**For HOL Light 2.20** **October 16, 2011**

# The HOL Light System
## REFERENCE

John Harrison University of Cambridge DSTO SRI International

# Preface

This volume is the reference manual for the HOL Light system. In contrast to the Tutorial, it is mainly intended for reference purposes, though some readers will find it productive to browse through it as part of the learning process. The main entries for the reference manual are generated from the same database that is used by the online HOL Light help system.

The entries that follow provide documentation on essentially all the pre-defined ML variable bindings in the HOL Light system. These include: general-purpose functions, such as ML functions for list processing, arithmetic, input/output, and interface configuration; functions for processing the types and terms of the HOL logic and for using the subgoal package; primitive and derived forward inference rules; tactics and tacticals; and pre-proved built-in theorems.

The manual entries for these ML identifiers are divided into two chapters. The first chapter is an alphabetical sequence of manual entries for all ML identifiers in the system except those identifiers that are bound to theorems (or pairs of theorems, etc.) The theorems are listed in the second chapter, roughly grouped into sections based on subject matter.

Our documentation does not cover basic functions in the OCaml toplevel, such as addition, string concatenation etc. In fact, relatively few native OCaml functions are used, and those are all documented in the Objective CAML Reference Manual:

`http://caml.inria.fr/pub/docs/manual-ocaml/index.html`

# Acknowledgements

This HOL Light Reference manual is derived from the original *REFERENCE* document for the HOL88 system, and generates the main body from online help entries in the same way and using essentially the same scripts. Many of these entries are minor edits of HOL88 originals, though plenty are also completely new. All in the latter group (and some of the former) were written by John Harrison. The re-use of the existing infrastructure was suggested by Steve Brackin.

The original HOL88 documentation project was managed by Mike Gordon at the Cambridge (UK) Research Center of SRI International, with the support of DSTO Australia. The main reference entries were written in a joint effort by members of the Cambridge HOL group. The original document design used LaTeX macros supplied by Elsa Gunter, Tom Melham and Larry Paulson. The typesetting of all three volumes was managed by Tom Melham. The conversion of the `troff` sources of *The ML Handbook* to LaTeX was done by Inder Dhingra and John Van Tassel. The cover design is by Arnold Smith, who used a photograph of a 'snow watching lantern' taken by Avra Cohn (in whose garden the original object resides). John Van Tassel composed the LaTeX picture of the lantern.

Many people other than those listed above have contributed to the HOL documentation effort, either by providing material, or by sending lists of errors in the first edition. Thanks to everyone who helped, and thanks to DSTO and SRI for their generous support.

# Contents

Chapter 1

# Pre-defined ML Identifiers

This chapter provides manual entries for all the pre-defined ML identifiers in the HOL system, except the identifiers that are bound to pre-proved theorems (for these, see chapter two). These include: general-purpose functions, such as functions for list processing, arithmetic, input/output, and interface configuration; functions for processing the types and terms of the HOL logic, for using the subgoal package; primitive and derived forward inference rules; and tactics and tacticals. The arrangement is alphabetical.

# ABBREV_TAC

`ABBREV_TAC : term -> (string * thm) list * term -> goalstate`

## Synopsis
Tactic to introduce an abbreviation.

## Description
The tactic `ABBREV_TAC ‘x = t‘` abbreviates any instances of the term `t` in the goal (assumptions or conclusion) with `x`, and adds a new assumption `t = x`. (Reversed so that rules like `ASM_REWRITE_TAC` will not immediately expand it again.) The LHS may be of the form `f x` in which case abstraction will happen first.

## Failure
Fails unless the left-hand side is a variable or a variable applied to a list of variable arguments.

## Example
```
# g ‘(12345 + 12345) + f(12345 + 12345) = f(12345 + 12345)‘;;
Warning: Free variables in goal: f
val it : goalstack = 1 subgoal (1 total)

‘(12345 + 12345) + f (12345 + 12345) = f (12345 + 12345)‘

#  e(ABBREV_TAC ‘n = 12345 + 12345‘);;
val it : goalstack = 1 subgoal (1 total)

 0 [‘12345 + 12345 = n‘]

‘n + f n = f n‘
```

## Uses
Convenient for abbreviating large and unwieldy expressions as a sort of ‘local definition’.

## See also
`EXPAND_TAC`.

# ABS_CONV

`ABS_CONV : conv -> conv`

## Synopsis
Applies a conversion to the body of an abstraction.

## Description
If `c` is a conversion that maps a term `` `t` `` to the theorem `|- t = t'`, then the conversion `ABS_CONV c` maps abstractions of the form `` `\x. t` `` to theorems of the form:

```
|- (\x. t) = (\x. t')
```

That is, `` ABS_CONV c `\x. t` `` applies `c` to the body of the abstraction `` `\x. t` ``.

## Failure
`ABS_CONV c tm` fails if `tm` is not an abstraction or if `tm` has the form `` `\x. t` `` but the conversion `c` fails when applied to the term `t`, or if the theorem returned has assumptions in which the abstracted variable `x` is free. The function returned by `ABS_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

## Example

```
# ABS_CONV SYM_CONV `\x. 1 = x`;;
val it : thm = |- (\x. 1 = x) = (\x. x = 1)
```

## See also
`GABS_CONV, RAND_CONV, RATOR_CONV, SUB_CONV.`

# ABS

`ABS : term -> thm -> thm`

## Synopsis
Abstracts both sides of an equation.

## Description

```
       A |- t1 = t2
------------------------  ABS `x`           [Where x is not free in A]
 A |- (\x.t1) = (\x.t2)
```

## Failure
If the theorem is not an equation, or if the variable `x` is free in the assumptions `A`.

### Example

```
# ABS `m:num` (REFL `m:num`);;
val it : thm = |- (\m. m) = (\m. m)
```

### Comments

This is one of HOL Light's 10 primitive inference rules.

### See also

ETA_CONV, EXT, MK_ABS.

## ABS_TAC

```
ABS_TAC : tactic
```

### Synopsis

Strips an abstraction from each side of an equational goal.

### Description

`ABS_TAC` reduces a goal of the form `A ?- (\x. s[x]) = (\y. t[y])` by stripping away the abstractions to give a new goal `A ?- s[x'] = t[x']` where `x'` is a variant of `x`, the bound variable on the left-hand side, chosen not to be free in the current goal's assumptions or conclusion. the function applications, giving the new goal `A ?- x = y`.

```
    A ?- (\x. s[x]) = (\y. t[y])
   ================================  ABS_TAC
        A ?- s[x'] = t[x']
```

### Failure

Fails unless the goal is equational, with both sides being abstractions.

### See also

AP_TERM_TAC, AP_THM_TAC, BINOP_TAC, MK_COMB_TAC.

## ACCEPT_TAC

```
ACCEPT_TAC : thm_tactic
```

## Synopsis
Solves a goal if supplied with the desired theorem (up to alpha-conversion).

## Description
`ACCEPT_TAC` maps a given theorem `th` to a tactic that solves any goal whose conclusion is alpha-convertible to the conclusion of `th`.

## Failure
`ACCEPT_TAC th (A ?- g)` fails if the term `g` is not alpha-convertible to the conclusion of the supplied theorem `th`.

## Example
The theorem `BOOL_CASES_AX = |- !t. (t <=> T) \/ (t <=> F)` can be used to solve the goal:

```
# g `!x. (x <=> T) \/ (x <=> F)`;;
```

by

```
# e(ACCEPT_TAC BOOL_CASES_AX);;
val it : goalstack = No subgoals
```

## Uses
Used for completing proofs by supplying an existing theorem, such as an axiom, or a lemma already proved. Often this can simply be done by rewriting, but there are times when greater delicacy is wanted.

## See also
`MATCH_ACCEPT_TAC`.

# AC

```
AC : thm -> term -> thm
```

## Synopsis
Proves equality of terms using associative, commutative, and optionally idempotence laws.

## Description

Suppose `_` is a function, which is assumed to be infix in the following syntax, and `acth` is a theorem expressing associativity and commutativity in the particular canonical form:

```
acth = |- m _ n = n _ m /\
          (m _ n) _ p = m _ n _ p /\
          m _ n _ p = n _ m _ p
```

Then `AC acth` will prove equations whose left and right sides can be made identical using these associative and commutative laws. If the input theorem also has idempotence property in this canonical form:

```
|- (p _ q = q _ p) /\
   ((p _ q) _ r = p _ q _ r) /\
   (p _ q _ r = q _ p _ r) /\
   (p _ p = p) /\
   (p _ p _ q = p _ q)
```

then idempotence will also be applied.

## Failure

Fails if the terms are not proved equivalent under the appropriate laws. This may happen because the input theorem does not have the correct canonical form. The latter problem will not in itself cause failure until it is applied to the term.

## Example

```
# AC ADD_AC `1 + 2 + 3 = 2 + 1 + 3`;;
val it : thm = |- 1 + 2 + 3 = 2 + 1 + 3
# AC CONJ_ACI `p /\ (q /\ p) <=> (p /\ q) /\ (p /\ q)`;;
val it : thm = |- p /\ q /\ p <=> (p /\ q) /\ p /\ q
```

## Comments

Note that pre-proved theorems in the correct canonical form for `AC` are already present for many standard operators, e.g. `ADD_AC`, `MULT_AC`, `INT_ADD_AC`, `INT_MUL_AC`, `REAL_ADD_AC`, `REAL_MUL_AC`, `CONJ_ACI`, `DISJ_ACI` and `INSERT_AC`. The underlying algorithm is not particularly delicate, and normalization under the associative/commutative/idempotent laws can be achieved by direct rewriting with the same canonical theorems. For some cases, specially optimized rules are available such as `CONJ_ACI_RULE` and `DISJ_ACI_RULE`.

## See also

`ASSOC_CONV`, `CONJ_ACI_RULE`, `DISJ_ACI_RULE`, `SYM_CONV`.

# aconv

```
aconv : term -> term -> bool
```

## Synopsis
Tests for alpha-convertibility of terms.

## Description
When applied to two terms, `aconv` returns `true` if they are alpha-convertible, and `false` otherwise.

## Failure
Never fails.

## Example
A simple case of alpha-convertibility is the renaming of a single quantified variable:

```
# aconv `?x. x <=> T` `?y. y <=> T`;;
val it : bool = true
```

but other cases can be more involved:

```
# aconv `\x y z. x + y + z` `\y x z. y + x + z`;;
val it : bool = true
```

## Comments
The code for alpha-conversion first checks for simple equality with pointer equality short-cutting, and can therefore often returns `true` without a full traversal.

In principle, most of the HOL Light deductive apparatus should work modulo alpha-conversion. At least, all the primitive inference rules do.

## See also
`ALPHA`, `ALPHA_CONV`.

# ADD_ASSUM

```
ADD_ASSUM : term -> thm -> thm
```

## Synopsis
Adds an assumption to a theorem.

## Description
When applied to a boolean term `s` and a theorem `A |- t`, the inference rule `ADD_ASSUM` returns the theorem `A u {s} |- t`.

```
       A |- t
   --------------  ADD_ASSUM `s`
    A u {s} |- t
```

`ADD_ASSUM` performs straightforward set union with the new assumption; it checks for identical assumptions, but not for alpha-equivalent ones. The position at which the new assumption is inserted into the assumption list should not be relied on.

## Failure
Fails unless the given term has type `bool`.

## Example

```
# ADD_ASSUM `q:bool` (ASSUME `p:bool`);;
val it : thm = p, q |- p
```

## See also
`ASSUME`, `UNDISCH`.

# a

```
a : 'a -> 'a list -> 'a * 'a list
```

## Synopsis
Parser that requires a specific item.

## Description
The call `a x` gives a parser that parses a single item that is exactly `x`, raising `Noparse` if the first item is something different.

## Failure
The call `a x` never fails, though the resulting parser may raise `Noparse`.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

# ALL_CONV

```
ALL_CONV : conv
```

## Synopsis

Conversion that always succeeds and leaves a term unchanged.

## Description

When applied to a term `` `t` ``, the conversion `ALL_CONV` returns the theorem `|- t = t`. It is just `REFL` explicitly regarded as a conversion.

## Failure

Never fails.

## Uses

Identity element for `THENC`.

## See also

`NO_CONV`, `REFL`.

# allpairs

```
allpairs : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

## Synopsis
Compute list of all results from applying function to pairs from two lists.

## Description
The call `allpairs f [x1;...;xm] [y1;...;yn]` returns the list of results `[f x1 y1; f x1 y2; ... ; f x1 yn`

## Failure
Never fails.

## Example

```
# allpairs (fun x y -> (x,y)) [1;2;3] [4;5];;
val it : (int * int) list = [(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]
```

## See also
`map2`, `zip`.

# ALL_TAC

```
ALL_TAC : tactic
```

## Synopsis
Passes on a goal unchanged.

## Description
`ALL_TAC` applied to a goal `g` simply produces the subgoal list `[g]`. It is the identity for the `THEN` tactical.

## Failure
Never fails.

## Example
Suppose we want to solve the goal:

```
# g `~(n MOD 2 = 0) <=> n MOD 2 = 1`;;
...
```

We could just solve it with `e ARITH_TAC`, but suppose we want to introduce a little lemma that `n MOD 2 < 2`, proving that by `ARITH_TAC`. We could do

```
# e(SUBGOAL_THEN `n MOD 2 < 2` ASSUME_TAC THENL
     [ARITH_TAC;
      ...rest of proof...]);;
```

However if we split off many lemmas, we get a deeply nested proof structure that's a bit confusing. In cases where the proofs of the lemmas are trivial one-liners like this we might just want to keep the proof basically linear with

```
# e(SUBGOL_THEN `n MOD 2 < 2` ASSUME_TAC THENL [ARITH_TAC; ALL_TAC] THEN
    ...rest of proof...);;
```

### Uses
Keeping proof structures linear, as in the above example, or convenient algebraic combinations in complicated tactic structures.

### See also
`NO_TAC, REPEAT, THENL.`

## ALL_THEN

`ALL_THEN : thm_tactical`

### Synopsis
Passes a theorem unchanged to a theorem-tactic.

### Description
For any theorem-tactic `ttac` and theorem `th`, the application `ALL_THEN ttac th` results simply in `ttac th`, that is, the theorem is passed unchanged to the theorem-tactic. `ALL_THEN` is the identity theorem-tactical.

### Failure
The application of `ALL_THEN` to a theorem-tactic never fails. The resulting theorem-tactic fails under exactly the same conditions as the original one

### Uses
Writing compound tactics or tacticals, e.g. terminating list iterations of theorem-tacticals.

### See also
`ALL_TAC, FAIL_TAC, NO_TAC, NO_THEN, THEN_TCL, ORELSE_TCL.`

# ALPHA_CONV

```
ALPHA_CONV : term -> term -> thm
```

## Synopsis

Renames the bound variable of a lambda-abstraction.

## Description

If `‘y‘` is a variable of type `ty` and `‘\x. t‘` is an abstraction in which the bound variable `x` also has type `ty` and `y` does not occur free in `t`, then `ALPHA_CONV ‘y‘ ‘\x. t‘` returns the theorem:

```
|- (\x. t) = (\y. t[y/x])
```

## Failure

Fails if the first argument is not a variable, the second is not an abstraction, if the types of the new variable and the bound variable in the abstraction differ, or if the new variable is already free in the body of the abstraction.

## Example

```
# ALPHA_CONV ‘y:num‘ ‘\x. x + 1‘;;
val it : thm = |- (\x. x + 1) = (\y. y + 1)

#  ALPHA_CONV ‘y:num‘ ‘\x. x + y‘;;
Exception: Failure "alpha: Invalid new variable".
```

## See also

`ALPHA`, `GEN_ALPHA_CONV`.

# alpha

```
alpha : term -> term -> term
```

## Synopsis

Changes the name of a bound variable.

## Description

The call `alpha ‘v’‘ ‘\v. t[v]‘` returns the second argument with the top bound variable changed to `v’`, and other variables renamed if necessary.

## Failure

Fails if the first term is not a variable, or if the second is not an abstraction, if the corresponding types are not the same, or if the desired new variable is already free in the abstraction.

## Example

```
# alpha ‘y:num‘ ‘\x y. x + y + 2‘;;
val it : term = ‘\y y’. y + y’ + 2‘

# alpha ‘y:num‘ ‘\x. x + y + 1‘;;
Exception: Failure "alpha: Invalid new variable".
```

## See also

`ALPHA`, `aconv`.

# ALPHA

```
ALPHA : term -> term -> thm
```

## Synopsis

Proves equality of alpha-equivalent terms.

## Description

When applied to a pair of terms `t1` and `t1’` which are alpha-equivalent, `ALPHA` returns the theorem `|- t1 = t1’`.

```
   -------------  ALPHA ‘t1‘ ‘t1’‘
    |- t1 = t1’
```

## Failure

Fails unless the terms provided are alpha-equivalent.

## Example

```
# ALPHA `!x:num. x = x` `!y:num. y = y`;;
val it : thm = |- (!x. x = x) <=> (!y. y = y)

# ALPHA `\w. w + z` `\z'. z' + z`;;
val it : thm = |- (\w. w + z) = (\z'. z' + z)
```

## See also

aconv, ALPHA_CONV, GEN_ALPHA_CONV.

# ANTE_RES_THEN

`ANTE_RES_THEN : thm_tactical`

## Synopsis

Resolves implicative assumptions with an antecedent.

## Description

Given a theorem-tactic `ttac` and a theorem `A |- t`, the function `ANTE_RES_THEN` produces a tactic that attempts to match `t` to the antecedent of each implication

```
Ai |- !x1...xn. ui ==> vi
```

(where `Ai` is just `!x1...xn. ui ==> vi`) that occurs among the assumptions of a goal. If the antecedent `ui` of any implication matches `t`, then an instance of `Ai u A |- vi` is obtained by specialization of the variables `x1`, ..., `xn` and type instantiation, followed by an application of modus ponens. Because all implicative assumptions are tried, this may result in several modus-ponens consequences of the supplied theorem and the assumptions. Tactics are produced using `ttac` from all these theorems, and these tactics are applied in sequence to the goal. That is,

```
ANTE_RES_THEN ttac (A |- t) g
```

has the effect of:

```
MAP_EVERY ttac [A1 u A |- v1; ...; Am u A |- vm] g
```

where the theorems `Ai u A |- vi` are all the consequences that can be drawn by a (single) matching modus-ponens inference from the implications that occur among the assumptions of the goal `g` and the supplied theorem `A |- t`.

## Failure

ANTE_RES_THEN ttac (A |- t) fails when applied to a goal g if any of the tactics produced by ttac (Ai u A |- vi), where Ai u A |- vi is the ith resolvent obtained from the theorem A |- t and the assumptions of g, fails when applied in sequence to g.

## See also

IMP_RES_THEN, MATCH_MP, MATCH_MP_TAC.

# ANTS_TAC

```
ANTS_TAC : tactic
```

## Synopsis

Split off antecedent of antecedent of goal as a new subgoal.

## Description

```
   A ?- (p ==> q) ==> r
  ====================== ANTS_TAC
   A ?- p   A ?- q ==> r
```

## Failure

Fails unless the goal is of the specified form.

## Uses

Convenient for focusing on assumptions of an implicational theorem that one wants to use.

## See also

MP_TAC.

# applyd

```
applyd : ('a, 'b) func -> ('a -> 'b) -> 'a -> 'b
```

## Synopsis

Applies a finite partial function, with a backup function for undefined points.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` is a finite partial function, `g` a conventional function and `x` an argument, `tryapply f g x` tries to apply `f` to `x` as with `apply f x`, but instead returns `g x` is `f` is undefined on `x`.

## Failure

Can only fail if the backup function fails.

## Example

```
# applyd undefined (fun x -> x) 1;;
val it : int = 1
# applyd (1 |=> 2) (fun x -> x) 1;;
val it : int = 2
```

## See also

`|->`, `|=>`, `apply`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# apply

```
apply : ('a, 'b) func -> 'a -> 'b
```

## Synopsis

Applies a finite partial function, failing on undefined points.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` is a finite partial function and `x` an argument, `apply f x` tries to apply `f` to `x` and fails if it is undefined.

## Example

```
# apply undefined 1;;
Exception: Failure "apply".
# apply (1 |=> 2) 1;;
val it : int = 2
```

## See also

|->, |=>, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefine, undefined.

# apply_prover

```
apply_prover : prover -> term -> thm
```

## Synopsis

Apply a prover to a term.

## Description

The HOL Light simplifier (e.g. as invoked by `SIMP_TAC`) allows provers of type `prover` to be installed into simpsets, to automatically dispose of side-conditions. These may maintain a state dynamically and augment it as more theorems become available (e.g. a theorem `p |- p` becomes available when simplifying the consequenct of an implication `` `p ==> q` ``). In order to allow maximal flexibility in the data structure used to maintain state, provers are set up in an 'object-oriented' style, where the context is part of the prover function itself. A call `` apply_prover p `tm` `` applies the prover with its current context to attempt to prove the term `tm`.

## Failure

The call `apply_prover p` never fails, but it may fail to prove the term.

## Uses

Mainly intended for users customizing the simplifier.

## Comments

I learned of this ingenious trick for maintaining context from Don Syme, who discovered it by reading some code written by Richard Boulton. I was told by Simon Finn that there are similar ideas in the functional language literature for simulating existential types.

## See also
augment, mk_prover, SIMP_CONV, SIMP_RULE, SIMP_TAC.

# AP_TERM

```
AP_TERM : term -> thm -> thm
```

## Synopsis
Applies a function to both sides of an equational theorem.

## Description
When applied to a term f and a theorem A |- x = y, the inference rule AP_TERM returns the theorem A |- f x = f y.

```
     A |- x = y
   ----------------  AP_TERM `f`
    A |- f x = f y
```

## Failure
Fails unless the theorem is equational and the supplied term is a function whose domain type is the same as the type of both sides of the equation.

## Example

```
# NUM_ADD_CONV `2 + 2`;;
val it : thm = |- 2 + 2 = 4

# AP_TERM `(+) 1` it;;
val it : thm = |- 1 + 2 + 2 = 1 + 4
```

## See also
AP_THM, MK_COMB.

# AP_TERM_TAC

```
AP_TERM_TAC : tactic
```

### Synopsis
Strips a function application from both sides of an equational goal.

### Description
`AP_TERM_TAC` reduces a goal of the form `A ?- f x = f y` by stripping away the function applications, giving the new goal `A ?- x = y`.

```
    A ?- f x = f y
   ================  AP_TERM_TAC
     A ?- x = y
```

### Failure
Fails unless the goal is equational, with both sides being applications of the same function.

### See also
`ABS_TAC`, `AP_TERM`, `AP_THM_TAC`, `BINOP_TAC`, `MK_COMB_TAC`.

## AP_THM

```
AP_THM : thm -> term -> thm
```

### Synopsis
Proves equality of equal functions applied to a term.

### Description
When applied to a theorem `A |- f = g` and a term `x`, the inference rule `AP_THM` returns the theorem `A |- f x = g x`.

```
      A |- f = g
   ----------------  AP_THM (A |- f = g) `x`
    A |- f x = g x
```

### Failure
Fails unless the conclusion of the theorem is an equation, both sides of which are functions whose domain type is the same as that of the supplied term.

## Example

```
# REWRITE_RULE[GSYM FUN_EQ_THM] ADD1;;
val it : thm = |- SUC = (\m. m + 1)

# AP_THM it `11`;;
val it : thm = |- SUC 11 = (\m. m + 1) 11
```

## See also

AP_TERM, ETA_CONV, EXT, MK_COMB.

# AP_THM_TAC

```
AP_THM_TAC : tactic
```

## Synopsis

Strips identical operands from functions on both sides of an equation.

## Description

When applied to a goal of the form `A ?- f x = g x`, the tactic `AP_THM_TAC` strips away the operands of the function application:

```
    A ?- f x = g x
   ================  AP_THM_TAC
      A ?- f = g
```

## Failure

Fails unless the goal has the above form, namely an equation both sides of which consist of function applications to the same argument.

## See also

ABS_TAC, AP_TERM_TAC, AP_THM, BINOP_TAC, MK_COMB_TAC.

# ARITH_RULE

```
ARITH_RULE : term -> thm
```

## Synopsis

Automatically proves natural number arithmetic theorems needing basic rearrangement and linear inequality reasoning only.

## Description

The function `ARITH_RULE` can automatically prove natural number theorems using basic algebraic normalization and inequality reasoning. For nonlinear equational reasoning use `NUM_RING`.

## Failure

Fails if the term is not boolean or if it cannot be proved using the basic methods employed, e.g. requiring nonlinear inequality reasoning.

## Example

```
# ARITH_RULE `x = 1 ==> y <= 1 \/ x < y`;;
val it : thm = |- x = 1 ==> y <= 1 \/ x < y

# ARITH_RULE `x <= 127 ==> ((86 * x) DIV 256 = x DIV 3)`;;
val it : thm = |- x <= 127 ==> (86 * x) DIV 256 = x DIV 3

# ARITH_RULE
   `2 * a * b EXP 2 <= b * a * b ==> (SUC c - SUC(a * b * b) <= c)`;;
val it : thm =
  |- 2 * a * b EXP 2 <= b * a * b ==> SUC c - SUC (a * b * b) <= c
```

## Uses

Disposing of elementary arithmetic goals.

## See also

`ARITH_CONV`, `ARITH_TAC`, `INT_ARITH`, `NUM_RING`, `REAL_ARITH`, `REAL_FIELD`, `REAL_RING`.

# ARITH_TAC

```
ARITH_TAC : tactic
```

## Synopsis

Tactic for proving arithmetic goals needing basic rearrangement and linear inequality reasoning only.

## Description

ARITH_TAC will automatically prove goals that require basic algebraic normalization and inequality reasoning over the natural numbers. For nonlinear equational reasoning use NUM_RING and derivatives.

## Failure

Fails if the automated methods do not suffice.

## Example

```
# g `1 <= x /\ x <= 3 ==> x = 1 \/ x = 2 \/ x = 3`;;
Warning: Free variables in goal: x
val it : goalstack = 1 subgoal (1 total)

`1 <= x /\ x <= 3 ==> x = 1 \/ x = 2 \/ x = 3`

# e ARITH_TAC;;
val it : goalstack = No subgoals
```

## Uses

Solving basic arithmetic goals.

## See also

ARITH_RULE, INT_ARITH_TAC, NUM_RING, REAL_ARITH_TAC.

# ASM_CASES_TAC

```
ASM_CASES_TAC : term -> tactic
```

## Synopsis

Given a term, produces a case split based on whether or not that term is true.

## Description

Given a term u, ASM_CASES_TAC applied to a goal produces two subgoals, one with u as an assumption and one with ~u:

```
              A ?-  t
================================  ASM_CASES_TAC `u`
 A u {u} ?- t   A u {~u} ?- t
```

## Failure

Fails if u does not have boolean type.

## Example

The tactic `ASM_CASES_TAC ‘&0 <= u‘` can be used to produce a case analysis on `‘&0 <= u‘`:

```
# g ‘&0 <= (u:real) pow 2‘;;
Warning: Free variables in goal: u
val it : goalstack = 1 subgoal (1 total)

‘&0 <= u pow 2‘

# e(ASM_CASES_TAC ‘&0 <= u‘);;
val it : goalstack = 2 subgoals (2 total)

 0 [‘~(&0 <= u)‘]

‘&0 <= u pow 2‘

 0 [‘&0 <= u‘]

‘&0 <= u pow 2‘
```

## Uses

Performing a case analysis according to whether a given term is true or false.

## See also

`BOOL_CASES_TAC`, `COND_CASES_TAC`, `ITAUT`, `DISJ_CASES_TAC`, `STRUCT_CASES_TAC`, `TAUT`.

# ASM

```
ASM : (thm list -> tactic) -> thm list -> tactic
```

## Synopsis

Augments a tactic's theorem list with the assumptions.

## Description

If `tac` is a tactic that expects a list of theorems as its arguments, e.g. `REWRITE_TAC` or `MESON_TAC`, then `ASM tac` converts it to a tactic where that list is augmented by the goal's assumptions.

## Failure

Never fails (though the resulting tactic may do).

## Example

```
The inbuilt {\small\verb%ASM_REWRITE_TAC%} is in fact defined as just {\small\verb%ASM REWRITE_
```

## See also

ASSUM_LIST, FREEZE_THEN.

# ASM_FOL_TAC

```
ASM_FOL_TAC : (string * thm) list * term -> goalstate
```

## Synopsis

Fix up function arities for first-order proof search.

## Description

This function attempts to make the assumptions of a goal more 'first-order'. Functions that are not consistently used with the same arity, e.g. a function `f` that is sometimes applied `f(a)` and sometimes used as an argument to other functions, `g(f)`, will be identified. Applications of the function will then be modified by the introduction of the identity function `I` (which can be thought of later as binary 'function application') so that `f(a)` becomes `I f a`. This gives a more natural formulation as a prelude to traditional first-order proof search.

## Failure

Never fails.

## Comments

This function is not intended for general use, but is part of the initial normalization in `MESON` and `MESON_TAC`.

## See also

MESON, MESON_TAC.

# ASM_MESON_TAC

```
ASM_MESON_TAC : thm list -> tactic
```

## Synopsis

Automated first-order proof search tactic using assumptions of goal.

## Description

A call to `ASM_MESON_TAC[theorems]` will attempt to establish the goal using pure first-order reasining, taking `theorems` and the assumptions of the goal as the starting-point. It will usually either solve the goal completely or run for an infeasible length of time before terminating, but it may sometimes fail quickly. For more details, see `MESON` or `MESON_TAC`.

## Failure

Fails if the goal is unprovable within the search bounds, though not necessarily in a feasible amount of time.

## See also

`GEN_MESON_TAC`, `MESON`, `MESON_TAC`.

# ASM_REWRITE_RULE

```
ASM_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis

Rewrites a theorem including built-in rewrites and the theorem's assumptions.

## Description

`ASM_REWRITE_RULE` rewrites with the tautologies in `basic_rewrites`, the given list of theorems, and the set of hypotheses of the theorem. All hypotheses are used. No ordering is specified among applicable rewrites. Matching subterms are searched for recursively, starting with the entire term of the conclusion and stopping when no rewritable expressions remain. For more details about the rewriting process, see `GEN_REWRITE_RULE`. To avoid using the set of basic tautologies, see `PURE_ASM_REWRITE_RULE`.

## Failure

`ASM_REWRITE_RULE` does not fail, but may result in divergence. To prevent divergence where it would occur, `ONCE_ASM_REWRITE_RULE` can be used.

## See also

`GEN_REWRITE_RULE`, `ONCE_ASM_REWRITE_RULE`, `PURE_ASM_REWRITE_RULE`,
`PURE_ONCE_ASM_REWRITE_RULE`, `REWRITE_RULE`.

# ASM_REWRITE_TAC

```
ASM_REWRITE_TAC : thm list -> tactic
```

## Synopsis

Rewrites a goal including built-in rewrites and the goal's assumptions.

## Description

`ASM_REWRITE_TAC` generates rewrites with the tautologies in `basic_rewrites`, the set of assumptions, and a list of theorems supplied by the user. These are applied top-down and recursively on the goal, until no more matches are found. The order in which the set of rewrite equations is applied is an implementation matter and the user should not depend on any ordering. Rewriting strategies are described in more detail under `GEN_REWRITE_TAC`. For omitting the common tautologies, see the tactic `PURE_ASM_REWRITE_TAC`.

## Failure

`ASM_REWRITE_TAC` does not fail, but it can diverge in certain situations. For rewriting to a limited depth, see `ONCE_ASM_REWRITE_TAC`. The resulting tactic may not be valid if the applicable replacement introduces new assumptions into the theorem eventually proved.

## Example

The use of assumptions in rewriting, specially when they are not in an obvious equational form, is illustrated below:

```
# g `P ==> (P /\ Q /\ R <=> R /\ Q /\ P)`;;
Warning: Free variables in goal: P, Q, R
val it : goalstack = 1 subgoal (1 total)

`P ==> (P /\ Q /\ R <=> R /\ Q /\ P)`

# e DISCH_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [`P`]

`P /\ Q /\ R <=> R /\ Q /\ P`

# e(ASM_REWRITE_TAC[]);;
val it : goalstack = 1 subgoal (1 total)

 0 [`P`]

`Q /\ R <=> R /\ Q`
```

## See also

`basic_rewrites`, `GEN_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_ASM_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

# ASM_SIMP_TAC

```
ASM_SIMP_TAC : thm list -> tactic
```

## Synopsis

Perform simplification of goal by conditional contextual rewriting using assumptions and built-in simplifications.

## Description

A call to `ASM_SIMP_TAC[theorems]` will apply conditional contextual rewriting with `theorems` and the current assumptions of the goal to the goal's conclusion, as well as the default simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of rewriting, see `SIMP_CONV`. If the extra generality of contextual conditional rewriting is not needed, `REWRITE_TAC` is usually more efficient.

## Failure
Never fails, but may loop indefinitely.

## See also
ASM_REWRITE_TAC, SIMP_CONV, SIMP_TAC, REWRITE_TAC.

# ASSOC_CONV

```
ASSOC_CONV : thm -> term -> thm
```

## Synopsis
Right-associates a term with respect to an associative binary operator.

## Description
The conversion ASSOC_CONV expects a theorem asserting that a certain binary operator is associative, in the standard form (with optional universal quantifiers):

```
x op (y op z) = (x op y) op z
```

It is then applied to a term, and will right-associate any toplevel combinations built up from the operator op. Note that if op is polymorphic, the type instance of the theorem needs to be the same as in the term to which it is applied.

## Failure
May fail if the theorem is malformed. On application to the term, it never fails, but returns a reflexive theorem when itis inapplicable.

## Example

```
# ASSOC_CONV ADD_ASSOC `((1 + 2) + 3) + (4 + 5) + (6 + 7)`;;
val it : thm = |- ((1 + 2) + 3) + (4 + 5) + 6 + 7 = 1 + 2 + 3 + 4 + 5 + 6 + 7

# ASSOC_CONV CONJ_ASSOC `((p /\ q) /\ (r /\ s)) /\ t`;;
val it : thm = |- ((p /\ q) /\ r /\ s) /\ t <=> p /\ q /\ r /\ s /\ t
```

## See also
AC, CNF_CONV, CONJ_ACI_RULE, DISJ_ACI_RULE, DNF_CONV.

# assocd

```
assocd : 'a -> ('a * 'b) list -> 'b -> 'b
```

## Synopsis

Looks up item in association list taking default in case of failure.

## Description

The call `assocd x [x1,y1; ...; xn,yn] y` returns the first `yi` in the list where the corresponding `xi` is the same as `x`. If there is no such item, it returns the value `y`. This is similar to `assoc` except that the latter will fail rather than take a default.

## Failure

Never fails.

## Example

```
# assocd 2 [1,2; 2,4; 3,6] (-1);;
val it : int = 4
# assocd 4 [1,2; 2,4; 3,6] (-1);;
val it : int = -1
```

## Uses

Simple lookup without exception handling.

## See also

`assoc, rev_assocd.`

# assoc

```
assoc : 'a -> ('a * 'b) list -> 'b
```

## Synopsis

Searches a list of pairs for a pair whose first component equals a specified value.

## Description

`assoc x [(x1,y1);...;(xn,yn)]` returns the first `yi` in the list such that `xi` equals `x`.

## Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

## Example

```
# assoc 2 [1,4; 3,2; 2,5; 2,6];;
val it : int = 5
```

## See also

`rev_assoc`, `find`, `mem`, `tryfind`, `exists`, `forall`.

# ASSUME

```
ASSUME : term -> thm
```

## Synopsis

Introduces an assumption.

## Description

When applied to a term `t`, which must have type `bool`, the inference rule `ASSUME` returns the theorem `t |- t`.

```
   --------  ASSUME `t`
    t |- t
```

## Failure

Fails unless the term `t` has type `bool`.

## Example

```
# ASSUME `p /\ q`;;
val it : thm = p /\ q |- p /\ q
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

`ADD_ASSUM`, `REFL`.

# ASSUME_TAC

```
ASSUME_TAC : thm_tactic
```

## Synopsis

Adds an assumption to a goal.

## Description

Given a theorem `th` of the form `A' |- u`, and a goal, `ASSUME_TAC th` adds `u` to the assumptions of the goal.

```
         A ?- t
    ==============  ASSUME_TAC (A' |- u)
     A u {u} ?- t
```

Note that unless `A'` is a subset of `A`, this tactic is invalid. The new assumption is unlabelled; for a named assumption use `LABEL_TAC`.

## Failure

Never fails.

## Example

One can add an external theorem as an assumption if desired, for example so that `ASM_REWRITE_TAC[]` will automatically apply it. But usually the theorem is derived from some theorem-tactical, e.g. by discharging the antecedent of an implication or doing forward inference on another assumption. For example iff faced with the goal:

```
  # g `0 = x ==> f(2 * x) = f(x * f(x))`;;
```

one might not want to just do `DISCH_TAC` or `STRIP_TAC` because the assumption will be

`0 = x`. One can swap it first then put it on the assumptions by:

```
# e(DISCH_THEN(ASSUME_TAC o SYM));;
val it : goalstack = 1 subgoal (1 total)

 0 [`x = 0`]

`f (2 * x) = f (x * f x)`
```

after which the goal can very easily be solved:

```
# e(ASM_REWRITE_TAC[MULT_CLAUSES]);;
val it : goalstack = No subgoals
```

## Uses

Useful as a parameter to various theorem-tacticals such as `X_CHOOSE_THEN`, `DISCH_THEN` etc. when it is simply desired to add the theorem that has been deduced to the assumptions rather than used further at once.

## See also

`ACCEPT_TAC`, `LABEL_TAC`, `STRIP_ASSUME_TAC`.

# ASSUM_LIST

```
ASSUM_LIST : (thm list -> tactic) -> tactic
```

## Synopsis

Applies a tactic generated from the goal's assumption list.

## Description

When applied to a function of type `thm list -> tactic` and a goal, `ASSUM_LIST` constructs a tactic by applying `f` to a list of `ASSUME`d assumptions of the goal, then applies that tactic to the goal.

```
ASSUM_LIST f ({A1;...;An} ?- t)
      = f [A1 |- A1; ... ; An |- An] ({A1;...;An} ?- t)
```

## Failure

Fails if the function fails when applied to the list of `ASSUME`d assumptions, or if the resulting tactic fails when applied to the goal.

## Comments

There is nothing magical about `ASSUM_LIST`: the same effect can usually be achieved just as conveniently by using `ASSUME a` wherever the assumption `a` is needed. If `ASSUM_LIST` is used, it is extremely unwise to use a function which selects elements from its argument list by number, since the ordering of assumptions should not be relied on.

## Example

The tactic:

```
ASSUM_LIST(MP_TAC o end_itlist CONJ)
```

adds a conjunction of all assumptions as an antecedent of a goal.

## Uses

Making more careful use of the assumption list than simply rewriting.

## See also

`ASM_REWRITE_TAC, EVERY_ASSUM, POP_ASSUM, POP_ASSUM_LIST, REWRITE_TAC.`

# atleast

```
atleast : int -> ('a -> 'b * 'a) -> 'a -> 'b list * 'a
```

## Synopsis

Parses at least a given number of successive items using given parser.

## Description

If `p` is a parser and `n` an integer, `atleast n p` is a new parser that attempts to parse at least `n` successive items using parser `p` and fails otherwise. Unless `n` is positive, this is equivalent to `many p`.

## Failure

The call to `atleast n p` itself never fails.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

### See also
`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

## aty

```
aty : hol_type
```

### Synopsis
The type variable `` `:A` ``.

### Description
This name is bound to the HOL type `:A`.

### Failure
Not applicable.

### Uses
Exploiting the very common type variable `:A` inside derived rules (e.g. an instantiation list for `inst` or `type_subst`) without the inefficiency or inconvenience of calling a quotation parser or explicit constructor.

### See also
`bty`, `bool_ty`.

## augment

```
augment : prover -> thm list -> prover
```

### Synopsis
Augments a prover's context with new theorems.

### Description
The HOL Light simplifier (e.g. as invoked by `SIMP_TAC`) allows provers of type `prover` to be installed into simpsets, to automatically dispose of side-conditions. These may maintain

a state dynamically and augment it as more theorems become available (e.g. a theorem `p |- p` becomes available when simplifying the consequenct of an implication `‘p ==> q‘`). In order to allow maximal flexibility in the data structure used to maintain state, provers are set up in an ‘object-oriented’ style, where the context is part of the prover function itself. A call `augment p thl` maps a prover `p` to a new prover with theorems `thl` added to the initial state.

## Failure
Never fails unless the prover is abnormal.

## Uses
This is mostly for experts wishing to customize the simplifier.

## Comments
I learned of this ingenious trick for maintaining context from Don Syme, who discovered it by reading some code written by Richard Boulton. I was told by Simon Finn that there are similar ideas in the functional language literature for simulating existential types.

## See also
`apply_prover`, `mk_prover`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# AUGMENT_SIMPSET

```
AUGMENT_SIMPSET : thm -> simpset -> simpset
```

## Synopsis
Augment context of a simpset with a list of theorems.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a ‘simpset’. Given a list of theorems `thl` and a simpset `ss`, the call `AUGMENT_SIMPSET thl ss` augments the state of the simpset, adding the theorems as new rewrite rules and also making any provers in the simpset process the new context appropriately.

## Failure
Never fails unless some of the simpset functions are ill-formed.

## Uses
Mostly for experts wishing to customize the simplifier.

### See also
augment, SIMP_CONV.

## axioms

```
axioms : unit -> thm list
```

### Synopsis
Returns the current set of axioms.

### Description
A call `axioms()` returns the current list of axioms.

### Failure
Never fails.

### Example
Under normal circumstances, the list of axioms will be as follows, containing the axioms of infinity, choice and extensionality.

```
# axioms();;
val it : thm list =
  [|- ?f. ONE_ONE f /\ ~ONTO f; |- !P x. P x ==> P ((@) P);
   |- !t. (\x. t x) = t]
```

If other axioms are used, the consistency of the resulting theory cannot be guaranteed. However, new definitions and type definitions are always safe and are not considered as true 'axioms'.

### See also
new_definition, define, the_definitions, new_axiom.

## basic_congs

```
basic_congs : unit -> thm list
```

### Synopsis
Lists the congruence rules used by the simplifier.

## Description

The HOL Light simplifier (as invoked by `SIMP_TAC` etc.) uses congruence rules to determine how it uses context when descending through a term. These are essentially theorems showing how to decompose one equality to a series of other inequalities in context. A call to `basic_congs()` returns those congruences that are built into the system.

## Failure

Never fails.

## Example

Here is the effect in HOL Light's initial state:

```
# basic_congs();;
val it : thm list =
  [|- (!x. x IN s ==> f x = g x) ==> sum s (\i. f i) = sum s g;
   |- (!i. a <= i /\ i <= b ==> f i = g i)
      ==> sum (a..b) (\i. f i) = sum (a..b) g;
   |- (!x. p x ==> f x = g x) ==> sum {y | p y} (\i. f i) = sum {y | p y} g;
   |- (!x. x IN s ==> f x = g x) ==> nsum s (\i. f i) = nsum s g;
   |- (!i. a <= i /\ i <= b ==> f i = g i)
      ==> nsum (a..b) (\i. f i) = nsum (a..b) g;
   |- (!x. p x ==> f x = g x) ==> nsum {y | p y} (\i. f i) = nsum {y | p y} g;
   |- (g <=> g')
      ==> (g' ==> t = t')
      ==> (~g' ==> e = e')
      ==> (if g then t else e) = (if g' then t' else e');
   |- (p <=> p') ==> (p' ==> (q <=> q')) ==> (p ==> q <=> p' ==> q')]
```

## See also

`extend_basic_congs, set_basic_congs, SIMP_CONV, SIMP_RULE, SIMP_TAC.`

# basic_convs

```
basic_convs : unit -> (string * (term * conv)) list
```

## Synopsis

List the current default conversions used in rewriting and simplification.

## Description

The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_` variants. A call to `basic_convs()` returns the current set of conversions.

### Failure
Never fails.

### Example
In the default HOL Light state the only conversion is for generalized beta reduction. All the other default simplifications are done by rewrite rules.

```
# basic_convs();;
val it : (string * (term * conv)) list =
  [("GEN_BETA_CONV", (`GABS (\a. b) c`, <fun>))]
```

### See also
`basic_rewrites`, `extend_basic_convs`, `set_basic_convs`.

## basic_net

`basic_net : unit -> gconv net`

### Synopsis
Returns the term net used to optimize access to default rewrites and conversions.

### Description
The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_` variants. Internally, these are maintained in a term net (see `enter` and `lookup` for more information), and a call to `basic_net()` returns that net.

### Failure
Never fails.

### Uses
Only useful for those who are delving deep into the implementation of rewriting.

### See also
`basic_convs`, `basic_rewrites`, `enter`, `lookup`.

## basic_prover

`basic_prover : (simpset -> 'a -> term -> thm) -> simpset -> 'a -> term -> thm`

## Synopsis
The basic prover use function used in the simplifier.

## Description
The HOL Light simplifier (e.g. as invoked by `SIMP_TAC`) allows provers of type `prover` to be installed into simpsets, to automatically dispose of side-conditions. There is another component of the simpset that controls how these are applied to unproven subgoals arising in simplification. The `basic_prover` function, which is used in all the standard simpsets, simply tries to simplify the goals with the rewrites as far as possible, then tries the provers one at a time on the resulting subgoals till one succeeds.

## Failure
Never fails, though the later application to a term may fail to prove it.

## See also
`mk_prover`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# basic_rectype_net

```
basic_rectype_net : (int * (term -> thm)) net ref
```

## Synopsis
Net of injectivity and distinctness properties for recursive type constructors.

## Description
HOL Light maintains a net of theorems used to simplify equations between elements of recursive datatypes; essentially these include injectivity and distinctness, e.g. `CONS_11` and `NOT_CONS_NIL` for lists. This net is used in some situations where such things need to be proved automatically, notably in `define`. A call to `basic_rectype_net()` returns that net. It is automatically updated whenever a type is defined by `define_type`.

## Failure
Never fails.

## See also
`cases`, `define`, `distinctness`, `GEN_BETA_CONV`, `injectivity`.

# basic_rewrites

`basic_rewrites : unit -> thm list`

## Synopsis
Returns the set of built-in theorems used, by default, in rewriting.

## Description
The list of theorems returned by `basic_rewrites()` is applied by default in rewriting conversions, rules and tactics such as `ONCE_REWRITE_CONV`, `REWRITE_RULE` and `SIMP_TAC`, though not in the 'pure' variants like `PURE_REWRITE_TAC`. This default set can be modified using `extend_basic_rewrites`, `set_basic_rewrites`. Other conversions, not necessarily expressible as rewriting with a theorem, can be added using `set_basic_convs` and `extend_basic_convs` and examined by `basic_convs`.

## Example
The following shows the list of default rewrites in the standard HOL Light state. Most of them are basic logical tautologies.

```
# basic_rewrites();;
val it : thm list =
  [|- FST (x,y) = x; |- SND (x,y) = y; |- FST x,SND x = x;
   |- (if x = x then y else z) = y; |- (if T then t1 else t2) = t1;
   |- (if F then t1 else t2) = t2; |- ~ ~t <=> t; |- ~T <=> F; |- ~F <=> T;
   |- (@y. y = x) = x; |- x = x <=> T; |- (T <=> t) <=> t;
   |- (t <=> T) <=> t; |- (F <=> t) <=> ~t; |- (t <=> F) <=> ~t; |- ~T <=> F;
   |- ~F <=> T; |- T /\ t <=> t; |- t /\ T <=> t; |- F /\ t <=> F;
   |- t /\ F <=> F; |- t /\ t <=> t; |- T \/ t <=> T; |- t \/ T <=> T;
   |- F \/ t <=> t; |- t \/ F <=> t; |- t \/ t <=> t; |- T ==> t <=> t;
   |- t ==> T <=> T; |- F ==> t <=> T; |- t ==> t <=> T; |- t ==> F <=> ~t;
   |- (!x. t) <=> t; |- (?x. t) <=> t; |- (\x. f x) y = f y;
   |- x = x ==> p <=> p]
```

## Uses
The `basic_rewrites` are included in the set of equations used by some of the rewriting tools.

## See also
`extend_basic_rewrites`, `set_basic_rewrites`, `set_basic_convs`, `extend_basic_convs`, `basic_convs`, `REWRITE_CONV`, `REWRITE_RULE`, `REWRITE_TAC`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# basic_ss

```
basic_ss : thm list -> simpset
```

## Synopsis
Construct a straightforward simpset from a list of theorems.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset'. A call `basic_ss thl` gives a straightforward simpset used by the default simplifier instances like `SIMP_TAC`, which has the given theorems as well as the basic rewrites and conversions, and no other provers.

## Failure
Never fails.

## See also
`basic_convs`, `basic_rewrites`, `empty_ss`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# b

```
b : unit -> goalstack
```

## Synopsis
Restores the proof state, undoing the effects of a previous expansion.

## Description
The function `b` is part of the subgoal package. It allows backing up from the last state change (caused by calls to `e`, `g`, `r`, `set_goal` etc.) The package maintains a backup list of previous proof states. A call to `b` restores the state to the previous state (which was on top of the backup list).

## Failure
The function `b` will fail if the backup list is empty.

## Example

```
# g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3]`

# e CONJ_TAC;;
val it : goalstack = 2 subgoals (2 total)

`TL [1; 2; 3] = [2; 3]`

`HD [1; 2; 3] = 1`

# b();;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3]`
```

## Uses

Back tracking in a goal-directed proof to undo errors or try different tactics.

## See also

e, g, p, r, set_goal, top_goal, top_thm.

# BETA_CONV

```
BETA_CONV : term -> thm
```

## Synopsis

Performs a simple beta-conversion.

## Description

The conversion BETA_CONV maps a beta-redex `(\x.u)v` to the theorem

```
|- (\x.u)v = u[v/x]
```

where u[v/x] denotes the result of substituting v for all free occurrences of x in u, after renaming sufficient bound variables to avoid variable capture. This conversion is one of the primitive inference rules of the HOL system.

## Failure

`BETA_CONV tm` fails if `tm` is not a beta-redex.

## Example

```
# BETA_CONV `(\x. x + 1) y`;;
val it : thm = |- (\x. x + 1) y = y + 1

# BETA_CONV `(\x y. x + y) y`;;
val it : thm = |- (\x y. x + y) y = (\y'. y + y')
```

## Comments

The HOL Light primitive rule `BETA` is the special case where the argument is the same as the bound variable. If you know that you are in this case, `BETA` is significantly more efficient. Though traditionally a primitive, `BETA_CONV` is actually a derived rule in HOL Light.

## See also

`GEN_BETA_CONV, BETA, BETA_RULE, BETA_TAC.`

# BETA

`BETA : term -> thm`

## Synopsis

Special primitive case of beta-reduction.

## Description

Given a term of the form `(\x. t[x]) x`, i.e. a lambda-term applied to exactly the same variable that occurs in the abstraction, `BETA` returns the theorem `|- (\x. t[x]) x = t[x]`.

## Failure

Fails if the term is not of the required form.

## Example

```
# BETA `(\n. n + 1) n`;;
val it : thm = |- (\n. n + 1) n = n + 1
```

Note that more general beta-reduction is not handled by `BETA`, but will be by `BETA_CONV`:

```
# BETA `(\n. n + 1) m`;;
Exception: Failure "BETA: not a trivial beta-redex".
# BETA_CONV `(\n. n + 1) m`;;
val it : thm = |- (\n. n + 1) m = m + 1
```

## Uses

This is more efficient than `BETA_CONV` in the special case in which it works, because no traversal and replacement of the body of the abstraction is needed.

## Comments

This is one of HOL Light's 10 primitive inference rules. The more general case of beta-reduction, where a lambda-term is applied to any term, is implemented by `BETA_CONV`, derived in terms of this primitive.

## See also

`BETA_CONV`.

# BETA_RULE

`BETA_RULE : thm -> thm`

## Synopsis

Beta-reduces all the beta-redexes in the conclusion of a theorem.

## Description

When applied to a theorem `A |- t`, the inference rule `BETA_RULE` beta-reduces all beta-redexes, at any depth, in the conclusion `t`. Variables are renamed where necessary to avoid free variable capture.

```
    A |- ....((\x. s1) s2)....
  ----------------------------  BETA_RULE
      A |- ....(s1[s2/x])....
```

## Failure

Never fails, but will have no effect if there are no beta-redexes.

## Example

The following example is a simple reduction which illustrates variable renaming:

```
# let x = ASSUME `f = ((\x y. x + y) y)`;;
val x : thm = f = (\x y. x + y) y |- f = (\x y. x + y) y

# BETA_RULE x;;
val it : thm = f = (\x y. x + y) y |- f = (\y'. y + y')
```

## See also

BETA_CONV, BETA_TAC, GEN_BETA_CONV.

# BETAS_CONV

BETAS_CONV : conv

## Synopsis

Beta conversion over multiple arguments.

## Description

Given a term t of the form `` `(\x1 ... xn. t[x1,...,xn]) s1 ... sn` ``, the call BETAS_CONV t returns

```
|- (\x1 ... xn. t[x1,...,xn]) s1 ... sn = t[s1,...,sn]
```

## Failure

Fails if the term is not of the form shown, for some n.

## Example

```
# BETAS_CONV `(\x y. x + y) 1 2`;;
val it : thm = |- (\x y. x + y) 1 2 = 1 + 2
```

## See also

BETA_CONV, RIGHT_BETAS.

# BETA_TAC

BETA_TAC : tactic

## Synopsis

Beta-reduces all the beta-redexes in the conclusion of a goal.

## Description

When applied to a goal `A ?- t`, the tactic `BETA_TAC` produces a new goal which results from beta-reducing all beta-redexes, at any depth, in `t`. Variables are renamed where necessary to avoid free variable capture.

```
    A ?- ...((\x. s1) s2)...
  ==========================  BETA_TAC
     A ?- ...(s1[s2/x])...
```

## Failure

Never fails, but will have no effect if there are no beta-redexes.

## Comments

Beta-reduction, and indeed, generalized beta reduction (`GEN_BETA_CONV`) are already among the basic rewrites, so happen anyway simply on `REWRITE_TAC[]`. But occasionally it is convenient to be able to invoke them separately.

## See also

`BETA_CONV`, `BETA_RULE`, `GEN_BETA_CONV`.

# BINDER_CONV

```
BINDER_CONV : conv -> term -> thm
```

## Synopsis

Applies conversion to the body of a binder.

## Description

If `c` is a conversion such that `c ‘t‘` returns `|- t = t’`, then `BINDER_CONV c ‘b (\x. t)‘` returns `|- b (\x. t) = b (\x. t’)`, i.e. applies the core conversion to the body of a ‘binder’. In fact, `b` here can be any term, but it is typically a binder constant such as a quantifier.

## Failure

Fails if the core conversion does, or if the theorem returned by it is not of the right form.

## Example

```
# BINDER_CONV SYM_CONV `@n. n = m + 1`;;
val it : thm = |- (@n. n = m + 1) = (@n. m + 1 = n)

# BINDER_CONV (REWR_CONV SWAP_FORALL_THM) `!x y z. x + y + z = y + x + z`;;
val it : thm =
  |- (!x y z. x + y + z = y + x + z) <=> (!x z y. x + y + z = y + x + z)
```

## See also

ABS_CONV, RAND_CONV, RATOR_CONV.

# binders

`binders : unit -> string list`

## Synopsis

Lists the binders.

## Description

The function `binders` returns a list of all the binders declared so far. A binder `b` is then parsed in constructs like `b x. t[x]` as an abbreviation for `(b) (\x. t[x])`. The set of binders can be changed with `parse_as_binder` and `unparse_as_binder`.

## Failure

Never fails

## Example

```
# binders();;
val it : string list = ["\\"; "!"; "?"; "?!"; "@"; "minimal"; "lambda"]
```

## See also

parse_as_binder, parses_as_binder, parse_as_infix, parse_as_prefix, unparse_as_binder.

# BINOP_CONV

`BINOP_CONV : (term -> thm) -> term -> thm`

## Synopsis

Applies a conversion to both arguments of a binary operator.

## Description

If `c` is a conversion where `c ‘l‘` returns `|- l = l’` and `c ‘r‘` returns `|- r = r’`, then `BINOP_CONV ‘op l r‘` returns `|- op l r = op l’ r’`. The term `op` is arbitrary, but is often a constant such as addition or conjunction.

## Failure

Never fails when applied to the conversion. But may fail when applied to the term if one of the core conversions fails or returns an inappropriate theorem on the subterms.

## Example

```
# BINOP_CONV NUM_ADD_CONV ‘(1 + 1) * (2 + 2)‘;;
val it : thm = |- (1 + 1) * (2 + 2) = 2 * 4
```

## See also

`ABS_CONV`, `COMB_CONV`, `COMB2_CONV`, `RAND_CONV`, `RATOR_CONV`.

# binops

```
binops : term -> term -> term list
```

## Synopsis

Repeatedly breaks apart an iterated binary operator into components.

## Description

The call `binops op t` repeatedly breaks down applications of the binary operator `op` within `t`. If `t` is of the form `(op l) r` (thinking of `op` as infix, `l op r`), then it recursively breaks down `l` and `r` in the same way and appends the results. Otherwise, a singleton list of the original term is returned.

## Failure

Never fails.

## Example

```
# binops `(+):num->num->num` `((1 + 2) + 3) + 4 + 5 + 6`;;
val it : term list = [`1`; `2`; `3`; `4`; `5`; `6`]

# binops `(+):num->num->num` `F`;;
val it : term list = [`F`]
```

## See also

dest_binop, mk_binop, striplist.

# BINOP_TAC

```
BINOP_TAC : tactic
```

## Synopsis

Breaks apart equation between binary operator applications into equality between their arguments.

## Description

Given a goal whose conclusion is an equation between applications of the same curried binary function `f`, the tactic `BINOP_TAC` breaks it down to two subgoals expressing equality of the corresponding arguments:

```
       A ?- f x1 y1 = f x2 y2
   ================================  BINOP_TAC
      A ?- x1 = x2    A ?- y1 = y2
```

## Failure

Fails if the conclusion of the goal is not an equation between applications of the same curried binary operator.

## Example

We can set up the following goal which is an equation between applications of the binary operator +:

```
# g `f(2 * x + 1) + w * z = f(SUC(x + 1) * 2 - 1) + z * w`;;
```

and it is simplest to prove if we split it up into two subgoals:

```
# e BINOP_TAC;;
val it : goalstack = 2 subgoals (2 total)

`w * z = z * w`

`f (2 * x + 1) = f (SUC (x + 1) * 2 - 1)`
```

the first of which can be solved by `ARITH_TAC`, and the second by `AP_TERM_TAC THEN ARITH_TAC`.

## See also
ABS_TAC, AP_TERM_TAC, AP_THM_TAC, MK_BINOP, MK_COMB_TAC.

# bndvar

```
bndvar : term -> term
```

## Synopsis
Returns the bound variable of an abstraction.

## Description
`bndvar `\var. t`` returns ``var``.

## Failure
Fails unless the term is an abstraction.

## Example

```
# bndvar `\x. x + 1`;;
val it : term = `x`
```

## See also
body, dest_abs.

# body

```
body : term -> term
```

## Synopsis
Returns the body of an abstraction.

## Description
`body ‘\var. t‘` returns `‘t‘`.

## Failure
Fails unless the term is an abstraction.

## Example

```
# body ‘\x. x + 1‘;;
val it : term = ‘x + 1‘
```

## See also
`bndvar`, `dest_abs`.

# BOOL_CASES_TAC

```
BOOL_CASES_TAC : term -> tactic
```

## Synopsis
Performs boolean case analysis on a (free) term in the goal.

## Description
When applied to a term `x` (which must be of type `bool` but need not be simply a variable), and a goal `A ?- t`, the tactic `BOOL_CASES_TAC` generates the two subgoals corresponding to `A ?- t` but with any free instances of `x` replaced by `F` and `T` respectively.

```
              A ?- t
   ============================  BOOL_CASES_TAC ‘x‘
    A ?- t[F/x]    A ?- t[T/x]
```

The term given does not have to be free in the goal, but if it isn't, `BOOL_CASES_TAC` will merely duplicate the original goal twice. Note that in the new goals, we don't have `x` and `~x` as assumptions; for that use `ASM_CASES_TAC`.

## Failure

Fails unless the term `x` has type `bool`.

## Example

The goal:

```
# g `(b ==> ~b) ==> (b ==> a)`;;
```

can be completely solved by using `BOOL_CASES_TAC` on the variable `b`, then simply rewriting the two subgoals using only the inbuilt tautologies, i.e. by applying the following tactic:

```
# e(BOOL_CASES_TAC `b:bool` THEN REWRITE_TAC[]);;
val it : goalstack = No subgoals
```

## Uses

Avoiding fiddly logical proofs by brute-force case analysis, possibly only over a key term as in the above example, possibly over all free boolean variables.

## See also

`ASM_CASES_TAC`, `COND_CASES_TAC`, `DISJ_CASES_TAC`, `ITAUT`, `STRUCT_CASES_TAC`, `TAUT`.

# bool_ty

```
bool_ty : hol_type
```

## Synopsis

The type `:bool`.

## Description

This name is bound to the HOL type `:bool`.

## Failure

Not applicable.

## Uses

Exploiting the very common type `:bool` inside derived rules without the inefficiency or inconvenience of calling a quotation parser or explicit constructor.

## See also

`aty`, `bty`.

# bty

`bty : hol_type`

## Synopsis
The type variable `‘:B‘`.

## Description
This name is bound to the HOL type `:B`.

## Failure
Not applicable.

## Uses
Exploiting the very common type variable `:B` inside derived rules (e.g. an instantiation list for `inst` or `type_subst`) without the inefficiency or inconvenience of calling a quotation parser or explicit constructor.

## See also
`aty, bool_ty.`

# butlast

`butlast : 'a list -> 'a list`

## Synopsis
Computes the sub-list of a list consisting of all but the last element.

## Description
`butlast [x1;...;xn]` returns `[x1;...;x(n-1)]`.

## Failure
Fails if the list is empty.

## See also
`last, hd, tl, el.`

# by

```
by : tactic -> refinement
```

## Synopsis
Converts a tactic to a refinement.

## Description
The call `by tac` for a tactic `tac` gives a refinement of the current list of subgoals that applies `tac` to the first subgoal.

## Comments
Only of interest to users who want to handle ‘refinements’ explicitly.

# CACHE_CONV

```
CACHE_CONV : (term -> thm) -> term -> thm
```

## Synopsis
Accelerates a conversion by cacheing previous results.

## Description
If `cnv` is any conversion, then `CACHE_CONV cnv` gives a new conversion that is functionally identical but keeps a cache of previous arguments and results, and simply returns the cached result if the same input is encountered again.

## Failure
Never fails, though the subsequent application to a term may.

## Example
The following call takes a while, making several applications to the same expression:

```
# time (DEPTH_CONV NUM_RED_CONV) `31 EXP 31 + 31 EXP 31 + 31 EXP 31`;;
CPU time (user): 1.542
val it : thm =
  |- 31 EXP 31 + 31 EXP 31 + 31 EXP 31 =
     51207522392169707875831929087177944268134203293
```

whereas the cached variant is faster since the result for `31 EXP 31` is stored away and re-used after the first call:

```
# time (DEPTH_CONV(CACHE_CONV NUM_RED_CONV))
       `31 EXP 31 + 31 EXP 31 + 31 EXP 31`;;
CPU time (user): 0.461
val it : thm =
  |- 31 EXP 31 + 31 EXP 31 + 31 EXP 31 =
     51207522392169707875831929087177944268134203293
```

## See also

# can

`can : ('a -> 'b) -> 'a -> bool`

## Synopsis
Tests for failure.

## Description
`can f x` evaluates to `true` if the application of `f` to `x` succeeds. It evaluates to `false` if the application with a `Failure _` exception.

## Failure
Never fails.

## Example

```
# can hd [1;2];;
val it : bool = true
# can hd [];;
val it : bool = false
```

## See also
`check`.

# cases

`cases : string -> thm`

## Synopsis
Produce cases theorem for an inductive type.

## Description
A call `cases "ty"` where `"ty"` is the name of a recursive type defined with `define_type`, returns a "cases" theorem asserting that each element of the type is an instance of one of the type constructors. The effect is exactly the same is if `prove_cases_thm` were applied to the induction theorem produced by `define_type`, and the documentation for `prove_cases_thm` gives a lengthier discussion.

## Failure
Fails if `ty` is not the name of a recursive type.

## Example

```
# cases "num";;
val it : thm = |- !m. m = 0 \/ (?n. m = SUC n)

# cases "list";;
val it : thm = |- !x. x = [] \/ (?a0 a1. x = CONS a0 a1)
```

## See also
`define_type`, `distinctness`, `injectivity`, `prove_cases_thm`.

# CCONTR

`CCONTR : term -> thm -> thm`

## Synopsis
Implements the classical contradiction rule.

## Description
When applied to a term `t` and a theorem `A |- F`, the inference rule `CCONTR` returns the theorem `A - {~t} |- t`.

```
      A |- F
   ---------------  CCONTR `t`
    A - {~t} |- t
```

## Failure
Fails unless the term has type `bool` and the theorem has `F` as its conclusion.

## Comments

The usual use will be when `~t` exists in the assumption list; in this case, `CCONTR` corresponds to the classical contradiction rule: if `~t` leads to a contradiction, then `t` must be true.

## See also

`CONTR`, `CONTR_TAC`, `NOT_ELIM`.

# C

```
C : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

## Synopsis

Permutes first two arguments to curried function: `C f x y` = `f y x`.

## Failure

Never fails.

## See also

`F_F`, `I`, `K`, `W`.

# CHANGED_CONV

```
CHANGED_CONV : conv -> conv
```

## Synopsis

Makes a conversion fail if applying it leaves a term unchanged.

## Description

For a conversion `cnv`, the construct `CHANGED_CONV c` gives a new conversion that has the same action as `cnv`, except that it will fail on terms `t` such that `cnv t` returns a reflexive theorem `|- t = t`, or more precisely `|- t = t'` where `t` and `t'` are alpha-equivalent.

## Failure

Never fails when applied to the conversion, but fails on further application to a term if the original conversion does or it returns a reflexive theorem.

## Example

```
# ONCE_DEPTH_CONV num_CONV `x + 0`;;
val it : thm = |- x + 0 = x + 0

# CHANGED_CONV(ONCE_DEPTH_CONV num_CONV) `x + 0`;;
Exception: Failure "CHANGED_CONV".

# CHANGED_CONV(ONCE_DEPTH_CONV num_CONV) `6`;;
val it : thm = |- 6 = SUC 5

# REPEATC(CHANGED_CONV(ONCE_DEPTH_CONV num_CONV)) `6`;;
val it : thm = |- 6 = SUC (SUC (SUC (SUC (SUC (SUC 0)))))
```

## Uses

`CHANGED_CONV` is used to transform a conversion that may leave terms unchanged, and therefore may cause a nonterminating computation if repeated, into one that can safely be repeated until application of it fails to substantially modify its input term, as in the last example above.

# CHANGED_TAC

```
CHANGED_TAC : tactic -> tactic
```

## Synopsis

Makes a tactic fail if it has no effect.

## Description

When applied to a tactic `t`, the tactical `CHANGED_TAC` gives a new tactic which is the same as `t` if that has any effect, and otherwise fails.

## Failure

The application of `CHANGED_TAC` to a tactic never fails. The resulting tactic fails if the basic tactic either fails or has no effect.

## Uses

Occasionally useful in controlling complicated tctic compositions. Also sometimes convenient just to check that a step did indeed modify a goal.

## See also

`TRY`, `VALID`.

# CHEAT_TAC

`CHEAT_TAC : tactic`

## Synopsis
Proves goal by asserting it as an axiom.

## Description
Given any goal `A ?- p`, the tactic `CHEAT_TAC` solves it by using `mk_thm`, which in turn involves essentially asserting the goal as a new axiom.

## Failure
Never fails.

## Uses
Temporarily plugging boring parts of a proof to deal with the interesting parts.

## Comments
Needless to say, this should be used with caution since once new axioms are asserted there is no guarantee that logical consistency is preserved.

## See also
`new_axiom`, `mk_thm`.

# check

`check : ('a -> bool) -> 'a -> 'a`

## Synopsis
Checks that a value satisfies a predicate.

## Description
`check p x` returns `x` if the application `p x` yields `true`. Otherwise, `check p x` fails.

## Failure
`check p x` fails with `Failure "check"` if the predicate `p` yields `false` when applied to the value `x`.

## Example

```
# check is_var ‘x:bool‘;;
val it : term = ‘x‘
# check is_var ‘x + 2‘;;
Exception: Failure "check".
```

## Uses

Can be used to filter out candidates from a set of terms, e.g. to apply theorem-tactics to assumptions with a certain pattern.

## See also

`can`.

# choose

`choose : ('a, 'b) func -> 'a * 'b`

## Synopsis

Picks an arbitrary element from the graph of a finite partial function.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` is a finite partial function, `choose f` picks an arbitrary pair of values from its graph, i.e. a pair `x,y` where `f` maps `x` to `y`. The particular choice is implementation-defined, and it is not likely to be the most obvious ‘first’ value.

## Failure

Fails if and only if the finite partial function is completely undefined.

## Example

```
# let f = itlist I [1 |-> 2; 2 |-> 3; 3 |-> 4] undefined;;
val f : (int, int) func = <func>
# choose f;;
val it : int * int = (2, 3)
```

## See also

`|->`, `|=>`, `apply`, `applyd`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# CHOOSE_TAC

`CHOOSE_TAC : thm_tactic`

## Synopsis

Adds the body of an existentially quantified theorem to the assumptions of a goal.

## Description

When applied to a theorem `A' |- ?x. t` and a goal, `CHOOSE_TAC` adds `t[x'/x]` to the assumptions of the goal, where `x'` is a variant of `x` which is not free in the assumption list; normally `x'` is just `x`.

```
        A ?- u
====================  CHOOSE_TAC (A' |- ?x. t)
 A u {t[x'/x]} ?- u
```

Unless `A'` is a subset of `A`, this is not a valid tactic.

## Failure

Fails unless the given theorem is existentially quantified.

## Example

Suppose we have a goal asserting that the output of an electrical circuit (represented as a boolean-valued function) will become high at some time:

```
?- ?t. output(t)
```

and we have the following theorems available:

```
t1 = |- ?t. input(t)
t2 = !t. input(t) ==> output(t+1)
```

Then the goal can be solved by the application of:

```
CHOOSE_TAC t1 THEN EXISTS_TAC `t+1` THEN
UNDISCH_TAC `input (t:num) :bool` THEN MATCH_ACCEPT_TAC t2
```

## See also

CHOOSE_THEN, X_CHOOSE_TAC.

# CHOOSE_THEN

```
CHOOSE_THEN : thm_tactical
```

## Synopsis

Applies a tactic generated from the body of existentially quantified theorem.

## Description

When applied to a theorem-tactic `ttac`, an existentially quantified theorem `A' |- ?x. t`, and a goal, `CHOOSE_THEN` applies the tactic `ttac (t[x'/x] |- t[x'/x])` to the goal, where `x'` is a variant of `x` chosen not to be free in the assumption list of the goal. Thus if:

```
    A ?- s1
   =========  ttac (t[x'/x] |- t[x'/x])
    B ?- s2
```

then

```
    A ?- s1
   ==========  CHOOSE_THEN ttac (A' |- ?x. t)
    B ?- s2
```

This is invalid unless `A'` is a subset of `A`.

## Failure

Fails unless the given theorem is existentially quantified, or if the resulting tactic fails when applied to the goal.

## Example

This theorem-tactical and its relatives are very useful for using existentially quantified theorems. For example one might use the inbuilt theorem

```
LT_EXISTS = |- !m n. m < n <=> (?d. n = m + SUC d)
```

to help solve the goal

```
# g `x < y ==> 0 < y * y`;;
```

by starting with the following tactic

```
# e(DISCH_THEN(CHOOSE_THEN SUBST1_TAC o REWRITE_RULE[LT_EXISTS]));;
```

reducing the goal to

```
val it : goalstack = 1 subgoal (1 total)

`0 < (x + SUC d) * (x + SUC d)`
```

which can then be finished off quite easily, by, for example just `ARITH_TAC`, or

```
# e(REWRITE_TAC[ADD_CLAUSES; MULT_CLAUSES; LT_0]);;
```

## See also

`CHOOSE_TAC`, `X_CHOOSE_THEN`.

# CHOOSE

```
CHOOSE : term * thm -> thm -> thm
```

## Synopsis

Eliminates existential quantification using deduction from a particular witness.

## Description

When applied to a term-theorem pair `(v,A1 |- ?x. s)` and a second theorem of the form `A2 u {s[v/x]} |- t`, the inference rule `CHOOSE` produces the theorem `A1 u A2 |- t`.

```
    A1 |- ?x. s[x]    A2 u {s[v/x]} |- t
   ---------------------------------------  CHOOSE (`v`,(A1 |- ?x. s))
                A1 u A2 |- t
```

Where `v` is not free in `A2` or `t`.

### Failure
Fails unless the terms and theorems correspond as indicated above; in particular, 1) `v` must be a variable and have the same type as the variable existentially quantified over, and it must not be free in `A2` or `t`; 2) the second theorem must have `s[v/x]` in its assumptions.

### Comments
For the special case of simply existentially quantifying an assumption over a variable, `SIMPLE_CHOOSE` is easier.

### See also
`CHOOSE_TAC`, `EXISTS`, `EXISTS_TAC`, `SIMPLE_CHOOSE`.

## chop_list

```
chop_list : int -> 'a list -> 'a list * 'a list
```

### Synopsis
Chops a list into two parts at a specified point.

### Description
`chop_list i [x1;...;xn]` returns `([x1;...;xi],[x(i+1);...;xn])`.

### Failure
Fails with `chop_list` if `i` is negative or greater than the length of the list.

### Example

```
# chop_list 3 [1;2;3;4;5];;
val it : int list * int list = ([1; 2; 3], [4; 5])
```

### See also
`partition`.

## CNF_CONV

```
CNF_CONV : conv
```

## Synopsis
Converts a term already in negation normal form into conjunctive normal form.

## Description
When applied to a term already in negation normal form (see `NNF_CONV`), meaning that all other propositional connectives have been eliminated in favour of conjunction, disjunction and negation, and negation is only applied to atomic formulas, `CNF_CONV` puts the term into an equivalent conjunctive normal form, which is a right-associated conjunction of disjunctions without repetitions. No reduction by subsumption is performed, however, e.g. from `a /\ (a \/ b)` to just `a`).

## Failure
Never fails; non-Boolean terms will just yield a reflexive theorem.

## Example

```
# CNF_CONV `(a /\ b) \/ (a /\ b /\ c) \/ d`;;
val it : thm =
  |- a /\ b \/ a /\ b /\ c \/ d <=>
     (a \/ d) /\ (a \/ b \/ d) /\ (a \/ c \/ d) /\ (b \/ d) /\ (b \/ c \/ d)
```

## See also
`DNF_CONV, NNF_CONV, WEAK_CNF_CONV, WEAK_DNF_CONV.`

# COMB2_CONV

```
COMB2_CONV : (term -> thm) -> (term -> thm) -> term -> thm
```

## Synopsis
Applies two conversions to the two sides of an application.

## Description
If `c1` and `c2` are conversions such that `` c1 `f` `` returns `|- f = f'` and `` c2 `x` `` returns `|- x = x'`, then `` COMB_CONV c1 c2 `f x` `` returns `|- f x = f' x'`. That is, the conversions `c1` and `c2` are applied respectively to the two immediate subterms.

## Failure
Never fails when applied to the initial conversions. On application to the term, it fails if either `c1` or `c2` does, or if either returns a theorem that is of the wrong form.

### See also
`BINOP_CONV`, `COMB_CONV`, `LAND_CONV`, `RAND_CONV`, `RATOR_CONV`

## COMB_CONV

```
COMB_CONV : conv -> conv
```

### Synopsis
Applies a conversion to the two sides of an application.

### Description
If `c` is a conversion such that `` c `f` `` returns `|- f = f'` and `` c `x` `` returns `|- x = x'`, then `` COMB_CONV c `f x` `` returns `|- f x = f' x'`. That is, the conversion `c` is applied to the two immediate subterms.

### Failure
Never fails when applied to the initial conversion. On application to the term, it fails if conversion given as the argument does, or if the theorem returned by it is inappropriate.

### See also
`BINOP_CONV`, `COMB2_CONV`, `LAND_CONV`, `RAND_CONV`, `RATOR_CONV`

## combine

```
combine : ('a -> 'a -> 'a) -> ('a -> bool) -> ('b, 'a) func -> ('b, 'a) func -> ('b, 'a) func
```

### Synopsis
Combine together two finite partial functions using symmetric pointwise operation.

### Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` and `g` are finite partial functions, then `combine op z f g` will combine them together in the following somewhat complicated way. If just one of the functions `f` and `g` is defined at point `x`, that will give the value of the combined function. If both `f` and `g` are defined at `x` with values `y1` and `y2`, the value of

the combined function will be `op y1 y2`, or possibly `y = op y2 y1` (hence, the operation `op` should be symmetric). However, if the resulting value `y` satisfies the predicate `z`, the new function will be undefined at that point; the intuition is that the two values `y1` and `y2` cancel each other out.

## Failure

Can only fail if the given operation fails.

## Example

```
# let f = itlist I [1 |-> 2; 2 |-> 3; 3 |-> 6] undefined
  and g = itlist I [1 |-> 5; 2 |-> -3] undefined;;
val f : (int, int) func = <func>
val g : (int, int) func = <func>

# graph(combine (+) (fun x -> x = 0) f g);;
val it : (int * int) list = [(1, 7); (3, 6)]
```

## Uses

When finite partial functions are used to represent values with a numeric domain (e.g. matrices or polynomials), this can be used to perform addition pointwise by using addition for the `op` argument. Using a zero test as the predicate `z` will ensure that no zero values are included in the result, giving a canonical representation.

## See also

`|->`, `|=>`, `apply`, `applyd`, `choose`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# comment_token

```
comment_token : lexcode ref
```

## Synopsis

HOL Light comment token.

## Description

Users may insert comments in HOL Light terms that are ignored in parsing. Comments are introduced by a special comment token and terminated by the next end of line. (There are no multi-line comments supported in HOL Light terms.) The reference `comment_token` stores the token that introduces a comment, which by default is `Resword "//"` as in BCPL, C++, Java etc. The user may change it to another token, though this should be done with care in case other proofs break.

### Failure
Not applicable.

### Example
Here we change the comment token to be '`--`' (as used in Ada, Eiffel, Haskell, Occam and several other programming languages):

```
# comment_token := Ident "--";;
val it : unit = ()
```

and we can test that it works:

```
# `let wordsize = 32 -- may change to 64 later
   and radix = 2     -- only care about binary
   in radix EXP wordsize`;;
val it : term = `let wordsize = 32 and radix = 2 in radix EXP wordsize`
```

### Comments
Comments are handled at the level of the lexical analyzer, so can also be used in types and the strings used for the specification of inductive types.

### See also
`define_type`, `lex`, `parse_inductive_type_specification`, `parse_term`, `parse_type`.

## compose_insts

`compose_insts : instantiation -> instantiation -> instantiation`

### Synopsis
Compose two instantiations.

### Description
Given two instantiations `i1` and `i2` (with type `instantiation`, as returned by `term_match` for example), the call `compose_insts i1 i2` will give a new instantiation that results from composing them, with `i1` applied first and then `i2`. For example, `instantiate (compose_insts i1 i2) t` should be the same as `instantiate i2 (instantiate i1 t)`.

## Failure
Never fails.

## Comments
Mostly of specialized interest; used in sequencing tactics like `THEN` to compose metavariable instantiations.

## See also
`instantiate`, `INSTANTIATE`, `INSTANTIATE_ALL`, `inst_goal`, `PART_MATCH`, `term_match`.

# concl

`concl : thm -> term`

## Synopsis
Returns the conclusion of a theorem.

## Description
When applied to a theorem `A |- t`, the function `concl` returns `t`.

## Failure
Never fails.

## Example

```
# ADD_SYM;;
val it : thm = |- !m n. m + n = n + m
# concl ADD_SYM;;
val it : term = `!m n. m + n = n + m`

# concl (ASSUME `1 = 0`);;
val it : term = `1 = 0`
```

## See also
`dest_thm`, `hyp`.

# COND_CASES_TAC

`COND_CASES_TAC : tactic`

## Synopsis
Induces a case split on a conditional expression in the goal.

## Description
`COND_CASES_TAC` searches for a free conditional subterm in the term of a goal, i.e. a subterm of the form `if p then u else v`, choosing some topmost one if there are several. It then induces a case split over `p` as follows:

```
                       A ?- t
    ==========================================  COND_CASES_TAC
     A u {p} ?- t[T/p; u/(if p then u else v)]
     A u {~p} ?- t[F/p; v/(if p then u else v)]
```

where `p` is not a constant, and the term `p then u else v` is free in `t`. Note that it both enriches the assumptions and inserts the assumed value into the conditional.

## Failure
`COND_CASES_TAC` fails if there is no conditional sub-term as described above.

## Example
We can prove the following just by `REAL_ARITH `!x y:real. x <= max x y``, but it is instructive to consider a manual proof.

```
  # g `!x y:real. x <= max x y`;;
  val it : goalstack = 1 subgoal (1 total)

  `!x y. x <= max x y`

  # e(REPEAT GEN_TAC THEN REWRITE_TAC[real_max]);;'
  val it : goalstack = 1 subgoal (1 total)

  `x <= (if x <= y then y else x)`

  # e COND_CASES_TAC;;
  val it : goalstack = 1 subgoal (1 total)

   0 [`~(x <= y)`]

  `x <= x`
```

## Uses
Useful for case analysis and replacement in one step, when there is a conditional sub-term in the term part of the goal. When there is more than one such sub-term and one in particular is to be analyzed, `COND_CASES_TAC` cannot always be depended on to choose the

'desired' one. It can, however, be used repeatedly to analyze all conditional sub-terms of a goal.

## Comments

Note that logically it should only be necessary for `p` to be free in the whole term, not the two branches `x` and `y`. However, as an artifact of the current implementation, we need them to be free too. The more sophisticated conversion `CONDS_ELIM_CONV` handles this better.

## See also

`ASM_CASES_TAC`, `COND_ELIM_CONV`, `CONDS_ELIM_CONV`, `DISJ_CASES_TAC`, `STRUCT_CASES_TAC`.

# COND_ELIM_CONV

```
COND_ELIM_CONV : term -> thm
```

## Synopsis

Conversion to eliminate one free conditional subterm.

## Description

When applied to a term `` `....(if p then x else y)...` `` containing a free conditional subterm, `COND_ELIM_CONV` returns a theorem asserting its equivalence to a term with the conditional eliminated:

```
|- ....(if p then x else y).... <=>
   (p ==> ....x....) /\ (~p ==> ....y....)
```

If the term contains many free conditional subterms, a topmost one will be used.

## Failure

Fails if there are no free conditional subterms.

## Example

We can prove the little equivalence noted by Dijkstra in EWD1176 automatically:

```
# REAL_ARITH `!a b:real. a + b >= max a b <=> a >= &0 /\ b >= &0`;;
val it : thm = |- !a b. a + b >= max a b <=> a >= &0 /\ b >= &0
```

However, if our automated tools were unfamiliar with `max`, we might expand its definition (theorem `real_max`) and then eliminate the resulting conditional by `COND_ELIM_CONV`:

```
# COND_ELIM_CONV `a + b >= (if a <= b then b else a) <=> a >= &0 /\ b >= &0`;;
val it : thm =
  |- (a + b >= (if a <= b then b else a) <=> a >= &0 /\ b >= &0) <=>
     (a <= b ==> (a + b >= b <=> a >= &0 /\ b >= &0)) /\
     (~(a <= b) ==> (a + b >= a <=> a >= &0 /\ b >= &0))
```

## Uses

Eliminating conditionals as a prelude to other automated proof steps that are not equipped to handle them.

## Comments

Note that logically it should only be necessary for `p` to be free in the whole term, not the two branches `x` and `y`. However, as an artifact of the current implementation, we need them to be free too. The more sophisticated `CONDS_ELIM_CONV` handles this better.

## See also

`COND_CASES_TAC`, `CONDS_ELIM_CONV`.

# CONDS_CELIM_CONV

```
CONDS_CELIM_CONV : conv
```

## Synopsis

Remove all conditional expressions from a Boolean formula.

## Description

When applied to a Boolean term, `CONDS_CELIM_CONV` identifies subterms that are conditional expressions of the form '`if p then x else y`', and eliminates them. First they are "pulled out" as far as possible, e.g. from '`f (if p then x else y)`' to '`if p then f(x) else f(y)`' and so on. When a quantifier that binds one of the variables in the expression is reached, the subterm is of Boolean type, say '`if p then q else r`', and it is replaced by a propositional equivalent of the form '`(~p \/ q) /\ (p \/ r)`'.

## Failure

Never fails, but will just return a reflexive theorem if the term is not Boolean.

## Example

```
# CONDS_CELIM_CONV `y <= z ==> !x. (if x <= y then y else x) <= z`;;
val it : thm =
  |- y <= z ==> (!x. (if x <= y then y else x) <= z) <=>
     y <= z ==> (!x. (~(x <= y) \/ y <= z) /\ (x <= y \/ x <= z))
```

## Uses

Mostly for initial normalization in automated rules, but may be helpful for other uses.

## Comments

The function `CONDS_ELIM_CONV` is functionally similar, but will do the final propositional splitting in a "disjunctive" rather than "conjunctive" way. The disjunctive way is usually better when the term will subsequently be passed to a refutation procedure, whereas the conjunctive form is better for non-refutation procedures. In each case, the policy is changed in an appropriate way after passing through quantifiers.

## See also

`COND_CASES_TAC`, `COND_ELIM_CONV`, `CONDS_ELIM_CONV`.

# CONDS_ELIM_CONV

```
CONDS_ELIM_CONV : conv
```

## Synopsis

Remove all conditional expressions from a Boolean formula.

## Description

When applied to a Boolean term, `CONDS_ELIM_CONV` identifies subterms that are conditional expressions of the form '`if p then x else y`', and eliminates them. First they are "pulled out" as far as possible, e.g. from '`f (if p then x else y)`' to '`if p then f(x) else f(y)`' and so on. When a quantifier that binds one of the variables in the expression is reached, the subterm is of Boolean type, say '`if p then q else r`', and it is replaced by a propositional equivalent of the form '`p /\ q \/ ~p /\ r`'.

## Failure

Never fails, but will just return a reflexive theorem if the term is not Boolean.

## Example

Note that in contrast to COND_ELIM_CONV, there are no freeness restrictions, and the Boolean split will be done inside quantifiers if necessary:

```
# CONDS_ELIM_CONV `!x y. (if x <= y then y else x) <= z ==> x <= z`;;
val it : thm =
  |- (!x y. (if x <= y then y else x) <= z ==> x <= z) <=>
     (!x y. ~(x <= y) \/ (y <= z ==> x <= z))
```

## Uses

Mostly for initial normalization in automated rules, but may be helpful for other uses.

## Comments

The function CONDS_CELIM_CONV is functionally similar, but will do the final propositional splitting in a "conjunctive" rather than "disjunctive" way. The disjunctive way is usually better when the term will subsequently be passed to a refutation procedure, whereas the conjunctive form is better for non-refutation procedures. In each case, the policy is changed in an appropriate way after passing through quantifiers.

## See also

COND_CASES_TAC, COND_ELIM_CONV, CONDS_CELIM_CCONV.

# CONJ_ACI_RULE

```
CONJ_ACI_RULE : term -> thm
```

## Synopsis

Proves equivalence of two conjunctions containing same set of conjuncts.

## Description

The call CONJ_ACI_RULE `t1 /\ ... /\ tn <=> u1 /\ ... /\ um`, where both sides of the equation are conjunctions of exactly the same set of conjuncts, (with arbitrary ordering, association, and repetitions), will return the corresponding theorem |- t1 /\ ... /\ tn <=> u1 /\ ...

## Failure

Fails if applied to a term that is not a Boolean equation or the two sets of conjuncts are different.

## Example

```
# CONJ_ACI_RULE `(a /\ b) /\ (a /\ c) <=> (a /\ (c /\ a)) /\ b`;;
val it : thm = |- (a /\ b) /\ a /\ c <=> (a /\ c /\ a) /\ b
```

## Comments

The same effect can be had with the more general `AC` construct. However, for the special case of conjunction, `CONJ_ACI_RULE` is substantially more efficient when there are many conjuncts involved.

## See also

`AC`, `CONJ_CANON_CONV`, `DISJ_ACI_RULE`.

# CONJ_CANON_CONV

```
CONJ_CANON_CONV : term -> thm
```

## Synopsis

Puts an iterated conjunction in canonical form.

## Description

When applied to a term, `CONJ_CANON_CONV` splits it into the set of conjuncts and produces a theorem asserting the equivalence of the term and the new term with the disjuncts right-associated without repetitions and in a canonical order.

## Failure

Fails if applied to a non-Boolean term. If applied to a term that is not a conjunction, it will trivially work in the sense of regarding it as a single conjunct and returning a reflexive theorem.

## Example

```
# CONJ_CANON_CONV `(a /\ b) /\ ((b /\ d) /\ a) /\ c`;;
val it : thm = |- (a /\ b) /\ ((b /\ d) /\ a) /\ c <=> a /\ b /\ c /\ d
```

## See also

`AC`, `CONJ_ACI_CONV`, `DISJ_CANON_CONV`.

# CONJ

```
CONJ : thm -> thm -> thm
```

## Synopsis
Introduces a conjunction.

## Description

```
    A1 |- t1      A2 |- t2
   ------------------------  CONJ
     A1 u A2 |- t1 /\ t2
```

## Failure
Never fails.

## Example

```
  # CONJ (NUM_REDUCE_CONV `2 + 2`) (ASSUME `p:bool`);;
  val it : thm = p |- 2 + 2 = 4 /\ p
```

## See also
CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJ_PAIR.

# CONJ_PAIR

```
CONJ_PAIR : thm -> thm * thm
```

## Synopsis
Extracts both conjuncts of a conjunction.

## Description

```
       A |- t1 /\ t2
   ----------------------  CONJ_PAIR
    A |- t1      A |- t2
```

The two resultant theorems are returned as a pair.

## Failure

Fails if the input theorem is not a conjunction.

## Example

```
# CONJ_PAIR(ASSUME `p /\ q`);;
val it : thm * thm = (p /\ q |- p, p /\ q |- q)
```

## See also

CONJUNCT1, CONJUNCT2, CONJ, CONJUNCTS.

# CONJ_TAC

```
CONJ_TAC : tactic
```

## Synopsis

Reduces a conjunctive goal to two separate subgoals.

## Description

When applied to a goal `A ?- t1 /\ t2`, the tactic `CONJ_TAC` reduces it to the two subgoals corresponding to each conjunct separately.

```
        A ?- t1 /\ t2
   ======================  CONJ_TAC
    A ?- t1      A ?- t2
```

## Failure

Fails unless the conclusion of the goal is a conjunction.

## See also

STRIP_TAC.

# CONJUNCT1

```
CONJUNCT1 : thm -> thm
```

### Synopsis
Extracts left conjunct of theorem.

### Description
```
   A |- t1 /\ t2
  ---------------  CONJUNCT1
      A |- t1
```

### Failure
Fails unless the input theorem is a conjunction.

### Example
```
# CONJUNCT1(ASSUME `p /\ q`);;
val it : thm = p /\ q |- p
```

### See also
CONJ_PAIR, CONJUNCT2, CONJ, CONJUNCTS.

## CONJUNCT2

```
CONJUNCT2 : thm -> thm
```

### Synopsis
Extracts right conjunct of theorem.

### Description
```
   A |- t1 /\ t2
  ---------------  CONJUNCT2
      A |- t2
```

### Failure
Fails unless the input theorem is a conjunction.

### Example
```
# CONJUNCT2(ASSUME `p /\ q`);;
val it : thm = p /\ q |- q
```

### See also
CONJ_PAIR, CONJUNCT1, CONJ, CONJUNCTS.

# conjuncts

`conjuncts : term -> term list`

## Synopsis
Iteratively breaks apart a conjunction.

## Description
If a term `t` is a conjunction `p /\ q`, then `conjuncts t` will recursively break down `p` and `q` into conjuncts and append the resulting lists. Otherwise it will return the singleton list `[t]`. So if `t` is of the form `t1 /\ ... /\ tn` with any reassociation, no `ti` itself being a conjunction, the list returned will be `[t1; ...; tn]`. But

```
conjuncts(list_mk_conj([t1;...;tn]))
```

will not return `[t1;...;tn]` if any of `t1`,...,`tn` is a conjunction.

## Failure
Never fails, even if the term is not boolean.

## Example

```
# conjuncts `((p /\ q) /\ r) /\ ((p /\ s /\ t) /\ u)`;;
val it : term list = [`p`; `q`; `r`; `p`; `s`; `t`; `u`]

# conjuncts(list_mk_conj [`a /\ b`; `c:bool`; `d /\ e /\ f`]);;
val it : term list = [`a`; `b`; `c`; `d`; `e`; `f`]
```

## Comments
Because `conjuncts` splits both the left and right sides of a conjunction, this operation is not the inverse of `list_mk_conj`. You can also use `splitlist dest_conj` to split in a right-associated way only.

## See also
`dest_conj, disjuncts, is_conj.`

# CONJUNCTS_THEN2

`CONJUNCTS_THEN2 : thm_tactic -> thm_tactic -> thm_tactic`

## Synopsis

Applies two theorem-tactics to the corresponding conjuncts of a theorem.

## Description

`CONJUNCTS_THEN2` takes two theorem-tactics, `f1` and `f2`, and a theorem `t` whose conclusion must be a conjunction. `CONJUNCTS_THEN2` breaks `t` into two new theorems, `t1` and `t2` which are `CONJUNCT1` and `CONJUNCT2` of `t` respectively, and then returns the tactic `f1 t1 THEN f2 t2`. Thus

```
   CONJUNCTS_THEN2 f1 f2 (A |- l /\ r) =  f1 (A |- l) THEN f2 (A |- r)
```

so if

```
   A1 ?- t1                    A2 ?- t2
  ==========  f1 (A |- l)     ==========  f2 (A |- r)
   A2 ?- t2                    A3 ?- t3
```

then

```
    A1 ?- t1
   ==========  CONJUNCTS_THEN2 f1 f2 (A |- l /\ r)
    A3 ?- t3
```

## Failure

`CONJUNCTS_THEN f` will fail if applied to a theorem whose conclusion is not a conjunction.

## Uses

The construction of complex `tacticals` like `CONJUNCTS_THEN`.

## See also

`CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJUNCTS_TAC, CONJUNCTS_THEN2, STRIP_THM_THEN.`

# CONJUNCTS_THEN

```
CONJUNCTS_THEN : thm_tactical
```

## Synopsis

Applies a theorem-tactic to each conjunct of a theorem.

## Description

CONJUNCTS_THEN takes a theorem-tactic ttac, and a theorem t whose conclusion must be a conjunction. CONJUNCTS_THEN breaks t into two new theorems, t1 and t2 which are CONJUNCT1 and CONJUNCT2 of t respectively, and then returns a new tactic: ttac t1 THEN ttac t2. That is,

```
   CONJUNCTS_THEN ttac (A |- l /\ r) =  ttac (A |- l) THEN ttac (A |- r)
```

so if

```
   A1 ?- t1                        A2 ?- t2
  ==========  ttac (A |- l)       ==========  ttac (A |- r)
   A2 ?- t2                        A3 ?- t3
```

then

```
    A1 ?- t1
   ==========  CONJUNCTS_THEN ttac (A |- l /\ r)
    A3 ?- t3
```

## Failure

CONJUNCTS_THEN ttac will fail if applied to a theorem whose conclusion is not a conjunction.

## Comments

CONJUNCTS_THEN ttac (A |- u1 /\ ... /\ un) results in the tactic:

```
   ttac (A |- u1) THEN ttac (A |- u2 /\ ... /\ un)
```

The iterated effect:

```
   ttac (A |- u1) THEN ... THEN ttac(A |- un)
```

can be achieved by

```
  REPEAT_TCL CONJUNCTS_THEN ttac (A |- u1 /\ ... /\ un)
```

## See also

CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJUNCTS_TAC, CONJUNCTS_THEN2, STRIP_THM_THEN.

# CONJUNCTS

```
CONJUNCTS : thm -> thm list
```

## Synopsis
Recursively splits conjunctions into a list of conjuncts.

## Description
Flattens out all conjuncts, regardless of grouping. Returns a singleton list if the input theorem is not a conjunction.

```
       A |- t1 /\ t2 /\ ... /\ tn
   -----------------------------------  CONJUNCTS
    A |- t1   A |- t2   ...   A |- tn
```

## Failure
Never fails.

## Example

```
# CONJUNCTS(ASSUME `(x /\ y) /\ z /\ w`);;
val it : thm list =
  [(x /\ y) /\ z /\ w |- x; (x /\ y) /\ z /\ w |- y; (x /\ y) /\ z /\ w
   |- z; (x /\ y) /\ z /\ w |- w]
```

## See also
CONJ, CONJUNCT1, CONJUNCT2, CONJ_PAIR.

# constants

```
constants : unit -> (string * hol_type) list
```

## Synopsis
Returns a list of the constants currently defined.

## Description
The call

```
constants();;
```

returns a list of all the constants that have been defined so far.

## Failure
Never fails.

## See also
`axioms`, `binders`, `infixes`.

# CONTRAPOS_CONV

```
CONTRAPOS_CONV : term -> thm
```

## Synopsis
Proves the equivalence of an implication and its contrapositive.

## Description
When applied to an implication `‘p ==> q‘`, the conversion `CONTRAPOS_CONV` returns the theorem:

```
|- (p ==> q) <=> (~q ==> ~p)
```

## Failure
Fails if applied to a term that is not an implication.

## Comments
The same effect can be had by `GEN_REWRITE_CONV I [GSYM CONTRAPOS_THM]`

## See also
`CONTRAPOS`.

# CONTR

```
CONTR : term -> thm -> thm
```

## Synopsis
Implements the intuitionistic contradiction rule.

## Description

When applied to a term `t` and a theorem `A |- F`, the inference rule `CONTR` returns the theorem `A |- t`.

```
 A |- F
--------  CONTR `t`
 A |- t
```

## Failure

Fails unless the term has type `bool` and the theorem has `F` as its conclusion.

## Example

```
# let th = REWRITE_RULE[ARITH] (ASSUME `1 = 0`);;
val th : thm = 1 = 0 |- F

#  CONTR `Russell:Person = Pope` th;;
val it : thm = 1 = 0 |- Russell = Pope
```

## See also

`CCONTR`, `CONTR_TAC`, `NOT_ELIM`.

# CONTR_TAC

```
CONTR_TAC : thm_tactic
```

## Synopsis

Solves any goal from contradictory theorem.

## Description

When applied to a contradictory theorem `A' |- F`, and a goal `A ?- t`, the tactic `CONTR_TAC` completely solves the goal. This is an invalid tactic unless `A'` is a subset of `A`.

```
 A ?- t
========  CONTR_TAC (A' |- F)
```

## Uses

One quite common pattern is to use a contradictory hypothesis via `FIRST_ASSUM CONTR_TAC`.

## Failure

Fails unless the theorem is contradictory, i.e. has `F` as its conclusion.

## See also

`CHECK_ASSUME_TAC`, `CONTR`, `CCONTR`, `CONTRAPOS`, `NOT_ELIM`.

# CONV_RULE

```
CONV_RULE : conv -> thm -> thm
```

## Synopsis

Makes an inference rule from a conversion.

## Description

If `c` is a conversion, then `CONV_RULE c` is an inference rule that applies `c` to the conclusion of a theorem. That is, if `c` maps a term `` `t` `` to the theorem `|- t = t'`, then the rule `CONV_RULE c` infers `|- t'` from the theorem `|- t`. More precisely, if `` c `t` `` returns `A' |- t = t'`, then:

```
       A |- t
   --------------  CONV_RULE c
    A u A' |- t'
```

Note that if the conversion `c` returns a theorem with assumptions, then the resulting inference rule adds these to the assumptions of the theorem it returns.

## Failure

`CONV_RULE c th` fails if `c` fails when applied to the conclusion of `th`. The function returned by `CONV_RULE c` will also fail if the ML function `c` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

## Example

```
  # CONV_RULE BETA_CONV (ASSUME `(\x. x < 2) 1`);;
  val it : thm = (\x. x < 2) 1 |- 1 < 2
```

## See also

`CONV_TAC`.

# CONV_TAC

```
CONV_TAC : conv -> tactic
```

## Synopsis

Makes a tactic from a conversion.

## Description

If `c` is a conversion, then `CONV_TAC c` is a tactic that applies `c` to the goal. That is, if `c` maps a term `‘g‘` to the theorem `|- g = g’`, then the tactic `CONV_TAC c` reduces a goal `g` to the subgoal `g’`. More precisely, if `c ‘g‘` returns `A’ |- g = g’`, then:

```
        A ?- g
   ===============  CONV_TAC c
        A ?- g’
```

In the special case where `‘g‘` is `‘T‘`, the call immediately solves the goal rather than generating a subgoal `A ?- T`. And in a slightly liberal interpretation of "conversion", the conversion may also just prove the goal and return `A’ |- g`, in which case again the goal will be completely solved.

Note that in all cases the conversion `c` should return a theorem whose assumptions are also among the assumptions of the goal (normally, the conversion will returns a theorem with no assumptions). `CONV_TAC` does not fail if this is not the case, but the resulting tactic will be invalid, so the theorem ultimately proved using this tactic will have more assumptions than those of the original goal.

## Failure

`CONV_TAC c` applied to a goal `A ?- g` fails if `c` fails when applied to the term `g`. The function returned by `CONV_TAC c` will also fail if the function `c` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t’`).

## Uses

`CONV_TAC` can be used to apply simplifications that can't be expressed as equations (rewrite rules). For example, a goal:

```
# g `abs(pi - &22 / &7) <= abs(&355 / &113 - &22 / &7)`;;
```

can be simplified by rational number arithmetic:

```
# e(CONV_TAC REAL_RAT_REDUCE_CONV);;
val it : goalstack = 1 subgoal (1 total)

`abs (pi - &22 / &7) <= &1 / &791`
```

It is also handy for invoking decision procedures that only have a "rule" form, and no special "tactic" form. (Indeed, the tactic form can be defined in terms of the rule form by using `CONV_TAC`.) For example, the goal:

```
# g `!x:real. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))`;;
```

can be solved by:

```
# e(CONV_TAC REAL_FIELD);;
...
val it : goalstack = No subgoals
```

## See also

`CONV_RULE`.

# current_goalstack

```
current_goalstack : goalstack ref
```

## Synopsis

Reference variable holding current goalstack.

## Description

The reference variable `current_goalstack` contains the current goalstack. A goalstack is a type containing a list of goalstates.

## Failure

Not applicable.

### Comments

Users will probably not often want to examine this variable explicitly, since various proof commands modify it in various ways.

### See also

`b`, `g`, `e`, `r`.

## curry

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

### Synopsis

Converts a function on a pair to a corresponding curried function.

### Description

The application `curry f` returns `\x y. f(x,y)`, so that

```
curry f x y = f(x,y)
```

### Failure

Never fails.

### Example

```
# curry mk_var;;
val it : string -> hol_type -> term = <fun>
# it "x";;
val it : hol_type -> term = <fun>
# it `:bool`;;
val it : term = `x`
```

### See also

`uncurry`.

## decreasing

```
decreasing : ('a -> 'b) -> 'a -> 'a -> bool
```

### Synopsis

When applied to a "measure" function `f`, the call `increasing f` returns a binary function ordering elements in a call `increasing f x y` by `f(y) <? f(x)`, where the ordering `<?` is the OCaml polymorphic ordering.

### Failure

Never fails unless the measure function does.

### Example

```
# let nums = -5 -- 5;;
val nums : int list = [-5; -4; -3; -2; -1; 0; 1; 2; 3; 4; 5]
# sort (decreasing abs) nums;;
val it : int list = [5; -5; 4; -4; 3; -3; 2; -2; 1; -1; 0]
```

### See also

`<?`, `increasing`, `sort`.

## DEDUCT_ANTISYM_RULE

```
DEDUCT_ANTISYM_RULE : thm -> thm -> thm
```

### Synopsis

Deduces logical equivalence from deduction in both directions.

### Description

When applied to two theorems, this rule deduces logical equivalence between their conclusions with a modified assumption list:

```
       A |- q        B |- p
----------------------------------
 (A - {p}) u (B - {q}) |- p <=> q
```

The special case when `A = {p}` and `B = {q}` is perhaps the easiest to understand:

```
     {p} |- q        {q} |- p
    --------------------------
           |- p <=> q
```

### Failure

Never fails.

## Example

```
# let th1 = SYM(ASSUME `x:num = y`)
  and th2 = SYM(ASSUME `y:num = x`);;
val th1 : thm = x = y |- y = x
val th2 : thm = y = x |- x = y
# DEDUCT_ANTISYM_RULE th1 th2;;
val it : thm = |- y = x <=> x = y
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

`IMP_ANTISYM_RULE`, `PROVE_HYP`.

# deep_alpha

`deep_alpha : (string * string) list -> term -> term`

## Synopsis

Modify bound variable according to renaming scheme.

## Description

When applied to a list of string-string pairs

```
deep_alpha ["x1'","x1"; ...; "xn'","xn"]
```

a conversion results that will attempt to traverse a term and systematically replace any bound variable called `xi` with one called `xi'`. It will quietly do nothing in cases where that is impossible because of variable capture.

## Example

```
# deep_alpha ["x'","x"; "y'","y"] `?x. x <=> !y. y = y`;;
Warning: inventing type variables
val it : term = `?x'. x' <=> (!y'. y' = y')`
```

## Comments

This is used inside `PART_MATCH` to try to achieve a resonable correspondence in bound variable names, e.g. so that the bound variable is still called 'n' rather than 'x' here:

```
# REWR_CONV NOT_FORALL_THM `~(!n. n < m)`;;
val it : thm = |- ~(!n. n < m) <=> (?n. ~(n < m))
```

## See also
alpha, PART_MATCH.

# defined

```
defined : ('a, 'b) func -> 'a -> bool
```

## Synopsis
Tests if a finite partial function is defined on a certain domain value.

## Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The call `defined f x` returns `true` if the finite partial function `f` is defined on domain value `x`, and `false` otherwise.

## Failure
Never fails.

## Example

```
# defined (1 |=> 2) 1;;
val it : bool = true

# defined (1 |=> 2) 2;;
val it : bool = false

# defined undefined 1;;
val it : bool = false
```

## See also
|->, |=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefined.

# define

```
define : term -> thm
```

## Synopsis

Defines a general recursive function.

## Description

The function `define` should be applied to a conjunction of 'definitional' clauses `` `def_1[f] /\ ... /\ def_n[ ``
for some variable `f`, where each clause `def_i` is a universally quantified equation with an application of `f` to arguments on the left-hand side. The idea is that these clauses define the action of `f` on arguments of various kinds, for example on an empty list and nonempty list:

```
(f [] = a) /\ (!h t. CONS h t = k[f,h,t])
```

or on even numbers and odd numbers:

```
(!n. f(2 * n) = a[f,n]) /\ (!n. f(2 * n + 1) = b[f,n])
```

The `define` function attempts to prove that there is indeed a function satisfying all these properties, and if it succceeds it defines a new function `f` and returns the input term with the variable `f` replaced by the newly defined constant.

## Failure

Fails if the definition is malformed or if some of the necessary conditions for the definition to be admissible cannot be proved automatically, or if there is already a constant of the given name.

## Example

This is a 'multifactorial' function:

```
# define
    `multifactorial m n =
        if m = 0 then 1
        else if n <= m then n else n * multifactorial m (n - m)`;;
val it : thm =
|- multifactorial m n =
   (if m = 0 then 1 else if n <= m then n else n * multifactorial m (n - m))
```

Note that it fails without the `m = 0` guard because then there's no reason to suppose that `n - m` decreases and hence the recursion is apparently illfounded. Perhaps a more surprising example is the Collatz function:

```
# define
   `!n. collatz(n) = if n <= 1 then n
                     else if EVEN(n) then collatz(n DIV 2)
                     else collatz(3 * n + 1)`;;
```

Note that the definition was made successfully because there provably is a function satisfying these recursion equations, notwithstanding the fact that it is unknown whether the recursion is wellfounded. (Tail-recursive functions are always logically consistent, though they may not have any useful provable properties.)

## Comments

Assuming the definition is well-formed and the constant name is unused, failure indicates that `define` was unable to prove one or both of the following two properties: (i) the clauses are not mutually inconsistent (more than one clause could apply to some arguments, and the results are not obviously the same), or (ii) the definition is recursive and no ordering justifying the recursion could be arrived at by the automated heuristic. In order to make progress in such cases, try applying `prove_recursive_function_exists` to the same definition with existential quantification over `f`, to see the unproven side-conditions. An example is in the documentation for `prove_recursive_function_exists`.

## See also

`new_definition, new_recursive_definition, new_specification, prove_recursive_function_exists.`

# define_finite_type

```
define_finite_type : int -> thm
```

## Synopsis

Defines a new type of a specified finite size.

## Description

The call `define_finite_type n` where `n` is a positive integer defines a new type also called simply '`n`', and returns a theorem asserting that its universe has size `n`, in the form:

```
|- (:n) HAS_SIZE n
```

where `(:n)` is the customary HOL Light printing of the universe set `UNIV:n->bool`.

## Failure

Fails if `n` is zero or negative, or if there is a type of the same name (unless it was also defined by the same call for `define_finite_type`, which is perfectly permissible), or if the names of the type constructor and destructor functions are already in use:

```
mk_auto_define_finite_type_n:num->n
dest_auto_define_finite_type_n:32->num
```

## Example

Here we define a 32-element type, perhaps useful for indexing the bits of a word:

```
# define_finite_type 32;;
val it : thm = |- (:32) HAS_SIZE 32
```

## Uses

In conjunction with Cartesian powers such as `real^3`, where only the size of the indexing type is relevant and the simple name `n` is intuitive.

## See also

`define_type`, `new_type_definition`.

# define_quotient_type

```
define_quotient_type : string -> string * string -> term -> thm * thm
```

## Synopsis

Defines a quotient type based on given equivalence relation.

## Description

The call `define_quotient_type "qty" ("abs","rep") ‘R‘`, where `R:A->A->bool` is a binary relation, defines a new "quotient type" `:qty` and two new functions `abs:(A->bool)->qty` and `rep:qty->(A->bool)`, and returns the pair of theorems `|- abs(rep a) = a` and `|- (?x. r = R x) <=> re`
Normally, `R` will be an equivalence relation (reflexive, symmetric and transitive), in which case the quotient type will be in bijection with the set of `R`-equivalence classes.

## Failure

Fails if there is already a type `qty` or if either `abs` or `rep` is already in use as a constant.

## Example

For some purposes we may want to use "multisets" or "bags". These are like sets in that order is irrelevant, but like lists in that multiplicity is counted. We can define a type of finite multisets as a quotient of lists by the relation:

```
# let multisame = new_definition
    `multisame l1 l2 <=> !a:A. FILTER (\x. x = a) l1 = FILTER (\x. x = a) l2`;;
```

as follows:

```
# let multiset_abs,multiset_rep =
     define_quotient_type "multiset" ("multiset_of_list","list_of_multiset")
     `multisame:A list -> A list -> bool`;;
val multiset_abs : thm = |- multiset_of_list (list_of_multiset a) = a
val multiset_rep : thm =
  |- (?x. r = multisame x) <=> list_of_multiset (multiset_of_list r) = r
```

For development of this example, see the documentation entries for `lift_function` and `lift_theorem` (in that order). Similarly we could define a type of finite sets by:

```
define_quotient_type "finiteset" ("finiteset_of_list","list_of_finiteset")
 `\l1 l2. !a:A. MEM a l1 <=> MEM a l2`;;
val it : thm * thm =
  (|- finiteset_of_list (list_of_finiteset a) = a,
   |- (?x. r = (\l1 l2. !a. MEM a l1 <=> MEM a l2) x) <=>
      list_of_finiteset (finiteset_of_list r) = r)
```

## Uses

Convenient creation of quotient structures. Using related functions `lift_function` and `lift_theorem`, functions, relations and theorems can be lifted from the representing type to the type of equivalence classes. As well as those shown above, characteristic applications are the definition of rationals as equivalence classes of pairs of integers under cross-multiplication, or of 'directions' as equivalence classes of vectors under parallelism.

## Comments

If `R` is not an equivalence relation, the basic operation of `define_quotient_type` will work equally well, but the usefulness of the new type will be limited. In particular, `lift_function` and `lift_theorem` may not be usable.

## See also

`lift_function`, `lift_theorem`.

# define_type

```
define_type : string -> thm * thm
```

## Synopsis

Automatically define user-specified inductive data types.

## Description

The function `define_type` automatically defines an inductive data type or a mutually inductive family of them. These may optionally contain nested instances of other inductive data types. The function returns two theorems that together identify the type up to isomorphism. The input is just a string indicating the desired pattern of recursion. The simplest case where we define a single type is:

```
"op = C1 ty ... ty | C2 ty ... ty | ... | Cn ty ... ty"
```

where `op` is the name of the type constant or type operator to be defined, `C1`, ..., `Cn` are identifiers, and each `ty` is either a (logical) type expression valid in the current theory (in which case `ty` must not contain `op`) or just the identifier "`op`' itself.

A string of this form describes an n-ary type operator `op`, where n is the number of distinct type variables in the types `ty` on the right hand side of the equation. If n is zero then `op` is a type constant; otherwise `op` is an n-ary type operator. The type described by the specification has `n` distinct constructors `C1`, ..., `Cn`. Each constructor `Ci` is a function that takes arguments whose types are given by the associated type expressions `ty` in the specification. If one or more of the type expressions `ty` is the type `op` itself, then the equation specifies a recursive data type. In any specification, at least one constructor must be non-recursive, i.e. all its arguments must have types which already exist in the current theory.

Each of the types `ty` above may be built from the type being defined using other inductive type operators already defined, e.g. `list`. Moreover, one can actually have a mutually recursive family of types, where the format is a sequence of specifications in the above form separated by semicolons:

```
"op1 = C1_1 ty ... ty | C1_2 ty ... ty | ... | C1_n1 ty ... ty;
 op2 = C2_1 ty ... ty | ... | C2_n2 ty ... ty;
 ...
 opk = Ck_1 ty ... ty | ... | ... | Ck_nk ty ... ty"
```

Given a type specification of the form described above, `define_type` makes an appropriate type definition for the type operator or type operators. It then makes appropriate definitions for the constants `Ci_j` and automatically proves and returns two theorems that characterize the type up to isomorphism. Roughly, the first theorem allows one to prove properties over the new (family of) types by (mutual) induction, while the latter allows one to defined functions by recursion. Rather than presenting these in full generality, it is probably easier to consider some simple examples.

## Failure

The evaluation fails if one of the types or constructor constants is already defined, or if there are certain improper kinds of recursion, e.g. involving function spaces of one of the types being defined.

## Example

The following call to `define_type` defines `tri` to be a simple enumerated type with exactly three distinct values:

```
# define_type "tri = ONE | TWO | THREE";;
val it : thm * thm =
  (|- !P. P ONE /\ P TWO /\ P THREE ==> (!x. P x),
   |- !f0 f1 f2. ?fn. fn ONE = f0 /\ fn TWO = f1 /\ fn THREE = f2)
```

The theorem returned is a degenerate 'primitive recursion' theorem for the concrete type `tri`. An example of a recursive type that can be defined using `define_type` is a type of binary trees:

```
# define_type "btree = LEAF A | NODE btree btree";;
val it : thm * thm =
  (|- !P. (!a. P (LEAF a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (NODE a0 a1))
          ==> (!x. P x),
   |- !f0 f1.
          ?fn. (!a. fn (LEAF a) = f0 a) /\
               (!a0 a1. fn (NODE a0 a1) = f1 a0 a1 (fn a0) (fn a1)))
```

The theorem returned by `define_type` in this case asserts the unique existence of functions defined by primitive recursion over labelled binary trees. For an example of nested recursion, here we use the type of lists in a nested fashion to define a type of first-order terms:

```
# define_type "term = Var num | Fn num (term list)";;
val it : thm * thm =
  (|- !P0 P1.
          (!a. P0 (Var a)) /\
          (!a0 a1. P1 a1 ==> P0 (Fn a0 a1)) /\
          P1 [] /\
          (!a0 a1. P0 a0 /\ P1 a1 ==> P1 (CONS a0 a1))
          ==> (!x0. P0 x0) /\ (!x1. P1 x1),
   |- !f0 f1 f2 f3.
          ?fn0 fn1.
              (!a. fn1 (Var a) = f0 a) /\
              (!a0 a1. fn1 (Fn a0 a1) = f1 a0 a1 (fn0 a1)) /\
              fn0 [] = f2 /\
              (!a0 a1. fn0 (CONS a0 a1) = f3 a0 a1 (fn1 a0) (fn0 a1)))
```

and here we have an example of mutual recursion, defining syntax trees for commands and expressions for a hypothetical programming language:

```
# define_type "command = Assign num expression
                       | Ite expression command command;
               expression = Variable num
                          | Constant num
                          | Plus expression expression
                          | Valof command";;
val it : thm * thm =
  (|- !P0 P1.
         (!a0 a1. P1 a1 ==> P0 (Assign a0 a1)) /\
         (!a0 a1 a2. P1 a0 /\ P0 a1 /\ P0 a2 ==> P0 (Ite a0 a1 a2)) /\
         (!a. P1 (Variable a)) /\
         (!a. P1 (Constant a)) /\
         (!a0 a1. P1 a0 /\ P1 a1 ==> P1 (Plus a0 a1)) /\
         (!a. P0 a ==> P1 (Valof a))
         ==> (!x0. P0 x0) /\ (!x1. P1 x1),
   |- !f0 f1 f2 f3 f4 f5.
         ?fn0 fn1.
             (!a0 a1. fn0 (Assign a0 a1) = f0 a0 a1 (fn1 a1)) /\
             (!a0 a1 a2.
                  fn0 (Ite a0 a1 a2) =
                  f1 a0 a1 a2 (fn1 a0) (fn0 a1) (fn0 a2)) /\
             (!a. fn1 (Variable a) = f2 a) /\
             (!a. fn1 (Constant a) = f3 a) /\
             (!a0 a1. fn1 (Plus a0 a1) = f4 a0 a1 (fn1 a0) (fn1 a1)) /\
             (!a. fn1 (Valof a) = f5 a (fn0 a)))
```

## See also

INDUCT_THEN, new_recursive_definition, new_type_abbrev, prove_cases_thm, prove_constructors_distinct, prove_constructors_one_one, prove_induction_thm, prove_rec_fn_exists.

# define_type_raw

```
define_type_raw : (hol_type * (string * hol_type list) list) list -> thm * thm
```

## Synopsis

Like `define_type` but from a more structured representation than a string.

### Description

The core functionality of `define_type_raw` is the same as `define_type`, but the input is a more structured format for the type specification. In fact, `define_type` is just the composition of `define_type_raw` and `parse_inductive_type_specification`.

### Failure

May fail for the usual reasons `define_type` may.

### Uses

Not intended for general use, but sometimes useful in proof tools that want to generate inductive types.

### See also

`define`, `parse_inductive_type_specification`.

## delete_parser

```
delete_parser : string -> unit
```

### Synopsis

Uninstall a user parser.

### Description

HOL Light allows user parsing functions to be installed, and will try them on all terms during parsing before the usual parsers. The call `delete_parser "s"` removes any parsers associated with string `"s"`.

### Failure

Never fails, regardless of whether there are any parsers associated with the string.

### See also

`install_parser`, `installed_parsers`, `try_user_parser`.

## delete_user_printer

```
delete_user_printer : string -> unit
```

## Synopsis

Remove user-defined printer from the HOL Light term printing.

## Description

HOL Light allows arbitrary user printers to be inserted into the toplevel printer so that they are invoked on all applicable subterms (see `install_user_printer`). The call `delete_user_printer s` removes any such printer associated with the tag `s`.

## Failure

Never fails, even if there is no printer with the given tag.

## Example

If a user printer has been installed as in the example given for `install_user_printer`, it can be removed again by:

```
# delete_user_printer "print_typed_var";;
val it : unit = ()
```

## See also

`install_user_printer`, `try_user_printer`.

# denominator

```
denominator : num -> num
```

## Synopsis

Returns denominator of rational number in canonical form.

## Description

Given a rational number as supported by the `Num` library, `denominator` returns the denominator $q$ from the rational number cancelled to its reduced form, $p/q$ where $q > 0$ and $p$ and $q$ have no common factor.

## Failure

Never fails.

## Example

```
# denominator(Int 22 // Int 7);;
val it : num = 7
# denominator(Int 0);;
val it : num = 1
# denominator(Int 100);;
val it : num = 1
# denominator(Int 4 // Int(-2));;
val it : num = 1
```

## See also
`numdom`, `numerator`.

# DENUMERAL

```
DENUMERAL : thm -> thm
```

## Synopsis
Remove instances of the `NUMERAL` constant from a theorem.

## Description
The call `DENUMERAL th` removes from the conclusion of `th` any instances of the constant `NUMERAL`, used in the internal representation of numerals.

## Failure
Never fails.

## Uses
Only intended for users manipulating the internal structure of numerals.

## See also
`NUM_REDUCE_CONV`.

# DEPTH_BINOP_CONV

```
DEPTH_BINOP_CONV : term -> (term -> thm) -> term -> thm
```

## Synopsis
Applied a conversion to the leaves of a tree of binary operator expressions.

## Description
If a term `t` is built up from terms `t1,...,tn` using a binary operator `op` (for example `op (op t1 t2) (op (op t3 t4) t5)`), the call `DEPTH_BINOP_CONV ‘op‘ cnv t` will apply the conversion `cnv` to each `ti` to give a theorem `|- ti = ti’`, and return the equational theorem `|- t = t’` where `t’` results from replacing each `ti` in `t` with the corresponding `ti’`.

## Failure
Fails only if the core conversion `cnv` fails on one of the chosen subterms.

## Example
One can always completely evaluate arithmetic expressions with `NUM_REDUCE_CONV`, e.g.

```
# NUM_REDUCE_CONV ‘(1 + 2) + (3 * (4 + 5) + 6) + (7 DIV 8)‘;;
val it : thm = |- (1 + 2) + (3 * (4 + 5) + 6) + 7 DIV 8 = 36
```

However, if one wants for some reason not to reduce the top-level combination of additions, one can do instead:

```
# DEPTH_BINOP_CONV ‘(+):num->num->num‘ NUM_REDUCE_CONV
 ‘(1 + 2)   + (3 * (4 + 5) + 6) + (7 DIV 8)‘;;
val it : thm =
  |- (1 + 2) + (3 * (4 + 5) + 6) + 7 DIV 8 = (1 + 2) + (27 + 6) + 0
  # NUM_REDUCE_CONV ‘(1 + 2) + (3 * (4 + 5) + 6) + (7 DIV 8)‘;;
```

Note that the subterm `‘3 * (4 + 5)‘` did get completely evaluated, because the addition was not part of the toplevel tree, but was nested inside a multiplication.

## See also
`BINOP_CONV`, `ONCE_DEPTH_CONV`, `PROP_ATOM_CONV`, `TOP_DEPTH_CONV`.

# DEPTH_CONV

```
DEPTH_CONV : conv -> conv
```

## Synopsis
Applies a conversion repeatedly to all the sub-terms of a term, in bottom-up order.

## Description

DEPTH_CONV c tm repeatedly applies the conversion c to all the subterms of the term tm, including the term tm itself. The supplied conversion is applied repeatedly (zero or more times, as is done by REPEATC) to each subterm until it fails. The conversion is applied to subterms in bottom-up order.

## Failure

DEPTH_CONV c tm never fails but can diverge if the conversion c can be applied repeatedly to some subterm of tm without failing.

## Example

The following example shows how DEPTH_CONV applies a conversion to all subterms to which it applies:

```
# DEPTH_CONV BETA_CONV `(\x. (\y. y + x) 1) 2`;;
val it : thm = |- (\x. (\y. y + x) 1) 2 = 1 + 2
```

Here, there are two beta-redexes in the input term, one of which occurs within the other. DEPTH_CONV BETA_CONV applies beta-conversion to innermost beta-redex (\y. y + x) 1 first. The outermost beta-redex is then (\x. 1 + x) 2, and beta-conversion of this redex gives 1 + 2.

Because DEPTH_CONV applies a conversion bottom-up, the final result may still contain subterms to which the supplied conversion applies. For example, in:

```
# DEPTH_CONV BETA_CONV `(\f x. (f x) + 1) (\y.y) 2`;;
val it : thm = |- (\f x. f x + 1) (\y. y) 2 = (\y. y) 2 + 1
```

the right-hand side of the result still contains a beta-redex, because the redex `(\y.y)2` is introduced by virtue an application of BETA_CONV higher-up in the structure of the input term. By contrast, in the example:

```
# DEPTH_CONV BETA_CONV `(\f x. (f x)) (\y.y) 2`;;
val it : thm = |- (\f x. f x) (\y. y) 2 = 2
```

all beta-redexes are eliminated, because DEPTH_CONV repeats the supplied conversion (in this case, BETA_CONV) at each subterm (in this case, at the top-level term).

## Uses

If the conversion c implements the evaluation of a function in logic, then DEPTH_CONV c will do bottom-up evaluation of nested applications of it. For example, the conversion ADD_CONV implements addition of natural number constants within the logic. Thus, the effect of:

```
# DEPTH_CONV NUM_ADD_CONV `(1 + 2) + (3 + 4 + 5)`;;
val it : thm = |- (1 + 2) + 3 + 4 + 5 = 15
```

is to compute the sum represented by the input term.

### See also
ONCE_DEPTH_CONV, REDEPTH_CONV, TOP_DEPTH_CONV, TOP_SWEEP_CONV.

# DEPTH_SQCONV

```
DEPTH_SQCONV : strategy
```

### Synopsis
Applies simplification repeatedly to all the sub-terms of a term, in bottom-up order.

### Description
HOL Light's simplification functions (e.g. SIMP_TAC) have their traversal algorithm controlled by a "strategy". DEPTH_SQCONV is a strategy corresponding to DEPTH_CONV for ordinary conversions: simplification is applied repeatedly to all the sub-terms of a term, in bottom-up order.

### Failure
Not applicable.

### See also
DEPTH_CONV, ONCE_DEPTH_SQCONV, REDEPTH_SQCONV, TOP_DEPTH_SQCONV, TOP_SWEEP_SQCONV.

# derive_nonschematic_inductive_relations

```
derive_nonschematic_inductive_relations : term -> thm
```

### Synopsis
Deduce inductive definitions properties from an explicit assignment.

## Description

Given a set of clauses as given to `new_inductive_definitions`, the call `derive_nonschematic_inductive_rela` will introduce explicit equational constraints ("definitions", though only assumptions of the theorem, not actually constant definitions) that allow it to deduce those clauses. It will in general have additional 'monotonicity' hypotheses, but these may be removable by `prove_monotonicity_hyps`. None of the arguments are treated as schematic.

## Failure

Fails if the format of the clauses is wrong.

## Example

Here we try one of the classic examples of a mutually inductive definition, defining oddness and even-ness of natural numbers:

```
# (prove_monotonicity_hyps o derive_nonschematic_inductive_relations)
    `even(0) /\ odd(1) /\
     (!n. even(n) ==> odd(n + 1)) /\ (!n. odd(n) ==> even(n + 1))`;;
val it : thm =
  odd =
  (\a0. !odd' even'.
            (!a0. a0 = 1 \/ (?n. a0 = n + 1 /\ even' n) ==> odd' a0) /\
            (!a1. a1 = 0 \/ (?n. a1 = n + 1 /\ odd' n) ==> even' a1)
            ==> odd' a0),
  even =
  (\a1. !odd' even'.
            (!a0. a0 = 1 \/ (?n. a0 = n + 1 /\ even' n) ==> odd' a0) /\
            (!a1. a1 = 0 \/ (?n. a1 = n + 1 /\ odd' n) ==> even' a1)
            ==> even' a1)
  |- (even 0 /\
      odd 1 /\
      (!n. even n ==> odd (n + 1)) /\
      (!n. odd n ==> even (n + 1))) /\
     (!odd' even'.
          even' 0 /\
          odd' 1 /\
          (!n. even' n ==> odd' (n + 1)) /\
          (!n. odd' n ==> even' (n + 1))
          ==> (!a0. odd a0 ==> odd' a0) /\ (!a1. even a1 ==> even' a1)) /\
     (!a0. odd a0 <=> a0 = 1 \/ (?n. a0 = n + 1 /\ even n)) /\
     (!a1. even a1 <=> a1 = 0 \/ (?n. a1 = n + 1 /\ odd n))
```

Note that the final theorem has two assumptions that one can think of as the appropriate explicit definitions of these relations, and the conclusion gives the rule, induction and cases theorems.

## Comments

Normally, use `prove_inductive_relations_exist` or `new_inductive_definition`. This function is only needed for a very fine level of control.

## See also

`new_inductive_definition`, `prove_inductive_relations_exist`, `prove_monotonicity_hyps`.

# dest_abs

```
dest_abs : term -> term * term
```

## Synopsis

Breaks apart an abstraction into abstracted variable and body.

## Description

`dest_abs` is a term destructor for abstractions: `dest_abs ‘\var. t‘` returns `(‘var‘,‘t‘)`.

## Failure

Fails with `dest_abs` if term is not an abstraction.

## Example

```
# dest_abs ‘\x. x + 1‘;;
val it : term * term = (‘x‘, ‘x + 1‘)
```

## See also

`dest_comb`, `dest_const`, `dest_var`, `is_abs`, `mk_abs`, `strip_abs`.

# dest_binary

```
dest_binary : string -> term -> term * term
```

## Synopsis

Breaks apart an instance of a binary operator with given name.

## Description

The call `dest_binary s tm` will, if `tm` is a binary operator application `(op l) r` where `op` is a constant with name `s`, return the two arguments to which it is applied as a pair `l,r`. Otherwise, it fails. Note that `op` is required to be a constant.

## Failure

Never fails.

## Example

This one succeeds:

```
# dest_binary "+" `1 + 2`;;
val it : term * term = (`1`, `2`)
```

## See also

`dest_binop, is_binary, is_comb, mk_binary.`

# dest_binder

```
dest_binder : string -> term -> term * term
```

## Synopsis

Breaks apart a “binder”.

## Description

Applied to a term `tm` of the form `` `c (\x. t)` `` where `c` is a constant whose name is `"s"`, the call `dest_binder "c" tm` returns `` (`x`,`t`) ``. Note that this is actually independent of whether the name parses as a binder, but the usual application is where it does.

## Failure

Fails if the term is not of the appropriate form with a constant of the same name.

## Example

The call `dest_binder "!"` is the same as `dest_forall`, and is in fact how that function is implemented.

## See also

`dest_abs, dest_comb, dest_const, dest_var.`

# dest_binop

```
dest_binop : term -> term -> term * term
```

## Synopsis

Breaks apart an application of a given binary operator to two arguments.

## Description

The call `dest_binop op t`, where `t` is of the form `(op l) r`, will return the pair `l,r`. If `t` is not of that form, it fails. Note that `op` can be any term; it need not be a constant nor parsed infix.

## Failure

Fails if the term is not a binary application of operator `op`.

## Example

```
# dest_binop `(+):num->num->num` `1 + 2 + 3`;;
val it : term * term = (`1`, `2 + 3`)
```

## See also

`dest_binary`, `is_binary`, `is_binop`, `mk_binary`, `mk_binop`.

# dest_comb

```
dest_comb : term -> term * term
```

## Synopsis

Breaks apart a combination (function application) into rator and rand.

## Description

`dest_comb` is a term destructor for combinations:

```
dest_comb `t1 t2`
```

returns (`t1`,`t2`).

### Failure
Fails with `dest_comb` if term is not a combination.

### Example

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

We can use `dest_comb` to reveal more about the internal representation of numerals:

```
# dest_comb `12`;;
val it : term * term = (`NUMERAL`, `BIT0 (BIT0 (BIT1 (BIT1 _0)))`)
```

### See also
`dest_abs`, `dest_const`, `dest_var`, `is_comb`, `list_mk_comb`, `mk_comb`, `strip_comb`.

## dest_cond

```
dest_cond : term -> term * (term * term)
```

### Synopsis
Breaks apart a conditional into the three terms involved.

### Description
`dest_cond` is a term destructor for conditionals:

```
dest_cond `if t then t1 else t2`
```

returns (`t`,`t1`,`t2`).

### Failure
Fails with `dest_cond` if term is not a conditional.

### See also
`mk_cond`, `is_cond`.

## dest_conj

```
dest_conj : term -> term * term
```

## Synopsis
Term destructor for conjunctions.

## Description
`dest_conj(‘t1 /\ t2‘)` returns `(‘t1‘,‘t2‘)`.

## Failure
Fails with `dest_conj` if term is not a conjunction.

## See also
`is_conj`, `mk_conj`.

# dest_cons

```
dest_cons : term -> term * term
```

## Synopsis
Breaks apart a ‘CONS pair’ into head and tail.

## Description
`dest_cons` is a term destructor for ‘CONS pairs’. When applied to a term representing a nonempty list `‘[t;t1;...;tn]‘` (which is equivalent to `‘CONS t [t1;...;tn]‘`), it returns the pair of terms `(‘t‘,‘[t1;...;tn]‘)`.

## Failure
Fails with `dest_cons` if the term is not a non-empty list.

## See also
`dest_list`, `is_cons`, `is_list`, `mk_cons`, `mk_list`.

# dest_const

```
dest_const : term -> string * hol_type
```

## Synopsis
Breaks apart a constant into name and type.

## Description

dest_const is a term destructor for constants:

```
dest_const `const:ty`
```

returns ("const",`:ty`).

## Failure

Fails with dest_const if term is not a constant.

## Example

```
# dest_const `T`;;
val it : string * hol_type = ("T", `:bool`)
```

## See also

dest_abs, dest_comb, dest_var, is_const, mk_const, mk_mconst, name_of.

# dest_disj

```
dest_disj : term -> term * term
```

## Synopsis

Breaks apart a disjunction into the two disjuncts.

## Description

dest_disj is a term destructor for disjunctions:

```
dest_disj `t1 \/ t2`
```

returns (`t1`,`t2`).

## Failure

Fails with dest_disj if term is not a disjunction.

## See also

is_disj, mk_disj.

# dest_eq

```
dest_eq : term -> term * term
```

## Synopsis
Term destructor for equality.

## Description
`dest_eq(‘t1 = t2‘)` returns `(‘t1‘,‘t2‘)`.

## Failure
Fails with `dest_eq` if term is not an equality.

## Example

```
# dest_eq ‘2 + 2 = 4‘;;
val it : term * term = (‘2 + 2‘, ‘4‘)
```

## See also
`is_eq, mk_eq.`

# dest_exists

```
dest_exists : term -> term * term
```

## Synopsis
Breaks apart an existentially quantified term into quantified variable and body.

## Description
`dest_exists` is a term destructor for existential quantification: `dest_exists ‘?var. t‘` returns `(‘var‘,‘t‘)`.

## Failure
Fails with `dest_exists` if term is not an existential quantification.

## See also
`is_exists, mk_exists, strip_exists.`

# dest_forall

```
dest_forall : term -> term * term
```

## Synopsis

Breaks apart a universally quantified term into quantified variable and body.

## Description

`dest_forall` is a term destructor for universal quantification: `dest_forall ‘!var. t‘` returns (`‘var‘`,`‘t‘`).

## Failure

Fails with `dest_forall` if term is not a universal quantification.

## See also

`is_forall`, `mk_forall`, `strip_forall`.

# dest_fun_ty

```
dest_fun_ty : hol_type -> hol_type * hol_type
```

## Synopsis

Break apart a function type into domain and range.

## Description

The call `dest_fun_ty ‘:s->t‘` breaks apart the function type `s->t` and returns the pair `‘:s‘,‘:t‘`.

## Failure

Fails if the type given as argument is not a function type (constructor `"fun"`).

## Example

```
# dest_fun_ty `:A->B`;;
val it : hol_type * hol_type = (`:A`, `:B`)

# dest_fun_ty `:num->num->bool`;;
val it : hol_type * hol_type = (`:num`, `:num->bool`)

# dest_fun_ty `:A#B`;;
Exception: Failure "dest_fun_ty".
```

## See also

dest_type, mk_fun_ty.

# dest_gabs

```
dest_gabs : term -> term * term
```

## Synopsis

Breaks apart a generalized abstraction into abstracted varstruct and body.

## Description

`dest_pabs` is a term destructor for generalized abstractions: for example with a paired varstruct the effect on `dest_pabs `\(v1..(..)..vn). t`` is to return the pair `(`(v1..(..)..vn)`,`t`` It will also act as for `dest_abs` on basic abstractions.

## Failure

Fails unless the term is a basic or generalized abstraction.

## Example

These are fairly typical applications:

```
# dest_gabs `\(x,y). x + y`;;
val it : term * term = (`x,y`, `x + y`)

# dest_gabs `\(CONS h t). h + 1`;;
val it : term * term = (`CONS h t`, `h + 1`)
```

while the following shows that it also works on basic abstractions:

```
# dest_gabs `\x. x`;;
Warning: inventing type variables
val it : term * term = (`x`, `x`)
```

## See also

`GEN_BETA_CONV`, `is_gabs`, `mk_gabs`, `strip_gabs`.

# dest_iff

```
dest_iff : term -> term * term
```

## Synopsis

Term destructor for logical equivalence.

## Description

`dest_iff(`t1 <=> t2`)` returns `(`t1`,`t2`)`.

## Failure

Fails with if term is not a logical equivalence, i.e. an equation between terms of Boolean type.

## Example

```
# dest_iff `x = y <=> y = 1`;;
val it : term * term = (`x = y`, `y = 1`)
```

## Comments

The function `dest_eq` has the same effect, but the present function checks that the types of the two sides are indeed Boolean, whereas `dest_eq` will break apart any equation.

## See also
`dest_eq, is_iff, mk_iff.`

# dest_imp

```
dest_imp : term -> term * term
```

## Synopsis
Breaks apart an implication into antecedent and consequent.

## Description
`dest_imp` is a term destructor for implications. Thus

```
dest_imp `t1 ==> t2`
```

returns

```
(`t1`,`t2`)
```

## Failure
Fails with `dest_imp` if term is not an implication.

## Comments
Previous version of `dest_imp` treats negations as an implication with `F` as the conclusion. This is renamed to `dest_neg_imp`.

## See also
`mk_imp, is_imp, strip_imp, dest_neg_imp.`

# dest_intconst

```
dest_intconst : term -> num
```

## Synopsis
Converts an integer literal of type `:int` to an OCaml number.

## Description

The call `dest_intconst t` where `t` is a canonical integer literal of type `:int`, returns the corresponding OCaml number (type `num`). The permissible forms of integer literals are ‘`&n`’ for a numeral `n` or ‘`-- &n`’ for a nonzero numeral `n`.

## Failure

Fails if applied to a term that is not a canonical integer literal of type `:int`.

## Example

```
# dest_intconst `-- &11 :int`;;
val it : num = -11
```

## See also

`dest_realintconst, is_intconst, mk_intconst.`

# dest_let

```
dest_let : term -> (term * term) list * term
```

## Synopsis

Breaks apart a let-expression.

## Description

`dest_let` is a term destructor for general let-expressions: `dest_let `let x1 = e1 and ... and xn = en in` returns a pair of the list `[`x1`,`e1`; ... ; `xn`,`en`]` and ``E``.

## Failure

Fails with `dest_let` if term is not a `let`-expression.

## Example

```
# dest_let `let m = 256 and n = 65536 in (x MOD m + y MOD m) MOD n`;;
val it : (term * term) list * term =
  ([(`m`, `256`); (`n`, `65536`)], `(x MOD m + y MOD m) MOD n`)
```

## See also

`mk_let, is_let.`

# dest_list

`dest_list : term -> term list`

## Synopsis
Iteratively breaks apart a list term.

## Description
`dest_list` is a term destructor for lists: `dest_list(‘[t1;...;tn]:(ty)list‘)` returns `[‘t1‘;...;‘tn‘]`.

## Failure
Fails with `dest_list` if the term is not a list.

## See also
`dest_cons, dest_setenum, is_cons, is_list, mk_cons, mk_list.`

# dest_neg

`dest_neg : term -> term`

## Synopsis
Breaks apart a negation, returning its body.

## Description
`dest_neg` is a term destructor for negations: `dest_neg ‘~t‘` returns `‘t‘`.

## Failure
Fails with `dest_neg` if term is not a negation.

## See also
`is_neg, mk_neg.`

# dest_numeral

`dest_numeral : term -> num`

## Synopsis
Converts a HOL numeral term to unlimited-precision integer.

## Description
The call `dest_numeral t` where `t` is the HOL numeral representation of `n`, returns `n` as an unlimited-precision intger (type `num`). It fails if the term is not a numeral.

## Failure
Fails if the term is not a numeral.

## Example
```
# dest_numeral `0`;;
val it : num = 0

# dest_numeral `18446744073709551616`;;
val it : num = 18446744073709551616
```

## Comments
The similar function `dest_small_numeral` maps to a machine integer, which means it may overflow. So the use of `dest_numeral` is better unless you are very sure of the range.

```
# dest_small_numeral `18446744073709551616`;;
Exception: Failure "int_of_big_int".
```

## See also
`dest_small_numeral`, `is_numeral`, `mk_numeral`, `mk_small_numeral`, `rat_of_term`.

# dest_pair

```
dest_pair : term -> term * term
```

## Synopsis
Breaks apart a pair into two separate terms.

## Description
`dest_pair` is a term destructor for pairs: `dest_pair `(t1,t2)`` returns (`` `t1`,`t2` ``).

## Failure
Fails with `dest_pair` if term is not a pair.

## Example

```
# dest_pair `(1,2),(3,4),(5,6)`;;
val it : term * term = (`1,2`, `(3,4),5,6`)
```

## See also

dest_cons, is_pair, mk_pair.

# dest_realintconst

dest_realintconst : term -> num

## Synopsis

Converts an integer literal of type :real to an OCaml number.

## Description

The call dest_realintconst t where t is a canonical integer literal of type :real, returns the corresponding OCaml number (type num). The permissible forms of integer literals are '&n' for a numeral n or '-- &n' for a nonzero numeral n.

## Failure

Fails if applied to a term that is not a canonical integer literal of type :real.

## Example

```
# dest_realintconst `-- &27 :real`;;
val it : num = -27
```

## See also

dest_intconst, is_realintconst, mk_realintconst, rat_of_term.

# dest_select

dest_select : term -> term * term

## Synopsis

Breaks apart a choice term into selected variable and body.

## Description

`dest_select` is a term destructor for choice terms:

```
dest_select `@var. t`
```

returns (`var`,`t`).

## Failure

Fails with `dest_select` if term is not an epsilon-term.

## See also

`mk_select`, `is_select`.

# dest_setenum

```
dest_setenum : term -> term list
```

## Synopsis

Breaks apart a set enumeration.

## Description

`dest_setenum` is a term destructor for set enumerations: `dest_setenum `{t1,...,tn}`` returns `[`t1`;...;`tn`]`. Note that the list follows the syntactic pattern of the set enumeration, even if it contains duplicates. (The underlying set is still a set logically, of course, but can be represented redundantly.)

## Failure

Fails if the term is not a set enumeration.

## Example

```
# dest_setenum `{1,2,3,4}`;;
val it : term list = [`1`; `2`; `3`; `4`]

# dest_setenum `{1,2,1,2}`;;
val it : term list = [`1`; `2`; `1`; `2`]
```

## See also

`dest_list`, `is_setenum`, `mk_fset`, `mk_setenum`.

# dest_small_numeral

```
dest_small_numeral : term -> int
```

## Synopsis
Converts a HOL numeral term to machine integer.

## Description
The call `dest_small_numeral t` where `t` is the HOL numeral representation of `n`, returns `n` as an OCaml machine integer. It fails if the term is not a numeral or the result doesn't fit in a machine integer.

## Failure
Fails if the term is not a numeral or if the result doesn't fit in a machine integer.

## Example

```
#  dest_small_numeral `12`;;
val it : int = 12

# dest_small_numeral `18446744073709551616`;;
Exception: Failure "int_of_big_int".
```

## Comments
If overflow is a danger, you may be better off using OCaml type `num` and the analogous function `dest_numeral`. However, none of HOL's inference rules depend on the behaviour of machine integers, so logical soundness is not an issue.

## See also
`dest_numeral`, `is_numeral`, `mk_numeral`, `mk_small_numeral`, `rat_of_term`.

# dest_thm

```
dest_thm : thm -> term list * term
```

## Synopsis
Breaks a theorem into assumption list and conclusion.

## Description

dest_thm (t1,...,tn |- t) returns ([‘t1‘;...;‘tn‘],‘t‘).

## Failure

Never fails.

## Example

```
# dest_thm (ASSUME ‘1 = 0‘);;
val it : term list * term = ([‘1 = 0‘], ‘1 = 0‘)
```

## See also

concl, hyp.

# dest_type

```
dest_type : hol_type -> string * hol_type list
```

## Synopsis

Breaks apart a type (other than a variable type).

## Description

dest_type(‘:(ty1,...,tyn)op‘) returns ("op",[‘:ty1‘;...;‘:tyn‘]).

## Failure

Fails with dest_type if the type is a type variable.

## Example

```
# dest_type ‘:bool‘;;
val it : string * hol_type list = ("bool", [])

# dest_type ‘:(bool)list‘;;
val it : string * hol_type list = ("list", [‘:bool‘])

# dest_type ‘:num -> bool‘;;
val it : string * hol_type list = ("fun", [‘:num‘; ‘:bool‘])
```

## See also

mk_type, dest_vartype.

# dest_uexists

```
dest_uexists : term -> term * term
```

## Synopsis
Breaks apart a unique existence term.

## Description
If `t` has the form `?!x. p[x]` (there exists a unique [xthen `dest_uexists t` returns the pair `x,p[x]`; otherwise it fails.

## Failure
Fails if the term is not a 'unique existence' term.

## See also
`dest_exists`, `dest_forall`, `is_uexists`.

# dest_var

```
dest_var : term -> string * hol_type
```

## Synopsis
Breaks apart a variable into name and type.

## Description
`dest_var `var:ty`` returns `("var",`:ty`)`.

## Failure
Fails with `dest_var` if term is not a variable.

## Example

```
# dest_var `x:num`;;
val it : string * hol_type = ("x", `:num`)
```

## See also
`mk_var`, `is_var`, `dest_const`, `dest_comb`, `dest_abs`, `name_of`.

# dest_vartype

`dest_vartype : hol_type -> string`

## Synopsis
Breaks a type variable down to its name.

## Description
`dest_vartype ":A"` returns `"A"` when `A` is a type variable.

## Failure
Fails with `dest_vartype` if the type is not a type variable.

## Example

```
# dest_vartype `:A`;;
val it : string = "A"

# dest_vartype `:A->B`;;
Exception: Failure "dest_vartype: type constructor not a variable".
```

## See also
`mk_vartype, is_vartype, dest_type.`

# DISCH_ALL

`DISCH_ALL : thm -> thm`

## Synopsis
Discharges all hypotheses of a theorem.

## Description

```
      A1, ..., An |- t
----------------------------  DISCH_ALL
 |- A1 ==> ... ==> An ==> t
```

## Failure
`DISCH_ALL` will not fail if there are no hypotheses to discharge, it will simply return the theorem unchanged.

## Example

```
# end_itlist CONJ (map ASSUME [`p:bool`; `q:bool`; `r:bool`]);;
val it : thm = p, q, r |- p /\ q /\ r

# DISCH_ALL it;;
val it : thm = |- r ==> q ==> p ==> p /\ q /\ r
```

## Comments

Users should not rely on the hypotheses being discharged in any particular order. Two or more alpha-convertible hypotheses will be discharged by a single implication; users should not rely on which hypothesis appears in the implication.

## See also

DISCH, DISCH_TAC, DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

# DISCH

DISCH : term -> thm -> thm

## Synopsis

Discharges an assumption.

## Description

```
       A |- t
--------------------  DISCH `u`
 A - {u} |- u ==> t
```

## Failure

DISCH will fail if `u` is not boolean.

## Comments

The term `u` need not be a hypothesis. Discharging `u` will remove all identical and alpha-equivalent hypotheses.

## Example

```
# DISCH `p /\ q` (CONJUNCT1(ASSUME `p /\ q`));;
val it : thm = |- p /\ q ==> p
```

## See also

DISCH_ALL, DISCH_TAC, DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

# DISCH_TAC

```
DISCH_TAC : tactic
```

## Synopsis

Moves the antecedent of an implicative goal into the assumptions.

## Description

```
    A ?- u ==> v
  ==============  DISCH_TAC
    A u {u} ?- v
```

Note that `DISCH_TAC` treats `‘~u‘` as `‘u ==> F‘`, so will also work when applied to a goal with a negated conclusion.

## Failure

`DISCH_TAC` will fail for goals which are not implications or negations.

## Uses

Solving goals of the form `‘u ==> v‘` by rewriting `‘v‘` with `‘u‘`, although the use of `DISCH_THEN` is usually more elegant in such cases.

## Comments

If the antecedent already appears in the assumptions, it will be duplicated.

## See also

`DISCH`, `DISCH_ALL`, `DISCH_THEN`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`, `UNDISCH_TAC`.

# DISCH_THEN

```
DISCH_THEN : thm_tactic -> tactic
```

## Synopsis

Undischarges an antecedent of an implication and passes it to a theorem-tactic.

## Description

`DISCH_THEN` removes the antecedent and then creates a theorem by `ASSUME`ing it. This new theorem is passed to the theorem-tactic given as `DISCH_THEN`'s argument. The consequent tactic is then applied. Thus:

```
DISCH_THEN ttac (asl ?- t1 ==> t2) = ttac (ASSUME ‘t1‘) (asl ?- t2)
```

For example, if

```
 A ?- t
========  ttac (ASSUME ‘u‘)
 B ?- v
```

then

```
 A ?- u ==> t
==============  DISCH_THEN ttac
    B ?- v
```

Note that `DISCH_THEN` treats `‘~u‘` as `‘u ==> F‘`.

## Failure

`DISCH_THEN` will fail for goals that are not implications or negations.

## Example

Given the goal:

```
# g ‘!x. x = 0 ==> f(x) * x = x + 2 * x‘;;
```

we can discharge the antecedent and substitute with it immediately by:

```
# e(GEN_TAC THEN DISCH_THEN SUBST1_TAC);;
val it : goalstack = 1 subgoal (1 total)

‘f 0 * 0 = 0 + 2 * 0‘
```

and now `REWRITE_TAC[ADD_CLAUSES; MULT_CLAUSES]` will finish the job.

## Comments

The tactical `REFUTE_THEN` provides a more general classical ‘assume otherwise’ function.

## See also

`DISCH`, `DISCH_ALL`, `DISCH_TAC`, `REFUTE_THEN`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`, `UNDISCH_TAC`.

# DISJ1

```
DISJ1 : thm -> term -> thm
```

## Synopsis
Introduces a right disjunct into the conclusion of a theorem.

## Description

```
      A |- t1
   ---------------  DISJ1 (A |- t1) `t2`
    A |- t1 \/ t2
```

## Failure
Fails unless the term argument is boolean.

## Example

```
  # DISJ1 TRUTH `F`;;
  val it : thm = |- T \/ F
```

## See also
DISJ1_TAC, DISJ2, DISJ2_TAC, DISJ_CASES.

# DISJ1_TAC

```
DISJ1_TAC : tactic
```

## Synopsis
Selects the left disjunct of a disjunctive goal.

## Description

```
   A ?- t1 \/ t2
  ===============  DISJ1_TAC
     A ?- t1
```

## Failure
Fails if the goal is not a disjunction.

## See also
DISJ1, DISJ2, DISJ2_TAC.

# DISJ2

```
DISJ2 : term -> thm -> thm
```

## Synopsis
Introduces a left disjunct into the conclusion of a theorem.

## Description

```
      A |- t2
   ---------------  DISJ2 `t1`
    A |- t1 \/ t2
```

## Failure
Fails if the term argument is not boolean.

## Example

```
  # DISJ2 `F` TRUTH;;
  val it : thm = |- F \/ T
```

## See also
DISJ1, DISJ1_TAC, DISJ2_TAC, DISJ_CASES.

# DISJ2_TAC

```
DISJ2_TAC : tactic
```

## Synopsis
Selects the right disjunct of a disjunctive goal.

## Description

```
    A ?- t1 \/ t2
  ===============  DISJ2_TAC
       A ?- t2
```

## Failure

Fails if the goal is not a disjunction.

## See also

DISJ1, DISJ1_TAC, DISJ2.

# DISJ_ACI_RULE

```
DISJ_ACI_RULE : term -> thm
```

## Synopsis

Proves equivalence of two disjunctions containing same set of disjuncts.

## Description

The call `DISJ_ACI_RULE ‘t1 \/ ... \/ tn <=> u1 \/ ... \/ um‘`, where both sides of the equation are disjunctions of exactly the same set of disjuncts, (with arbitrary ordering, association, and repetitions), will return the corresponding theorem `|- t1 \/ ... \/ tn <=> u1 \/ ... \/`

## Failure

Fails if applied to a term that is not a Boolean equation or the two sets of disjuncts are different.

## Example

```
# DISJ_ACI_RULE ‘(p \/ q) \/ (q \/ r) <=> r \/ q \/ p‘;;
val it : thm = |- (p \/ q) \/ q \/ r <=> r \/ q \/ p
```

## Comments

The same effect can be had with the more general `AC` construct. However, for the special case of disjunction, `DISJ_ACI_RULE` is substantially more efficient when there are many disjuncts involved.

## See also

AC, CONJ_ACI_RULE, DISJ_CANON_CONV.

# DISJ_CANON_CONV

```
DISJ_CANON_CONV : term -> thm
```

## Synopsis
Puts an iterated disjunction in canonical form.

## Description
When applied to a term, `DISJ_CANON_CONV` splits it into the set of disjuncts and produces a theorem asserting the equivalence of the term and the new term with the disjuncts right-associated without repetitions and in a canonical order.

## Failure
Fails if applied to a non-Boolean term. If applied to a term that is not a disjunction, it will trivially work in the sense of regarding it as a single disjunct and returning a reflexive theorem.

## Example

```
# DISJ_CANON_CONV `(c \/ a \/ b) \/ (b \/ a \/ d)`;;
val it : thm = |- (c \/ a \/ b) \/ b \/ a \/ d <=> a \/ b \/ c \/ d
```

## See also
`AC`, `CONJ_CANON_CONV`, `DISJ_ACI_CONV`.

# DISJ_CASES

```
DISJ_CASES : thm -> thm -> thm -> thm
```

## Synopsis
Eliminates disjunction by cases.

## Description
The rule `DISJ_CASES` takes a disjunctive theorem, and two 'case' theorems, each with one of the disjuncts as a hypothesis while sharing alpha-equivalent conclusions. A new

theorem is returned with the same conclusion as the 'case' theorems, and the union of all assumptions excepting the disjuncts.

```
    A |- t1 \/ t2     A1 u {t1} |- t      A2 u {t2} |- t
   ------------------------------------------------------  DISJ_CASES
                    A u A1 u A2 |- t
```

## Failure

Fails if the first argument is not a disjunctive theorem, or if the conclusions of the other two theorems are not alpha-convertible.

## Example

Let us create several theorems. Note that `th1` and `th2` draw the same conclusion from different hypotheses, while `th` proves the disjunction of the two hypotheses:

```
# let [th; th1; th2] = map (UNDISCH_ALL o REAL_FIELD)
    [`~(x = &0) \/ x = &0`;
     `~(x = &0) ==> x * (inv(x) * x - &1) = &0`;
     `x = &0 ==> x * (inv(x) * x - &1) = &0`];;
  ...
val th : thm = |- ~(x = &0) \/ x = &0
val th1 : thm = ~(x = &0) |- x * (inv x * x - &1) = &0
val th2 : thm = x = &0 |- x * (inv x * x - &1) = &0
```

so we can apply `DISJ_CASES`:

```
# DISJ_CASES th th1 th2;;
val it : thm = |- x * (inv x * x - &1) = &0
```

## Comments

Neither of the 'case' theorems is required to have either disjunct as a hypothesis, but otherwise `DISJ_CASES` is pointless.

## See also

`DISJ_CASES_TAC`, `DISJ_CASES_THEN`, `DISJ_CASES_THEN2`, `DISJ1`, `DISJ2`, `SIMPLE_DISJ_CASES`.

# DISJ_CASES_TAC

```
DISJ_CASES_TAC : thm_tactic
```

## Synopsis

Produces a case split based on a disjunctive theorem.

## Description

Given a theorem `th` of the form `A |- u \/ v`, `DISJ_CASES_TAC th` applied to a goal produces two subgoals, one with `u` as an assumption and one with `v`:

```
            A ?- t
 =================================  DISJ_CASES_TAC (A |- u \/ v)
  A u {u} ?- t   A u {v}?- t
```

## Failure

Fails if the given theorem does not have a disjunctive conclusion.

## Example

Given the simple fact about arithmetic `th`, `|- m = 0 \/ (?n. m = SUC n)`, the tactic `DISJ_CASES_TAC th` can be used to produce a case split:

```
# let th = SPEC `m:num` num_CASES;;
val th : thm = |- m = 0 \/ (?n. m = SUC n)

# g `(P:num -> bool) m`;;
Warning: Free variables in goal: P, m
val it : goalstack = 1 subgoal (1 total)

`P m`

# e(DISJ_CASES_TAC th);;
val it : goalstack = 2 subgoals (2 total)

 0 [`?n. m = SUC n`]

`P m`

 0 [`m = 0`]

`P m`
```

## Uses

Performing a case analysis according to a disjunctive theorem.

## See also

`ASSUME_TAC`, `ASM_CASES_TAC`, `COND_CASES_TAC`, `DISJ_CASES_THEN`, `STRUCT_CASES_TAC`.

# DISJ_CASES_THEN2

```
DISJ_CASES_THEN2 : thm_tactic -> thm_tactic -> thm_tactic
```

## Synopsis

Applies separate theorem-tactics to the two disjuncts of a theorem.

## Description

If the theorem-tactics `ttac1` and `ttac2`, applied to the `ASSUME`d left and right disjunct of a theorem `|- u \/ v` respectively, produce results as follows when applied to a goal `(A ?- t)`:

```
    A ?- t                                      A ?- t
   =========  ttac1 (u |- u)      and          =========  ttac2 (v |- v)
    A ?- t1                                     A ?- t2
```

then applying `DISJ_CASES_THEN2 ttac1 ttac2 (|- u \/ v)` to the goal `(A ?- t)` produces two subgoals.

```
           A ?- t
   ======================  DISJ_CASES_THEN2 ttac1 ttac2 (|- u \/ v)
    A ?- t1      A ?- t2
```

## Failure

Fails if the theorem is not a disjunction. An invalid tactic is produced if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

## Example

Given the theorem

```
# let th = SPEC `m:num` num_CASES;;
val th : thm = |- m = 0 \/ (?n. m = SUC n)
```

and a goal:

```
# g `PRE m = m <=> m = 0`;;
```

the following produces two subgoals:

```
# e(DISJ_CASES_THEN2 SUBST1_TAC MP_TAC th);;
val it : goalstack = 2 subgoals (2 total)

`(?n. m = SUC n) ==> (PRE m = m <=> m = 0)`

`PRE 0 = 0 <=> 0 = 0`
```

The first subgoal has had the disjunct `m = 0` used for a substitution, and the second has added the disjunct as an antecedent. Alternatively, we can make the second theorem-tactic also choose a witness for the existential quantifier and follow by also substituting:

```
# e(DISJ_CASES_THEN2 SUBST1_TAC (CHOOSE_THEN SUBST1_TAC) th);;
val it : goalstack = 2 subgoals (2 total)

`PRE (SUC n) = SUC n <=> SUC n = 0`

`PRE 0 = 0 <=> 0 = 0`
```

Either subgoal can be finished with `ARITH_TAC`, but the way, but so could the initial goal.

## Uses

Building cases tacticals. For example, `DISJ_CASES_THEN` could be defined by:

```
let DISJ_CASES_THEN f = DISJ_CASES_THEN2 f f
```

## See also

`STRIP_THM_THEN`, `CHOOSE_THEN`, `CONJUNCTS_THEN`, `CONJUNCTS_THEN2`, `DISJ_CASES_THEN`.

# DISJ_CASES_THEN

```
DISJ_CASES_THEN : thm_tactical
```

## Synopsis

Applies a theorem-tactic to each disjunct of a disjunctive theorem.

## Description

If the theorem-tactic `f:thm->tactic` applied to either `ASSUME`d disjunct produces results as follows when applied to a goal `(A ?- t)`:

```
    A ?- t                                A ?- t
   =========  f (u |- u)      and        =========  f (v |- v)
    A ?- t1                               A ?- t2
```

then applying `DISJ_CASES_THEN f (|- u \/ v)` to the goal `(A ?- t)` produces two subgoals.

```
           A ?- t
   =====================  DISJ_CASES_THEN f (|- u \/ v)
    A ?- t1      A ?- t2
```

## Failure

Fails if the theorem is not a disjunction. An invalid tactic is produced if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

## Example

Given the theorem

```
th = |- (m = 0) \/ (?n. m = SUC n)
```

and a goal of the form `?- (PRE m = m) = (m = 0)`, applying the tactic

```
DISJ_CASES_THEN MP_TAC th
```

produces two subgoals, each with one disjunct as an added antecedant

```
# let th = SPEC `m:num` num_CASES;;
val th : thm = |- m = 0 \/ (?n. m = SUC n)
# g `PRE m = m <=> m = 0`;;
Warning: Free variables in goal: m
val it : goalstack = 1 subgoal (1 total)

`PRE m = m <=> m = 0`

# e(DISJ_CASES_THEN MP_TAC th);;
val it : goalstack = 2 subgoals (2 total)

`(?n. m = SUC n) ==> (PRE m = m <=> m = 0)`

`m = 0 ==> (PRE m = m <=> m = 0)`
```

## Uses

Building cases tactics. For example, `DISJ_CASES_TAC` could be defined by:

```
let DISJ_CASES_TAC = DISJ_CASES_THEN ASSUME_TAC
```

## Comments

Use `DISJ_CASES_THEN2` to apply different tactic generating functions to each case.

## See also

STRIP_THM_THEN, CHOOSE_THEN, CONJUNCTS_THEN, CONJUNCTS_THEN2, DISJ_CASES_TAC, DISJ_CASES_THEN2, DISJ_CASES_THENL.

# disjuncts

```
disjuncts : term -> term list
```

## Synopsis

Iteratively breaks apart a disjunction.

## Description

If a term `t` is a disjunction `p \/ q`, then `disjuncts t` will recursively break down `p` and `q` into disjuncts and append the resulting lists. Otherwise it will return the singleton list `[t]`. So if `t` is of the form `t1 \/ ... \/ tn` with any reassociation, no `ti` itself being a disjunction, the list returned will be `[t1; ...; tn]`. But

```
disjuncts(list_mk_disj([t1;...;tn]))
```

will not return `[t1;...;tn]` if any of `t1`,...,`tn` is a disjunction.

## Failure

Never fails, even if the term is not boolean.

## Example

```
# list_mk_disj [`a \/ b`;`c \/ d`;`e \/ f`];;
val it : term = `(a \/ b) \/ (c \/ d) \/ e \/ f`

# disjuncts it;;
val it : term list = [`a`; `b`; `c`; `d`; `e`; `f`]

# disjuncts `1`;;
val it : term list = [`1`]
```

## Comments

Because `disjuncts` splits both the left and right sides of a disjunction, this operation is not the inverse of `list_mk_disj`. You can also use `splitlist dest_disj` to split in a right-associated way only.

## See also

`conjuncts`, `dest_disj`, `list_mk_disj`.

# distinctness

```
distinctness : string -> thm
```

## Synopsis

Produce distinctness theorem for an inductive type.

## Description

A call `distinctness "ty"` where `"ty"` is the name of a recursive type defined with `define_type`, returns a "distinctness" theorem asserting that elements constructed by different type constructors are always different. The effect is exactly the same is if `prove_constructors_distinct` were applied to the recursion theorem produced by `define_type`, and the documentation for `prove_constructors_distinct` gives a lengthier discussion.

## Failure

Fails if `ty` is not the name of a recursive type, or if the type has only one constructor.

## Example

```
# distinctness "num";;
val it : thm = |- !n'. ~(0 = SUC n')

# distinctness "list";;
val it : thm = |- !a0' a1'. ~([] = CONS a0' a1')
```

## See also

`cases, define_type, injectivity, prove_constructors_distinct.`

# distinctness_store

```
distinctness_store : (string * thm) list ref
```

## Synopsis

Internal theorem list of distinctness theorems.

## Description

This list contains all the distinctness theorems (see `distinct`) for the recursive types defined so far. It is automatically extended by `define_type` and used as a cache by `distinct`.

## Failure

Not applicable.

## See also

`define_type, distinctness, extend_rectype_net, injectivity_store.`

# DNF_CONV

```
DNF_CONV : conv
```

## Synopsis
Converts a term already in negation normal form into disjunctive normal form.

## Description
When applied to a term already in negation normal form (see `NNF_CONV`), meaning that all other propositional connectives have been eliminated in favour of disjunction, disjunction and negation, and negation is only applied to atomic formulas, `DNF_CONV` puts the term into an equivalent disjunctive normal form, which is a right-associated disjunction of conjunctions without repetitions. No reduction by subsumption is performed, however, e.g. from `a \/ a /\ b` to just `a /\ b`).

## Failure
Never fails; non-Boolean terms will just yield a reflexive theorem.

## Example

```
# DNF_CONV `(a \/ b) /\ (a \/ c /\ e)`;;
val it : thm =
  |- (a \/ b) /\ (a \/ c /\ e) <=> a \/ a /\ b \/ a /\ c /\ e \/ b /\ c /\ e
```

## See also
`CNF_CONV`, `NNF_CONV`, `WEAK_CNF_CONV`, `WEAK_DNF_CONV`.

# do_list

```
do_list : ('a -> 'b) -> 'a list -> unit
```

## Synopsis
Apply imperative function to each element of a list.

## Description
The call `do_list f [x1; ... ; xn]` evaluates in sequence the expressions `f x1`, ..., `f xn` in that order, discarding the results. Presumably the applications will have some side-effect, such as printing something to the terminal.

## Example

```
# do_list (fun x -> print_string x; print_newline()) (explode "john");;
j
o
h
n
val it : unit = ()

# do_list (fun x -> print_string x) (rev(explode "nikolas"));;
salokinval it : unit = ()
```

## Uses

Running imperative code parametrized by list members.

## See also

`map`.

# dom

```
dom : ('a, 'b) func -> 'a list
```

## Synopsis

Returns domain of a finite partial function.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The `dom` operation returns the domain of such a function, i.e. the set of points on which it is defined.

## Failure

Attempts to sort the resulting list, so may fail if the domain type does not admit comparisons.

### Example

```
# dom (1 |=> "1");;
val it : int list = [1]
# dom(itlist I [2|->4; 3|->6] undefined);;
val it : int list = [2; 3]
```

### See also

|->, |=>, apply, applyd, choose, combine, defined, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefine, undefined.

## dpty

```
dpty : pretype
```

### Synopsis

Dummy pretype.

### Description

This is a dummy pretype, intended as a placeholder until the correct one is discovered.

### Failure

Not applicable.

### See also

pretype_of_type, type_of_pretype.

## e

```
e : tactic -> goalstack
```

### Synopsis

Applies a tactic to the current goal, stacking the resulting subgoals.

### Description

The function `e` is part of the subgoal package. It applies a tactic to the current goal to give a new proof state. The previous state is stored on the backup list. If the tactic produces subgoals, the new proof state is formed from the old one by adding a new level consisting of its subgoals.

The tactic applied is a validating version of the tactic given. It ensures that the justification of the tactic does provide a proof of the goal from the subgoals generated by the tactic. It will cause failure if this is not so. The tactical `VALID` performs this validation.

For a description of the subgoal package, see `set_goal`.

## Failure

`e tac` fails if the tactic `tac` fails for the top goal. It will diverge if the tactic diverges for the goal. It will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved. It will also fail in cases when the tactic is invalid.

## Example

```
# g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3]`

# e CONJ_TAC;;
val it : goalstack = 2 subgoals (2 total)

`TL [1; 2; 3] = [2; 3]`

`HD [1; 2; 3] = 1`

# e (REWRITE_TAC[HD]);;
val it : goalstack = 1 subgoal (1 total)

`TL [1; 2; 3] = [2; 3]`

# e (REWRITE_TAC[TL]);;
val it : goalstack = No subgoals
```

## Uses

Doing a step in an interactive goal-directed proof.

## See also

`b`, `g`, `p`, `r`, `top_goal`, `top_thm`.

# el

`el : int -> 'a list -> 'a`

## Synopsis
Extracts a specified element from a list.

## Description
`el i [x0;x1;...;xn]` returns `xi`. Note that the elements are numbered starting from `0`, not `1`.

## Failure
Fails with `el` if the integer argument is negative or greater than the length of the list.

## Example

```
# el 3 [1;2;7;1];;
val it : int = 1
```

## See also
`hd`, `tl`.

# elistof

`elistof : ('a -> 'b * 'a) -> ('a -> 'c * 'a) -> string -> 'a -> 'b list * 'a`

## Synopsis
Parses a possibly empty separated list of items.

## Description
If `p` is a parser for "items" of some kind, `s` is a parser for a "separator", and `e` is an error message, then `elistof p s e` parses a possibly empty list of successive items using `p`, where adjacent items are separated by something parseable by `s`. If a separator is parsed successfully but there is no following item that can be parsed by `s`, an exception `Failure e` is raised. (So note that the separator must not terminate the final element.)

## Failure
The call `elistof p s e` itself never fails, though the resulting parser may.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++, ||, >>, a, atleast, finished, fix, leftbin, listof, many, nothing, possibly, rightbin, some.`

# empty_net

```
empty_net : 'a net
```

## Synopsis

Empty term net.

## Description

Term nets (type `'a net`) are a lookup structure associating objects of type `'a`, e.g. conversions, with a corresponding 'pattern' term. For a given term, one can then relatively quickly look up all objects whose pattern terms might possibly match to it. This is used, for example, in rewriting to quickly filter out obviously inapplicable rewrites rather than attempting each one in turn. The (polymorphic) object `empty_net` is the term net with no objects defined; it can then be augmented by `enter` or `merge_nets` and used in `lookup`.

## Failure

Not applicable.

## See also

`enter, lookup, merge_nets.`

# empty_ss

```
empty_ss : simpset
```

## Synopsis

Simpset consisting of only the default rewrites and conversions.

## Description

In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset'. The simpset `empty_ss` has just the basic rewrites and conversions (see `basic_rewrites` and `basic_convs`), and no other provers.

## Failure

Not applicable.

## See also

`basic_convs`, `basic_rewrites`, `basic_ss`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# end_itlist

```
end_itlist : ('a -> 'a -> 'a) -> 'a list -> 'a
```

## Synopsis

List iteration function. Applies a binary function between adjacent elements of a list.

## Description

`end_itlist f [x1;...;xn]` returns `f x1 ( ... (f x(n-1) xn)...)`. Returns `x` for a one-element list `[x]`.

## Failure

Fails with `end_itlist` if list is empty.

## Example

```
# end_itlist (+) [1;2;3;4];;
val it : int = 10
```

## See also

`itlist`, `rev_itlist`.

# enter

```
enter : term list -> term * 'a -> 'a net -> 'a net
```

## Synopsis

Enter an object and its pattern term into a term net.

## Description

Term nets (type `'a net`) are a lookup structure associating objects of type `'a`, e.g. conversions, with a corresponding 'pattern' term. For a given term, one can then relatively quickly look up all objects whose pattern terms might possibly match to it. This is used, for example, in rewriting to quickly filter out obviously inapplicable rewrites rather than attempting each one in turn. The call `enter lconsts (pat,obj) net` enters the item `obj` into a net `obj` with indexing pattern term `pat`. The list `lconsts` lists variables that should be considered 'local constants' when matching, so will only match terms with exactly the same variable in corresponding places.

## Failure

Never fails.

## Example

Here we construct a net with the conversions for various arithmetic operations on numerals, each with a pattern term to identify the class of terms to which it might apply:

```
let arithnet = itlist (enter [])
   [`SUC n`,NUM_SUC_CONV;
    `m + n:num`,NUM_ADD_CONV;
    `m - n:num`,NUM_SUB_CONV;
    `m * n:num`,NUM_MULT_CONV;
    `m EXP n`,NUM_EXP_CONV;
    `m DIV n`,NUM_DIV_CONV;
    `m MOD n`,NUM_MOD_CONV]
   empty_net;;
```

Now we can define a conversion that uses lookup in this net as a first-stage filter and tries to apply the results.

```
let NUM_ARITH_CONV tm = FIRST_CONV (lookup tm arithnet) tm;;
```

Note that this is functionally not really different from just

```
let NUM_ARITH_CONV' =
  FIRST_CONV [NUM_SUC_CONV; NUM_ADD_CONV; NUM_SUB_CONV; NUM_MULT_CONV;
              NUM_EXP_CONV; NUM_DIV_CONV; NUM_MOD_CONV];;
```

but it may be significantly more efficient because instead of successive attempts to apply the conversions, each one is only invoked when the term has the right pattern.

```
# let tm = funpow 5 (fun x -> mk_binop `(*):num->num->num` x x) `12`;;
...
time (DEPTH_CONV NUM_ARITH_CONV) term;;
CPU time (user): 0.12
...
time (DEPTH_CONV NUM_ARITH_CONV') term;;
CPU time (user): 0.22
...
```

In situations with very many conversions, each one quite fast, the difference can be much more striking.

### See also
empty_net, lookup, merge_nets.

## EQF_ELIM

```
EQF_ELIM : thm -> thm
```

### Synopsis
Replaces equality with `F` by negation.

### Description

```
    A |- tm <=> F
   ---------------  EQF_ELIM
     A |- ~tm
```

### Failure
Fails if the argument theorem is not of the form `A |- tm <=> F`.

### Example

```
# EQF_ELIM(REFL `F`);;
val it : thm = |- ~F
```

### See also
EQF_INTRO, EQT_ELIM, EQT_INTRO, NOT_ELIM, NOT_INTRO.

# EQF_INTRO

```
EQF_INTRO : thm -> thm
```

## Synopsis

Converts negation to equality with F.

## Description

```
     A |- ~tm
   ---------------  EQF_INTRO
    A |- tm <=> F
```

## Failure

Fails if the argument theorem is not a negation.

## Example

```
  # let th = ASSUME `~p`;;
  val th : thm = ~p |- ~p

  # EQF_INTRO th;;
  val it : thm = ~p |- p <=> F
```

## See also

EQF_ELIM, EQT_ELIM, EQT_INTRO, NOT_ELIM, NOT_INTRO.

# EQ_IMP_RULE

```
EQ_IMP_RULE : thm -> thm * thm
```

## Synopsis

Derives forward and backward implication from equality of boolean terms.

## Description

When applied to a theorem `A |- t1 <=> t2`, where `t1` and `t2` both have type `bool`, the inference rule `EQ_IMP_RULE` returns the theorems `A |- t1 ==> t2` and `A |- t2 ==> t1`.

```
             A |- t1 <=> t2
 -----------------------------------  EQ_IMP_RULE
  A |- t1 ==> t2     A |- t2 ==> t1
```

## Failure

Fails unless the conclusion of the given theorem is an equation between boolean terms.

## Example

```
# SPEC_ALL CONJ_SYM;;
val it : thm = |- t1 /\ t2 <=> t2 /\ t1

# EQ_IMP_RULE it;;
val it : thm * thm = (|- t1 /\ t2 ==> t2 /\ t1, |- t2 /\ t1 ==> t1 /\ t2)
```

## See also

`EQ_MP`, `EQ_TAC`, `IMP_ANTISYM_RULE`.

# EQ_MP

```
EQ_MP : thm -> thm -> thm
```

## Synopsis

Equality version of the Modus Ponens rule.

## Description

When applied to theorems `A1 |- t1 <=> t2` and `A2 |- t1`, the inference rule `EQ_MP` returns the theorem `A1 u A2 |- t2`.

```
  A1 |- t1 <=> t2   A2 |- t1
 ----------------------------  EQ_MP
       A1 u A2 |- t2
```

## Failure

Fails unless the first theorem is equational and its left side is the same as the conclusion of the second theorem (and is therefore of type `bool`), up to alpha-conversion.

## Example

```
# let th1 = SPECL [`p:bool`; `q:bool`] CONJ_SYM
  and th2 = ASSUME `p /\ q`;;
val th1 : thm = |- p /\ q <=> q /\ p
val th2 : thm = p /\ q |- p /\ q
# EQ_MP th1 th2;;
val it : thm = p /\ q |- q /\ p
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

EQ_IMP_RULE, IMP_ANTISYM_RULE, MP, PROVE_HYP.

# =?

```
(=?) : 'a -> 'a -> bool
```

## Synopsis

Reflexive short-cutting equality test.

## Description

This is functionally identical to the OCaml equality test `=` except that it is reflexive even on floating-point NaN. More importantly, it will more efficiently short-cut comparisons of large data structures where subcomponents are identical (pointer equivalent).

## Failure

May fail when applied to functions.

## Example

```
# let x = 0.0 /. 0.0;;
val x : float = nan
# x = x;;
val it : bool = false
# x =? x;;
val it : bool = true
```

## See also

<?, <=?, >?, >=?.

# EQ_TAC

```
EQ_TAC : tactic
```

## Synopsis
Reduces goal of equality of boolean terms to forward and backward implication.

## Description
When applied to a goal `A ?- t1 <=> t2`, where `t1` and `t2` have type `bool`, the tactic `EQ_TAC` returns the subgoals `A ?- t1 ==> t2` and `A ?- t2 ==> t1`.

```
            A ?- t1 <=> t2
   =================================  EQ_TAC
    A ?- t1 ==> t2   A ?- t2 ==> t1
```

## Failure
Fails unless the conclusion of the goal is an equation between boolean terms.

## See also
`EQ_IMP_RULE`, `IMP_ANTISYM_RULE`.

# EQT_ELIM

```
EQT_ELIM : thm -> thm
```

## Synopsis
Eliminates equality with `T`.

## Description

```
    A |- tm <=> T
   ---------------  EQT_ELIM
      A |- tm
```

## Failure
Fails if the argument theorem is not of the form `A |- tm <=> T`.

## Example

```
# REFL `T`;;
val it : thm = |- T <=> T

# EQT_ELIM it;;
val it : thm = |- T
```

## See also

EQF_ELIM, EQF_INTRO, EQT_INTRO.

# EQT_INTRO

```
EQT_INTRO : thm -> thm
```

## Synopsis

Introduces equality with T.

## Description

```
      A |- tm
   ---------------  EQF_INTRO
    A |- tm <=> T
```

## Failure

Never fails.

## Example

```
# EQT_INTRO (REFL `2`);;
val it : thm = |- 2 = 2 <=> T
```

## See also

EQF_ELIM, EQF_INTRO, EQT_ELIM.

# equals_goal

```
equals_goal : goal -> goal -> bool
```

### Synopsis
Equality test on goals.

### Description
The relation `equals_goal` tests if two goals have exactly the same structure, with the same assumptions, conclusions and even labels, with the assumptions in the same order. The only respect in which this differs from a pure equality tests is that the various term components are tested modulo alpha-conversion.

### Failure
Never fails.

### Comments
Probably not generally useful. Used inside `CHANGED_TAC`.

### See also
`CHANGED_TAC`, `equals_thm`.

## equals_thm

```
equals_thm : thm -> thm -> bool
```

### Synopsis
Equality test on theorems.

### Description
The call `equals_thm th1 th2` returns `true` if and only if both the conclusions and assumptions of the two theorems `th1` and `th2` are exactly the same. The same can be achieved by a simple equality test, but it is better practice to use this function because it will also work in the proof recording version of HOL Light (see the `Proofrecording` subdirectory).

### Failure
Never fails.

### See also
`=?`.

## ETA_CONV

```
ETA_CONV : term -> thm
```

## Synopsis

Performs a toplevel eta-conversion.

## Description

ETA_CONV maps an eta-redex `\x. t x`, where x does not occur free in t, to the theorem |- (\x. t x) = t.

## Failure

Fails if the input term is not an eta-redex.

## Example

```
# ETA_CONV `\n. SUC n`;;
val it : thm = |- (\n. SUC n) = SUC

# ETA_CONV `\n. 1 + n`;;
val it : thm = |- (\n. 1 + n) = (+) 1

# ETA_CONV `\n. n + 1`;;
Exception: Failure "ETA_CONV".
```

## Comments

The same basic effect can be achieved by rewriting with ETA_AX. The theorem ETA_AX is one of HOL Light's three mathematical axioms.

# EVERY_ASSUM

```
EVERY_ASSUM : thm_tactic -> tactic
```

## Synopsis

Sequentially applies all tactics given by mapping a function over the assumptions of a goal.

## Description

When applied to a theorem-tactic f and a goal ({A1;...;An} ?- C), the EVERY_ASSUM tactical maps f over the list of assumptions then applies the resulting tactics, in sequence, to the goal:

```
EVERY_ASSUM f ({A1;...;An} ?- C)
 = (f(.. |- A1) THEN ... THEN f(.. |- An)) ({A1;...;An} ?- C)
```

If the goal has no assumptions, then EVERY_ASSUM has no effect.

## Failure

The application of EVERY_ASSUM to a theorem-tactic and a goal fails if the theorem-tactic fails when applied to any of the assumptions of the goal, or if any of the resulting tactics fail when applied sequentially.

## See also

ASSUM_LIST, MAP_EVERY, MAP_FIRST, THEN.

# EVERY_CONV

```
EVERY_CONV : conv list -> conv
```

## Synopsis

Applies in sequence all the conversions in a given list of conversions.

## Description

`EVERY_CONV [c1;...;cn] ‘t‘` returns the result of applying the conversions `c1`, ..., `cn` in sequence to the term `‘t‘`. The conversions are applied in the order in which they are given in the list. In particular, if `ci ‘ti‘` returns `|- ti=ti+1` for `i` from `1` to `n`, then `EVERY_CONV [c1;...;cn] ‘t1‘` returns `|- t1=t(n+1)`. If the supplied list of conversions is empty, then `EVERY_CONV` returns the identity conversion. That is, `EVERY_CONV [] ‘t‘` returns `|- t=t`.

## Failure

`EVERY_CONV [c1;...;cn] ‘t‘` fails if any one of the conversions `c1`, ..., `cn` fails when applied in sequence as specified above.

## Example

```
# EVERY_CONV [BETA_CONV; NUM_ADD_CONV] ‘(\x. x + 2) 5‘;;
val it : thm = |- (\x. x + 2) 5 = 7
```

## See also

THENC.

# EVERY

```
EVERY : tactic list -> tactic
```

## Synopsis
Sequentially applies all the tactics in a given list of tactics.

## Description
When applied to a list of tactics `[t1; ... ;tn]`, and a goal `g`, the tactical `EVERY` applies each tactic in sequence to every subgoal generated by the previous one. This can be represented as:

```
EVERY [t1;...;tn] = t1 THEN ... THEN tn
```

If the tactic list is empty, the resulting tactic has no effect.

## Failure
The application of `EVERY` to a tactic list never fails. The resulting tactic fails iff any of the component tactics do.

## Comments
It is possible to use `EVERY` instead of `THEN`, but probably stylistically inferior. `EVERY` is more useful when applied to a list of tactics generated by a function.

## See also
`FIRST`, `MAP_EVERY`, `THEN`.

# EVERY_TCL

```
EVERY_TCL : thm_tactical list -> thm_tactical
```

## Synopsis
Composes a list of theorem-tacticals.

## Description
When given a list of theorem-tacticals and a theorem, `EVERY_TCL` simply composes their effects on the theorem. The effect is:

```
EVERY_TCL [ttl1;...;ttln] = ttl1 THEN_TCL ... THEN_TCL ttln
```

In other words, if:

```
ttl1 ttac th1 = ttac th2  ...  ttln ttac thn = ttac thn'
```

then:

```
EVERY_TCL [ttl1;...;ttln] ttac th1 = ttac thn'
```

If the theorem-tactical list is empty, the resulting theorem-tactical behaves in the same way as ALL_THEN, the identity theorem-tactical.

### Failure
The application to a list of theorem-tacticals never fails.

### See also
FIRST_TCL, ORELSE_TCL, REPEAT_TCL, THEN_TCL.

## EXISTENCE

```
EXISTENCE : thm -> thm
```

### Synopsis
Deduces existence from unique existence.

### Description
When applied to a theorem with a unique-existentially quantified conclusion, EXISTENCE returns the same theorem with normal existential quantification over the same variable.

```
    A |- ?!x. p
   -------------  EXISTENCE
    A |- ?x. p
```

### Failure
Fails unless the conclusion of the theorem is unique-existentially quantified.

### Example

```
  # let th = MESON[] `?!n. n = m`;;
  ...
  val th : thm = |- ?!n. n = m

  # EXISTENCE th;;
  val it : thm = |- ?n. n = m
```

### See also
EXISTS, SIMPLE_EXISTS.

# exists

`exists : ('a -> bool) -> 'a list -> bool`

## Synopsis
Tests a list to see if some element satisfy a predicate.

## Description
`exists p [x1;...;xn]` returns `true` if `(p xi)` is true for some `xi` in the list. Otherwise, for example if the list is empty, it returns `false`.

## Failure
Never fails.

## Example

```
# exists (fun n -> n mod 2 = 0) [2;3;5;7;11;13;17];;
val it : bool = true
# exists (fun n -> n mod 2 = 0) [3;5;7;9;11;13;15];;
val it : bool = false
```

## See also
`find`, `forall`, `tryfind`, `mem`, `assoc`, `rev_assoc`.

# EXISTS_EQUATION

`EXISTS_EQUATION : term -> thm -> thm`

## Synopsis
Derives existence from explicit equational constraint.

## Description
Given a term `` `x = t` `` where `x` does not occur free in `t`, and a theorem `A |- p[x]`, the rule `EXISTS_EQUATION` returns `A - {x = t} |- ?x. p[x]`. Normally, the equation `x = t` is one of the hypotheses of the theorem, so this rule allows one to derive an existence assertion ignoring the actual "definition".

### Failure
Fails if the term is not an equation, if the LHS is not a variable, or if the variable occurs free in the RHS.

### Example

```
# let th = (UNDISCH o EQT_ELIM o SIMP_CONV[ARITH])
    `x = 3 ==> ODD(x) /\ x > 2`;;
val th : thm = x = 3 |- ODD x /\ x > 2

# EXISTS_EQUATION `x = 3` th;;
val it : thm = |- ?x. ODD x /\ x > 2
```

Note that it is not obligatory for the term to be an assumption:

```
# EXISTS_EQUATION `x = 1` (REFL `x:num`);;
val it : thm = |- ?x. x = x
```

### See also
EXISTS, SIMPLE_EXISTS.

## EXISTS_TAC

```
EXISTS_TAC : term -> tactic
```

### Synopsis
Reduces existentially quantified goal to one involving a specific witness.

### Description
When applied to a term `u` and a goal `A ?- ?x. t`, the tactic `EXISTS_TAC` reduces the goal to `A ?- t[u/x]` (substituting `u` for all free instances of `x` in `t`, with variable renaming if necessary to avoid free variable capture).

```
    A ?- ?x. t
   =============  EXISTS_TAC `u`
    A ?- t[u/x]
```

### Failure
Fails unless the goal's conclusion is existentially quantified and the term supplied has the same type as the quantified variable in the goal.

## Example

The goal:

```
# g `?x. 1 < x /\ x < 3`;;
```

can be solved by:

```
# e(EXISTS_TAC `2` THEN ARITH_TAC);;
val it : goalstack = No subgoals
```

## See also

EXISTS.

# EXISTS

```
EXISTS : term * term -> thm -> thm
```

## Synopsis

Introduces existential quantification given a particular witness.

## Description

When applied to a pair of terms and a theorem, the first term an existentially quantified pattern indicating the desired form of the result, and the second a witness whose substitution for the quantified variable gives a term which is the same as the conclusion of the theorem, EXISTS gives the desired theorem.

```
    A |- p[u/x]
  -------------  EXISTS (`?x. p`,`u`)
    A |- ?x. p
```

## Failure

Fails unless the substituted pattern is the same as the conclusion of the theorem.

## Example

The following examples illustrate how it is possible to deduce different things from the same theorem:

```
# EXISTS (`?x. x <=> T`,`T`) (REFL `T`);;
val it : thm = |- ?x. x <=> T

# EXISTS (`?x:bool. x = x`,`T`) (REFL `T`);;
val it : thm = |- ?x. x <=> x
```

## See also

CHOOSE, EXISTS_TAC, SIMPLE_EXISTS.

# EXPAND_CASES_CONV

```
EXPAND_CASES_CONV : conv
```

## Synopsis

Expand a numerical range `` `!i. i < n ==> P[i]` ``.

## Description

When applied to a term of the form `` `!i. i < n ==> P[i]` `` for some `P[i]` and a numeral `n`, the conversion `EXPAND_CASES_CONV` returns

```
|- (!i. i < n ==> P[i]) <=> P[0] /\ ... /\ P[n-1]
```

## Failure

Fails if applied to a term that is not of the right form.

## Example

```
# EXPAND_CASES_CONV `(!n. n < 5 ==> ~(n = 0) ==> 12 MOD n = 0)`;;
val it : thm =
  |- (!n. n < 5 ==> ~(n = 0) ==> 12 MOD n = 0) <=>
     (~(1 = 0) ==> 12 MOD 1 = 0) /\
     (~(2 = 0) ==> 12 MOD 2 = 0) /\
     (~(3 = 0) ==> 12 MOD 3 = 0) /\
     (~(4 = 0) ==> 12 MOD 4 = 0)

# (EXPAND_CASES_CONV THENC NUM_REDUCE_CONV)
    `(!n. n < 5 ==> ~(n = 0) ==> 12 MOD n = 0)`;;
val it : thm = |- (!n. n < 5 ==> ~(n = 0) ==> 12 MOD n = 0) <=> T
```

## See also

NUM_REDUCE_CONV.

# EXPAND_TAC

`EXPAND_TAC : string -> tactic`

## Synopsis
Expand an abbreviation in the hypotheses.

## Description
The tactic `EXPAND_TAC "x"`, applied to a goal, looks for a hypothesis of the form `‘t = x‘` where `x` is a variable with the given name. It then replaces `x` by `t` throughout the conclusion of the goal.

## Failure
Fails if there is no suitable assumption in the goal.

## Example
Consider the final goal in the example given for `ABBREV_TAC`:

```
val it : goalstack = 1 subgoal (1 total)

 0 [‘12345 + 12345 = n‘]

‘n + f n = f n‘
```

If we expand it, we get:

```
# e(EXPAND_TAC "n");;
val it : goalstack = 1 subgoal (1 total)

 0 [‘12345 + 12345 = n‘]

‘(12345 + 12345) + f (12345 + 12345) = f (12345 + 12345)‘
```

## See also
`ABBREV_TAC`.

# explode

`explode : string -> string list`

## Synopsis
Converts a string into a list of single-character strings.

## Description
`explode s` returns the list of single-character strings that make up `s`, in the order in which they appear in `s`. If `s` is the empty string, then an empty list is returned.

## Failure
Never fails.

## Example

```
# explode "example";;
val it : string list = ["e"; "x"; "a"; "m"; "p"; "l"; "e"]
```

## See also
`implode`.

# extend_basic_congs

```
extend_basic_congs : thm list -> unit
```

## Synopsis
Extends the set of congruence rules used by the simplifier.

## Description
The HOL Light simplifier (as invoked by `SIMP_TAC` etc.) uses congruence rules to determine how it uses context when descending through a term. These are essentially theorems showing how to decompose one equality to a series of other inequalities in context. A call to `extend_basic_congs thl` adds the congruence rules in `thl` to the defaults.

## Failure
Never fails.

## Example
By default, the simplifier uses context `p` when simplifying `q` within an implication `p ==> q`. Some users would like the simplifier to do likewise for a conjunction `p /\ q`, which is not

done by default:

```
# SIMP_CONV[] `x = 1 /\ x < 2`;;
val it : thm = |- x = 1 /\ x < 2 <=> x = 1 /\ x < 2
```

You can make it do so with

```
# extend_basic_congs
    [TAUT `(p <=> p') ==> (p' ==> (q <=> q')) ==> (p /\ q <=> p' /\ q')`];;
val it : unit = ()
```

as you can see:

```
# SIMP_CONV[] `x = 1 /\ x < 2`;;
val it : thm = |- x = 1 /\ x < 2 <=> x = 1 /\ 1 < 2

# SIMP_CONV[ARITH] `x = 1 /\ x < 2`;;
val it : thm = |- x = 1 /\ x < 2 <=> x = 1
```

## See also

`basic_congs`, `set_basic_congs`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# extend_basic_convs

```
extend_basic_convs : string * (term * conv) -> unit
```

## Synopsis

Extend the set of default conversions used by rewriting and simplification.

## Description

The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_` variants. The latter are normally term transformations that cannot be expressed as single (conditional or unconditional) rewrite rules. A call to

```
extend_basic_convs("name",(`pat`,conv))
```

will add the conversion `conv` into the default set, using the name `name` to refer to it and restricting it to subterms encountered that match `pat`.

## Failure

Never fails.

## Example

By default, no arithmetic is done in rewriting, though rewriting with the theorem `ARITH` gives that effect.

```
# REWRITE_CONV[] `x = 1 + 2 + 3 + 4`;;
val it : thm = |- x = 1 + 2 + 3 + 4 <=> x = 1 + 2 + 3 + 4
```

You can add `NUM_ADD_CONV` to the set of default conversions by

```
# extend_basic_convs("addition on nat",(`m + n:num`,NUM_ADD_CONV));;
val it : unit = ()
```

and now it happens by default:

```
# REWRITE_CONV[] `x = 1 + 2 + 3 + 4`;;
val it : thm = |- x = 1 + 2 + 3 + 4 <=> x = 10
```

## See also

`basic_convs`, `extend_basic_rewrites`, `set_basic_convs`.

# extend_basic_rewrites

```
extend_basic_rewrites : thm list -> unit
```

## Synopsis

Extend the set of default rewrites used by rewriting and simplification.

## Description

The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_` variants. A call to `extend_basic_rewrites thl` extends the former with the list of theorems `thl`, so they will thereafter happen by default.

## Failure

Never fails.

## See also

`basic_rewrites`, `extend_basic_convs`, `set_basic_rewrites`.

# extend_rectype_net

```
extend_rectype_net : string * ('a * 'b * thm) -> unit
```

## Synopsis
Extends internal store of distinctness and injectivity theorems for a new inductive type.

## Description
HOL Light maintains several data structures based on the current set of distinctness and injectivity theorems for the inductive data type so far defined. A call `extend_rectype_net ("tyname"` where `rth` is the recursion theorem for the type as returned as the second item from `define_type`, extend these structures for a new type. Two arguments are ignored just for regularity with some other internal data structures.

## Failure
Never fails, even if the theorem is malformed.

## Comments
This functoin is called automatically by `define_type`, and normally users will not need to invoke it explicitly.

## See also
`basic_rectype_net`, `define_type`, `distinctness_store`, `injectivity_store`.

# fail

```
fail : unit -> 'a
```

## Synopsis
Fail with empty string.

## Description
In HOL Light, the class of exceptions `Failure "string"` is used consistently. This makes it easy to catch all HOL-related exceptions by a `Failure _` pattern without accidentally catching others. In general, the failure can be generated by `failwith "string"`, but the special case of an empty string is bound to the function `fail`.

### Failure
Always fails.

### Uses
Useful when there is no intention to propagate helpful information about the cause of the exception, for example because you know it will be caught and handled without discrimination.

### See also

## FAIL_TAC

```
FAIL_TAC : string -> tactic
```

### Synopsis
Tactic that always fails, with the supplied string.

### Description
Whatever goal it is applied to, `FAIL_TAC "s"` always fails with `Failure "s"`.

### Failure
The application of `FAIL_TAC` to a string never fails; the resulting tactic always fails.

### Example
The following example uses the fact that if a tactic `t1` solves a goal, then the tactic `t1 THEN t2` never results in the application of `t2` to anything, because `t1` produces no subgoals. In attempting to solve the following goal:

```
# g `if x then T else T`;;
```

the tactic

```
# e(REWRITE_TAC[] THEN FAIL_TAC "Simple rewriting failed to solve goal");;
Exception: Failure "Simple rewriting failed to solve goal".
```

fails with the message provided, whereas the following quietly solves the goal:

```
# e(REWRITE_TAC[COND_ID] THEN FAIL_TAC "Using that failed to solve goal");;
val it : goalstack = No subgoals
```

### See also
`ALL_TAC`, `NO_TAC`.

# f_f_

```
f_f_ : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'b * 'd
```

## Synopsis

Non-infix version of `F_F`.

## See also

`F_F`.

# F_F

```
(F_F) : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'b * 'd
```

## Synopsis

Infix operator. Applies two functions to a pair: `(f F_F g) (x,y) = (f x, g y)`.

## Description

## Failure

Never fails.

## Example

## Uses

## Comments

## See also

`f_f_`

# file_on_path

```
file_on_path : string list -> string -> string
```

## Synopsis
Expands relative filename to first available one in path.

## Description
When given an absolute filename, (e.g. on Linux/Unix one starting with a slash or tilde), this function returns it unchanged. Otherwise it tries to find the file in one of the directories in the path argument.

## Failure
Fails if no file is found on the path.

## Example

```
# file_on_path (!load_path) "Examples/analysis.ml";;
val it : string = "/home/johnh/holl/Examples/analysis.ml"
# file_on_path (!load_path) "Examples/wibble.ml";;
Exception: Not_found.
```

## See also
`load_path`, `loads`, `loadt`, `needs`.

# filter

`filter : ('a -> bool) -> 'a list -> 'a list`

## Synopsis
Filters a list to the sublist of elements satisfying a predicate.

## Description
`filter p l` applies `p` to every element of `l`, returning a list of those that satisfy `p`, in the order they appeared in the original list.

## Failure
Fails if the predicate fails on any element.

## See also
`mapfilter`, `partition`, `remove`.

# FIND_ASSUM

```
FIND_ASSUM : thm_tactic -> term -> tactic
```

## Synopsis

Apply a theorem-tactic to the the first assumption equal to given term.

## Description

The tactic `FIND_ASSUM ttac ‘t‘` finds the first assumption whose conclusion is `t`, and applies `ttac` to it. If there is no such assumption, the call fails.

## Failure

Fails if there is no assumption the same as the given term, or if the theorem-tactic itself fails on the assumption.

## Example

Suppose we set up this goal:

```
# g ‘0 = x /\ y = 0 ==> f(x + f(y)) = f(f(f(x) * x * y))‘;;
```

and move the hypotheses into the assumption list:

```
# e STRIP_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [‘0 = x‘]
 1 [‘y = 0‘]

‘f (x + f y) = f (f (f x * x * y))‘
```

We can’t just use `ASM_REWRITE_TAC[]` to solve the goal, but we can more directly use the assumptions:

```
# e(FIND_ASSUM SUBST1_TAC `y = 0` THEN
    FIND_ASSUM (SUBST1_TAC o SYM) `0 = x`);;
val it : goalstack = 1 subgoal (1 total)

 0 [`0 = x`]
 1 [`y = 0`]

`f (0 + f 0) = f (f (f 0 * 0 * 0))`
```

after which simple rewriting solves the goal:

```
# e(REWRITE_TAC[ADD_CLAUSES; MULT_CLAUSES]);;
val it : goalstack = No subgoals
```

## Uses

Identifying an assumption to use by explicitly quoting it.

## Comments

A similar effect can be achieved by `ttac(ASSUME `t`)`. The use of `FIND_ASSUM` may be considered preferable because it immediately fails if there is no assumption `t`, whereas the `ASSUME` construct only generates a validity failure. Still, the the above example, it would have been a little briefer to write:

```
# e(REWRITE_TAC[ASSUME `y = 0`; SYM(ASSUME `0 = x`);
                ADD_CLAUSES; MULT_CLAUSES]);;
```

## See also

`ASSUME`, `VALID`.

# find

```
find : ('a -> bool) -> 'a list -> 'a
```

## Synopsis

Returns the first element of a list which satisfies a predicate.

## Description

`find p [x1;...;xn]` returns the first `xi` in the list such that `(p xi)` is `true`.

## Failure
Fails with `find` if no element satisfies the predicate. This will always be the case if the list is empty.

## See also
`tryfind`, `mem`, `exists`, `forall`, `assoc`, `rev_assoc`.

# find_path

```
find_path : (term -> bool) -> term -> string
```

## Synopsis
Returns a path to some subterm satisfying a predicate.

## Description
The call `find_path p t` traverses the term `t` top-down until it finds a subterm satisfying the predicate `p`. It then returns a path indicating how to reach it; this is just a string with each character interpreted as:

- `"b"`: take the body of an abstraction
- `"l"`: take the left (rator) path in an application
- `"r"`: take the right (rand) path in an application

## Failure
Fails if there is no subterm satisfying `p`.

## Example

```
# find_path is_list `!x. ~(x = []) ==> CONS (HD x) (TL x) = x`;;
Warning: inventing type variables
val it : string = "rblrrr"
```

## See also
`follow_path`, `PATH_CONV`.

# find_term

```
find_term : (term -> bool) -> term -> term
```

### Synopsis
Searches a term for a subterm that satisfies a given predicate.

### Description
The largest subterm, in a depth-first, left-to-right search of the given term, that satisfies the predicate is returned.

### Failure
Fails if no subterm of the given term satisfies the predicate.

### Example

```
# find_term is_var `x + y + z`;;
val it : term = `x`
```

### See also
`find_terms`.

# find_terms

```
find_terms : (term -> bool) -> term -> term list
```

### Synopsis
Searches a term for all subterms that satisfy a predicate.

### Description
A list of subterms of a given term that satisfy the predicate is returned.

### Failure
Never fails.

## Example

This is a simple example:

```
# find_terms is_var `x + y + z`;;
val it : term list = [`z`; `y`; `x`]
```

while the following shows that the terms returned may overlap or contain each other:

```
# find_terms is_comb `x + y + z`;;
val it : term list = [`(+) y`; `y + z`; `(+) x`; `x + y + z`]
```

## See also

`find_term`.

# finished

```
finished : 'a list -> int * 'a list
```

## Synopsis

Parser that checks emptiness of the input.

## Description

The function `finished` tests if its input is the empty list, and if so returns a pair of zero and that input. Otherwise it fails.

## Failure

Fails on nonempty input.

## Uses

This function is intended to check that some parsing operation has absorbed all the input.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

# FIRST_ASSUM

```
FIRST_ASSUM : thm_tactic -> tactic
```

## Synopsis

Applied theorem-tactic to first assumption possible.

## Description

The tactic

```
FIRST_ASSUM ttac ([A1; ...; An], g)
```

has the effect of applying the first tactic which can be produced by `ttac` from the assumptions `(.. |- A1)`, ..., `(.. |- An)` and which succeeds when applied to the goal. Failures of `ttac` to produce a tactic are ignored. The similar function `FIRST_X_ASSUM` is the same except that the assumption used is then removed from the goal.

## Failure

Fails if `ttac (.. |- Ai)` fails for every assumption `Ai`, or if the assumption list is empty, or if all the tactics produced by `ttac` fail when applied to the goal.

## Example

The tactic

```
FIRST_ASSUM (fun asm -> CONTR_TAC asm  ORELSE  ACCEPT_TAC asm)
```

searches the assumptions for either a contradiction or the desired conclusion. The tactic

```
FIRST_ASSUM MATCH_MP_TAC
```

searches the assumption list for an implication whose conclusion matches the goal, reducing the goal to the antecedent of the corresponding instance of this implication.

## See also

`ASSUM_LIST`, `EVERY`, `EVERY_ASSUM`, `FIRST`, `FIRST_X_ASSUM`, `MAP_EVERY`, `MAP_FIRST`.

# FIRST_CONV

```
FIRST_CONV : conv list -> conv
```

## Synopsis
Apply the first of the conversions in a given list that succeeds.

## Description
`FIRST_CONV [c1;...;cn] ‘t‘` returns the result of applying to the term `‘t‘` the first conversion `ci` that succeeds when applied to `‘t‘`. The conversions are tried in the order in which they are given in the list.

## Failure
`FIRST_CONV [c1;...;cn] ‘t‘` fails if all the conversions `c1`, ..., `cn` fail when applied to the term `‘t‘`. `FIRST_CONV cs ‘t‘` also fails if `cs` is the empty list.

## Example

```
# FIRST_CONV [NUM_ADD_CONV; NUM_MULT_CONV; NUM_EXP_CONV] ‘12 * 12‘;;
val it : thm = |- 12 * 12 = 144
```

## See also
`ORELSEC`.

# FIRST

```
FIRST : tactic list -> tactic
```

## Synopsis
Applies the first tactic in a tactic list that succeeds.

## Description
When applied to a list of tactics `[t1;...;tn]`, and a goal `g`, the tactical `FIRST` tries applying the tactics to the goal until one succeeds. If the first tactic which succeeds is `tm`,

then the effect is the same as just `tm`. Thus `FIRST` effectively behaves as follows:

```
FIRST [t1;...;tn] = t1 ORELSE ... ORELSE tn
```

### Failure

The application of `FIRST` to a tactic list never fails. The resulting tactic fails iff all the component tactics do when applied to the goal, or if the tactic list is empty.

### See also

`EVERY`, `ORELSE`.

## FIRST_TCL

```
FIRST_TCL : thm_tactical list -> thm_tactical
```

### Synopsis

Applies the first theorem-tactical in a list that succeeds.

### Description

When applied to a list of theorem-tacticals, a theorem-tactic and a theorem, `FIRST_TCL` returns the tactic resulting from the application of the first theorem-tactical to the theorem-tactic and theorem that succeeds. The effect is the same as:

```
FIRST_TCL [ttl1;...;ttln] = ttl1 ORELSE_TCL ... ORELSE_TCL ttln
```

### Failure

`FIRST_TCL` fails iff each tactic in the list fails when applied to the theorem-tactic and theorem. This is trivially the case if the list is empty.

### See also

`EVERY_TCL`, `ORELSE_TCL`, `REPEAT_TCL`, `THEN_TCL`.

## FIRST_X_ASSUM

```
FIRST_X_ASSUM : thm_tactic -> tactic
```

## Synopsis
Applied theorem-tactic to first assumption possible, extracting assumption.

## Description
The tactic

```
FIRST_X_ASSUM ttac ([A1; ...; An], g)
```

has the effect of applying the first tactic which can be produced by `ttac` from the assumptions `(.. |- A1)`, ..., `(.. |- An)` and which succeeds when applied to the goal with that assumption removed. Failures of `ttac` to produce a tactic are ignored. The similar function `FIRST_ASSUM` is the same except that the assumption used is not removed from the goal.

## Failure
Fails if `ttac (.. |- Ai)` fails for every assumption `Ai`, or if the assumption list is empty, or if all the tactics produced by `ttac` fail when applied to the goal.

## Example
The tactic

```
FIRST_X_ASSUM MATCH_MP_TAC
```

searches the assumption list for an implication whose conclusion matches the goal, removing that assumption and reducing the goal to the antecedent of the corresponding instance of this implication.

## See also
`ASSUM_LIST, EVERY, EVERY_ASSUM, FIRST, FIRST_ASSUM, MAP_EVERY, MAP_FIRST.`

# fix

```
fix : string -> ('a -> 'b) -> 'a -> 'b
```

## Synopsis
Applies parser and fails if it raises `Noparse`.

## Description
Parsers raise `Noparse` to indicate that they were not able to make any progress at all. If `p` is such a parser, `fix s p` gives a new parser where a `Noparse` exception from `p` will result in a `Failure s` exception, but is otherwise the same as `p`.

### Failure
The immediate call `fix s p` never fails, but the resulting parser may.

### Comments
This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

### See also
`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

## flat

```
flat : 'a list list -> 'a list
```

### Synopsis
Flattens a list of lists into one long list.

### Description
`flat [l1;...;ln]` returns `(l1 @ ... @ ln)` where each li is a list and `@` is list concatenation.

### Failure
Never fails.

### Example

```
# flat [[1;2];[3;4;5];[6]];;
val it : int list = [1; 2; 3; 4; 5; 6]
```

## flush_goalstack

```
flush_goalstack : unit -> unit
```

### Synopsis
Eliminate all but the current goalstate from the current goalstack.

### Description
Normally, the current goalstack has the current goalstate at the head and all previous intermediate states further back in the list. This function `flush_goalstack()` keeps just the current goalstate and eliminates all previous states.

### Failure
Fails if there is no current goalstate, i.e. if the goalstack is empty.

### See also
`b, g, r.`

## foldl

```
foldl : ('a -> 'b -> 'c -> 'a) -> 'a -> ('b, 'c) func -> 'a
```

### Synopsis
Folds an operation iteratively over the graph of a finite partial function.

### Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If a finite partial function `p` has graph `[x1,y1; ...; xn,yn]` then the application `foldl f a p` returns

```
f (f ... (f (f a x1 y1) x2 y2) ...) xn yn
```

Note that the order in which the pairs are operated on depends on the internal structure of the finite partial function, and is often not the most obvious.

### Failure
Fails if one of the embedded function applications does.

### Example
The `graph` function is implemented based on the following invocation of `foldl`, with an

additional sorting phase afterwards:

```
# let f = (1 |-> 2) (2 |=> 3);;
val f : (int, int) func = <func>

# graph f;;
val it : (int * int) list = [(1, 2); (2, 3)]

#  foldl (fun a x y -> (x,y)::a) [] f;;
val it : (int * int) list = [(1, 2); (2, 3)]
```

Note that in this case the order happened to be the same, but this is an accident.

## See also

|->, |=>, apply, applyd, choose, combine, defined, dom, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefine, undefined.

# foldr

```
foldr : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) func -> 'c -> 'c
```

## Synopsis

Folds an operation iteratively over the graph of a finite partial function.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If a finite partial function `p` has graph `[x1,y1; ...; xn,yn]` then the application `foldl f p a` returns

```
f x1 y1 (f x2 y2 (f x3 y3 (f ... (f xn yn a) ... )))
```

Note that the order in which the pairs are operated on depends on the internal structure of the finite partial function, and is often not the most obvious.

## Failure

Fails if one of the embedded function applications does.

## Example

```
# let f = (1 |-> 2) (2 |=> 3);;
val f : (int, int) func = <func>

# graph f;;
val it : (int * int) list = [(1, 2); (2, 3)]

# foldr (fun x y a -> (x,y)::a) f [];;
val it : (int * int) list = [(2, 3); (1, 2)]
```

Note how the pairs are actually processed in the opposite order to the order in which they are presented by `graph`. The order will in general not be obvious, and generally this is applied to operations with appropriate commutativity properties.

## See also

`|->`, `|=>`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# follow_path

`follow_path : string -> term -> term`

## Synopsis

Find the subterm of a given term indicated by a path.

## Description

A call `follow_path p t` follows path `p` inside `t` and returns the subterm encountered. The path is a string with the successive characters interpreted as follows:

- `"b"`: take the body of an abstraction
- `"l"`: take the left (rator) path in an application
- `"r"`: take the right (rand) path in an application

## Failure

Fails if the path is not meaningful for the term, e.g. if a `"b"` is encountered for a subterm that is not an abstraction.

## Example

```
# follow_path "rrlr" `1 + 2 + 3 + 4 + 5`;;
val it : term = `3`
```

## See also

find_path, PATH_CONV.

# forall2

```
forall2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

## Synopsis

Tests if corresponding elements of two lists all satisfy a relation.

## Description

`forall p [x1;...;xn] [y1;...;yn]` returns `true` if `(p xi yi)` is true for all corresponding `xi` and `yi` in the list. Otherwise, or if the lengths of the lists are different, it returns `false`.

## Failure

Never fails.

## Example

Here we check whether all elements of the first list are less than the corresponding element of the second:

```
# forall2 (<) [1;2;3] [2;3;4];;
val it : bool = true

# forall2 (<) [1;2;3;4] [5;4;3;2];;
val it : bool = false

# forall2 (<) [1] [2;3];;
val it : bool = false
```

## See also

exists, forall.

# forall

```
forall : ('a -> bool) -> 'a list -> bool
```

## Synopsis
Tests a list to see if all its elements satisfy a predicate.

## Description
`forall p [x1;...;xn]` returns `true` if `(p xi)` is true for all `xi` in the list. Otherwise it returns `false`. If the list is empty, this function always returns true.

## Failure
Never fails.

## Example

```
# forall (fun x -> x <= 2) [0;1;2];;
val it : bool = true
# forall (fun x -> x <= 2) [1;2;3];;
val it : bool = false
```

## See also
`exists`, `find`, `tryfind`, `mem`, `assoc`, `rev_assoc`.

# free_in

```
free_in : term -> term -> bool
```

## Synopsis
Tests if one term is free in another.

## Description
When applied to two terms `t1` and `t2`, the function `free_in` returns `true` if `t1` is free in `t2`, and `false` otherwise. It is not necessary that `t1` be simply a variable.

## Failure
Never fails.

## Example

In the following example `free_in` returns `false` because the `x` in `SUC x` in the second term is bound:

```
# free_in `SUC x` `!x. SUC x = x + 1`;;
val it : bool = false
```

whereas the following call returns `true` because the first instance of `x` in the second term is free, even though there is also a bound instance:

```
# free_in `x:bool` `x /\ (?x. x=T)`;;
val it : bool = true
```

## Comments

If the term `t1` is a variable, the rule `vfree_in` is more basic and probably more efficient.

## See also

`frees`, `freesin`, `freesl`, `thm_frees`, `vfree_in`.

# frees

```
frees : term -> term list
```

## Synopsis

Returns a list of the variables free in a term.

## Description

When applied to a term, `frees` returns a list of the free variables in that term. There are no repetitions in the list produced even if there are multiple free instances of some variables.

## Failure

Never fails.

## Example

Clearly in the following term, `x` and `y` are free, whereas `z` is bound:

```
# frees `x = 1 /\ y = 2 /\ !z. z >= 0`;;
val it : term list = [`x`; `y`]
```

## See also

`freesl`, `free_in`, `thm_frees`, `variables`.

# freesin

```
freesin : term list -> term -> bool
```

## Synopsis
Tests if all free variables of a term appear in a list.

## Description
The call `freesin l t` tests whether all free variables of `t` occur in the list `l`. The special case where `l = []` will therefore test whether `t` is closed (i.e. contains no free variables).

## Failure
Never fails.

## Example

```
# freesin [] `!x y. x + y >= 0`;;
val it : bool = true
# freesin [] `x + y >= 0`;;
val it : bool = false
# freesin [`x:num`; `y:num`; `z:num`] `x + y >= 0`;;
val it : bool = true
```

## Uses
Can be attractive to fold together some free-variable tests without explicitly constructing the set of free variables in a term.

## See also
`frees`, `freesl`, `vfree_in`.

# freesl

```
freesl : term list -> term list
```

## Synopsis
Returns a list of the free variables in a list of terms.

## Description

When applied to a list of terms, `freesl` returns a list of the variables which are free in any of those terms. There are no repetitions in the list produced even if several terms contain the same free variable.

## Failure

Never fails.

## Example

In the following example there are free instances of each of `w`, `x` and `y`, whereas the only instances of `z` are bound:

```
# freesl [`x + y = 2`; `!z. z >= x - w`];;
val it : term list = [`y`; `x`; `w`]
```

## See also

`frees`, `free_in`, `thm_frees`.

# FREEZE_THEN

```
FREEZE_THEN : thm_tactical
```

## Synopsis

'Freezes' a theorem to prevent instantiation of its free variables.

## Description

`FREEZE_THEN` expects a tactic-generating function `f:thm->tactic` and a theorem `(A1 |- w)` as arguments. The tactic-generating function `f` is applied to the theorem `(w |- w)`. If this tactic generates the subgoal:

```
    A ?- t
   =========  f (w |- w)
    A ?- t1
```

then applying `FREEZE_THEN f (A1 |- w)` to the goal `(A ?- t)` produces the subgoal:

```
    A ?- t
   =========  FREEZE_THEN f (A1 |- w)
    A ?- t1
```

Since the term `w` is a hypothesis of the argument to the function `f`, none of the free variables present in `w` may be instantiated or generalized. The hypothesis is discharged by `PROVE_HYP` upon the completion of the proof of the subgoal.

## Failure

Failures may arise from the tactic-generating function. An invalid tactic arises if the hypotheses of the theorem are not alpha-convertible to assumptions of the goal.

## Uses

Used in serious proof hacking to limit the matches achievable by rewriting etc.

## See also

`ASSUME`, `IMP_RES_TAC`, `PROVE_HYP`, `RES_TAC`, `REWR_CONV`.

# funpow

```
funpow : int -> ('a -> 'a) -> 'a -> 'a
```

## Synopsis

Iterates a function a fixed number of times.

## Description

`funpow n f x` applies `f` to `x`, `n` times, giving the result `f (f ... (f x)...)` where the number of `f`'s is `n`. `funpow 0 f x` returns `x`. If `n` is negative, it is treated as zero.

## Failure

`funpow n f x` fails if any of the `n` applications of f fail.

## Example

Apply `tl` three times to a list:

```
# funpow 3 tl [1;2;3;4;5];;
val it : int list = [4; 5]
```

Apply `tl` zero times:

```
# funpow 0 tl [1;2;3;4;5];;
val it : int list = [1; 2; 3; 4; 5]
```

Apply `tl` six times to a list of only five elements:

```
# funpow 6 tl [1;2;3;4;5];;
Exception: Failure "tl".
```

# GABS_CONV

```
GABS_CONV : conv -> term -> thm
```

## Synopsis

Applies a conversion to the body of a generalized abstraction.

## Description

If `c` is a conversion that maps a term `‘t‘` to the theorem `|- t = t’`, then the conversion `ABS_CONV c` maps generalized abstractions of the form `‘\vs. t‘` to theorems of the form:

```
|- (\vs. t) = (\vs. t’)
```

That is, `ABS_CONV c ‘\vs. t‘` applies `c` to the body of the generalized abstraction `‘\vs. t‘`. It is permissible to use it on a basic abstraction, in which case the effect is the same as `ABS_CONV`.

## Failure

Fails if applied to a term that is not a generalized abstraction (or a basic one), or if the conversion `c` fails when applied to the term `t`, or if the theorem returned has assumptions in which one of the variables in the abstraction varstruct is free.

## Example

```
# GABS_CONV SYM_CONV `\(x,y,z). x + y + z = 7`;;
val it : thm = |- (\(x,y,z). x + y + z = 7) = (\(x,y,z). 7 = x + y + z)
```

## See also
ABS_CONV, RAND_CONV, RATOR_CONV, SUB_CONV.

# gcd

```
gcd : int -> int -> int
```

## Synopsis
Computes greatest common divisor of two integers.

## Description
The call `gcd m n` for two integers `m` and `n` returns the (nonnegative) greatest common divisor of `m` and `n`. If `m` and `n` are both zero, it returns zero.

## Failure
Never fails.

## Example

```
# gcd 10 12;;
val it : int = 2

# gcd 11 27;;
val it : int = 1

# gcd (-33) 76;;
val it : int = 1

# gcd 0 99;;
val it : int = 99

# gcd 0 0;;
val it : int = 0
```

## See also
gcd_num, lcm_num.

# gcd_num

`gcd_num : num -> num -> num`

## Synopsis

Computes greatest common divisor of two unlimited-precision integers.

## Description

The call `gcd_num m n` for two unlimited-precision (type `num`) integers `m` and `n` returns the (positive) greatest common divisor of `m` and `n`. If both `m` and `n` are zero, it returns zero.

## Failure

Fails if either number is not an integer (the type `num` supports arbitrary rationals).

## Example

```
# gcd_num (Int 35) (Int(-77));;
val it : num = 7

# gcd_num (Int 11) (Int 0);;
val it : num = 11

# gcd_num (Int 22 // Int 7) (Int 2);;
Exception: Failure "big_int_of_ratio".
```

## See also

`gcd`, `lcm_num`.

# g

`g : term -> goalstack`

## Synopsis

Initializes the subgoal package with a new goal which has no assumptions.

## Description

The call

```
g `tm`
```

is equivalent to

```
set_goal([],`tm`)
```

and clearly more convenient if a goal has no assumptions. For a description of the subgoal package, see `set_goal`.

## Failure

Fails unless the argument term has type `bool`.

## Example

```
# g `HD[1;2;3] = 1 /\ TL[1;2;3] = [2;3]`;;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3]`
```

## See also

`b, e, p, r, set_goal, top_goal, top_thm.`

# GEN_ALL

```
GEN_ALL : thm -> thm
```

## Synopsis

Generalizes the conclusion of a theorem over its own free variables.

## Description

When applied to a theorem `A |- t`, the inference rule `GEN_ALL` returns the theorem `A |- !x1...xn. t`, where the `xi` are all the variables, if any, which are free in `t` but not in the assumptions.

```
        A |- t
  ------------------  GEN_ALL
   A |- !x1...xn. t
```

## Failure

Never fails.

### Example

```
# let th = ARITH_RULE `x < y ==> 2 * x + y + 1 < 3 * y`;;
val th : thm = |- x < y ==> 2 * x + y + 1 < 3 * y

# GEN_ALL th;;
val it : thm = |- !x y. x < y ==> 2 * x + y + 1 < 3 * y
```

### See also
GEN, GENL, GEN_ALL, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

## GEN_ALPHA_CONV

```
GEN_ALPHA_CONV : term -> term -> thm
```

### Synopsis
Renames the bound variable of an abstraction or binder.

### Description
The conversion `GEN_ALPHA_CONV` provides alpha conversion for lambda abstractions of the form `` `\x. t` ``, as well as other terms of the form `` `b (\x. t)` `` such as quantifiers and other binders. (Note that whether `b` is a constant or parses as a binder is irrelevant, though this is usually the case in applications.) The call `` GEN_ALPHA_CONV `y` `\x. t` `` returns

```
|- (\x. t)  = (\y. t[y/x])
```

while `` GEN_ALPHA_CONV `y` `b (\x. t)` `` returns

```
|- b (\x. t)  = b (\y. t[y/x])
```

### Failure
`` GEN_ALPHA_CONV `y` tm `` fails if `y` is not a variable, or if `tm` does not have one of the forms `` `\x. t` `` or `` `b (\x. t)` ``, or if the types of `x` and `y` differ, or if `y` is already free in the body `t`.

### See also
alpha, ALPHA, ALPHA_CONV.

# GEN_BETA_CONV

```
GEN_BETA_CONV : term -> thm
```

## Synopsis
Beta-reduces general beta-redexes (e.g. paired ones).

## Description
The conversion `GEN_BETA_CONV` will perform beta-reduction of simple beta-redexes in the manner of `BETA_CONV`, or of generalized beta-redexes such as paired redexes.

## Failure
`GEN_BETA_CONV tm` fails if `tm` is neither a simple nor a tupled beta-redex.

## Example
The following examples show the action of `GEN_BETA_CONV` on tupled redexes and others:

```
# GEN_BETA_CONV `(\x. x + 1) 2`;;
val it : thm = |- (\x. x + 1) 2 = 2 + 1

# GEN_BETA_CONV `(\(x,y,z). x + y + z) (1,2,3)`;;
val it : thm = |- (\(x,y,z). x + y + z) (1,2,3) = 1 + 2 + 3

# GEN_BETA_CONV `(\[a;b;c]. b) [1;2;3]`;;
val it : thm = |- (\[a; b; c]. b) [1; 2; 3] = 2
```

However, it will fail if there is a mismatch between the varstruct and the argument, or if it is unable to make sense of the generalized abstraction:

```
# GEN_BETA_CONV `(\(SUC n). n) 3`;;
Exception: Failure "term_pmatch".

# GEN_BETA_CONV `(\(x,y,z). x + y + z) (1,x)`;;
Exception: Failure "dest_comb: not a combination".
```

## See also
`BETA_CONV`, `PAIRED_BETA_CONV`.

# GEN

```
GEN : term -> thm -> thm
```

## Synopsis
Generalizes the conclusion of a theorem.

## Description
When applied to a term `x` and a theorem `A |- t`, the inference rule `GEN` returns the theorem `A |- !x. t`, provided `x` is a variable not free in any of the assumptions. There is no compulsion that `x` should be free in `t`.

```
      A |- t
   ------------  GEN `x`                [where x is not free in A]
    A |- !x. t
```

## Failure
Fails if `x` is not a variable, or if it is free in any of the assumptions.

## Example
This is a basic example:

```
  # GEN `x:num` (REFL `x:num`);;
  val it : thm = |- !x. x = x
```

while the following example shows how the above side-condition prevents the derivation of the theorem `x <=> T |- !x. x <=> T`, which is invalid.

```
  # let t = ASSUME `x <=> T`;;
  val t : thm = x <=> T |- x <=> T

  # GEN `x:bool` t;;
  Exception: Failure "GEN".
```

## See also
`GENL`, `GEN_ALL`, `GEN_TAC`, `SPEC`, `SPECL`, `SPEC_ALL`, `SPEC_TAC`.

# GENERAL_REWRITE_CONV

```
GENERAL_REWRITE_CONV : bool -> (conv -> conv) -> gconv net -> thm list -> conv
```

## Synopsis
Rewrite with theorems as well as an existing net.

## Description

The call `GENERAL_REWRITE_CONV b cnvl net thl` will regard `thl` as rewrite rules, and if `b = true`, also potentially as conditional rewrite rules. These extra rules will be incorporated into the existing `net`, and rewriting applied with a search strategy `cnvl` (e.g. `DEPTH_CONV`).

## Comments

This is mostly for internal use, but it can sometimes be more efficient when rewriting with large sets of theorems repeatedly if they are first composed into a net and then augmented like this.

## See also

GEN_REWRITE_CONV, REWRITES_CONV.

# GENL

```
GENL : term list -> thm -> thm
```

## Synopsis

Generalizes zero or more variables in the conclusion of a theorem.

## Description

When applied to a term list `[x1;...;xn]` and a theorem `A |- t`, the inference rule `GENL` returns the theorem `A |- !x1...xn. t`, provided none of the variables `xi` are free in any of the assumptions. It is not necessary that any or all of the `xi` should be free in `t`.

```
         A |- t
   ------------------  GENL `[x1;...;xn]`       [where no xi is free in A]
    A |- !x1...xn. t
```

## Failure

Fails unless all the terms in the list are variables, none of which are free in the assumption list.

### Example

```
# SPEC `m + p:num` ADD_SYM;;
val it : thm = |- !n. (m + p) + n = n + m + p

# GENL [`m:num`; `p:num`] it;;
val it : thm = |- !m p n. (m + p) + n = n + m + p
```

### See also

GEN, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

## GEN_MESON_TAC

```
GEN_MESON_TAC : int -> int -> int -> thm list -> tactic
```

### Synopsis

First-order proof search with specified search limits and increment.

### Description

This is a slight generalization of the usual tactics for first-order proof search. Normally MESON, MESON_TAC and ASM_MESON_TAC explore the search space by successively increasing a size limit from 0, increasing it by 1 per step and giving up when a depth of 50 is reached. The more general tactic GEN_MESON_TAC allows the user to specify the starting, finishing and stepping value, but is otherwise identical to ASM_MESON_TAC. In fact, that is defined as:

```
# let ASM_MESON_TAC = GEN_MESON_TAC 0 50 1;;
```

### Failure

If the goal is unprovable, GEN_MESON_TAC will fail, though not necessarily in a feasible amount of time.

### Uses

Normally, the defaults built into MESON_TAC and ASM_MESON_TAC are reasonably effective. However, very occasionally a goal exhibits a small search space yet still requires a deep proof, so you may find GEN_MESON_TAC with a larger "maximum" value than 50 useful. Another potential use is to start the search at a depth that you know will succeed, to reduce the search time when a proof is re-run. However, the inconvenience of doing this is seldom repaid by a dramatic improvement in performance, since exploration is normally at least exponential with the size bound.

## See also

ASM_MESON_TAC, MESON, MESON_TAC.

# GEN_NNF_CONV

```
GEN_NNF_CONV : bool -> conv * (term -> thm * thm) -> conv
```

## Synopsis

General NNF (negation normal form) conversion.

## Description

The function `GEN_NNF_CONV` is a highly general conversion for putting a term in 'negation normal form' (NNF). This means that other propositional connectives are eliminated in favour of conjunction ('/\'), disjunction ('\/') and negation ('~'), and the negations are pushed down to the level of atomic formulas, also through universal and existential quantifiers, with double negations eliminated.

This function is very general. The first, boolean, argument determines how logical equivalences '`p <=> q`' are split. If the flag is `true`, toplevel equivalences are split "conjunctively" into '`(p \/ ~q) /\ (~p \/ q)`', while if it is false they are split "disjunctively" into '`(p /\ q) \/ (~p /\ ~q)`'. At subformulas, the effect is modified appropriately in order to make the resulting formula simpler in conjunctive normal form (if the flag is true) or disjunctive normal form (if the flag is false).

The second argument has two components. The first is a conversion to apply to literals, that is atomic formulas or their negations. The second is a slightly more elaborate variant of the same thing, taking an atomic formula `p` and returning desired equivalences for both `p` and `~p` in a pair. This interface avoids multiple recomputations in terms involving many nested logical equivalences, where otherwise the core conversion would be called several times.

## Failure

Never fails but may have no effect.

## Comments

The simple functions like `NNF_CONV` should be adequate most of the time, with this somewhat intricate interface being reserved for special situations.

## See also
NNF_CONV, NNFC_CONV.

# GEN_REAL_ARITH

```
GEN_REAL_ARITH : ((thm list * thm list * thm list -> positivstellensatz -> thm) -> thm list * t
```

## Synopsis
Initial normalization and proof reconstruction wrapper for real decision procedure.

## Description
The function `GEN_REAL_ARITH` takes two arguments, the first of which is an underlying 'prover', and the second a term to prove. This function is mainly intended for internal use: the function `REAL_ARITH` is essentially implemented as

```
GEN_REAL_ARITH REAL_LINEAR_PROVER
```

The wrapper `GEN_REAL_ARITH` performs various initial normalizations, such as eliminating `max`, `min` and `abs`, and passes to the prover a proof reconstruction function, say `reconstr`, and a triple of theorem lists to refute. The theorem lists are respectively a list of equations of the form `A_i |- p_i = &0`, a list of non-strict inequalities of the form `B_j |- q_i >= &0`, and a list of strict inequalities of the form `C_k |- r_k > &0`, with both sides being real in each case. The underlying prover merely needs to find a "Positivstellensatz" refutation, and pass the triple of theorems actually used and the Positivstellensatz refutation back to the reconstruction function `reconstr`. A Positivstellensatz refutation is essentially a representation of how to add and multiply equalities or inequalities chosen from the list to reach a trivially false equation or inequality such as `&0 > &0`. Note that the underlying prover may choose to augment the list of inequalities before proceeding with the proof, e.g. `REAL_LINEAR_PROVER` adds theorems `|- &0 <= &n` for relevant numeral terms `&n`. This is why the interface passes in a reconstruction function rather than simply expecting a Positivstellensatz refutation back.

## Failure
Never fails at this stage, though it may fail when subsequently applied to a term.

## Example
As noted, the built-in decision procedure `REAL_ARITH` is a simple application. See also the file `Examples/sos.ml`, where a more sophisticated nonlinear prover is plugged into `GEN_REAL_ARITH` in place of `REAL_LINEAR_PROVER`.

## Comments

Mainly intended for experts.

## See also

REAL_ARITH, REAL_LINEAR_PROVER, REAL_POLY_CONV.

# GEN_REWRITE_CONV

```
GEN_REWRITE_CONV : (conv -> conv) -> thm list -> conv
```

## Synopsis

Rewrites a term, selecting terms according to a user-specified strategy.

## Description

Rewriting in HOL is based on the use of equational theorems as left-to-right replacements on the subterms of an object theorem. This replacement is mediated by the use of `REWR_CONV`, which finds matches between left-hand sides of given equations in a term and applies the substitution.

Equations used in rewriting are obtained from the theorem lists given as arguments to the function. These are at first transformed into a form suitable for rewriting. Conjunctions are separated into individual rewrites. Theorems with conclusions of the form `` `~t` `` are transformed into the corresponding equations `` `t = F` ``. Theorems `` `t` `` which are not equations are cast as equations of form `` `t = T` ``.

If a theorem is used to rewrite a term, its assumptions are added to the assumptions of the returned theorem. The matching involved uses variable instantiation. Thus, all free variables are generalized, and terms are instantiated before substitution. Theorems may have universally quantified variables.

The theorems with which rewriting is done are divided into two groups, to facilitate implementing other rewriting tools. However, they are considered in an order-independent fashion. (That is, the ordering is an implementation detail which is not specified.)

The search strategy for finding matching subterms is the first argument to the rule. Matching and substitution may occur at any level of the term, according to the specified search strategy: the whole term, or starting from any subterm. The search strategy also specifies the depth of the search: recursively up to an arbitrary depth until no matches occur, once over the selected subterm, or any more complex scheme.

## Failure

`GEN_REWRITE_CONV` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used.

## Uses

This conversion is used in the system to implement all other rewritings conversions, and may provide a user with a method to fine-tune rewriting of terms.

## Example

Suppose we have a term of the form:

```
`(1 + 2) + 3 = (3 + 1) + 2`
```

and we would like to rewrite the left-hand side with the theorem `ADD_SYM` without changing the right hand side. This can be done by using:

```
GEN_REWRITE_CONV (RATOR_CONV o ONCE_DEPTH_CONV) [ADD_SYM] mythm
```

Other rules, such as `ONCE_REWRITE_CONV`, would match and substitute on both sides, which would not be the desirable result.

## See also

`ONCE_REWRITE_CONV`, `PURE_REWRITE_CONV`, `REWR_CONV`, `REWRITE_CONV`.

# GEN_REWRITE_RULE

```
GEN_REWRITE_RULE : (conv -> conv) -> thm list -> thm -> thm
```

## Synopsis

Rewrites a theorem, selecting terms according to a user-specified strategy.

## Description

Rewriting in HOL is based on the use of equational theorems as left-to-right replacements on the subterms of an object theorem. This replacement is mediated by the use of `REWR_CONV`, which finds matches between left-hand sides of given equations in a term and applies the substitution.

Equations used in rewriting are obtained from the theorem lists given as arguments to the function. These are at first transformed into a form suitable for rewriting. Conjunctions are separated into individual rewrites. Theorems with conclusions of the form `` `~t` `` are transformed into the corresponding equations `` `t = F` ``. Theorems `` `t` `` which are not equations are cast as equations of form `` `t = T` ``.

If a theorem is used to rewrite the object theorem, its assumptions are added to the assumptions of the returned theorem, unless they are alpha-convertible to existing assumptions. The matching involved uses variable instantiation. Thus, all free variables are generalized, and terms are instantiated before substitution. Theorems may have universally quantified variables.

The theorems with which rewriting is done are divided into two groups, to facilitate implementing other rewriting tools. However, they are considered in an order-independent fashion. (That is, the ordering is an implementation detail which is not specified.)

The search strategy for finding matching subterms is the first argument to the rule. Matching and substitution may occur at any level of the term, according to the specified search strategy: the whole term, or starting from any subterm. The search strategy also specifies the depth of the search: recursively up to an arbitrary depth until no matches occur, once over the selected subterm, or any more complex scheme.

### Failure

`GEN_REWRITE_RULE` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used.

### Uses

This rule is used in the system to implement all other rewriting rules, and may provide a user with a method to fine-tune rewriting of theorems.

### Example

Suppose we have a theorem of the form:

```
thm = |- (1 + 2) + 3 = (3 + 1) + 2
```

and we would like to rewrite the left-hand side with the theorem `ADD_SYM` without changing the right hand side. This can be done by using:

```
GEN_REWRITE_RULE (RATOR_CONV o ONCE_DEPTH_CONV) []  [ADD_SYM] mythm
```

Other rules, such as `ONCE_REWRITE_RULE`, would match and substitute on both sides, which would not be the desirable result.

### See also

`ASM_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWR_CONV`, `REWRITE_RULE`.

## GEN_REWRITE_TAC

```
GEN_REWRITE_TAC : (conv -> conv) -> thm list -> tactic
```

## Synopsis
Rewrites a goal, selecting terms according to a user-specified strategy.

## Description
Distinct rewriting tactics differ in the search strategies used in finding subterms on which to apply substitutions, and the built-in theorems used in rewriting. In the case of `REWRITE_TAC`, this is a recursive traversal starting from the body of the goal's conclusion part, while in the case of `ONCE_REWRITE_TAC`, for example, the search stops as soon as a term on which a substitution is possible is found. `GEN_REWRITE_TAC` allows a user to specify a more complex strategy for rewriting.

The basis of pattern-matching for rewriting is the notion of conversions, through the application of `REWR_CONV`. Conversions are rules for mapping terms with theorems equating the given terms to other semantically equivalent ones.

When attempting to rewrite subterms recursively, the use of conversions (and therefore rewrites) can be automated further by using functions which take a conversion and search for instances at which they are applicable. Examples of these functions are `ONCE_DEPTH_CONV` and `RAND_CONV`. The first argument to `GEN_REWRITE_TAC` is such a function, which specifies a search strategy; i.e. it specifies how subterms (on which substitutions are allowed) should be searched for.

The second and third arguments are lists of theorems used for rewriting. The order in which these are used is not specified. The theorems need not be in equational form: negated terms, say `"~ t"`, are transformed into the equivalent equational form `"t = F"`, while other non-equational theorems with conclusion of form `"t"` are cast as the corresponding equations `"t = T"`. Conjunctions are separated into the individual components, which are used as distinct rewrites.

## Failure
`GEN_REWRITE_TAC` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used. The resulting tactic is invalid when a theorem which matches the goal (and which is thus used for rewriting it with) has a hypothesis which is not alpha-convertible to any of the assumptions of the goal. Applying such an invalid tactic may result in a proof of a theorem which does not correspond to the original goal.

## Uses
Detailed control of rewriting strategy, allowing a user to specify a search strategy.

## Example
Given a goal such as:

```
?- a - (b + c) = a - (c + b)
```

we may want to rewrite only one side of it with a theorem, say `ADD_SYM`. Rewriting tactics which operate recursively result in divergence; the tactic ONCE_REWRITE_TAC [ADD_SYM] rewrites on both sides to produce the following goal:

```
?- a - (c + b) = a - (b + c)
```

as ADD_SYM matches at two positions. To rewrite on only one side of the equation, the following tactic can be used:

```
GEN_REWRITE_TAC (RAND_CONV o ONCE_DEPTH_CONV) [ADD_SYM]
```

which produces the desired goal:

```
?- a - (c + b) = a - (c + b)
```

## See also

ASM_REWRITE_TAC, GEN_REWRITE_RULE, ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWR_CONV, REWRITE_TAC,

# GEN_SIMPLIFY_CONV

```
GEN_SIMPLIFY_CONV : strategy -> simpset -> int -> thm list -> conv
```

## Synopsis

General simplification with given strategy and simpset and theorems.

## Description

The call GEN_SIMPLIFY_CONV strat ss n thl incorporates the rewrites and conditional rewrites derived from thl into the simpset ss, then simplifies using that simpset, controlling the traversal of the term by strat, and starting at level n.

## Failure

Never fails unless some component is malformed.

## See also

GEN_REWRITE_CONV, ONCE_SIMPLIFY_CONV, SIMPLIFY_CONV, SIMP_CONV, SIMP_RULE, SIMP_TAC.

# GEN_TAC

```
GEN_TAC : tactic
```

### Synopsis
Strips the outermost universal quantifier from the conclusion of a goal.

### Description
When applied to a goal `A ?- !x. t`, the tactic `GEN_TAC` reduces it to `A ?- t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
     A ?- !x. t
   ==============  GEN_TAC
    A ?- t[x'/x]
```

### Failure
Fails unless the goal's conclusion is universally quantified.

### Uses
The tactic `REPEAT GEN_TAC` strips away any universal quantifiers, and is commonly used before tactics relying on the underlying term structure.

### See also
`GEN`, `GENL`, `GEN_ALL`, `SPEC`, `SPECL`, `SPEC_ALL`, `SPEC_TAC`, `STRIP_TAC`, `X_GEN_TAC`.

## genvar

```
genvar : hol_type -> term
```

### Synopsis
Returns a 'fresh' variable with specified type.

### Description
When given a type, `genvar` returns a variable of that type whose name has not previously been produced by `genvar`.

### Failure
Never fails.

## Example

The following indicates the typical stylized form of the names (this should not be relied on, of course):

```
# genvar `:bool`;;
val it : term = `_56799`
```

There is no guard against users' own variables clashing, but if the user avoids names in the same lexical style, that can be guaranteed.

## Uses

The unique variables are useful in writing derived rules, for specializing terms without having to worry about such things as free variable capture. If the names are to be visible to a typical user, the function `variant` can provide rather more meaningful names.

## See also

`variant`.

# >=?

```
(>=?) : 'a -> 'a -> bool
```

## Synopsis

Reflexive short-cutting inequality test.

## Description

This is functionally identical to the OCaml polymorphic inequality test `>=` except that it is total (hence reflexive) even on floating-point NaNs. More importantly, it will more efficiently short-cut comparisons of large data structures where subcomponents are identical (pointer equivalent).

## Failure

May fail when applied to functions.

## Example

```
# let x = 0.0 /. 0.0;;
val x : float = nan
# x >= x;;
val it : bool = false
# x >=? x;;
val it : bool = true
```

## See also

=?, <=, <?, >?.

# get_const_type

```
get_const_type : string -> hol_type
```

## Synopsis

Gets the generic type of a constant from the name of the constant.

## Description

`get_const_type "c"` returns the generic type of `‘c‘`, if `‘c‘` is a constant.

## Failure

`get_const_type st` fails if `st` is not the name of a constant.

## Example

```
  # get_const_type "COND";;
  val it : hol_type = ‘:bool->A->A->A‘
```

## See also

dest_const, is_const.

# get_infix_status

```
get_infix_status : string -> int * string
```

## Synopsis
Get the precedence and associativity of an infix operator.

## Description
Certain identifiers are treated as infix operators with a given precedence and associativity (left or right). The call `get_infix_status "op"` looks up `op` in this list and returns a pair consisting of its precedence and its associativity; the latteris one of the strings `"left"` or `"right"`.

## Failure
Fails if the given string does not have infix status.

## Example

```
# get_infix_status "/";;
val it : int * string = (22, "left")
# get_infix_status "UNION";;
val it : int * string = (16, "right")
```

## See also
`infixes`, `parse_as_infix`, `unparse_as_infix`.

# get_type_arity

```
get_type_arity : string -> int
```

## Synopsis
Returns the arity of a type constructor.

## Description
When applied to the name of a type constructor, `arity` returns its arity, i.e. how many types it is supposed to be applied to. Base types like `:bool` are regarded as constructors with zero arity.

## Failure
Fails if there is no type constructor of that name.

## Example

```
# get_type_arity "bool";;
val it : int = 0

# get_type_arity "fun";;
val it : int = 2

# get_type_arity "nocon";;
Exception: Failure "find".
```

## See also

`new_type`, `new_type_definition`, `types`.

# graph

`graph : ('a, 'b) func -> ('a * 'b) list`

## Synopsis

Returns the graph of a finite partial function.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The `graph` function takes a finite partial function that maps `x1` to `y1`, ..., `xn` to `yn` and returns its graph as a set/list: `[x1,y1; ...; xn,yn]`.

## Failure

Attempts to sort the resulting list, so may fail if the type of the pairs does not permit comparison.

## Example

```
# graph undefined;;
val it : ('a * 'b) list = []
# graph (1 |=> 2);;
val it : (int * int) list = [(1, 2)]
```

## See also

`|->`, `|=>`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# GSYM

```
GSYM : thm -> thm
```

## Synopsis

Reverses the first equation(s) encountered in a top-down search.

## Description

The inference rule `GSYM` reverses the first equation(s) encountered in a top-down search of the conclusion of the argument theorem. An equation will be reversed iff it is not a proper subterm of another equation. If a theorem contains no equations, it will be returned unchanged.

```
    A |- ..(s1 = s2)...(t1 = t2)..
   --------------------------------  GSYM
    A |- ..(s2 = s1)...(t2 = t1)..
```

## Failure

Never fails, and never loops infinitely.

## Example

```
# ADD;;
val it : thm = |- (!n. 0 + n = n) /\ (!m n. SUC m + n = SUC (m + n))

# GSYM ADD;;
val it : thm = |- (!n. n = 0 + n) /\ (!m n. SUC (m + n) = SUC m + n)
```

## See also

NOT_EQ_SYM, REFL, SYM.

# >?

```
(>?) : 'a -> 'a -> bool
```

## Synopsis

Reflexive short-cutting inequality test.

## Description

This is functionally identical to the OCaml polymorphic inequality test `>` except that it is total even on floating-point NaNs. More importantly, it will more efficiently short-cut comparisons of large data structures where subcomponents are identical (pointer equivalent).

## Failure

May fail when applied to functions.

## Example

```
# 1.0 > nan or nan > 1.0;;
val it : bool = false
# 1.0 >? nan;;
val it : bool = true
# nan >? 1.0;;
val it : bool = false
```

## See also

`=?`, `<?`, `<=?`, `>=?`.

# HAS_SIZE_CONV

```
HAS_SIZE_CONV : term -> thm
```

## Synopsis

Converts statement about set's size into existential enumeration.

## Description

Given a term of the form `` `s HAS_SIZE n` `` for a numeral `n`, the conversion `HAS_SIZE_CONV` returns an equivalent form postulating the existence of `n` pairwise distinct elements that make up the set.

## Failure

Fails if applied to a term of the wrong form.

## Example

```
# HAS_SIZE_CONV `s HAS_SIZE 1`;;
...
val it : thm = |- s HAS_SIZE 1 <=> (?a. s = {a})

# HAS_SIZE_CONV `t HAS_SIZE 3`;;
...
val it : thm =
  |- t HAS_SIZE 3 <=>
     (?a a' a''. ~(a' = a'') /\ ~(a = a') /\ ~(a = a'') /\ t = {a, a', a''})
```

# hd

`hd : 'a list -> 'a`

## Synopsis
Computes the first element (the head) of a list.

## Description
`hd [x1;...;xn]` returns `x1`.

## Failure
Fails with `hd` if the list is empty.

## See also
`tl`, `el`.

# help

`help : string -> unit`

## Synopsis
Displays help on a given identifier in the system.

## Description
A call `help "s"` will attempt to display the help file associated with a particular identifier `s` in the system. If there is no entry for identifier `s`, the call responds instead with some

possibly helpful suggestions as to what you might have meant, based on a simple 'edit distance' criterion.

Help files are stored in the `Help` subdirectory of HOL Light. Normally the help file for an identifier `name` would be called `name.doc`, but there are a few exceptions, because some identifiers have characters that cannot be put in filenames and some platforms like Cygwin have inadequate case sensitivity.

## Failure

Never fails.

## Example

Here is a successful call:

```
# help "lhs";;
-------------------------------------------------------------------

lhs : term -> term

SYNOPSIS

Returns the left-hand side of an equation.

DESCRIPTION

lhs `t1 = t2` returns `t1`.

FAILURE CONDITIONS

Fails with lhs if the term is not an equation.

EXAMPLES


  # lhs `2 + 2 = 4`;;
  val it : term = `2 + 2`


SEE ALSO
dest_eq, lhand, rand, rhs.

-------------------------------------------------------------------
val it : unit = ()
```

and here is one for a non-existent identifier:

```
# help "IMP_TAC";;
-------------------------------------------------------------------
No help found for "IMP_TAC"; did you mean:

help "SIMP_TAC";;
help "MP_TAC";;
help "IMP_TRANS";;

?
--------------------------------------------------------------------
```

## See also

hol_version.

# hide_constant

```
hide_constant : string -> unit
```

## Synopsis
Stops the quotation parser from recognizing a constant.

## Description
A call `hide_constant "c"` where `c` is the name of a constant, will prevent the quotation parser from parsing it as such; it will just be parsed as a variable. The effect can be reversed by `unhide_constant "c"`.

## Failure
Fails if the given name is not a constant of the current theory, or if the named constant is already hidden.

## Comments
The hiding of a constant only affects the quotation parser; the constant is still there in a theory, and may not be redefined.

## See also
`unhide_constant`.

# HIGHER_REWRITE_CONV

```
HIGHER_REWRITE_CONV : thm list -> bool -> term -> thm
```

## Synopsis
Rewrite once using more general higher order matching.

## Description
The call `HIGHER_REWRITE_CONV [th1;...;thn] flag t` will find a higher-order match for the whole term `t` against one of the left-hand sides of the equational theorems in the list `[th1;...;thn]`. Each such theorem should be of the form `|- P pat <=> t` where `f` is a variable. A free subterm `pat'` of `t` will be found that matches (in the usual restricted higher-order sense) the pattern `pat`. If the `flag` argument is true, this will be some topmost matchable term, while if it is false, some innermost matchable term will be selected. The

rewrite is then applied by instantiating `P` to a lambda-term reflecting how `t` is built up from `pat'`, and beta-reducing as in normal higher-order matching. However, this process is more general than HOL Light's normal higher-order matching (as in `REWRITE_CONV` etc., with core behaviour inherited from `PART_MATCH`), because `pat'` need not be uniquely determined by bound variable correspondences.

## Failure

Fails if no match is found.

## Example

The theorem `COND_ELIM_THM` can be applied to eliminate conditionals:

```
# COND_ELIM_THM;;
val it : thm = |- P (if c then x else y) <=> (c ==> P x) /\ (~c ==> P y)
```

in a term like this:

```
# let t = `z = if x = 0 then if y = 0 then 0 else x + y else x + y`;;
val t : term = `z = (if x = 0 then if y = 0 then 0 else x + y else x + y)`
```

either outermost first:

```
# HIGHER_REWRITE_CONV[COND_ELIM_THM] true t;;
val it : thm =
  |- z = (if x = 0 then if y = 0 then 0 else x + y else x + y) <=>
     (x = 0 ==> z = (if y = 0 then 0 else x + y)) /\ (~(x = 0) ==> z = x + y)
```

or innermost first:

```
# HIGHER_REWRITE_CONV[COND_ELIM_THM] false t;;
val it : thm =
  |- z = (if x = 0 then if y = 0 then 0 else x + y else x + y) <=>
     (y = 0 ==> z = (if x = 0 then 0 else x + y)) /\
     (~(y = 0) ==> z = (if x = 0 then x + y else x + y))
```

## Uses

Applying general simplification patterns without manual instantiation.

## See also

`PART_MATCH`, `REWRITE_CONV`.

# hol_dir

```
hol_dir : string ref
```

## Synopsis
Base directory in which HOL Light is installed.

## Description
This reference variable holds the directory (folder) for the base of the HOL Light distribution. This information is used, for example, when loading files with `loads`. Normally set to the current directory when HOL Light is loaded or built, but picked up from the system variable `HOLDIR` if it is defined.

## Failure
Not applicable.

## Example
On my laptop, the value is:

```
# !hol_dir;;
val it : string = "/home/johnh/holl"
```

## Uses
Ensuring that HOL Light can find any libraries or other system files needed to support proofs.

## See also
`load_path`, `loads`.

# hol_version

```
hol_version : string
```

## Synopsis
A string indicating the version of HOL Light.

## Description
This string is a numeric version number for HOL Light.

## Failure
Not applicable.

## Example

On my laptop, the value is:

```
# hol_version;;
val it : string = "2.10"
```

## See also

startup_banner.

# hyp

```
hyp : thm -> term list
```

## Synopsis

Returns the hypotheses of a theorem.

## Description

When applied to a theorem `A |- t`, the function `hyp` returns `A`, the list of hypotheses of the theorem.

## Failure

Never fails.

## Example

```
# let th = ADD_ASSUM `x = 1` (ASSUME `y = 2`);;
val th : thm = y = 2, x = 1 |- y = 2

# hyp th;;
val it : term list = [`y = 2`; `x = 1`]
```

## See also

dest_thm, concl.

# ideal_cofactors

```
ideal_cofactors : (term -> num) * (num -> term) * conv * term * term * term * term * ter
```

### Synopsis
Generic procedure to compute cofactors for ideal membership.

### Description
The `ideal_cofactors` function takes first the same set of arguments as `RING`, defining a suitable ring for it to operate over. (See the entry for `RING` for details.) It then yields a function that given a list of terms `[p1; ...; pn]` and another term `p`, all of which have the right type to be considered as polynomials over the ring, attempts to find a corresponding set of 'cofactors' `[q1; ...; qn]` such that the following is an algebraic ring identity:

```
p = p1 * q1 + ... + pn * qn
```

That is, it provides a concrete certificate for the fact that `p` is in the ideal generated by the `p1,...,pn`. If `p` is not in this ideal, the function will fail.

### Failure
Fails if the 'polynomials' are of the wrong type, or if ideal membership does not hold.

### Example
For an example of the real-number instantiation in action, see `real_ideal_cofactors`.

### See also
`real_ideal_cofactors`, `RING`, `RING_AND_IDEAL_CONV`.

## I

```
I : 'a -> 'a
```

### Synopsis
Performs identity operation: `I x = x`.

### Failure
Never fails.

### See also
`C`, `K`, `F_F`, `o`, `W`.

## ignore_constant_varstruct

```
ignore_constant_varstruct : bool ref
```

## Synopsis

Interpret a simple varstruct as a variable, even if there is a constant of that name.

## Description

As well as conventional abstractions `‘\x. t‘` where `x` is a variable, HOL Light permits generalized abstractions where the varstruct is a more complex term, e.g. `‘\(x,y). x + y‘`. This includes the degenerate case of just a constant. However, one may want a regular abstraction whose bound variable happens to be in use as a constant. When parsing a quotation `"\c. t"` where `c` is the name of a constant, HOL Light interprets it as a simple abstraction with a variable `c` when the flag `ignore_constant_varstruct` is `true`, as it is by default. It will interpret it as a degenerate generalized abstraction, only useful when applied to the constant `c`, if the flag is `false`.

## Failure

Not applicable.

## See also

`GEN_BETA_CONV, is_abs, is_gabs.`

# IMP_ANTISYM_RULE

```
IMP_ANTISYM_RULE : thm -> thm -> thm
```

## Synopsis

Deduces equality of boolean terms from forward and backward implications.

## Description

When applied to the theorems `A1 |- t1 ==> t2` and `A2 |- t2 ==> t1`, the inference rule `IMP_ANTISYM_RULE` returns the theorem `A1 u A2 |- t1 <=> t2`.

```
   A1 |- t1 ==> t2     A2 |- t2 ==> t1
  -------------------------------------  IMP_ANTISYM_RULE
          A1 u A2 |- t1 <=> t2
```

## Failure

Fails unless the theorems supplied are a complementary implicative pair as indicated above.

## Example

```
# let th1 = TAUT `p /\ q ==> q /\ p`
  and th2 = TAUT `q /\ p ==> p /\ q`;;
val th1 : thm = |- p /\ q ==> q /\ p
val th2 : thm = |- q /\ p ==> p /\ q

# IMP_ANTISYM_RULE th1 th2;;
val it : thm = |- p /\ q <=> q /\ p
```

## See also

EQ_IMP_RULE, EQ_MP, EQ_TAC.

# implode

`implode : string list -> string`

## Synopsis

Concatenates a list of strings into one string.

## Description

`implode [s1;...;sn]` returns the string formed by concatenating the strings `s1 ... sn`. If `n` is zero (the list is empty), then the empty string is returned.

## Failure

Never fails; accepts empty or multi-character component strings.

## Example

```
# implode ["e";"x";"a";"m";"p";"l";"e"];;
val it : string = "example"
# implode ["ex";"a";"mpl";"";"e"];;
val it : string = "example"
```

## See also

explode.

# IMP_RES_THEN

`IMP_RES_THEN : thm_tactical`

## Synopsis

Resolves an implication with the assumptions of a goal.

## Description

The function `IMP_RES_THEN` is the basic building block for resolution in HOL. This is not full higher-order, or even first-order, resolution with unification, but simply one way simultaneous pattern-matching (resulting in term and type instantiation) of the antecedent of an implicative theorem to the conclusion of another theorem (the candidate antecedent).

Given a theorem-tactic `ttac` and a theorem `th`, the theorem-tactical `IMP_RES_THEN` produces a tactic that, when applied to a goal `A ?- g` attempts to match each antecedent `ui` to each assumption `aj |- aj` in the assumptions `A`. If the antecedent `ui` of any implication matches the conclusion `aj` of any assumption, then an instance of the theorem `Ai u {aj} |- vi`, called a 'resolvent', is obtained by specialization of the variables `x1`, ..., `xn` and type instantiation, followed by an application of modus ponens. There may be more than one canonical implication and each implication is tried against every assumption of the goal, so there may be several resolvents (or, indeed, none).

Tactics are produced using the theorem-tactic `ttac` from all these resolvents (failures of `ttac` at this stage are filtered out) and these tactics are then applied in an unspecified sequence to the goal. That is,

```
IMP_RES_THEN ttac th  (A ?- g)
```

has the effect of:

```
MAP_EVERY (mapfilter ttac [... ; (Ai u {aj} |- vi) ; ...]) (A ?- g)
```

where the theorems `Ai u {aj} |- vi` are all the consequences that can be drawn by a (single) matching modus-ponens inference from the assumptions of the goal `A ?- g` and the implications derived from the supplied theorem `th`. The sequence in which the theorems `Ai u {aj} |- vi` are generated and the corresponding tactics applied is unspecified.

## Failure

Evaluating `IMP_RES_THEN ttac th` fails if the supplied theorem `th` is not an implication, or if no implications can be derived from `th` by the transformation process described under the entry for `RES_CANON`. Evaluating `IMP_RES_THEN ttac th (A ?- g)` fails if no assumption of the goal `A ?- g` can be resolved with the implication or implications derived from `th`. Evaluation also fails if there are resolvents, but for every resolvent `Ai u {aj} |- vi` evaluating the application `ttac (Ai u {aj} |- vi)` fails—that is, if for every resolvent `ttac` fails to produce a tactic. Finally, failure is propagated if any of the tactics that are produced from the resolvents by `ttac` fails when applied in sequence to the goal.

## Example

The following example shows a straightforward use of `IMP_RES_THEN` to infer an equational consequence of the assumptions of a goal, use it once as a substitution in the conclusion of goal, and then 'throw it away'. Suppose the goal is:

```
# g `!a n. a + n = a ==> !k. k - n = k`;;
```

and we start out with:

```
# e(REPEAT GEN_TAC THEN DISCH_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`a + n = a`]

`!k. k - n = k`
```

By using the theorem:

```
# let ADD_INV_0 = ARITH_RULE `!m n. m + n = m ==> n = 0`;;
```

the assumption of this goal implies that `n` equals `0`. A single-step resolution with this theorem followed by substitution:

```
# e(IMP_RES_THEN SUBST1_TAC ADD_INV_0);;
val it : goalstack = 1 subgoal (1 total)

 0 [`a + n = a`]

`!k. k - 0 = k`
```

Here, a single resolvent `a + n = a |- n = 0` is obtained by matching the antecedent of `ADD_INV_0` to the assumption of the goal. This is then used to substitute `0` for `n` in the conclusion of the goal. The goal is now solvable by `ARITH_TAC` (as indeed was the original goal).

## See also

IMP_RES_THEN, MATCH_MP, MATCH_MP_TAC.

# IMP_REWR_CONV

```
IMP_REWR_CONV : thm -> term -> thm
```

## Synopsis

Basic conditional rewriting conversion.

## Description

Given an equational theorem `A |- !x1...xn. p ==> s = t` that expresses a conditional rewrite rule, the conversion `IMP_REWR_CONV` gives a conversion that applied to any term `s'` will attempt to match the left-hand side of the equation `s = t` to `s'`, and return the corresponding theorem `A |- p' ==> s' = t'`.

## Failure

Fails if the theorem is not of the right form or the two terms cannot be matched, for example because the variables that need to be instantiated are free in the hypotheses `A`.

## Example

We use the following theorem:

```
# DIV_MULT;;
val it : thm = |- !m n. ~(m = 0) ==> (m * n) DIV m = n
```

to make a conditional rewrite:

```
# IMP_REWR_CONV DIV_MULT `(2 * x) DIV 2`;;
val it : thm = |- ~(2 = 0) ==> (2 * x) DIV 2 = x
```

## Uses

One of the building-blocks for conditional rewriting as implemented by `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC` etc.

## See also

`ORDERED_IMP_REWR_CONV`, `REWR_CONV`, `SIMP_CONV`.

# IMP_TRANS

```
IMP_TRANS : thm -> thm -> thm
```

## Synopsis

Implements the transitivity of implication.

## Description

When applied to theorems `A1 |- t1 ==> t2` and `A2 |- t2 ==> t3`, the inference rule `IMP_TRANS` returns the theorem `A1 u A2 |- t1 ==> t3`.

```
    A1 |- t1 ==> t2   A2 |- t2 ==> t3
   -----------------------------------  IMP_TRANS
         A1 u A2 |- t1 ==> t3
```

## Failure

Fails unless the theorems are both implicative, with the consequent of the first being the same as the antecedent of the second (up to alpha-conversion).

## Example

```
# let th1 = TAUT `p /\ q /\ r ==> p /\ q`
  and th2 = TAUT `p /\ q ==> p`;;
val th1 : thm = |- p /\ q /\ r ==> p /\ q
val th2 : thm = |- p /\ q ==> p

# IMP_TRANS th1 th2;;
val it : thm = |- p /\ q /\ r ==> p
```

## See also

`IMP_ANTISYM_RULE, SYM, TRANS.`

# increasing

```
increasing : ('a -> 'b) -> 'a -> 'a -> bool
```

## Synopsis

Returns a total ordering based on a measure function

## Description

When applied to a "measure" function `f`, the call `increasing f` returns a binary function ordering elements in a call `increasing f x y` by `f(x) <? f(y)`, where the ordering `<?` is the OCaml polymorphic ordering.

## Failure

Never fails unless the measure function does.

### Example

```
# let nums = -5 -- 5;;
val nums : int list = [-5; -4; -3; -2; -1; 0; 1; 2; 3; 4; 5]
# sort (increasing abs) nums;;
val it : int list = [0; 1; -1; 2; -2; 3; -3; 4; -4; 5; -5]
```

### See also

`<?`, `decreasing`, `sort`.

## index

`index : 'a -> 'a list -> int`

### Synopsis

Returns position of given element in list.

### Description

The call `index x l` where l is a list returns the position number of the first instance of x in the list, failing if there is none. The indices start at zero, corresponding to `el`.

### Example

```
# index "j" (explode "abcdefghijklmnopqrstuvwxyz");;
val it : int = 9
```

This is a sort of inverse to the indexing into a string by `el`:

```
# el 9 (explode "abcdefghijklmnopqrstuvwxyz");;
val it : string = "j"
```

### See also

`el`, `find`.

## inductive_type_store

`inductive_type_store : (string * (int * thm * thm)) list ref`

## Synopsis

List of inductive types defined with corresponding theorems.

## Description

The reference variable `inductive_type_store` holds an association list that associates with the name of each inductive type defined so far (e.g. `"list"` or `"1"`) a triple: the number of constructors, the induction theorem and the recursion theorem for it. The two theorems are exactly of the form returned by `define_type`.

## Failure

Not applicable.

## Example

This example is characteristic:

```
# assoc "list" (!inductive_type_store);;
val it : int * thm * thm =
  (2, |- !P. P [] /\ (!a0 a1. P a1 ==> P (CONS a0 a1)) ==> (!x. P x),
   |- !NIL' CONS'.
          ?fn. fn [] = NIL' /\
               (!a0 a1. fn (CONS a0 a1) = CONS' a0 a1 (fn a1)))
```

while the following shows that there is an entry for the Boolean type, for the sake of regularity, even though it is not normally considered an inductive type:

```
# assoc "bool" (!inductive_type_store);;
val it : int * thm * thm =
  (2, |- !P. P F /\ P T ==> (!x. P x), |- !a b. ?f. f F = a /\ f T = b)
```

## Uses

This list is mainly for internal use. For example it is employed by `define` to automatically prove the existence of recursive functions over inductive types. Users may find the information helpful to implement their own proof tools. However, while the list may be inspected, it should not be modified explicitly or there may be unwanted side-effects on `define`.

## See also

`define`, `define_type`, `new_recursive_definition`, `prove_recursive_functions_exist`.

# INDUCT_TAC

```
INDUCT_TAC : tactic
```

## Synopsis

Performs tactical proof by mathematical induction on the natural numbers.

## Description

`INDUCT_TAC` reduces a goal `A ?- !n. P[n]`, where `n` has type `num`, to two subgoals corresponding to the base and step cases in a proof by mathematical induction on `n`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `INDUCT_TAC` is:

```
                A ?- !n. P
  ========================================  INDUCT_TAC
   A ?- P[0/n]     A u {P} ?- P[SUC n'/n]
```

where `n'` is a primed variant of `n` that does not appear free in the assumptions `A` (usually, `n'` is just `n`).

## Failure

`INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form `‘!n. t‘`, where the variable `n` has type `num`.

## Example

Suppose we want to prove the classic ‘sum of the first `n` integers’ theorem:

```
# g ‘!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2‘;;
```

This is a classic example of an inductive proof. If we apply induction, we get two subgoals:

```
# e INDUCT_TAC;;
val it : goalstack = 2 subgoals (2 total)

 0 [‘nsum (1 .. n) (\i. i) = (n * (n + 1)) DIV 2‘]

‘nsum (1 .. SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2‘

‘nsum (1 .. 0) (\i. i) = (0 * (0 + 1)) DIV 2‘
```

each of which can be solved by just:

```
# e(ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

## Comments

Essentially the same effect can be had by `MATCH_MP_TAC num_INDUCTION`. This does not subsequently break down the goal in such a convenient way, but gives more control over choice of variable. You can also equally well use it for other kinds of induction, e.g. use `MATCH_MP_TAC num_WF` for wellfounded (complete, noetherian) induction.

### See also
`LIST_INDUCT_TAC`, `MATCH_MP_TAC`, `WF_INDUCT_TAC`.

## infixes

```
infixes : unit -> (string * (int * string)) list
```

### Synopsis
Lists the infixes currently recognized by the parser.

### Description
The function `infixes` should be applied to the unit `()` and will then return a list of all the infixes currently recognized by the parser together with their precedence and associativity (left or right).

### Failure
Never fails.

### See also
`get_infix_status`, `parse_as_infix`, `unparse_as_infix`.

## injectivity

```
injectivity : string -> thm
```

### Synopsis
Produce injectivity theorem for an inductive type.

### Description
A call `injectivity "ty"` where `"ty"` is the name of a recursive type defined with `define_type`, returns a "injectivity" theorem asserting that elements constructed by different type constructors are always different. The effect is exactly the same is if `prove_constructors_injective` were applied to the recursion theorem produced by `define_type`, and the documentation for `prove_constructors_injective` gives a lengthier discussion.

### Failure
Fails if `ty` is not the name of a recursive type, or if all its constructors are nullary.

## Example

```
# injectivity "num";;
val it : thm = |- !n n'. SUC n = SUC n' <=> n = n'

# injectivity "list";;
val it : thm =
  |- !a0 a1 a0' a1'. CONS a0 a1 = CONS a0' a1' <=> a0 = a0' /\ a1 = a1'
```

## See also

cases, define_type, distinctness, prove_constructors_injective.

# injectivity_store

```
injectivity_store : (string * thm) list ref
```

## Synopsis

Internal theorem list of injectivity theorems.

## Description

This list contains all the injectivity theorems (see `injectivity`) for the recursive types defined so far. It is automatically extended by `define_type` and used as a cache by `injectivity`.

## Failure

Not applicable.

## See also

define_type, distinctness_store, extend_rectype_net, injectivity.

# insert

```
insert : 'a -> 'a list -> 'a list
```

## Synopsis

Adds element to the head of a list if not already present.

## Description

The call `insert x l` returns just `l` if `x` is already in the list, and otherwise returns `x::l`.

## Example

```
# insert 5 (1--10);;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# insert 15 (1--10);;
val it : int list = [15; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

## Uses

An analog to the basic list constructor `::` but treating the list more like a set.

## See also

`union`, `intersect`, `subtract`.

# insert'

`insert' : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list`

## Synopsis

Insert element into list unless it contains an equivalent one already.

## Description

If `r` is a binary relation, `x` an element and `l` a list, the call `insert' r x l` will add `x` to the head of the list, unless the list already contains an element `x'` with `r x x'`; if it does, the list is returned unchanged. The function `insert` is the special case where `r` is equality.

## Failure

Fails only if the relation fails.

## Example

```
# insert' (fun x y -> abs(x) = abs(y)) (-1) [1;2;3];;
val it : int list = [1; 2; 3]

# insert' (fun x y -> abs(x) = abs(y)) (-1) [2;3;4];;
val it : int list = [-1; 2; 3; 4]
```

## See also

`insert`, `mem'`, `subtract'`, `union'`, `unions'`.

# installed_parsers

```
installed_parsers : unit -> (string * (lexcode list -> preterm * lexcode list)) list
```

## Synopsis
List the user parsers currently installed.

## Description
HOL Light allows user parsing functions to be installed, and will try them on all terms during parsing before the usual parsers. The call `installed_parsers()` lists the parsing functions that have been so installed.

## Failure
Never fails.

## See also
`delete_parser`, `install_parser`, `try_user_parser`.

# install_parser

```
install_parser : string * (lexcode list -> preterm * lexcode list) -> unit
```

## Synopsis
Install a user parser.

## Description
HOL Light allows user parsing functions to be installed, and will try them on all terms during parsing before the usual parsers. The call `install_parser(s,p)` installs the parser `p` among the user parsers to try in this way. The string `s` is there so that the parser can conveniently be deleted again.

## Failure
Never fails.

## See also
`delete_parser`, `installed_parsers`, `try_user_parser`.

# install_user_printer

`install_user_printer : string * (term -> unit) -> unit`

## Synopsis

Install a user-defined printing function into the HOL Light term printer.

## Description

The call `install_user_printer(s,pr)` sets up `pr` inside the HOL Light toplevel printer. On each subterm encountered, `pr` will be tried first, and only if it fails with `Failure ...` will the normal HOL Light printing be invoked. The additional string argument `s` is just to provide a convenient handle for later removal through `delete_user_printer`. However, any previous user printer with the same string tag will be removed when `install_user_printer` is called.

## Failure

Never fails.

## Example

The user might wish to print every variable with its type:

```
# let print_typed_var tm =
    let s,ty = dest_var tm in
    print_string("("^s^":"^string_of_type ty^")") in
  install_user_printer("print_typed_var",print_typed_var);;

val it : unit = ()
# ADD_ASSOC;;
val it : thm =
  |- !(m:num) (n:num) (p:num).
         (m:num) + (n:num) + (p:num) = ((m:num) + (n:num)) + (p:num)
```

## Uses

Modification of printing in this way is particularly useful when the HOL logic is used to embed some other formalism such as a programming language, hardware description language or other logic. This can then be printed in a “native” fashion without any artifacts of its HOL formalization.

## Comments

Since user printing functions are tried on every subterm encountered in the regular printing function, it is important that they fail quickly when inapplicable, or the printing process can be slowed. They should also not generate exceptions other than `Failure ...` or the toplevel printer will start to fail.

### See also
`delete_user_printer`, `try_user_printer`.

## INSTANTIATE_ALL

```
INSTANTIATE_ALL : instantiation -> thm -> thm
```

### Synopsis
Apply a higher-order instantiation to assumptions and conclusion of a theorem.

### Description
The call `INSTANTIATE_ALL i t`, where `i` is an instantiation as returned by `term_match`, will perform the instantiation indicated by `i` in the conclusion of the theorem `th`: types and terms will be instantiated and the beta-reductions that are part of higher-order matching will be applied.

### Failure
Never fails on a valid instantiation.

### Comments
This is not intended for general use. `PART_MATCH` is generally a more convenient packaging. The function `INSTANTIATE` is almost the same but does not instantiate hypotheses and may fail if type variables or term variables free in the hypotheses make the instantiation impossible.

### See also
`INSTANTIATE`, `INSTANTIATE_ALL`, `PART_MATCH`, `term_match`.

## instantiate_casewise_recursion

```
instantiate_casewise_recursion : term -> thm
```

### Synopsis
Instantiate the general scheme for a recursive function existence assertion.

## Description

The function `instantiate_casewise_recursion` should be applied to an existentially quantified term `` `?f. def_1[f] /\ ... /\ def_n[f]` ``, where each clause `def_i` is a universally quantified equation with an application of `f` to arguments on the left-hand side. The idea is that these clauses define the action of `f` on arguments of various kinds, for example on an empty list and nonempty list:

```
?f. (f [] = a) /\ (!h t. CONS h t = k[f,h,t])
```

or on even numbers and odd numbers:

```
?f. (!n. f(2 * n) = a[f,n]) /\ (!n. f(2 * n + 1) = b[f,n])
```

The returned value is a theorem whose conclusion matches the input term, with an assumption sufficient for the existence assertion. This is not normally in a very convenient form for the user.

## Failure

Fails only if the definition is malformed. However it is possible that for an inadmissible definition the assumption of the theorem may not hold.

## Uses

This is seldom a convenient function for users. Normally, use `prove_general_recursive_function_exists` to prove something like this while attempting to discharge the side-conditions automatically, or `define` to actually make a definition. In situations where the automatic discharge of the side-conditions fails, one may prefer instead `pure_prove_recursive_function_exists`. The even more minimal `instantiate_casewise_recursion` is for the rare cases where one wants to force no processing at all of the side-conditions to be undertaken.

## See also

`define`, `prove_general_recursive_function_exists`,
`pure_prove_recursive_function_exists`.

# instantiate

```
instantiate : instantiation -> term -> term
```

## Synopsis

Apply a higher-order instantiation to a term.

## Description

The call `instantiate i t`, where `i` is an instantiation as returned by `term_match`, will perform the instantiation indicated by `i` in the term `t`: types and terms will be instantiated and the beta-reductions that are part of higher-order matching will be applied.

## Failure

Should never fail on a valid instantiation.

## Example

We first compute an instantiation:

```
# let t = `(!x. P x) <=> ~(?x. P x)`;;
Warning: inventing type variables
val t : term = `(!x. P x) <=> ~(?x. P x)`

# let i = term_match [] (lhs t) `!p. prime(p) ==> p = 2 \/ ODD(p)`;;
val i : instantiation =
  ([(1, `P`)], [(`\p. prime p ==> p = 2 \/ ODD p`, `P`)],
   [(`:num`, `:?61195`)])
```

and now apply it. Notice that the type variable name is not corrected, as is done inside `PART_MATCH`

```
# instantiate i t;;
val it : term =
  `(!x. prime x ==> x = 2 \/ ODD x) <=> ~(?x. prime x ==> x = 2 \/ ODD x)`
```

## Comments

This is probably not useful for most users.

## See also

`compose_insts`, `INSTANTIATE`, `INSTANTIATE_ALL`, `inst_goal`, `PART_MATCH`, `term_match`.

# INSTANTIATE

```
INSTANTIATE : instantiation -> thm -> thm
```

## Synopsis

Apply a higher-order instantiation to conclusion of a theorem.

## Description

The call `INSTANTIATE i t`, where `i` is an instantiation as returned by `term_match`, will perform the instantiation indicated by `i` in the conclusion of the theorem `th`: types and terms will be instantiated and the beta-reductions that are part of higher-order matching will be applied.

## Failure

Fails if the instantiation is impossible because of free term or type variables in the hypotheses.

## Example

```
# let t = lhs(concl(SPEC_ALL NOT_FORALL_THM));;
val t : term = `~(!x. P x)`
# let i = term_match [] t `~(!n. prime(n) ==> ODD(n))`;;
val i : instantiation =
  ([(1, `P`)], [(`\n. prime n ==> ODD n`, `P`)], [(`:num`, `:A`)])

# INSTANTIATE i (SPEC_ALL NOT_FORALL_THM);;
val it : thm = |- ~(!x. prime x ==> ODD x) <=> (?x. ~(prime x ==> ODD x))
```

## Comments

This is not intended for general use. `PART_MATCH` is generally a more convenient packaging.

## See also

`instantiate`, `INSTANTIATE_ALL`, `PART_MATCH`, `term_match`.

# inst

```
inst : (hol_type * hol_type) list -> term -> term
```

## Synopsis

Instantiate type variables in a term.

## Description

The call `inst [ty1,tv1; ...; tyn,tvn] t` will systematically replace each type variable `tvi` by the corresponding type `tyi` inside the term `t`. Bound variables will be renamed if necessary to avoid capture.

## Failure

Never fails. Repeated type variables in the instantiation list are not detected, and the first such element will be used.

## Example

Here is a simple example:

```
# inst [`:num`,`:A`] `x:A = x`;;
val it : term = `x = x`

# type_of(rand it);;
val it : hol_type = `:num`
```

To construct an example where variable renaming is necessary we need to construct terms with identically-named variables of different types, which cannot be done directly in the term parser:

```
# let tm = mk_abs(`x:A`,`x + 1`);;
val tm : term = `\x. x + 1`
```

Note that the two variables `x` are different; this is a constant boolean function returning `x + 1`. Now if we instantiate type variable `:A` to `:num`, we still get a constant function, thanks to variable renaming:

```
# inst [`:num`,`:A`] tm;;
val it : term = `\x'. x + 1`
```

It would have been incorrect to just keep the same name, for that would have been the successor function, something different.

## See also

`subst`, `type_subst`, `vsubst`.

# inst_goal

```
inst_goal : instantiation -> goal -> goal
```

## Synopsis

Apply higher-order instantiation to a goal.

## Description

The call `inst_goal i g` where `i` is an instantiation (as returned by `term_match` for example), will perform the instantiation indicated by `i` in both assumptions and conclusion of the goal `g`.

## Failure

Should never fail on a valid instantiation.

## Comments

Probably only of specialist interest to those writing tactics from scratch.

## See also

`compose_insts`, `instantiate`, `INSTANTIATE`, `INSTANTIATE_ALL`, `PART_MATCH`, `term_match`.

# INST_TYPE

```
INST_TYPE : (hol_type * hol_type) list -> thm -> thm
```

## Synopsis

Instantiates types in a theorem.

## Description

`INST_TYPE [ty1,tv1;...;tyn,tvn]` will systematically replaces all instances of each type variable `tvi` by the corresponding type `tyi` in both assumptions and conclusions of a theorem:

```
              A |- t
  -----------------------------------  INST_TYPE [ty1,tv1;...;tyn,tvn]
   A[ty1,...,tyn/tv1,...,tvn]
       |- t[ty1,...,tyn/tv1,...,tvn]
```

Variables will be renamed if necessary to prevent variable capture.

## Failure

Never fails.

## Uses

`INST_TYPE` is employed to make use of polymorphic theorems.

## Example

Suppose one wanted to specialize the theorem EQ_SYM_EQ for particular values, the first attempt could be to use SPECL as follows:

```
# SPECL [`a:num`; `b:num`] EQ_SYM_EQ ;;
Exception: Failure "SPECL".
```

The failure occurred because EQ_SYM_EQ contains polymorphic types. The desired specialization can be obtained by using INST_TYPE:

```
# SPECL [`a:num`; `b:num`] (INST_TYPE [`:num`,`:A`] EQ_SYM_EQ) ;;
val it : thm = |- a = b <=> b = a
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

INST, ISPEC, ISPECL.

# INST

```
INST : (term * term) list -> thm -> thm
```

## Synopsis

Instantiates free variables in a theorem.

## Description

When INST [t1,x1; ...; tn,xn] is applied to a theorem, it gives a new theorem that systematically replaces free instances of each variable xi with the corresponding term ti in both assumptions and conclusion.

```
              A |- t
-----------------------------------  INST_TYPE [t1,x1;...;tn,xn]
 A[t1,...,tn/x1,...,xn]
     |- t[t1,...,tn/x1,...,xn]
```

Bound variables will be renamed if necessary to avoid capture. All variables are substituted in parallel, so there is no problem if there is an overlap between the terms ti and xi.

### Failure

Fails if any of the pairs ti,xi in the instantiation list has xi and ti with different types, or xi a non-variable. Multiple instances of the same xi in the list are not trapped, but only the first one will be used consistently.

### Example

Here is a simple example

```
# let th = SPEC_ALL ADD_SYM;;
val th : thm = |- m + n = n + m
# INST [`1`,`m:num`; `x:num`,`n:num`] th;;
val it : thm = |- 1 + x = x + 1
```

and here is one where bound variable renaming is needed.

```
# let th = SPEC_ALL LE_EXISTS;;
val th : thm = |- m <= n <=> (?d. n = m + d)
# INST [`d:num`,`m:num`] th;;
val it : thm = |- d <= n <=> (?d'. n = d + d')
```

### Uses

This is the most efficient way to obtain instances of a theorem; though sometimes more convenient, SPEC and SPECL are significantly slower.

### Comments

This is one of HOL Light's 10 primitive inference rules.

### See also

INST_TYPE, ISPEC, ISPECL, SPEC, SPECL.

## INT_ABS_CONV

INT_ABS_CONV : conv

### Synopsis

Conversion to produce absolute value of an integer literal of type :int.

### Description

The call INT_ABS_CONV `abs c`, where c is an integer literal of type :int, returns the theorem |- abs c = d where d is the canonical integer literal that is equal to c's absolute value. The literal `c` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not the negation of one of the permitted forms of integer literal of type `:int`.

## Example

```
# INT_ABS_CONV `abs(-- &42)`;;
val it : thm = |- abs (-- &42) = &42
```

## See also

INT_REDUCE_CONV, REAL_RAT_ABS_CONV.

# INT_ADD_CONV

```
INT_ADD_CONV : conv
```

## Synopsis

Conversion to perform addition on two integer literals of type `:int`.

## Description

The call `INT_ADD_CONV `c1 + c2`` where `c1` and `c2` are integer literals of type `:int`, returns `|- c1 + c2 = d` where `d` is the canonical integer literal that is equal to `c1 + c2`. The literals `c1` and `c2` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not the sum of two permitted integer literals of type `:int`.

## Example

```
# INT_ADD_CONV `-- &17 + &25`;;
val it : thm = |- -- &17 + &25 = &8
```

## See also

INT_REDUCE_CONV, REAL_RAT_ADD_CONV.

# INT_ARITH

```
INT_ARITH : term -> thm
```

## Synopsis
Proves integer theorems needing basic rearrangement and linear inequality reasoning only.

## Description
`INT_ARITH` is a rule for automatically proving natural number theorems using basic algebraic normalization and inequality reasoning.

## Failure
Fails if the term is not boolean or if it cannot be proved using the basic methods employed, e.g. requiring nonlinear inequality reasoning.

## Example

```
# INT_ARITH `!x y:int. x <= y + &1 ==> x + &2 < y + &4`;;
val it : thm = |- !x y. x <= y + &1 ==> x + &2 < y + &4

# INT_ARITH `(x + y:int) pow 2 = x pow 2 + &2 * x * y + y pow 2`;;
val it : thm = |- (x + y) pow 2 = x pow 2 + &2 * x * y + y pow 2
```

## Uses
Disposing of elementary arithmetic goals.

## See also
`ARITH_RULE`, `INT_ARITH_TAC`, `NUM_RING`, `REAL_ARITH`, `REAL_FIELD`, `REAL_RING`.

# INT_ARITH_TAC

```
INT_ARITH_TAC : tactic
```

## Synopsis
Attempt to prove goal using basic algebra and linear arithmetic over the integers.

## Description
The tactic `INT_ARITH_TAC` is the tactic form of `INT_ARITH`. Roughly speaking, it will automatically prove any formulas over the reals that are efectively universally quantified

and can be proved valid by algebraic normalization and linear equational and inequality reasoning. See REAL_ARITH for more information about the algorithm used and its scope.

## Failure

Fails if the goal is not in the subset solvable by these means, or is not valid.

## Example

Here is a goal that holds by virtue of pure algebraic normalization:

```
# prioritize_int();;
val it : unit = ()

# g `(x1 pow 2 + x2 pow 2 + x3 pow 2 + x4 pow 2) *
     (y1 pow 2 + y2 pow 2 + y3 pow 2 + y4 pow 2) =
     (x1 * y1 - x2 * y2 - x3 * y3 - x4 * y4) pow 2  +
     (x1 * y2 + x2 * y1 + x3 * y4 - x4 * y3) pow 2  +
     (x1 * y3 - x2 * y4 + x3 * y1 + x4 * y2) pow 2  +
     (x1 * y4 + x2 * y3 - x3 * y2 + x4 * y1) pow 2`;;
```

and here is one that holds by linear inequality reasoning:

```
# g `!x y:int. abs(x + y) < abs(x) + abs(y) + &1`;;
```

so either goal is solved simply by:

```
# e INT_ARITH_TAC;;
val it : goalstack = No subgoals
```

## See also

ARITH_TAC, INT_ARITH, REAL_ARITH_TAC.

# INTEGER_RULE

```
INTEGER_RULE : term -> thm
```

## Synopsis

Automatically prove elementary divisibility property over the integers.

## Description

INTEGER_RULE is a partly heuristic rule that can often automatically prove elementary "divisibility" properties of the integers. The precise subset that is dealt with is difficult to describe rigorously, but many universally quantified combinations of `divides`, `coprime`, `gcd` and congruences `(x == y) (mod n)` can be proved automatically, as well as some existentially quantified goals. The examples below may give a feel for what can be done.

## Failure

Fails if the goal is not accessible to the methods used.

## Example

All sorts of elementary Boolean combinations of divisibility and congruence properties can be solved, e.g.

```
# INTEGER_RULE
   `!x y n:int. (x == y) (mod n) ==> (n divides x <=> n divides y)`;;
...
val it : thm = |- !x y n. (x == y) (mod n) ==> (n divides x <=> n divides y)

# INTEGER_RULE
   `!a b d:int. d divides gcd(a,b) <=> d divides a /\ d divides b`;;
...
val it : thm =
 |- !a b d. d divides gcd (a,b) <=> d divides a /\ d divides b
```

including some less obvious ones:

```
# INTEGER_RULE
   `!x y. coprime(x * y,x pow 2 + y pow 2) <=> coprime(x,y)`;;
...
val it : thm = |- !x y. coprime (x * y,x pow 2 + y pow 2) <=> coprime (x,y)
```

A limited class of existential goals is solvable too, e.g. a classic sufficient condition for a linear congruence to have a solution:

```
# INTEGER_RULE `!a b n:int. coprime(a,n) ==> ?x. (a * x == b) (mod n)`;;
...
val it : thm = |- !a b n. coprime (a,n) ==> (?x. (a * x == b) (mod n))
```

or the two-number Chinese Remainder Theorem:

```
# INTEGER_RULE
  `!a b u v:int. coprime(a,b) ==> ?x. (x == u) (mod a) /\ (x == v) (mod b)`;;
...
val it : thm =
|- !a b u v. coprime (a,b) ==> (?x. (x == u) (mod a) /\ (x == v) (mod b))
```

## See also

`ARITH_RULE`, `INTEGER_TAC`, `INT_ARITH`, `INT_RING`, `NUMBER_RULE`.

# INTEGER_TAC

```
INTEGER_TAC : tactic
```

## Synopsis

Automated tactic for elementary divisibility properties over the integers.

## Description

The tactic `INTEGER_TAC` is a partly heuristic tactic that can often automatically prove elementary "divisibility" properties of the integers. The precise subset that is dealt with is difficult to describe rigorously, but many universally quantified combinations of `divides`, `coprime`, `gcd` and congruences `(x == y) (mod n)` can be proved automatically, as well as some existentially quantified goals. See the documentation for `INTEGER_RULE` for a larger set of representative examples.

## Failure

Fails if the goal is not accessible to the methods used.

## Example

A typical elementary divisibility property is that if `a * x` and `a * y` are congruent modulo `n` and the two numbers `a` and `n` are coprime (share no common factor besides `1`), then in fact `x` and `y` are congruent:

```
# g `!a n x y:int. (a * x == a * y) (mod n) /\ coprime(a,n)
                   ==> (x == y) (mod n)`;;
...
```

It can be solved automatically using `NUMBER_TAC`:

```
# e NUMBER_TAC;;
...
val it : goalstack = No subgoals
```

## See also

`INTEGER_RULE`, `INT_ARITH_TAC`, `INT_RING`, `NUMBER_RULE`.

# INT_EQ_CONV

```
INT_EQ_CONV : conv
```

## Synopsis
Conversion to prove whether one integer literal of type `:int` is equal to another.

## Description
The call `INT_EQ_CONV ‘c1 < c2‘` where `c1` and `c2` are integer litersls of type `:int`, returns whichever of `|- c1 = c2 <=> T` or `|- c1 = c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

## Failure
Fails if applied to a term that is not an equality comparison on two permitted integer literals of type `:int`.

## Example
```
# INT_EQ_CONV ‘&1 = &2‘;;
val it : thm = |- &1 = &2 <=> F

# INT_EQ_CONV ‘-- &1 = -- &1‘;;
val it : thm = |- -- &1 = -- &1 <=> T
```

## Comments
The related function `REAL_RAT_EQ_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
`INT_REDUCE_CONV`, `REAL_RAT_EQ_CONV`.

# intersect

```
intersect : 'a list -> 'a list -> 'a list
```

## Synopsis
Computes the intersection of two ‘sets’.

## Description
`intersect l1 l2` returns a list consisting of those elements of `l1` that also appear in `l2`. If both sets are free of repetitions, this can be considered a set-theoretic intersection operation.

## Failure
Never fails.

## Comments
Duplicate elements in the first list will still be present in the result.

## Example

```
# intersect [1;2;3] [3;5;4;1];;
val it : int list = [1; 3]
# intersect [1;2;4;1] [1;2;3;2];;
val it : int list = [1; 2; 1]
```

## See also
setify, set_equal, union, subtract.

# INT_GE_CONV

```
INT_GE_CONV : conv
```

## Synopsis
Conversion to prove whether one integer literal of type :int is >= another.

## Description
The call INT_GE_CONV ‘c1 >= c2‘ where c1 and c2 are integer litersls of type :int, returns whichever of |- c1 >= c2 <=> T or |- c1 >= c2 <=> F is true. By an integer literal we mean either &n or -- &n where n is a numeral.

## Failure
Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type :int.

## Example

```
# INT_GE_CONV ‘&7 >= &6‘;;
val it : thm = |- &7 >= &6 <=> T
```

## See also
INT_REDUCE_CONV, REAL_RAT_GE_CONV.

# INT_GT_CONV

```
INT_GT_CONV : conv
```

## Synopsis
Conversion to prove whether one integer literal of type `:int` is `<` another.

## Description
The call `INT_GT_CONV ‘c1 > c2‘` where `c1` and `c2` are integer litersls of type `:int`, returns whichever of `|- c1 > c2 <=> T` or `|- c1 > c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

## Failure
Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type `:int`.

## Example

```
# INT_GT_CONV ‘&1 > &2‘;;
val it : thm = |- &1 > &2 <=> F
```

## See also
`INT_REDUCE_CONV`, `REAL_RAT_GT_CONV`.

# int_ideal_cofactors

```
int_ideal_cofactors : term list -> term -> term list
```

## Synopsis
Produces cofactors proving that one integer polynomial is in the ideal generated by others.

## Description
The call `int_ideal_cofactors [‘p1‘; ...; ‘pn‘] ‘p‘`, where all the terms have type `:int` and can be considered as polynomials, will test whether `p` is in the ideal generated by the `p1,...,pn`. If so, it will return a corresponding list `[‘q1‘; ...; ‘qn‘]` of ‘cofactors’

such that the following is an algebraic identity provable by `INT_RING` or a slight elaboration of `INT_POLY_CONV`, for example)

```
p = p1 * q1 + ... + pn * qn
```

hence providing an explicit certificate for the ideal membership. If ideal membership does not hold, `int_ideal_cofactors` fails. The test is performed using a Gröbner basis procedure.

## Failure

Fails if the terms are ill-typed, or if ideal membership fails. At present this is a generic version for fields, and in rare cases it may fail because cofactors are found involving non-trivial rational numbers even where there are integer cofactors. This imperfection should be fixed eventually, and is not usually a problem in practice.

## Example

In the case of a singleton list, ideal membership just amounts to polynomial divisibility, e.g.

```
# int_ideal_cofactors
   [`r * x * (&1 - x) - x`]
   `r * (r * x * (&1 - x)) * (&1 - r * x * (&1 - x)) - x`;;
[`&1 * r pow 2 * x pow 2 +
  -- &1 * r pow 2 * x +
  -- &1 * r * x +
  &1 * r +
  &1`]
```

## Comments

When we say that terms can be 'considered as polynomials', we mean that initial normalization, essentially in the style of `INT_POLY_CONV`, will be applied, but some complex constructs such as conditional expressions will be treated as atomic.

## See also

`ideal_cofactors`, `INT_IDEAL_CONV`, `INT_RING`, `real_ideal_cofactors`, `RING`, `RING_AND_IDEAL_CONV`.

# INT_LE_CONV

```
INT_LE_CONV : conv
```

### Synopsis
Conversion to prove whether one integer literal of type `:int` is `<=` another.

### Description
The call `INT_LE_CONV ‘c1 <= c2‘` where `c1` and `c2` are integer litersls of type `:int`, returns whichever of `|- c1 <= c2 <=> T` or `|- c1 <= c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

### Failure
Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type `:int`.

### Example
```
# INT_LE_CONV ‘&11 <= &77‘;;
val it : thm = |- &11 <= &77 <=> T
```

### See also
`INT_REDUCE_CONV`, `REAL_RAT_LE_CONV`.

## INT_LT_CONV

```
INT_LT_CONV : conv
```

### Synopsis
Conversion to prove whether one integer literal of type `:int` is `<` another.

### Description
The call `INT_LT_CONV ‘c1 < c2‘` where `c1` and `c2` are integer litersls of type `:int`, returns whichever of `|- c1 < c2 <=> T` or `|- c1 < c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

### Failure
Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type `:int`.

## Example

```
# INT_LT_CONV `-- &18 < &64`;;
val it : thm = |- -- &18 < &64 <=> T
```

## Comments

The related function `REAL_RAT_LT_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

`INT_REDUCE_CONV`, `REAL_RAT_LT_CONV`.

# INT_MUL_CONV

```
INT_MUL_CONV : conv
```

## Synopsis

Conversion to perform multiplication on two integer literals of type `:int`.

## Description

The call `INT_MUL_CONV `c1 * c2`` where `c1` and `c2` are integer literals of type `:int`, returns `|- c1 * c2 = d` where `d` is the canonical integer literal that is equal to `c1 * c2`. The literals `c1` and `c2` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not the product of two permitted integer literals of type `:int`.

## Example

```
# INT_MUL_CONV `&6 * -- &9`;;
val it : thm = |- &6 * -- &9 = -- &54
```

## See also

`INT_REDUCE_CONV`, `REAL_RAT_MUL_CONV`.

# INT_NEG_CONV

`INT_NEG_CONV : conv`

## Synopsis
Conversion to negate an integer literal of type `:int`.

## Description
The call `INT_NEG_CONV ‘--c‘`, where `c` is an integer literal of type `:int`, returns the theorem `|- --c = d` where `d` is the canonical integer literal that is equal to `c`'s negation. The literal `c` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not the negation of one of the permitted forms of integer literal of type `:int`.

## Example

```
# INT_NEG_CONV ‘-- (-- &3 / &2)‘;;
val it : thm = |- --(-- &3 / &2) = &3 / &2
```

## Comments
The related function `REAL_RAT_NEG_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
`INT_REDUCE_CONV`, `REAL_RAT_NEG_CONV`.

# INT_OF_REAL_THM

`INT_OF_REAL_THM : thm -> thm`

## Synopsis
Map a universally quantified theorem from reals to integers.

## Description

We often regard integers as a subset of the reals, so any universally quantified theorem over the reals also holds for the integers, and indeed any other subset. In HOL, integers and reals are completely separate types (`int` and `real` respectively). However, there is a natural injection (actually called `dest_int`) from integers to reals that maps integer operations to their real counterparts, and using this we can similarly show that any universally quantified formula over the reals also holds over the integers with operations mapped to the right type. The rule `INT_OF_REAL_THM` embodies this procedure; given a universally quantified theorem over the reals, it maps it to a corresponding theorem over the integers.

## Failure

Never fails.

## Example

```
# REAL_ABS_TRIANGLE;;
val it : thm = |- !x y. abs (x + y) <= abs x + abs y
# map dest_var (variables(concl it));;
val it : (string * hol_type) list = [("y", `:real`); ("x", `:real`)]

# INT_OF_REAL_THM REAL_ABS_TRIANGLE;;
val it : thm = |- !x y. abs (x + y) <= abs x + abs y
# map dest_var (variables(concl it));;
val it : (string * hol_type) list = [("y", `:int`); ("x", `:int`)]
```

## See also

`ARITH_RULE`, `INT_ARITH`, `INT_ARITH_CONV`, `INT_ARITH_TAC`, `NUM_TO_INT_CONV`, `REAL_ARITH`.

# INT_POLY_CONV

```
INT_POLY_CONV : term -> thm
```

## Synopsis

Converts a integer polynomial into canonical form.

## Description

Given a term of type `:int` that is built up using addition, subtraction, negation and multiplication, `INT_POLY_CONV` converts it into a canonical polynomial form and returns a theorem asserting the equivalence of the original and canonical terms. The basic elements need not simply be variables or constants; anything not built up using the operators given above will be considered 'atomic' for the purposes of this conversion. The canonical polynomial form is a 'multiplied out' sum of products, with the monomials (product terms) ordered according to the canonical OCaml order on terms. In particular, it is just `&0` if the polynomial is identically zero.

## Failure

Never fails, even if the term has the wrong type; in this case it merely returns a reflexive theorem.

## Example

This illustrates how terms are 'multiplied out':

```
# INT_POLY_CONV `(x + y) pow 3`;;
val it : thm =
  |- (x + y) pow 3 = x pow 3 + &3 * x pow 2 * y + &3 * x * y pow 2 + y pow 3
```

while the following verifies a remarkable 'sum of cubes' identity due to Yasutoshi Kohmoto:

```
 # INT_POLY_CONV
     `(&1679616 * a pow 16 - &66096 * a pow 10 * b pow 6 +
       &153 * a pow 4 * b pow 12) pow 3 +
      (-- &1679616 * a pow 16 - &559872 * a pow 13 * b pow 3 -
       &27216 * a pow 10 * b pow 6 + &3888 * a pow 7 * b pow 9 +
       &63 * a pow 4 * b pow 12 - &3 * a * b pow 15) pow 3 +
      (&1679616 * a pow 15 * b + &279936 * a pow 12 * b pow 4 -
       &11664 * a pow 9 * b pow 7 -
       &648 * a pow 6 * b pow 10 + &9 * a pow 3 * b pow 13 + b pow 16) pow 3`;;
val it : thm =
 |- ... =
    b pow 48
```

## Uses

Keeping terms in normal form. For simply proving equalities, `INT_RING` is more powerful and usually more convenient.

## See also

`INT_ARITH`, `INT_RING`, `REAL_POLY_CONV`, `SEMIRING_NORMALIZERS_CONV`.

# INT_POW_CONV

```
INT_POW_CONV : conv
```

## Synopsis
Conversion to perform exponentiation on a integer literal of type `:int`.

## Description
The call `INT_POW_CONV ‘c pow n‘` where `c` is an integer literal of type `:int` and `n` is a numeral of type `:num`, returns `|- c pow n = d` where `d` is the canonical integer literal that is equal to `c` raised to the `n`th power. The literal `c` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not a permitted integer literal of type `:int` raised to a numeral power.

## Example
```
# INT_POW_CONV ‘(-- &2) pow 77‘;;
val it : thm = |- -- &2 pow 77 = -- &151115727451828646838272
```

## See also
`INT_POW_CONV`, `INT_REDUCE_CONV`.

# INT_RED_CONV

```
INT_RED_CONV : term -> thm
```

## Synopsis
Performs one arithmetic or relational operation on integer literals of type `:int`.

## Description
When applied to any of the terms `‘--c‘`, `‘abs c‘`, `‘c1 + c2‘`, `‘c1 - c2‘`, `‘c1 * c2‘`, `‘c pow n‘`, `‘c1 <= c2‘`, `‘c1 < c2‘`, `‘c1 >= c2‘`, `‘c1 > c2‘`, `‘c1 = c2‘`, where `c`, `c1` and `c2` are integer literals of type `:int` and `n` is a numeral of type `:num`, `INT_RED_CONV` returns a theorem asserting the equivalence of the term to a canonical integer (for the arithmetic operators) or a truth-value (for the relational operators). The integer literals are terms of the form `&n` or `-- &n` (with nonzero `n` in the latter case).

## Failure
Fails if applied to an inappropriate term.

### Uses

More convenient for most purposes is INT_REDUCE_CONV, which applies these evaluation conversions recursively at depth, or still more generally REAL_RAT_REDUCE_CONV which applies to any rational numbers, not just integers. Still, access to this 'one-step' reduction can be handy if you want to add a conversion conv for some other operator on int number literals, which you can conveniently incorporate it into INT_REDUCE_CONV with

```
# let INT_REDUCE_CONV' =
    DEPTH_CONV(INT_RED_CONV ORELSEC conv);;
```

### See also

INT_REDUCE_CONV, REAL_RAT_RED_CONV.

## INT_REDUCE_CONV

```
INT_REDUCE_CONV : conv
```

### Synopsis

Evaluate subexpressions built up from integer literals of type :int, by proof.

### Description

When applied to a term, INT_REDUCE_CONV performs a recursive bottom-up evaluation by proof of subterms built from integer literals of type :int using the unary operators '--', 'inv' and 'abs', and the binary arithmetic ('+', '-', '*', '/', 'pow') and relational ('<', '<=', '>', '>=', '=') operators, as well as propagating literals through logical operations, e.g. T /\ x <=> x, returning a theorem that the original and reduced terms are equal. The permissible integer literals are of the form &n or -- &n for numeral n, nonzero in the negative case.

### Failure

Never fails, but may have no effect.

### Example

```
# INT_REDUCE_CONV
   `if &5 pow 4 < &4 pow 5 then (&2 pow 3 - &1) pow 2 + &1 else &99`;;
val it : thm =
  |- (if &5 pow 4 < &4 pow 5 then (&2 pow 3 - &1) pow 2 + &1 else &99) = &50
```

### Comments

The corresponding INT_REDUCE_CONV works for the type of integers. The more general function REAL_RAT_REDUCE_CONV works similarly over :int but for arbitrary rational literals.

## See also

INT_RED_CONV, REAL_RAT_REDUCE_CONV.

# INT_RING

```
INT_RING : term -> thm
```

## Synopsis

Ring decision procedure instantiated to integers.

## Description

The rule INT_RING should be applied to a formula that, after suitable normalization, can be considered a universally quantified Boolean combination of equations and inequations between terms of type :int. If that formula holds in all integral domains, INT_RING will prove it. Any “alien” atomic formulas that are not integer equations will not contribute to the proof but will not in themselves cause an error. The function is a particular instantiation of RING, which is a more generic procedure for ring and semiring structures.

## Failure

Fails if the formula is unprovable by the methods employed. This does not necessarily mean that it is not valid for :int, but rather that it is not valid on all integral domains (see below).

## Example

Here is a nice identity taken from one of Ramanujan's notebooks:

```
# INT_RING
   `!a b c:int.
     a + b + c = &0
     ==> &2 * (a * b + a * c + b * c) pow 2 =
           a pow 4 + b pow 4 + c pow 4 /\
         &2 * (a * b + a * c + b * c) pow 4 =
           (a * (b - c)) pow 4 + (b * (a - c)) pow 4 + (c * (a - b)) pow 4`;;
...
val it : thm =
  |- !a b c.
         a + b + c = &0
         ==> &2 * (a * b + a * c + b * c) pow 2 = a pow 4 + b pow 4 + c pow 4 /\
             &2 * (a * b + a * c + b * c) pow 4 =
             (a * (b - c)) pow 4 + (b * (a - c)) pow 4 + (c * (a - b)) pow 4
```

The reasoning `INT_RING` is capable of includes, of course, the degenerate case of simple algebraic identity, e.g. Brahmagupta's identity:

```
# INT_RING `(a pow 2 + b pow 2) * (c pow 2 + d pow 2) =
            (a * c - b * d) pow 2 + (a * d + b * c) pow 2`;;
```

or the more complicated 4-squares variant:

```
# INT_RING
  `(x1 pow 2 + x2 pow 2 + x3 pow 2 + x4 pow 2) *
   (y1 pow 2 + y2 pow 2 + y3 pow 2 + y4 pow 2) =
   (x1 * y1 - x2 * y2 - x3 * y3 - x4 * y4) pow 2 +
   (x1 * y2 + x2 * y1 + x3 * y4 - x4 * y3) pow 2 +
   (x1 * y3 - x2 * y4 + x3 * y1 + x4 * y2) pow 2 +
   (x1 * y4 + x2 * y3 - x3 * y2 + x4 * y1) pow 2`;;
...
```

Note that formulas depending on specific features of the integers are not always provable by this generic ring procedure. For example we cannot prove:

```
# INT_RING `x pow 2 = &2 ==> F`;;
1 basis elements and 0 critical pairs
Exception: Failure "find".
```

Although it is possible to deal with special cases like this, there can be no general algorithm for testing such properties over the integers: the set of true universally quantified equations over the integers with addition and multiplication is not recursively enumerable. (This follows from Matiyasevich's results on diophantine sets leading to the undecidability of Hilbert's 10th problem.)

## See also
INT_ARITH, INT_ARITH_TAC, int_ideal_cofactors, NUM_RING, REAL_RING, REAL_FIELD.

# INT_SUB_CONV

```
INT_SUB_CONV : conv
```

## Synopsis
Conversion to perform subtraction on two integer literals of type `:int`.

## Description
The call `INT_SUB_CONV ‘c1 - c2‘` where `c1` and `c2` are integer literals of type `:int`, returns `|- c1 - c2 = d` where `d` is the canonical integer literal that is equal to `c1 - c2`. The literals `c1` and `c2` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not the difference of two permitted integer literals of type `:int`.

## Example
```
# INT_SUB_CONV ‘&33 - &77‘;;
val it : thm = |- &33 - &77 = -- &44
```

## See also
INT_REDUCE_CONV, REAL_RAT_SUB_CONV.

# is_abs

```
is_abs : term -> bool
```

## Synopsis
Tests a term to see if it is an abstraction.

## Description
`is_abs ‘\var. t‘` returns `true`. If the term is not an abstraction the result is `false`.

### Failure
Never fails.

### Example

```
# is_abs `\x. x + 1`;;
val it : bool = true

# is_abs `!x. x >= 0`;;
val it : bool = false
```

### See also
mk_abs, dest_abs, is_var, is_const, is_comb.

## isalnum

```
isalnum : string -> bool
```

### Synopsis
Tests if a one-character string is alphanumeric.

### Description
The call `isalnum s` tests whether the first character of string `s` (normally it is the only character) is alphanumeric, i.e. an uppercase or lowercase letter, a digit, an underscore or a prime character.

### Failure
Fails if the string is empty.

### See also
isalpha, isbra, isnum, issep, isspace, issymb.

## isalpha

```
isalpha : string -> bool
```

## Synopsis

Tests if a one-character string is alphabetic.

## Description

The call `isalpha s` tests whether the first character of string `s` (normally it is the only character) is alphabetic, i.e. an uppercase or lowercase letter, an underscore or a prime character.

## Failure

Fails if the string is empty.

## See also

`isalnum, isbra, isnum, issep, isspace, issymb.`

# is_binary

```
is_binary : string -> term -> bool
```

## Synopsis

Tests if a term is an application of a named binary operator.

## Description

The call `is_binary s tm` tests if term `tm` is an instance of a binary operator `(op l) r` where `op` is a constant with name `s`. If so, it returns true; otherwise it returns false. Note that `op` is required to be a constant.

## Failure

Never fails.

## Example

This one succeeds:

```
# is_binary "+" `1 + 2`;;
val it : bool = true
```

but this one fails unless `f` has been declared a constant:

```
# is_binary "f" `f x y`;;
Warning: inventing type variables
val it : bool = false
```

## See also

`dest_binary, is_binop, is_comb, mk_binary.`

# is_binder

`is_binder : string -> term -> bool`

## Synopsis
Tests if a term is a binder construct with named constant.

## Description
The call `is_binder "c" t` tests whether the term `t` has the form of an application of a constant `c` to an abstraction. Note that this has nothing to do with the parsing status of the name `c` as a binder, but only the form of the term.

## Failure
Never fails.

## Example

```
# is_binder "!" `!x. x >= 0`;;
val it : bool = true
```

Note how only the basic logical form is tested, even taking in things that we wouldn't really think of as binders:

```
# is_binder "=" `(=) (\x. x + 1)`;;
val it : bool = true
```

## See also
`dest_binder`, `mk_binder`.

# is_binop

`is_binop : term -> term -> bool`

## Synopsis
Tests if a term is an application of the given binary operator.

## Description

The call `is_binop op t` returns `true` if the term `t` is of the form `(op l) r` for any two terms `l` and `r`, and `false` otherwise.

## Failure

Never fails.

## Example

This is a fairly typical example:

```
# is_binop `(/\)` `p /\ q`;;
val it : bool = true
```

but note that the operator needn't be a constant:

```
  # is_binop `f:num->num->num` `(f:num->num->num) x y`;;
  val it : bool = true
```

## See also

`dest_binary`, `dest_binop`, `is_binary`, `mk_binary`, `mk_binop`.

# isbra

```
isbra : string -> bool
```

## Synopsis

Tests if a one-character string is some kind of bracket.

## Description

The call `isbra s` tests whether the first character of string `s` (normally it is the only character) is a bracket, meaning an opening or closing parenthesis, square bracket or curly brace.

## Failure

Fails if the string is empty.

## See also

`isalnum`, `isalpha`, `isnum`, `issep`, `isspace`, `issymb`.

# is_comb

`is_comb : term -> bool`

## Synopsis
Tests a term to see if it is a combination (function application).

## Description
`is_comb "t1 t2"` returns `true`. If the term is not a combination the result is `false`.

## Failure
Never fails

## Example

```
# is_comb `x + 1`;;
val it : bool = true
# is_comb `T`;;
val it : bool = false
```

## See also
`dest_comb`, `is_var`, `is_const`, `is_abs`, `mk_comb`.

# is_cond

`is_cond : term -> bool`

## Synopsis
Tests a term to see if it is a conditional.

## Description
`is_cond `if t then t1 else t2`` returns `true`. If the term is not a conditional the result is `false`.

## Failure
Never fails.

## See also
`mk_cond`, `dest_cond`.

# is_conj

```
is_conj : term -> bool
```

## Synopsis
Tests a term to see if it is a conjunction.

## Description
`is_conj` `‘t1 /\ t2‘` returns `true`. If the term is not a conjunction the result is `false`.

## Failure
Never fails.

## See also
`dest_conj`, `mk_conj`.

# is_cons

```
is_cons : term -> bool
```

## Synopsis
Tests a term to see if it is an application of `CONS`.

## Description
`is_cons` returns `true` of a term representing a non-empty list. Otherwise it returns `false`.

## Failure
Never fails.

## See also
`dest_cons`, `dest_list`, `is_list`, `mk_cons`, `mk_list`.

# is_const

`is_const : term -> bool`

## Synopsis
Tests a term to see if it is a constant.

## Description
`is_const ‘const:ty‘` returns `true`. If the term is not a constant the result is `false`.

## Failure
Never fails.

## Example

```
# is_const ‘T‘;;
val it : bool = true
# is_const ‘x:bool‘;;
val it : bool = false
```

Note that numerals are not constants; they are composite constructs hidden by prettyprinting:

```
# is_const ‘0‘;;
val it : bool = false
# is_numeral ‘12345‘;;
val it : bool = true
```

## See also
`dest_const`, `is_abs`, `is_comb`, `is_numeral`, `is_var`, `mk_const`.

# is_disj

`is_disj : term -> bool`

## Synopsis
Tests a term to see if it is a disjunction.

## Description

`is_disj ‘t1 \/ t2‘` returns `true`. If the term is not a disjunction the result is `false`.

## Failure

Never fails.

## See also

`dest_disj`, `mk_disj`.

# is_eq

```
is_eq : term -> bool
```

## Synopsis

Tests a term to see if it is an equation.

## Description

`is_eq ‘t1 = t2‘` returns `true`. If the term is not an equation the result is `false`. Note that logical equivalenceis just equality on type `:bool`, even though it is printed as `<=>`.

## Failure

Never fails.

## Example

```
# is_eq ‘2 + 2 = 4‘;;
val it : bool = true

# is_eq ‘p /\ q <=> q /\ p‘;;
val it : bool = true

#  is_eq ‘p ==> p‘;;
val it : bool = false
```

## See also

`dest_eq`, `is_beq`, `mk_eq`.

# is_exists

```
is_exists : term -> bool
```

### Synopsis
Tests a term to see if it as an existential quantification.

### Description
`is_exists ‘?var. t‘` returns `true`. If the term is not an existential quantification the result is `false`.

### Failure
Never fails.

### See also
`dest_exists`, `mk_exists`.

## is_forall

```
is_forall : term -> bool
```

### Synopsis
Tests a term to see if it is a universal quantification.

### Description
`is_forall ‘!var. t‘` returns `true`. If the term is not a universal quantification the result is `false`.

### Failure
Never fails.

### See also
`dest_forall`, `mk_forall`.

## is_gabs

```
is_gabs : term -> bool
```

### Synopsis
Tests if a term is a basic or generalized abstraction.

## Description

The call `is_gabs t` tests if `t` is either a basic logical abstraction (as identified by `is_abs`) or a generalized one (a standard composite logical structure to support a non-variable vastruct). If so, it returns `true`, and otherwise it returns `false`.

## Failure

Never fails.

## Example

This shows that ordinary abstractions are allowed:

```
# is_gabs `\x. x + 1`;;
val it : bool = true
```

while the following shows a more typical case:

```
# is_gabs `\(x,y,z). x + y + z + 1`;;
val it : bool = true
```

## See also

`GEN_BETA_CONV`, `dest_gabs`, `mk_gabs`.

# is_hidden

```
is_hidden : string -> bool
```

## Synopsis

Determines whether a constant is hidden.

## Description

This predicate returns `true` if the named `ML` constant has been hidden by the function `hide_constant`; it returns `false` if the constant is not hidden. Hiding a constant forces the quotation parser to treat the constant as a variable (lexical rules permitting).

## Example

```
# is_hidden "SUC";;
val it : bool = false

# hide_constant "SUC";;
val it : unit = ()

# is_hidden "SUC";;
val it : bool = true
```

## See also

hide_constant, unhide_constant

# is_iff

```
is_iff : term -> bool
```

## Synopsis

Tests if a term is an equation between Boolean terms (iff / logical equivalence).

## Description

Recall that in HOL, the Boolean operation variously called logical equivalence, bi-implication or 'if and only if' (iff) is simply the equality relation on Boolean type. The call `is_iff t` returns `true` if `t` is an equality between terms of Boolean type, and `false` otherwise.

## Failure

Never fails.

## Example

```
# is_iff `p = T`;;
val it : bool = true

# is_iff `p <=> q`;;
val it : bool = true

# is_iff `0 = 1`;;
val it : bool = false
```

## See also

dest_iff, is_eq, mk_iff.

# is_imp

```
is_imp : term -> bool
```

## Synopsis

Tests if a term is an application of implication.

## Description

The call `is_imp t` returns `true` if `t` is of the form `p ==> q` for some `p` and `q`, and returns `false` otherwise.

## Failure

Never fails.

## See also

`dest_imp`.

# is_intconst

```
is_intconst : term -> bool
```

## Synopsis

Tests if a term is an integer literal of type `:int`.

## Description

The call `is_intconst t` tests whether the term `t` is a canonical integer literal of type `:int`, i.e. either '`&n`' for a numeral `n` or '`-- &n`' for a nonzero numeral `n`. If so it returns `true`, otherwise `false`.

## Failure

Never fails.

## Example

```
# is_intconst `-- &3 :int`;;
val it : bool = true
# is_intconst `-- &0 :int`;;
val it : bool = false
```

## See also

`dest_intconst`, `is_realintconst`, `mk_intconst`.

# is_let

```
is_let : term -> bool
```

## Synopsis
Tests a term to see if it is a `let`-expression.

## Description
`is_let ‘let x1 = e1 and ... and xn = en in E‘` returns `true`. If the term is not a `let`-expression of any kind, the result is `false`.

## Failure
Never fails.

## Example

```
# is_let ‘let x = 1 in x + x‘;;
val it : bool = true

# is_let ‘let x = 2 and y = 3 in y + x‘;;
val it : bool = true
```

## See also
`mk_let`, `dest_let`.

# is_list

```
is_list : term -> bool
```

## Synopsis
Tests a term to see if it is a list.

## Description
`is_list` returns `true` of a term representing a list. Otherwise it returns `false`.

## Failure
Never fails.

## See also
dest_cons, dest_list, is_cons, mk_cons, mk_list.

# is_neg

```
is_neg : term -> bool
```

## Synopsis
Tests a term to see if it is a logical negation.

## Description
is_neg `~t` returns true. If the term is not a logical negation the result is false.

## Failure
Never fails.

## See also
dest_neg, mk_neg.

# isnum

```
isnum : string -> bool
```

## Synopsis
Tests if a one-character string is a decimal digit.

## Description
The call isnum s tests whether the first character of string s (normally it is the only character) is a decimal digit.

## Failure
Fails if the string is empty.

## See also
isalnum, isalpha, isbra, issep, isspace, issymb.

# is_numeral

`is_numeral : term -> bool`

## Synopsis
Tests if a term is a natural number numeral.

## Description
When applied to a term, `is_numeral` returns `true` if and only if the term is a canonical natural number numeral (`0`, `1`, `2` etc.)

## Failure
Never fails.

## See also
`dest_numeral`, `is_numeral`.

# is_pair

`is_pair : term -> bool`

## Synopsis
Tests a term to see if it is a pair.

## Description
`is_pair ‘(t1,t2)‘` returns `true`. If the term is not a pair the result is `false`.

## Failure
Never fails.

## Example

```
# is_pair ‘1,2,3‘;;
val it : bool = true

# is_pair ‘[1;2;3]‘;;
val it : bool = false
```

## See also
`dest_pair`, `is_cons`, `mk_pair`.

# ISPEC

```
ISPEC : term -> thm -> thm
```

## Synopsis
Specializes a theorem, with type instantiation if necessary.

## Description
This rule specializes a quantified variable as does `SPEC`; it differs from it in also instantiating the type if needed:

```
     A |- !x:ty.tm
  -----------------------  ISPEC `t:ty'`
      A |- tm[t/x]
```

(where `t` is free for `x` in `tm`, and `ty'` is an instance of `ty`).

## Failure
`ISPEC` fails if the input theorem is not universally quantified, if the type of the given term is not an instance of the type of the quantified variable, or if the type variable is free in the assumptions.

## Example

```
# ISPEC `0` EQ_REFL;;
val it : thm = |- 0 = 0
```

Note that the corresponding call to `SPEC` would fail because of the type mismatch:

```
# SPEC `0` EQ_REFL;;
Exception: Failure "SPEC".
```

## See also
`INST`, `INST_TYPE`, `ISPECL`, `SPEC`, `type_match`.

# ISPECL

```
ISPECL : term list -> thm -> thm
```

## Synopsis

Specializes a theorem zero or more times, with type instantiation if necessary.

## Description

ISPECL is an iterative version of ISPEC

```
       A |- !x1...xn.t
----------------------------  ISPECL [`t1`,...,`tn`]
  A |- t[t1,...tn/x1,...,xn]
```

(where ti is free for xi in tm).

## Failure

ISPECL fails if the list of terms is longer than the number of quantified variables in the term, if the type instantiation fails, or if the type variable being instantiated is free in the assumptions.

## Example

```
# ISPECL [`x:num`; `2`] EQ_SYM_EQ;;
val it : thm = |- x = 2 <=> 2 = x
```

Note that the corresponding call to SPECL would fail because of the type mismatch:

```
# SPECL [`x:num`; `2`] EQ_SYM_EQ;;
Exception: Failure "SPECL".
```

## See also

INST_TYPE, INST, ISPEC, SPEC, SPECL, type_match.

# is_prefix

```
is_prefix : string -> bool
```

## Synopsis

Tests if an identifier has prefix status.

## Description

Certain identifiers c have prefix status, meaning that combinations of the form c f x will be parsed as c (f x) rather than the usual (c f) x. The call is_prefix "c" tests if c is one of those identifiers.

## Failure
Never fails.

## See also
`parse_as_prefix`, `prefixes`, `unparse_as_prefix`.

# is_ratconst

```
is_ratconst : term -> bool
```

## Synopsis
Tests if a term is a canonical rational literal of type `:real`.

## Description
The call `is_ratconst t` tests whether the term `t` is a canonical rational literal of type `:real`. This means an integer literal `&n` for numeral `n`, `-- &n` for a nonzero numeral `n`, or a ratio `&p / &q` or `-- &p / &q` where `p` is nonzero, `q > 1` and `p` and `q` share no common factor. If so, `is_ratconst` returns `true`, and otherwise `false`.

## Failure
Never fails.

## Example

```
# is_ratconst `&22 / &7`;;
val it : bool = true
# is_ratconst `&4 / &2`;;
val it : bool = false
```

## See also
`is_realintconst`, `rat_of_term`, `REAL_RAT_REDUCE_CONV`, `term_of_rat`.

# is_realintconst

```
is_realintconst : term -> bool
```

## Synopsis

Tests if a term is an integer literal of type `:real`.

## Description

The call `is_realintconst t` tests whether the term `t` is a canonical integer literal of type `:real`, i.e. either '`&n`' for a numeral `n` or '`-- &n`' for a nonzero numeral `n`. If so it returns `true`, otherwise `false`.

## Failure

Never fails.

## Example

```
# is_realintconst `-- &3 :real`;;
val it : bool = true
# is_realintconst `&1 :int`;;
val it : bool = false
```

## See also

`dest_realintconst, is_intconst, is_ratconst, mk_realintconst.`

# is_reserved_word

`is_reserved_word : string -> bool`

## Synopsis

Tests if a string is one of the reserved words.

## Description

Certain identifiers in HOL are reserved, e.g. '`if`', '`let`' and '`|`', meaning that they are special to the parser and cannot be used as ordinary identifiers. The call `is_reserved_word s` tests if the string `s` is one of them.

## Failure

Never fails.

## See also

`reserved_words, reserve_words, unreserve_words.`

# is_select

`is_select : term -> bool`

## Synopsis
Tests a term to see if it is a choice binding.

## Description
`is_select` `` `@var. t` `` returns `true`. If the term is not an epsilon-term the result is `false`.

## Failure
Never fails.

## See also
`mk_select`, `dest_select`.

# issep

`issep : string -> bool`

## Synopsis
Tests if a one-character string is a separator.

## Description
The call `issep s` tests whether the first character of string `s` (normally it is the only character) is one of the separators '`,`' or '`;`'.

## Failure
Fails if the string is empty.

## See also
`isalnum`, `isalpha`, `isbra`, `isnum`, `isspace`, `issymb`.

# is_setenum

`is_setenum : term -> bool`

### Synopsis
Tests if a term is a set enumeration.

### Description
When applied to a term that is an explicit set enumeration `‘{t1,...,tn}‘`, the function `is_setenum` returns `true`; otherwise it returns `false`.

### Failure
Never fails.

### Example

```
# is_setenum ‘1 INSERT 2 INSERT {\small\verb%%}‘;;
val it : bool = true

# is_setenum ‘{1,2,3,4,1,2,3,4}‘;;
val it : bool = true

# is_setenum ‘1 INSERT 2 INSERT s‘;;
val it : bool = false
```

### See also
`dest_setenum`, `mk_fset`, `mk_setenum`.

## isspace

```
isspace : string -> bool
```

### Synopsis
Tests if a one-character string is some kind of space.

### Description
The call `isspace s` tests whether the first character of string `s` (normally it is the only character) is a ‘space’ of some kind, including tab and newline.

### Failure
Fails if the string is empty.

### See also
`isalnum`, `isalpha`, `isbra`, `isnum`, `issep`, `issymb`.

# issymb

`issymb : string -> bool`

## Synopsis

Tests if a one-character string is a symbol other than bracket or separator.

## Description

The call `issymb s` tests whether the first character of string `s` (normally it is the only character) is "symbolic". This means that it is one of the usual ASCII characters but is not alphanumeric, not an underscore or prime character, and is also not one of the two separators '`,`' or '`;`' nor any bracket, parenthesis or curly brace. More explicitly, the set of symbolic characters is:

```
\ ! @ # $ % ^ & * - + | \ \ < = > / ? ~ . :
```

## Failure

Fails if the string is empty.

## See also

`isalnum, isalpha, isbra, isnum, issep, isspace.`

# is_type

`is_type : hol_type -> bool`

## Synopsis

Tests whether a type is an instance of a type constructor.

## Description

`is_type ty` returns `true` if `ty` is a base type or constructed by an outer type constructor, and `false` if it is a type variable.

## Failure

Never fails.

## Example

```
# is_type `:bool`;;
val it : bool = true

# is_type `:bool->int`;;
val it : bool = true

# is_type `:Tyvar`;;
val it : bool = false
```

## See also

get_type_arity, is_vartype.

# is_uexists

is_uexists : term -> bool

## Synopsis

Tests if a term is of the form 'there exists a unique ...'

## Description

If t has the form ?!x. p[x] (there exists a unique x such that p[x] then is_uexists t returns true, otherwise false.

## Failure

Never fails.

## See also

dest_uexists, is_exists, is_forall.

# is_undefined

is_undefined : ('a, 'b) func -> bool

## Synopsis

Tests if a finite partial function is defined nowhere.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The predicate `is_undefined` tests if the argument is the completely undefined function.

## Failure

Never fails.

## Example

```
# let x = undefined and y = (1 |=> 2);;
val x : ('a, 'b) func = <func>
val y : (int, int) func = <func>

# is_undefined x;;
val it : bool = true

# is_undefined y;;
val it : bool = false
```

## See also

`|->`, `|=>`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# is_var

```
is_var : term -> bool
```

## Synopsis

Tests a term to see if it is a variable.

## Description

`is_var `var:ty`` returns `true`. If the term is not a variable the result is `false`.

## Failure

Never fails.

## Example

```
# is_var `x:bool`;;
val it : bool = true
# is_var `T`;;
val it : bool = false
```

## See also

mk_var, dest_var, is_const, is_comb, is_abs.

# is_vartype

```
is_vartype : hol_type -> bool
```

## Synopsis

Tests a type to see if it is a type variable.

## Description

Returns `true` if applied to a type variable. For types that are not type variables it returns `false`.

## Failure

Never fails.

## Example

```
# is_vartype `:A`;;
val it : bool = true

# is_vartype `:bool`;;
val it : bool = false

# is_vartype (mk_vartype "bool");;
val it : bool = true
```

## See also

mk_vartype, dest_vartype.

# ITAUT

```
ITAUT : term -> thm
```

## Synopsis

Attempt to prove term using intuitionistic first-order logic.

## Description

The call `ITAUT `p`` attempts to prove `p` using a basic tableau-type proof search for intuitionistic first-order logic. The restriction to intuitionistic logic means that no principles such as the "law of the excluded middle" or "law of double negation" are used.

## Failure

Fails if the goal is non-Boolean. May also fail if it's unprovable, though more usually this results in indefinite looping.

## Example

This is intuitionistically valid, so it works:

```
# ITAUT `~(~(~p)) ==> ~p`;;
...
val it : thm = |- ~ ~ ~p ==> ~p
```

whereas this, one of the main non-intuitionistic principles, is not:

```
# ITAUT `~(~p) ==> p`;;
Searching with limit 0
Searching with limit 1
Searching with limit 2
Searching with limit 3
...
```

so the procedure loops; you can as usual terminate such loops with control-C.

## Comments

Normally, first-order reasoning should be performed by `MESON[]`, which is much more powerful, complete for all classical logic, and handles equality. The function `ITAUT` is mainly for "bootstrapping" purposes. Nevertheless it may sometimes be intellectually interesting to see that certain logical formulas are provable intuitionistically.

## See also

`BOOL_CASES_TAC`, `ITAUT_TAC`, `MESON`, `MESON_TAC`.

# ITAUT_TAC

```
ITAUT_TAC : tactic
```

## Synopsis

Simple intuitionistic logic prover.

## Description

The tactic ITAUT attempts to prove the goal using a basic tableau-type proof search for intuitionistic first-order logic. The restriction to intuitionistic logic means that no principles such as the "law of the excluded middle" or "law of double negation" are used.

## Failure

May fail if the goal is unprovable, e.g. for purely propositional problems. For unsolvable problems with quantifiers it usually just loops.

## Example

Suppose we try to prove the logical equivalence of "contraposition", already embedded in the pre-proved theorem CONTRAPOS_THM:

```
# g `!p q. (p ==> q) <=> (~q ==> ~p)`;;
```

by splitting it into two subgoals:

```
# e(REPEAT GEN_TAC THEN EQ_TAC);;
val it : goalstack = 2 subgoals (2 total)

`(~q ==> ~p) ==> p ==> q`

`(p ==> q) ==> ~q ==> ~p`
```

The first subgoal (printed at the bottom) can be solved by ITAUT_TAC, indicating that it's intuitionistically valid:

```
# e ITAUT_TAC;;
...
val it : goalstack = 1 subgoal (1 total)

`(~q ==> ~p) ==> p ==> q`
```

but the other one isn't, though it is solvable by full classical logic:

```
# e(MESON_TAC[]);;
val it : goalstack = No subgoals
```

## Comments

Normally, first-order reasoning should be performed by `MESON_TAC[]`, which is much more powerful, complete for all classical logic, and handles equality. The function `ITAUT_TAC` is mainly for "bootstrapping" purposes. Nevertheless it may sometimes be intellectually interesting to see that certain logial formulas are provable intuitionistically.

## See also

`ITAUT`, `MESON_TAC`.

# `it`

```
it : 'a
```

## Synopsis

Binds the value of the last expression evaluated at top level.

## Description

The identifier `it` is bound to the value of the last expression evaluated at top level. Declarations do not effect the value of `it`.

## Example

```
# 2 + 3;;
val it : int = 5
# let x = 2*3;;
val x : int = 6
# it;;
val it : int = 5
# it + 12;;
val it : int = 17
```

## Uses

Used in evaluating expressions that require the value of the last evaluated expression.

# itlist2

`itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c`

## Synopsis

Applies a paired function between adjacent elements of 2 lists.

## Description

`itlist2 f ([x1;...;xn],[y1;...;yn]) z` returns

```
f x1 y1 (f x2 y2 ... (f xn yn z)...)
```

It returns `z` if both lists are empty.

## Failure

Fails if the two lists are of different lengths.

## Example

This takes a 'dot product' of two vectors of integers:

```
# let dot v w = itlist2 (fun x y z -> x * y + z) v w 0;;
val dot : int list -> int list -> int = <fun>
# dot [1;2;3] [4;5;6];;
val it : int = 32
```

## See also

`itlist`, `rev_itlist`, `end_itlist`, `uncurry`.

# itlist

```
itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

## Synopsis

List iteration function. Applies a binary function between adjacent elements of a list.

## Description

`itlist f [x1;...;xn] y` returns

```
f x1 (f x2 ... (f xn y)...)
```

It returns `y` if list is empty.

## Failure

Never fails.

## Example

```
# itlist (+) [1;2;3;4;5] 0;;
val it : int = 15
# itlist (+) [1;2;3;4;5] 6;;
val it : int = 21
```

## See also

`rev_itlist`, `end_itlist`.

# ++

```
(++) : ('a -> 'b * 'c) -> ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
```

## Synopsis

Sequentially compose two parsers.

## Description

If `p1` and `p2` are two parsers, `p1 ++ p2` is a new parser that parses as much of the input as possible using `p1` and then as much of what remains using `p2`, returning the pair of parse results and the unparsed input.

### Failure
Never fails.

### Comments
This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

### See also
`++`, `>>`, `||`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

## K

```
K : 'a -> 'b -> 'a
```

### Synopsis
Forms a constant function: `(K x) y = x`.

### Failure
Never fails.

### See also
`C`, `F_F`, `I`, `o`, `W`.

## LABEL_TAC

```
LABEL_TAC : string -> thm_tactic
```

### Synopsis
Add an assumption with a named label to a goal.

### Description
Given a theorem `th`, the tactic `LABEL_TAC "name" th` will add `th` as a new hypothesis, just as `ASSUME_TAC` does, but will also give it `name` as a label. The name will show up when

the goal is printed, and can be used to refer to the theorem in tactics like `USE_THEN` and `REMOVE_THEN`.

## Failure

Never fails, though may be invalid if the theorem has assumptions that are not a subset of those in the goal, up to alpha-equivalence.

## Example

Suppose we want to prove that a binary relation `<<=` that is antisymmetric and has a strong wellfoundedness property is also antisymmetric and transitive, and hence a wellorder:

```
# g `(!x y. x <<= y /\ y <<= x ==> x = y) /\
     (!s. ~(s = {}) ==> ?a:A. a IN s /\ !x. x IN s ==> a <<= x)
     ==> (!x y. x <<= y \/ y <<= x) /\
         (!x y z. x <<= y /\ y <<= z ==> x <<= z)`;;
```

We might start by putting the two hypotheses on the assumption list with intuitive names:

```
# e(DISCH_THEN(CONJUNCTS_THEN2 (LABEL_TAC "antisym") (LABEL_TAC "wo")));;
val it : goalstack = 1 subgoal (1 total)

 0 [`!x y. x <<= y /\ y <<= x ==> x = y`] (antisym)
 1 [`!s. ~(s = {}) ==> (?a. a IN s /\ (!x. x IN s ==> a <<= x))`] (wo)

`(!x y. x <<= y \/ y <<= x) /\ (!x y z. x <<= y /\ y <<= z ==> x <<= z)`
```

Now we break down the goal a bit

```
# e(REPEAT STRIP_TAC);;
val it : goalstack = 2 subgoals (2 total)

 0 [`!x y. x <<= y /\ y <<= x ==> x = y`] (antisym)
 1 [`!s. ~(s = {}) ==> (?a. a IN s /\ (!x. x IN s ==> a <<= x))`] (wo)
 2 [`x <<= y`]
 3 [`y <<= z`]

`x <<= z`

 0 [`!x y. x <<= y /\ y <<= x ==> x = y`] (antisym)
 1 [`!s. ~(s = {}) ==> (?a. a IN s /\ (!x. x IN s ==> a <<= x))`] (wo)

`x <<= y \/ y <<= x`
```

We want to specialize the wellordering assumption to an appropriate set for each case, and we can identify it using the label `wo`; the problem is then simple set-theoretic reasoning:

```
# e(USE_THEN "wo" (MP_TAC o SPEC `{\small\verb%x:A,y:A%}`) THEN SET_TAC[]);;
...
val it : goalstack = 1 subgoal (1 total)

 0 [`!x y. x <<= y /\ y <<= x ==> x = y`] (antisym)
 1 [`!s. ~(s = {}) ==> (?a. a IN s /\ (!x. x IN s ==> a <<= x))`] (wo)
 2 [`x <<= y`]
 3 [`y <<= z`]

`x <<= z`
```

Similarly for the other one:

```
# e(USE_THEN "wo" (MP_TAC o SPEC `{\small\verb%x:A,y:A,z:A%}`) THEN SET_TAC[]);;
...
val it : goalstack = No subgoals
```

## Uses

Convenient for referring to an assumption explicitly, just as in mathematics books one sometimes marks a theorem with an asterisk or dagger, then refers to it using that symbol.

## Comments

There are other ways of identifying assumptions than by label, but they are not always convenient. For example, explicitly doing `ASSUME `asm`` is cumbersome if `asm` is large, and using its number in the assumption list can make proofs very brittle under later changes.

## See also

`ASSUME_TAC`, `REMOVE_THEN`, `USE_THEN`.

# LAMBDA_ELIM_CONV

```
LAMBDA_ELIM_CONV : conv
```

## Synopsis

Eliminate lambda-terms that are not part of quantifiers from Boolean term.

## Description

When applied to a Boolean term, `LAMBDA_ELIM_CONV` returns an equivalent version with 'bare' lambda-terms (those not part of quantifiers) removed. They are replaced with new 'function' variables and suitable hypotheses about them; for example a lambda-term `\x. t[x]` is replaced by a function `f` with an additional hypothesis `\all{\small\verbx`

### Failure
Never fails.

### Example

```
#  LAMBDA_ELIM_CONV `MAP (\x. x + 1) l = l'`;;
val it : thm =
  |- MAP (\x. x + 1) l = l' <=>
     (!_73141. (!x. _73141 x = x + 1) ==> MAP _73141 l = l')
```

### Uses
This is mostly intended for normalization prior to automated proof procedures, and is used by `MESON`, for example. However, it may sometimes be useful in itself.

### See also
`SELECT_ELIM_TAC, CONDS_ELIM_CONV.`

## LAND_CONV

```
LAND_CONV : conv -> conv
```

### Synopsis
Apply a conversion to left-hand argument of binary operator.

### Description
If `c` is a conversion where `c `l`` gives `|- l = l'`, then `LAND_CONV c `op l r`` gives `|- op l r = op l' r`

### Failure
Fails if the underlying conversion does or returns an inappropriate theorem (i.e. is not really a conversion).

### Example

```
# LAND_CONV NUM_ADD_CONV `(2 + 2) + (2 + 2)`;;
val it : thm = |- (2 + 2) + 2 + 2 = 4 + 2 + 2
```

### See also
`ABS_CONV, COMB_CONV, COMB_CONV2, RAND_CONV, RATOR_CONV, SUB_CONV.`

# last

`last : 'a list -> 'a`

## Synopsis
Computes the last element of a list.

## Description
`last [x1;...;xn]` returns `xn`.

## Failure
Fails with `last` if the list is empty.

## See also
`butlast, hd, tl, el.`

# lcm_num

`lcm_num : num -> num -> num`

## Synopsis
Computes lowest common multiple of two unlimited-precision integers.

## Description
The call `lcm_num m n` for two unlimited-precision (type `num`) integers `m` and `n` returns the (positive) lowest common multiple of `m` and `n`. If either `m` or `n` (or both) are both zero, it returns zero.

## Failure
Fails if either number is not an integer (the type `num` supports arbitrary rationals).

## Example

```
# lcm_num (Int 35) (Int(-77));;
val it : num = 385
```

## See also
`gcd, gcd_num.`

# leftbin

`leftbin : (’a -> ’b * ’c) -> (’c -> ’d * ’a) -> (’d -> ’b -> ’b -> ’b) -> string -> ’a -`

## Synopsis

Parses iterated left-associated binary operator.

## Description

If `p` is a parser for “items” of some kind, `s` is a parser for some “separator”, `c` is a ‘constructor’ function taking an element as parsed by `s` and two other elements as parsed by `p` and giving a new such element, and `e` is an error message, then `leftbin p s c e` will parse an iterated sequence of items by `p` and separated by something parsed with `s`. It will repeatedly apply the constructor function `c` to compose these elements into one, associating to the left. For example, the input:

```
<p1> <s1> <p2> <s2> <p3> <s3> <p4>
```

meaning successive segments `pi` that are parsed by `p` and `sj` that are parsed by `s`, will result in

```
c (c s2 (c s1 p1 p2) p3) p4
```

## Failure

The call `leftbin p s c e` never fails, though the resulting parser may.

## Comments

This is one of a suite of combinators for manipulating “parsers”. A parser is simply a function whose OCaml type is some instance of `:(’a)list -> ’b * (’a)list`. The function should take a list of objects of type `:’a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `listof`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

# LE_IMP

`LE_IMP : thm -> thm`

### Synopsis
Perform transitivity chaining for non-strict natural number inequality.

### Description
When applied to a theorem `A |- s <= t` where `s` and `t` have type `num`, the rule `LE_IMP` returns `A |- !x1...xn z. t <= z ==> s <= z`, where `z` is some variable and the `x1,...,xn` are free variables in `s` and `t`.

### Failure
Fails if applied to a theorem whose conclusion is not of the form `‘s <= t‘` for some natural number terms `s` and `t`.

### Example

```
# LE_IMP (ARITH_RULE ‘n <= SUC(m + n)‘);;
val it : thm = |- !m n p. SUC (m + n) <= p ==> n <= p
```

### Uses
Can make transitivity chaining in goals easier, e.g. by `FIRST_ASSUM(MATCH_MP_TAC o LE_IMP)`.

### See also
`ARITH_RULE`, `REAL_LE_IMP`, `REAL_LET_IMP`.

## length

```
length : 'a list -> int
```

### Synopsis
Computes the length of a list: `length [x1;...;xn]` returns `n`.

### Failure
Never fails.

## <=?

```
(<=?) : 'a -> 'a -> bool
```

### Synopsis
Reflexive short-cutting inequality test.

## Description

This is functionally identical to the OCaml polymorphic inequality test `<=` except that it is total (hence reflexive) even on floating-point NaNs. More importantly, it will more efficiently short-cut comparisons of large data structures where subcomponents are identical (pointer equivalent).

## Failure

May fail when applied to functions.

## Example

```
# let x = 0.0 /. 0.0;;
val x : float = nan
# x <= x;;
val it : bool = false
# x <=? x;;
val it : bool = true
```

## See also

`=?`, `<?`, `>?`, `>=?`.

# let_CONV

```
let_CONV : term -> thm
```

## Synopsis

Evaluates `let`-terms in the HOL logic.

## Description

The conversion `let_CONV` implements evaluation of object-language `let`-terms. When applied to a `let`-term of the form:

```
let v1 = t1 and ... and vn = tn in t
```

where `v1`, ..., `vn` are variables, `let_CONV` proves and returns the theorem:

```
|- (let v1 = t1 and ... and vn = tn in t) = t[t1,...,tn/v1,...,vn]
```

where `t[t1,...,tn/v1,...,vn]` denotes the result of substituting `ti` for `v1` in parallel in `t`, with automatic renaming of bound variables to prevent free variable capture.

`let_CONV` also works on `let`-terms that bind tuples of variables to tuples of values. That is, if `<tup>` is an arbitrarily-nested tuple of distinct variables `v1`, ..., `vn` and `<val>` is a structurally similar tuple of values, that is <val> equals <tup>[t1,...,tn/v1,...,vn] for some terms t1, ..., tn, then:

```
let_CONV `let <tup> = <val> in t`
```

returns

```
|- (let <tup> = <val> in t) = t[t1,...,tn/v1,...,vn]
```

That is, the term ti is substituted for the corresponding variable vi in t. This form of let-reduction also works with simultaneous binding of tuples using and.

## Failure

let_CONV tm fails if tm is not a reducible let-term of one of the forms specified above.

## Example

A simple example of the use of let_CONV to eliminate a single local variable is the following:

```
# let_CONV `let x = 1 in x+y`;;
val it : thm = |- (let x = 1 in x + y) = 1 + y
```

and an example showing a tupled binding is:

```
# let_CONV `let (x,y) = (1,2) in x+y`;;
val it : thm = |- (let x,y = 1,2 in x + y) = 1 + 2
```

Simultaneous introduction of two bindings is illustrated by:

```
# let_CONV `let x = 1 and y = 2 in x + y + z`;;
val it : thm = |- (let x = 1 and y = 2 in x + y + z) = 1 + 2 + z
```

## See also

BETA_CONV, PAIRED_BETA_CONV.

# LET_TAC

```
LET_TAC : tactic
```

## Synopsis

Eliminates a let binding in a goal by introducing equational assumptions.

## Description

Given a goal A ?- t where t contains a free let-expression let x1 = E1 and ... let xn = En in E, the tactic LET_TAC replaces that subterm by simply E but adds new assumptions E1 = x1, ..., En = xn. That is, the local let bindings are replaced with new assumptions, put in reverse order so that ASM_REWRITE_TAC will not immediately expand them. In cases where the term contains several let-expression candidates, a topmost one will be selected. In particular, if let-expressions are nested, the outermost one will be handled.

## Failure

Fails if the goal contains no eligible let-term.

## Example

```
# g `let x = 2 and y = 3 in x + 1 <= y`;;
val it : goalstack = 1 subgoal (1 total)

`let x = 2 and y = 3 in x + 1 <= y`

# e LET_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [`2 = x`]
 1 [`3 = y`]

`x + 1 <= y`
```

## See also

ABBREV_TAC, EXPAND_TAC, let_CONV.

# lex

```
lex : string list -> lexcode list
```

## Synopsis

Lexically analyze an input string.

## Description

The function lex expects a list of single-character strings representing input (as produced by explode, for example) and analyzes it into a sequence of tokens according to HOL Light lexical conventions. A token is either Ident "s" or Resword "s"; in each case this encodes a string but in the latter case indicates that the string is a reserved word.

Lexical analysis essentially regards any number of alphanumeric characters (see isalnum) or any number of symbolic characters (see issymb) as a single token, except that certain brackets (see isbra) are only allowed to be single-character tokens and other separators (see issep) can only be combined with multiple instances of themselves not other characters. Whitespace including spaces, tabs and newlines (see isspace) is eliminated and serves only to separate tokens that would otherwise be one. Comments introduced by the comment token (see `comment_token`) are removed.

## Failure

Fails if the input is highly malformed, e.g. contains illegal characters.

## Example

```
# lex(explode "if p+1=2 then x + 1 else y - 1");;
val it : lexcode list =
  [Resword "if"; Ident "p"; Ident "+"; Ident "1"; Ident "="; Ident "2";
   Resword "then"; Ident "x"; Ident "+"; Ident "1"; Resword "else";
   Ident "y"; Ident "-"; Ident "1"]
```

## See also

`comment_token`, `explode`, `isalnum`, `isbra`, `issep`, `isspace`, `issymb`,
`is_reserved_word`, `parse_term`, `parse_type`.

# lhand

```
lhand : term -> term
```

## Synopsis

Take left-hand argument of a binary operator.

## Description

When applied to a term `t` that is an application of a binary operator to two arguments, i.e. is of the form `(op l) r`, the call `lhand t` will return the left-hand argument `l`. The terms `op` and `r` are arbitrary, though in many applications `op` is a constant such as addition or equality.

## Failure

Fails if the term is not of the indicated form.

## Example

```
# lhand `1 + 2`;;
val it : term = `1`

# lhand `2 + 2 = 4`;;
val it : term = `2 + 2`

# lhand `f x y z`;;
Warning: inventing type variables
val it : term = `y`

# lhand `if p then q else r`;;
Warning: inventing type variables
val it : term = `q`
```

## Comments

On equations, `lhand` has the same effect as `lhs`, but may be slightly quicker because it does not check whether the operator `op` is indeed the equality constant.

## See also

`lhs`, `rand`, `rhs`.

# lhs

```
lhs : term -> term
```

## Synopsis

Returns the left-hand side of an equation.

## Description

`` lhs `t1 = t2` `` returns `` `t1` ``.

## Failure

Fails with `lhs` if the term is not an equation.

## Example

```
# lhs `2 + 2 = 4`;;
val it : term = `2 + 2`
```

## See also

`dest_eq`, `lhand`, `rand`, `rhs`.

# lift_function

```
lift_function : thm -> thm * thm -> string -> thm -> thm * thm
```

## Synopsis

Lift a function on representing type to quotient type of equivalence classes.

## Description

Suppose type `qty` is a quotient type of `rty` under an equivalence relation `R:rty->rty->bool`, as defined by `define_quotient_type`, and `f` is a function `f:ty1->...->tyn->ty`, some `tyi` being the representing type `rty`. The term `lift_function` should be applied to (i) a theorem of the form `|- (?x. r = R x) <=> rep(abs r) = r` as returned by `define_quotient_type`, (ii) a pair of theorems asserting that `R` is reflexive and transitive, (iii) a desired name for the counterpart of `f` lifted to the type of equivalence classes, and (iv) a theorem asserting that `f` is "welldefined", i.e. respects the equivalence class. This last theorem essentially asserts that the value of `f` is independent of the choice of representative: any `R`-equivalent inputs give an equal output, or an `R`-equivalent one. Syntactically, the welldefinedness theorem should be of the form:

```
|- !x1 x1' .. xn xn'. (x1 == x1') /\ ... /\ (xn == xn')
                      ==> (f x1 .. xn == f x1' .. f nx')
```

where each `==` may be either equality or the relation `R`, the latter of course only if the type of that argument is `rty`. The reflexivity and transitivity theorems should be

```
|- !x. R x x
```

and

```
|- !x y z. R x y /\ R y z ==> R x z
```

It returns two theorems, a definition and a consequential theorem that can be used by `lift_theorem` later.

## Failure

Fails if the theorems are malformed or if there is already a constant of the given name.

## Example

Suppose that we have defined a type of finite multisets as in the documentation for `define_quotient_type`, based on the equivalence relation `multisame` on lists. First we prove that the equivalence relation `multisame` is indeed reflexive and transitive:

```
# let MULTISAME_REFL,MULTISAME_TRANS = (CONJ_PAIR o prove)
   (`(!l:(A)list. multisame l l) /\
     (!l1 l2 l3:(A)list.
          multisame l1 l2 /\ multisame l2 l3 ==> multisame l1 l3)`,
    REWRITE_TAC[multisame] THEN MESON_TAC[]);;
```

We would like to define the multiplicity of an element in a multiset. First we define this notion on the representing type of lists:

```
# let listmult = new_definition
   `listmult a l = LENGTH (FILTER (\x:A. x = a) l)`;;
```

and prove that it is welldefined. Note that the second argument is the only one we want to lift to the quotient type, so that's the only one for which we use the relation `multisame`. For the first argument and the result we only use equality:

```
# let LISTMULT_WELLDEF = prove
   (`!a a':A l l'.
      a = a' /\ multisame l l' ==> listmult a l = listmult a' l'`,
    SIMP_TAC[listmult; multisame]);;
```

Now we can lift it to a multiplicity function on the quotient type:

```
# let multiplicity,multiplicity_th =
    lift_function multiset_rep (MULTISAME_REFL,MULTISAME_TRANS)
    "multiplicity" LISTMULT_WELLDEF;;
val multiplicity : thm =
  |- multiplicity a l = (@u. ?l. listmult a l = u /\ list_of_multiset l l)
val multiplicity_th : thm =
  |- listmult a l = multiplicity a (multiset_of_list (multisame l))
```

Another example is the 'union' of multisets, which we can consider as the lifting of the `APPEND` operation on lists, which we show is welldefined:

```
# let APPEND_WELLDEF = prove
   (`!l l' m m' :A list.
       multisame l l' /\ multisame m m'
       ==> multisame (APPEND l m) (APPEND l' m')`,
    SIMP_TAC[multisame; FILTER_APPEND]);;
```

and lift as follows:

```
# let munion,munion_th =
    lift_function multiset_rep (MULTISAME_REFL,MULTISAME_TRANS)
    "munion" APPEND_WELLDEF;;
val munion : thm =
  |- munion l m =
     multiset_of_list
     (\u. ?l m.
              multisame (APPEND l m) u /\
              list_of_multiset l l /\
              list_of_multiset m m)
val munion_th : thm =
  |- multiset_of_list (multisame (APPEND l m)) =
     munion (multiset_of_list (multisame l)) (multiset_of_list (multisame m))
```

For continuation of this example, showing how to lift theorems from the representing functions to the functions on the quotient type, see the documentation entry for `lift_theorem`.

## Comments

If, as in these examples, the representing type is parametrized by type variables, make sure that the same type variables are used consistently in the various theorems.

## See also

`define_quotient_type`, `lift_theorem`.

# lift_theorem

```
lift_theorem : thm * thm -> thm * thm * thm -> thm list -> thm -> thm
```

## Synopsis

Lifts a theorem to quotient type from representing type.

## Description

The function `lift_theorem` should be applied (i) a pair of type bijection theorems as returned by `define_quotient_type` for equivalence classes over a binary relation `R`, (ii) a triple of theorems asserting that the relation `R` is reflexive, symmetric and transitive in exactly the following form:

```
|- !x. R x x
|- !x y. R x y <=> R y x
|- !x y z. R x y /\ R y z ==> R x z
```

and (iii) the list of theorems returned as the second component of the pairs from `lift_function` for all functions that should be mapped. Finally, it is then applied to a theorem about the representing type. It automatically maps it over to the quotient type, appropriately modifying quantification over the representing type into quantification over the new quotient type, and replacing functions over the representing type with their corresponding lifted counterparts. Note that all variables should be bound by quantifiers; these may be existential or universal but if any types involve the representing type `rty` it must be just `rty` and not a composite or higher-order type such as `rty->rty` or `rty#num`.

## Failure

Fails if any of the input theorems are malformed (e.g. symmetry stated with implication instead of equivalence) or fail to correspond (e.g. different polymorphic type variables in the type bijections and the equivalence theorem). Otherwise it will not fail, but if used improperly may not map the theorem across cleanly.

## Example

This is a continuation of the example in the documentation entries for `define_quotient_type` and `lift_function`, where a type of finite multisets is defined as the quotient of the type of lists by a suitable equivalence relation `multisame`. We can take the theorems asserting that this is indeed reflexive, symmetric and transitive:

```
# let [MULTISAME_REFL;MULTISAME_SYM;MULTISAME_TRANS] = (CONJUNCTS o prove)
   (`(!l:(A)list. multisame l l) /\
     (!l l':(A)list. multisame l l' <=> multisame l' l) /\
     (!l1 l2 l3:(A)list.
          multisame l1 l2 /\ multisame l2 l3 ==> multisame l1 l3)`,
    REWRITE_TAC[multisame] THEN MESON_TAC[]);;
```

and can now lift theorems. For example, we know that `APPEND` is itself associative, and so in particular:

```
# let MULTISAME_APPEND_ASSOC = prove
   (`!l m n. multisame (APPEND l (APPEND m n)) (APPEND (APPEND l m) n)`,
    REWRITE_TAC[APPEND_ASSOC; MULTISAME_REFL]);;
```

and we can easily show how list multiplicity interacts with APPEND:

```
# let LISTMULT_APPEND = prove
   (`!a l m. listmult a (APPEND l m) = listmult a l + listmult a m`,
    REWRITE_TAC[listmult; LENGTH_APPEND; FILTER_APPEND]);;
```

These theorems and any others like them can now be lifted to equivalence classes:

```
# let [MULTIPLICITY_MUNION;MUNION_ASSOC] =
      map (lift_theorem (multiset_abs,multiset_rep)
                        (MULTISAME_REFL,MULTISAME_SYM,MULTISAME_TRANS)
                        [multiplicity_th; munion_th])
          [LISTMULT_APPEND; MULTISAME_APPEND_ASSOC];;
val ( MULTIPLICITY_MUNION ) : thm =
  |- !a l m.
         multiplicity a (munion l m) = multiplicity a l + multiplicity a m
val ( MUNION_ASSOC ) : thm =
  |- !l m n. munion l (munion m n) = munion (munion l m) n
```

## See also
define_quotient_type, lift_function.

# LIST_CONV

```
LIST_CONV : conv -> conv
```

## Synopsis
Apply a conversion to each element of a list.

## Description
If cnv `ti` returns |- ti = ti' for i ranging from 1 to n, then LIST_CONV cnv `[t1; ...; tn]` returns |- [t1; ...; tn] = [t1'; ...; tn'].

## Failure
Fails if the conversion fails on any list element.

## Example

```
# LIST_CONV num_CONV `[1;2;3;4;5]`;;
val it : thm = |- [1; 2; 3; 4; 5] = [SUC 0; SUC 1; SUC 2; SUC 3; SUC 4]
```

## Uses

Applying a conversion more delicately than simply by DEPTH_CONV etc.

## See also

DEPTH_BINOP_CONV, DEPTH_CONV, ONCE_DEPTH_CONV, REDEPTH_CONV, TOP_DEPTH_CONV, TOP_SWEEP_CONV.

# LIST_INDUCT_TAC

```
LIST_INDUCT_TAC : tactic
```

## Synopsis

Performs tactical proof by structural induction on lists.

## Description

LIST_INDUCT_TAC reduces a goal A ?- !l. P[l], where l ranges over lists, to two subgoals corresponding to the base and step cases in a proof by structural induction on l. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of LIST_INDUCT_TAC is:

```
                    A ?- !l. P
   =====================================================  LIST_INDUCT_TAC
    A |- P[[]/l]   A u {P[t/l]} ?- P[CONS h t/l]
```

## Failure

LIST_INDUCT_TAC g fails unless the conclusion of the goal g has the form `!l. t`, where the variable l has type (ty)list for some type ty.

## Example

Many simple list theorems can be proved simply by list induction then just first-order reasoning (or even rewriting) with definitions of the operations involved. For example if we want to prove that mapping a composition of functions over a list is the same as successive mapping of the two functions:

```
# g `!l f:A->B g:B->C. MAP (g o f) l = MAP g (MAP f l)`;;
```

we can start by list induction:

```
# e LIST_INDUCT_TAC;;
val it : goalstack = 2 subgoals (2 total)

 0 [`!f g. MAP (g o f) t = MAP g (MAP f t)`]

`!f g. MAP (g o f) (CONS h t) = MAP g (MAP f (CONS h t))`

`!f g. MAP (g o f) [] = MAP g (MAP f [])`
```

and each resulting subgoal is just solved at once by:

```
# e(ASM_REWRITE_TAC[MAP; o_THM]);;
```

## Comments

Essentially the same effect can be had by `MATCH_MP_TAC list_INDUCT`. This does not subsequently break down the goal in such a convenient way, but gives more control over choice of variable. For example, starting with the same goal:

```
# g `!l f:A->B g:B->C. MAP (g o f) l = MAP g (MAP f l)`;;
```

we get:

```
# e(MATCH_MP_TAC list_INDUCT);;
val it : goalstack = 1 subgoal (1 total)

`(!f g. MAP (g o f) [] = MAP g (MAP f [])) /\
 (!a0 a1.
      (!f g. MAP (g o f) a1 = MAP g (MAP f a1))
      ==> (!f g. MAP (g o f) (CONS a0 a1) = MAP g (MAP f (CONS a0 a1))))`
```

and after getting rid of some trivia:

```
# e(REWRITE_TAC[MAP]);;
val it : goalstack = 1 subgoal (1 total)

`!a0 a1.
     (!f g. MAP (g o f) a1 = MAP g (MAP f a1))
     ==> (!f g.
              CONS ((g o f) a0) (MAP (g o f) a1) =
              CONS (g (f a0)) (MAP g (MAP f a1)))`
```

we can carefully choose the variable names:

```
# e(MAP_EVERY X_GEN_TAC [`k:A`; `l:A list`]);;
val it : goalstack = 1 subgoal (1 total)

`(!f g. MAP (g o f) l = MAP g (MAP f l))
 ==> (!f g.
          CONS ((g o f) k) (MAP (g o f) l) =
          CONS (g (f k)) (MAP g (MAP f l)))`
```

This kind of control can be useful when the sub-proof is more challenging. Here of course the same simple pattern as before works:

```
# e(SIMP_TAC[o_THM]);;
val it : goalstack = No subgoals
```

## See also

INDUCT_TAC, MATCH_MP_TAC, WF_INDUCT_TAC.

# list_mk_abs

`list_mk_abs : term list * term -> term`

## Synopsis
Iteratively constructs abstractions.

## Description
`list_mk_abs([‘x1‘;...;‘xn‘],‘t‘)` returns `‘\x1 ... xn. t‘`.

## Failure
Fails with `list_mk_abs` if the terms in the list are not variables.

## Example

```
# list_mk_abs([‘m:num‘; ‘n:num‘],‘m + n + 1‘);;
val it : term = ‘\m n. m + n + 1‘
```

## See also
`dest_abs`, `mk_abs`, `strip_abs`.

# list_mk_binop

`list_mk_binop : term -> term list -> term`

## Synopsis
Makes an iterative application of a binary operator.

## Description
The call `list_mk_binop op [t1; ...; tn]` constructs the term `op t1 (op t2 (op ... (op tn-1 tn) ...)))`. If we think of `op` as an infix operator we can write it `t1 op t2 op t3 ... op tn`, but the call will work for any term `op` compatible with all the types.

## Failure
Fails if the list of terms is empty or if the types would not work for the composite term. In particular, if the list contains at least three items, all the types must be the same.

## Example

This example is typical:

```
# list_mk_binop `(+):num->num->num` (map mk_small_numeral (1--10));;
val it : term = `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10`
```

while these show that for smaller lists, one can just regard it as `mk_comb` or `mk_binop`:

```
# list_mk_binop `SUC` [`0`];;
val it : term = `0`

 # list_mk_binop `f:A->B->C` [`x:A`; `y:B`];;
val it : term = `f x y`
```

## See also

binops, mk_binop.

# list_mk_comb

```
list_mk_comb : term * term list -> term
```

## Synopsis

Iteratively constructs combinations (function applications).

## Description

`list_mk_comb(`t`,[`t1`;...;`tn`])` returns `` `t t1 ... tn` ``.

## Failure

Fails with `list_mk_comb` if the types of `t1`,...,`tn` are not equal to the argument types of `t`. It is not necessary for all the arguments of `t` to be given. In particular the list of terms `t1`,...,`tn` may be empty.

## Example

```
# list_mk_comb(`1`,[]);;
val it : term = `1`

# list_mk_comb(`(/\)`,[`T`]);;
val it : term = `(/\) T`

# list_mk_comb(`(/\)`,[`1`]);;
Exception: Failure "mk_comb: types do not agree".
```

## See also

list_mk_icomb, mk_comb, strip_comb.

# list_mk_conj

```
list_mk_conj : term list -> term
```

## Synopsis
Constructs the conjunction of a list of terms.

## Description
`list_mk_conj([‘t1‘;...;‘tn‘])` returns `‘t1 /\ ... /\ tn‘`.

## Failure
Fails with `list_mk_conj` if the list is empty or if the list has more than one element, one or more of which are not of type `‘:bool‘`.

## Example

```
# list_mk_conj [‘T‘;‘F‘;‘T‘];;
val it : term = ‘T /\ F /\ T‘

# list_mk_conj [‘T‘;‘1‘;‘F‘];;
Exception: Failure "mk_binary".

# list_mk_conj [‘1‘];;
val it : term = ‘1‘
```

## See also
`conjuncts`, `mk_conj`.

# list_mk_disj

```
list_mk_disj : term list -> term
```

## Synopsis
Constructs the disjunction of a list of terms.

## Description
`list_mk_disj([‘t1‘;...;‘tn‘])` returns `‘t1 \/ ... \/ tn‘`.

## Failure
Fails with `list_mk_disj` if the list is empty or if the list has more than one element, one or more of which are not of type `‘:bool‘`.

## Example

```
# list_mk_disj [`T`;`F`;`T`];;
val it : term = `T \/ F \/ T`

# list_mk_disj [`T`;`1`;`F`];;
Exception: Failure "mk_binary".

# list_mk_disj [`1`];;
val it : term = `1`
```

## See also

disjuncts, is_disj, mk_disj.

# list_mk_exists

```
list_mk_exists : term list * term -> term
```

## Synopsis

Multiply existentially quantifies both sides of an equation using the given variables.

## Description

When applied to a list of terms `[x1;...;xn]`, where the `ti` are all variables, and a theorem `A |- t1 = t2`, the inference rule `LIST_MK_EXISTS` existentially quantifies both sides of the equation using the variables given, none of which should be free in the assumption list.

```
              A |- t1 <=> t2
   ----------------------------------------  LIST_MK_EXISTS [`x1`;...;`xn`]
    A |- (?x1...xn. t1) <=> (?x1...xn. t2)
```

## Failure

Fails if any term in the list is not a variable or is free in the assumption list, or if the theorem is not equational.

## See also

EXISTS_EQ, MK_EXISTS.

# list_mk_forall

```
list_mk_forall : term list * term -> term
```

## Synopsis
Iteratively constructs a universal quantification.

## Description
`list_mk_forall([‘x1‘;...;‘xn‘],‘t‘)` returns `‘!x1 ... xn. t‘`.

## Failure
Fails if any term in the list is not a variable or if `t` is not of type `‘:bool‘` and the list of terms is non-empty. If the list of terms is empty the type of `t` can be anything.

## Example

```
# list_mk_forall([‘x:num‘; ‘y:num‘],‘x + y + 1 = SUC z‘);;
val it : term = ‘!x y. x + y + 1 = SUC z‘
```

## See also
`mk_forall, strip_forall.`

# list_mk_gabs

```
list_mk_gabs : term list * term -> term
```

## Synopsis
Iteratively makes a generalized abstraction.

## Description
The call `list_mk_gabs([vs1; ...; vsn],t)` constructs an interated generalized abstraction `\vs1. \vs2. ... \vsn. t`. See `mk_gabs` for more details on constructing generalized abstractions.

## Failure
Never fails.

## Example

```
# list_mk_gabs([‘(x:num,y:num)‘; ‘(w:num,z:num)‘],‘x + w + 1‘);;
val it : term = ‘\(x,y). \(w,z). x + w + 1‘
```

## See also
`dest_gabs, is_gabs, mk_gabs.`

# list_mk_icomb

```
list_mk_icomb : string -> term list -> term
```

## Synopsis
Applies constant to list of arguments, instantiating constant type as needed.

## Description
The call `list_mk_icomb "c" [a1; ...; an]` will make the term `c a1 ... an` where c is a constant, after first instantiating `c`'s generic type so that the types are compatible.

## Failure
Fails if `c` is not a constant or if the types cannot be instantiated to match up with the argument list.

## Example
This would fail with the basic `list_mk_comb` function

```
# list_mk_icomb "=" [`1`; `2`];;
val it : term = `1 = 2`
```

## See also
`list_mk_comb`, `mk_mconst`, `mk_icomb`.

# listof

```
listof : ('a -> 'b * 'c) -> ('c -> 'd * 'a) -> string -> 'a -> 'b list * 'c
```

## Synopsis
Parses a separated list of items.

## Description
If `p` is a parser for "items" of some kind, `s` is a parser for a "separator", and `e` is an error message, then `listof p s e` parses a nonempty list of successive items using `p`, where adjacent items are separated by something parseable by `s`. If a separator is parsed successfully but there is no following item that can be parsed by `s`, an exception `Failure e` is raised. (So note that the separator must not terminate the final element.)

## Failure
The call `listof p s e` itself never fails, though the resulting parser may.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `many`, `nothing`, `possibly`, `rightbin`, `some`.

# loaded_files

```
loaded_files : (string * Digest.t) list ref
```

## Synopsis

List of files loaded so far.

## Description

This reference variable stores a list of previously loaded files together with MD5 digests. It is updated by all the main loading functions `load_on_path`, `loads`, `loadt` and `needs`, and is used by `needs` to avoid reloading the same file multiple times.

## Failure

Not applicable.

## Uses

Not really intended for average users to examine or modify.

## See also

`load_on_path`, `loads`, `loadt`, `needs`.

# load_on_path

```
load_on_path : string list -> string -> unit
```

## Synopsis

Finds a file on a path and loads it into HOL Light.

## Description
When given a filename and a path, the file is found either directly by its filename or on the given path, as explained in `file_on_path`. It is then loaded into HOL, updating the list of loaded files.

## Failure
Fails if the file is not found or generates an exception when loaded (e.g. a syntax problem or runtime exception).

## See also
`loads, loadt, needs.`

# load_path

```
load_path : string list ref
```

## Synopsis
Path where HOL Light tries to find files to load.

## Description
The reference variable `load_path` gives a list of directories. When HOL loads files with `loadt`, it will try these places in order on all non-absolute filenames.

## Failure
Not applicable.

## See also
`load_on_path, loads, loadt.`

# loads

```
loads : string -> unit
```

## Synopsis
Load a file from the HOL Light system tree.

## Description
Finds the named file, either by its absolute pathname or by starting in the base of the HOL installation stored by `hol_dir`, and loads it.

## Failure
Fails if the file is not found or generates an exception.

## Example
To load a library with more number theory:

```
# loads "Examples/prime.ml";;
- : unit = ()
val ( MULT_MONO_EQ ) : thm = |- !m i n. SUC n * m = SUC n * i <=> m = i
...
...
val ( GCD_CONV ) : term -> thm = <fun>
val it : unit = ()
```

## Uses
Loading HOL Light standard libraries without accidentally picking up other files of the same name in the current directory or on `load_path`

## See also
`loadt`, `needs`.

# loadt

```
loadt : string -> unit
```

## Synopsis
Finds a file on the load path and loads it.

## Description
The function `loadt` takes a string indicating an OCaml file name as argument and loads it. If the filename is relative, it is found on the load path `load_path`, and it is then loaded, updating the list of loaded files.

## Failure
`loadt` will fail if the file named by the argument does not exist in the search path. It will of course fail if the file is not a valid OCaml file. Failure in the OCaml file will also terminate loading.

## Example
If we have an ML file called `foo.ml` on the load path, e.g. in the current directory, which

contains the line

```
let x=2+2;;
```

this can be loaded as follows:

```
# loadt "foo.ml";;
```

and the system would respond with:

```
# loadt "foo.ml";;
val x : int = 4
val it : unit = ()
```

## See also

`loads`, `needs`.

# `lookup`

```
lookup : term -> 'a net -> 'a list
```

## Synopsis

Look up term in a term net.

## Description

Term nets (type `'a net`) are a lookup structure associating objects of type `'a`, e.g. conversions, with a corresponding 'pattern' term. For a given term, one can then relatively quickly look up all objects whose pattern terms might possibly match to it. This is used, for example, in rewriting to quickly filter out obviously inapplicable rewrites rather than attempting each one in turn. The call `lookup t net` for a term `t` returns the list of objects whose patterns might possibly be matchable to `t`. Note that this is conservative: if the pattern could be matched (even higher-order matched) in the sense of `term_match`, it will be in the list, but it is possible that some non-matchable objects will be returned. (For example, a pattern term `x + x` will match any term of the form `a + b`, even if `a` and `b` are the same.) It is intended that nets are a first-level filter for efficiency; finer discrimination may be embodied in the subsequent action with the list of returned objects.

## Failure

Never fails.

## Example

If we want to create ourselves the kind of automated rewriting with the basic rewrites that is done by `REWRITE_CONV`, we could simply try in succession all the rewrites:

```
# let BASIC_REWRITE_CONV' = FIRST_CONV (map REWR_CONV (basic_rewrites()));;
val ( BASIC_REWRITE_CONV' ) : conv = <fun>
```

However, it would be more efficient to use the left-hand sides as patterns in a term net to organize the different rewriting conversions:

```
# let rewr_net =
    let enter_thm th = enter (freesl(hyp th)) (lhs(concl th),REWR_CONV th) in
    itlist enter_thm (basic_rewrites()) empty_net;;
```

Now given a term, we get only the items with matchable patterns, usually much less than the full list:

```
# lookup `(\x. x + 1) 2` rewr_net;;
val it : (term -> thm) list = [<fun>]

# lookup `T /\ T` rewr_net;;
val it : (term -> thm) list = [<fun>; <fun>; <fun>]
```

The three items returned in the last call are rewrites based on the theorems `|- T /\ t <=> t`, `|- t /\ T <=> t` and `|- t /\ t <=> t`, which are the only ones matchable. We can use this net for a more efficient version of the same conversion:

```
# let BASIC_REWRITE_CONV tm = FIRST_CONV (lookup tm rewr_net) tm;;
val ( BASIC_REWRITE_CONV ) : term -> conv = <fun>
```

To see that it is indeed more efficient, consider:

```
# let tm = funpow 8 (fun x -> mk_conj(x,x)) `T`;;
...
time (DEPTH_CONV BASIC_REWRITE_CONV) tm;;
CPU time (user): 0.08
...
time (DEPTH_CONV BASIC_REWRITE_CONV') tm;;
CPU time (user): 1.121
...
```

## See also

`empty_net`, `enter`, `merge_nets`.

# <?

```
(<?) : 'a -> 'a -> bool
```

## Synopsis
Reflexive short-cutting inequality test.

## Description
This is functionally identical to the OCaml polymorphic inequality test `<` except that it is total even on floating-point NaNs. More importantly, it will more efficiently short-cut comparisons of large data structures where subcomponents are identical (pointer equivalent).

## Failure
May fail when applied to functions.

## Example

```
# 1.0 < nan or nan < 1.0;;
val it : bool = false
# 1.0 <? nan;;
val it : bool = false
# nan <? 1.0;;
val it : bool = true
```

## See also
`=?`, `<=?`, `>?`, `>=?`.

# make_args

```
make_args : string -> term list -> hol_type list -> term list
```

## Synopsis
Make a list of terms with stylized variable names

## Description
The call `make_args "s" avoids [ty0; ...; tyn]` constructs a list of variables of types `ty0`, ..., `tyn`, normally called `s0`, ..., `sn` but primed if necessary to avoid clashing with any in `avoids`

## Failure
Never fails.

## Example

```
# make_args "arg" [`arg2:num`] [`:num`; `:num`; `:num`];;
val it : term list = [`arg0`; `arg1`; `arg2'`]
```

## Uses
Constructing arbitrary but relatively natural names for argument lists.

## See also
`genvar`, `variant`.

# make_overloadable

```
make_overloadable : string -> hol_type -> unit
```

## Synopsis
Makes a symbol overloadable within the specified type skeleton.

## Description
HOL Light allows the same identifier to denote several different underlying constants, with the choice being determined by types and/or an order of priority (see `prioritize_overload`). However, any identifier `ident` to be overloaded must first be declared overloadable using `make_overloadable "ident" ‘` The "type skeleton" argument `` `:ty` `` is the most general type that the various instances may have.

The type skeleton can simply be a type variable, in which case any type is acceptable, but it is good practice to constrain it where possible to allow more information to be inferred during typechecking. For example, the symbol '`+`' has the type skeleton `` `:A->A->A` `` (as you can find out by examining the list `the_overload_skeletons`) indicating that it is always overloaded to a binary operator that returns and element of the same type as its two arguments.

## Failure
Fails if the symbol has previously been made overloadable but with a different type skeleton.

## Example

```
# make_overloadable "<=" `:A->A->bool`;;
val it : unit = ()
```

## See also
`overload_interface`, `override_interface`, `prioritize_overload`, `reduce_interface`, `remove_interface`, `the_interface`, `the_overload_skeletons`.

# many

```
many : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
```

## Synopsis

Parses zero or more successive items using given parser.

## Description

If `p` is a parser then `many p` gives a new parser that parses a series of successive items using `p` and returns the result as a list, with the expected left-to-right order.

## Failure

The immediate call `many` never fails. The resulting parser may fail when applied, though any `Noparse` exception in the core parser will be trapped.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `nothing`, `possibly`, `rightbin`, `some`.

# map2

```
map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

## Synopsis

Maps a binary function over two lists to create one new list.

## Description

`map2 f ([x1;...;xn],[y1;...;yn])` returns `[f(x1,y1);...;f(xn,yn)]`.

## Failure

Fails with `map2` if the two lists are of different lengths.

## Example

```
# map2 (+) [1;2;3] [30;20;10];;
val it : int list = [31; 22; 13]
```

## See also

map, uncurry.

# map

`map : ('a -> 'b) -> 'a list -> 'b list`

## Synopsis

Applies a function to every element of a list.

## Description

`map f [x1;...;xn]` returns `[(f x1);...;(f xn)]`.

## Failure

Never fails.

## Example

```
# map (fun x -> x * 2) [];;
val it : int list = []
# map (fun x -> x * 2) [1;2;3];;
val it : int list = [2; 4; 6]
```

# MAP_EVERY

`MAP_EVERY : ('a -> tactic) -> 'a list -> tactic`

## Synopsis

Sequentially applies all tactics given by mapping a function over a list.

## Description

When applied to a tactic-producing function `f` and an operand list `[x1;...;xn]`, the elements of which have the same type as `f`'s domain type, `MAP_EVERY` maps the function `f` over the list, producing a list of tactics, then applies these tactics in sequence as in the case of `EVERY`. The effect is:

```
MAP_EVERY f [x1;...;xn] = (f x1) THEN ... THEN (f xn)
```

If the operand list is empty, then `MAP_EVERY` has no effect.

## Failure

The application of `MAP_EVERY` to a function and operand list fails iff the function fails when applied to any element in the list. The resulting tactic fails iff any of the resulting tactics fails.

## Example

A convenient way of doing case analysis over several boolean variables is:

```
MAP_EVERY BOOL_CASES_TAC [`v1:bool`;...;`vn:bool`]
```

## See also

`EVERY, FIRST, MAP_FIRST, THEN.`

# mapf

```
mapf : ('a -> 'b) -> ('c, 'a) func -> ('c, 'b) func
```

## Synopsis

Maps a function over the range of a finite partial function

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The function `mapf f p` applies the (ordinary OCaml) function `f` to all the range elements of a finite partial function, so if it originally mapped `xi` to `yi` for it now maps `xi` to `f(yi)`.

## Failure

Fails if the function fails on one of the `yi`.

## Example

```
# let f = (1 |=> 2);;
val f : (int, int) func = <func>
# mapf string_of_int f;;
val it : (int, string) func = <func>
# apply it 1;;
```

## See also

`|->, |=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, ran, tryapplyd, undefine, undefined.`

# mapfilter

```
mapfilter : ('a -> 'b) -> 'a list -> 'b list
```

## Synopsis
Applies a function to every element of a list, returning a list of results for those elements for which application succeeds.

## Failure
Fails if an exception not of the form `Failure _` is generated by any application to the elements.

## Example

```
# mapfilter hd [[1;2;3];[4;5];[];[6;7;8];[]];;
val it : int list = [1; 4; 6]

# mapfilter (fun (h::t) -> h) [[1;2;3];[4;5];[];[6;7;8];[]];;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception: Match_failure ("", 24547, -35120).
```

## See also
`filter`, `map`.

# MAP_FIRST

```
MAP_FIRST : ('a -> tactic) -> 'a list -> tactic
```

## Synopsis
Applies first tactic that succeeds in a list given by mapping a function over a list.

## Description
When applied to a tactic-producing function `f` and an operand list `[x1;...;xn]`, the elements of which have the same type as `f`'s domain type, `MAP_FIRST` maps the function `f` over the list,

producing a list of tactics, then tries applying these tactics to the goal till one succeeds. If `f(xm)` is the first to succeed, then the overall effect is the same as applying `f(xm)`. Thus:

```
MAP_FIRST f [x1;...;xn] = (f x1) ORELSE ... ORELSE (f xn)
```

## Failure

The application of `MAP_FIRST` to a function and tactic list fails iff the function does when applied to any of the elements of the list. The resulting tactic fails iff all the resulting tactics fail when applied to the goal.

## Example

Using the definition of integer-valued real numbers:

```
# needs "Examples/floor.ml";;
```

we have a set of ‘composition’ theorems asserting that the predicate is closed under various arithmetic operations:

```
# INTEGER_CLOSED;;
val it : thm =
  |- (!n. integer (&n)) /\
     (!x y. integer x /\ integer y ==> integer (x + y)) /\
     (!x y. integer x /\ integer y ==> integer (x - y)) /\
     (!x y. integer x /\ integer y ==> integer (x * y)) /\
     (!x r. integer x ==> integer (x pow r)) /\
     (!x. integer x ==> integer (--x)) /\
     (!x. integer x ==> integer (abs x))
```

if we want to prove that some composite term has integer type:

```
# g `integer(x) /\ integer(y)
     ==> integer(&2 * (x - &1) pow 7 + &11 * (y + &1))`;;
...
# e(REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`integer x`]
 1 [`integer y`]

`integer (&2 * (x - &1) pow 7 + &11 * (y + &1))`
```

A direct proof using `ASM_MESON_TAC[INTEGER_CLOSED]` works fine. However if we want to control the application of composition theorems more precisely we might do:

```
# let INT_CLOSURE_TAC =
    MAP_FIRST MATCH_MP_TAC (CONJUNCTS(CONJUNCT2 INTEGER_CLOSED)) THEN
    TRY CONJ_TAC;;
```

and then could solve the goal by:

```
e(REPEAT INT_CLOSURE_TAC THEN ASM_REWRITE_TAC[CONJUNCT1 INTEGER_CLOSED]);;
```

## See also

EVERY, FIRST, MAP_EVERY, ORELSE.

# MATCH_ACCEPT_TAC

MATCH_ACCEPT_TAC : thm_tactic

## Synopsis

Solves a goal which is an instance of the supplied theorem.

## Description

When given a theorem A' |- t and a goal A ?- t' where t can be matched to t' by instantiating variables which are either free or universally quantified at the outer level, including appropriate type instantiation, MATCH_ACCEPT_TAC completely solves the goal.

```
    A ?- t'
   =========  MATCH_ACCEPT_TAC (A' |- t)
```

Unless A' is a subset of A, this is an invalid tactic.

## Failure

Fails unless the theorem has a conclusion which is instantiable to match that of the goal.

## Example

The following example shows variable and type instantiation at work. Suppose we have the following simple goal:

```
# g `HD [1;2] = 1`;;
```

we can do it via the polymorphic theorem HD = |- !h t. HD(CONS h t) = h:

```
# e(MATCH_ACCEPT_TAC HD);;
```

## See also

ACCEPT_TAC.

# MATCH_MP

```
MATCH_MP : thm -> thm -> thm
```

## Synopsis
Modus Ponens inference rule with automatic matching.

## Description
When applied to theorems `A1 |- !x1...xn. t1 ==> t2` and `A2 |- t1'`, the inference rule `MATCH_MP` matches `t1` to `t1'` by instantiating free or universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- !xa..xk. t2'`, where `t2'` is a correspondingly instantiated version of `t2`. Polymorphic types are also instantiated if necessary.

Variables free in the consequent but not the antecedent of the first argument theorem will be replaced by variants if this is necessary to maintain the full generality of the theorem, and any which were universally quantified over in the first argument theorem will be universally quantified over in the result, and in the same order.

```
    A1 |- !x1..xn. t1 ==> t2   A2 |- t1'
   --------------------------------------  MATCH_MP
          A1 u A2 |- !xa..xk. t2'
```

## Failure
Fails unless the first theorem is a (possibly repeatedly universally quantified) implication whose antecedent can be instantiated to match the conclusion of the second theorem, without instantiating any variables which are free in `A1`, the first theorem's assumption list.

## Example
In this example, automatic renaming occurs to maintain the most general form of the theorem, and the variant corresponding to `z` is universally quantified over, since it was universally quantified over in the first argument theorem.

```
# let ith = ARITH_RULE `!x z:num. x = y ==> (w + z) + x = (w + z) + y`;;
val ith : thm = |- !x z. x = y ==> (w + z) + x = (w + z) + y

# let th = ASSUME `w:num = z`;;
val th : thm = w = z |- w = z

# MATCH_MP ith th;;
val it : thm = w = z |- !z'. (w + z') + w = (w + z') + z
```

## See also
`EQ_MP`, `MATCH_MP_TAC`, `MP`, `MP_TAC`.

# MATCH_MP_TAC

```
MATCH_MP_TAC : thm_tactic
```

## Synopsis
Reduces the goal using a supplied implication, with matching.

## Description
When applied to a theorem of the form

```
   A' |- !x1...xn. s ==> t
```

`MATCH_MP_TAC` produces a tactic that reduces a goal whose conclusion `t'` is a substitution and/or type instance of `t` to the corresponding instance of `s`. Any variables free in `s` but not in `t` will be existentially quantified in the resulting subgoal:

```
     A ?- t'
  ======================  MATCH_MP_TAC (A' |- !x1...xn. s ==> t)
     A ?- ?z1...zp. s'
```

where `z1`, ..., `zp` are (type instances of) those variables among `x1`, ..., `xn` that do not occur free in `t`. Note that this is not a valid tactic unless `A'` is a subset of `A`.

## Example
The following goal might be solved by case analysis:

```
  # g `!n:num. n <= n * n`;;
```

We can "manually" perform induction by using the following theorem:

```
  # num_INDUCTION;;
  val it : thm = |- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)
```

which is useful with `MATCH_MP_TAC` because of higher-order matching:

```
  # e(MATCH_MP_TAC num_INDUCTION);;
  val it : goalstack = 1 subgoal (1 total)

  `0 <= 0 * 0 /\ (!n. n <= n * n ==> SUC n <= SUC n * SUC n)`
```

The goal can be finished with `ARITH_TAC`.

## Failure
Fails unless the theorem is an (optionally universally quantified) implication whose consequent can be instantiated to match the goal.

## See also
`EQ_MP`, `MATCH_MP`, `MP`, `MP_TAC`, `PART_MATCH`.

# mem

```
mem : 'a -> 'a list -> bool
```

## Synopsis
Tests whether a list contains a certain member.

## Description
`mem x [x1;...;xn]` returns `true` if some `xi` in the list is equal to `x`. Otherwise it returns `false`.

## Failure
Never fails.

## See also
`find`, `tryfind`, `exists`, `forall`, `assoc`, `rev_assoc`.

# mem'

```
mem' : ('a -> 'b -> bool) -> 'a -> 'b list -> bool
```

## Synopsis
Tests if an element is equivalent to a member of a list w.r.t. some relation.

## Description
If `r` is a binary relation, `x` an element and `l` a list, the call `mem' r x l` tests if there is an element in the list `l` that is equivalent to `x` according to `r`, that is, if `r x x'` holds for some `x'` in `l`. The function `mem` is the special case where the relation is equality.

## Failure
Fails only if the relation `r` fails.

## Example

```
# mem' (fun x y -> abs(x) = abs(y)) (-1) [1;2;3];;
val it : bool = true
# mem' (fun x y -> abs(x) = abs(y)) (-1) [2;3;4];;
val it : bool = false
```

## Uses
Set operations modulo some equivalence such as alpha-equivalence.

## See also
`insert'`, `mem`, `subtract'`, `union'`, `unions'`.

# merge

```
merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

## Synopsis
Merges together two sorted lists with respect to a given ordering.

## Description
If two lists `l1` and `l2` are sorted with respect to the given ordering `ord`, then `merge ord l1 l2` will merge them into a sorted list of all the elements. The merge keeps any duplicates; it is not a set operation.

## Failure
Never fails, but if the lists are not appropriately sorted the results will not in general be correct.

## Example

```
# merge (<) [1;2;3;4;5;6] [2;4;6;8];;
val it : int list = [1; 2; 2; 3; 4; 4; 5; 6; 6; 8]
```

## See also
`mergesort`, `sort`, `uniq`.

# merge_nets

```
merge_nets : 'a net * 'a net -> 'a net
```

## Synopsis
Merge together two term nets.

## Description
Term nets (type `'a net`) are a lookup structure associating objects of type `'a`, e.g. conversions, with a corresponding 'pattern' term. For a given term, one can then relatively quickly look up all objects whose pattern terms might possibly match to it. This is used, for example, in rewriting to quickly filter out obviously inapplicable rewrites rather than attempting each one in turn. The call `merge_nets(net1,net2)` merges two nets together; the list of objects is the union of those objects in the two nets `net1` and `net2`, with the term patterns adjusted appropriately.

## Failure
Never fails.

## Example
If we have one term net containing the addition conversion:

```
# let net1 = enter [] (`x + y`,NUM_ADD_CONV) empty_net;;
...
```

and another with beta-conversion:

```
# let net2 = enter [] (`(\x. t) y`,BETA_CONV) empty_net;;
...
```

we can combine them into a single net:

```
# let net = merge_nets(net1,net2);;
...
```

## See also
`empty_net`, `enter`, `lookup`.

# mergesort

`mergesort : ('a -> 'a -> bool) -> 'a list -> 'a list`

## Synopsis
Sorts the list with respect to given ordering using mergesort algorithm.

## Description
If `ord` is a total order, a call `mergesort ord l` will sort the list `l` according to the order `ord`. It works internally by a mergesort algorithm. From a user's point of view, this just means: (i) its worst-case performance is much better than `sort`, which uses quicksort, but (ii) it will not reliably topologically sort for a non-total order, whereas `sort` will.

## Failure
Never fails unless the ordering function fails.

## Example

```
# mergesort (<) [6;2;5;9;2;5;3];;
val it : int list = [2; 2; 3; 5; 5; 6; 9]
```

## See also
`merge`, `sort`.

# meson_brand

```
meson_brand : bool ref
```

## Synopsis
Makes MESON handle equations using Brand's transformation.

## Description
This is one of several parameters determining the behavior of MESON, MESON_TAC and related rules and tactics. When meson_brand is true, equations are handled inside MESON by applying Brand's transformation. When it is false, as it is by default, equations are handled in a more "naive" way, which nevertheless appears generally better.

## Failure
Not applicable.

## Uses
For users requiring fine control over the algorithms used in MESON's first-order proof search.

## Comments
For more details of Brand's modification, see his paper "Proving theorems with the modification method", SIAM Journal on Computing volume 4, 1975. See also Moser and Steinbach's Munich technical report "STE-modification revisited" (AR-97-03, 1997) for another look at the subject.

## See also
meson_chatty, meson_dcutin, meson_depth, meson_prefine, meson_skew, meson_split_limit,

# meson_chatty

```
meson_chatty : bool ref
```

## Synopsis
Make MESON's output more verbose and detailed.

## Description
This is one of several parameters determining the behavior of MESON, MESON_TAC and related rules and tactics. When meson_chatty is set to true, MESON provides more verbose output, reporting at each level of iterative deepening search the current size limit and number of inferences on a fresh line. When meson_chatty is false, as it is by default, the core inference numbers are condensed into 1-line output.

## Failure
Not applicable.

## See also
`meson_brand`, `meson_dcutin`, `meson_depth`, `meson_prefine`, `meson_skew`,
`meson_split_limit`,

# meson_dcutin

```
meson_dcutin : int ref
```

## Synopsis
Determines cut-in point for divide-and-conquer refinement in `MESON`.

## Description
This is one of several parameters determining the behavior of `MESON`, `MESON_TAC` and related rules and tactics. This number (by default 1) determines the number of current subgoals at which point a special divide-and-conquer refinement will be invoked.

## Failure
Not applicable.

## Uses
For users requiring fine control over the algorithms used in `MESON`'s first-order proof search.

## Comments
For more details of this optimization, see Harrison's paper "Optimizing Proof Search in Model Elimination", CADE-13, 1996.

## See also
`meson_brand`, `meson_chatty`, `meson_depth`, `meson_prefine`, `meson_skew`,
`meson_split_limit`,

# meson_depth

```
meson_depth : bool ref
```

## Synopsis
Make `MESON`'s search algorithm work by proof depth rather than size.

## Description

This is one of several parameters determining the behavior of MESON, MESON_TAC and related rules and tactics. The basic search strategy is iterated deepening, searching for proofs with higher and higher limits on the search space. The flag meson_depth, when set to true, limits the search space based on proof depth, i.e. the longest branch. When set to false, as it is by default, the proof is limited based on total size.

## Failure

Not applicable.

## Uses

For users requiring fine control over the algorithms used in MESON's first-order proof search.

## See also

meson_brand, meson_chatty, meson_dcutin, meson_prefine, meson_skew, meson_split_limit,

# MESON

```
MESON : thm list -> term -> thm
```

## Synopsis

Attempt to prove a term by first-order proof search.

## Description

A call MESON[theorems] `tm` will attempt to prove tm using pure first-order reasining, taking theorems as the starting-point. It will usually either solve the goal completely or run for an infeasible length of time before terminating, but it may sometimes fail quickly.

Although MESON is capable of some fairly non-obvious pieces of first-order reasoning, and will handle equality adequately, it does purely logical reasoning. It will exploit no special properties of the constants in the goal, other than equality and logical primitives. Any properties that are needed must be supplied explicitly in the theorem list, e.g. LE_REFL to tell it that <= on natural numbers is reflexive, or REAL_ADD_SYM to tell it that addition on real numbers is symmetric.

## Failure

Will fail if the term is not provable, but not necessarily in a feasible amount of time.

## Example

A typical application is to prove some elementary logical lemma for use inside a tactic proof:

```
# MESON[] `!P. P F /\ P T ==> !x. P x`;;
...
val it : thm = |- !P. P F /\ P T ==> (!x. P x)
```

To prove the following lemma, we need to provide the key property of real negation:

```
# MESON[REAL_NEG_NEG] `(!x. P(--x)) ==> !x:real. P x`;;
...
val it : thm = |- (!x. P (--x)) ==> (!x. P x)
```

If the lemma is not supplied, `MESON` will fail:

```
# MESON[] `(!x. P(--x)) ==> !x:real. P x`;;
...
Exception: Failure "solve_goal: Too deep".
```

`MESON` is also capable of proving less straightforward results; see the documentation for `MESON_TAC` to find more examples.

## Uses

Generating simple logical lemmas as part of a large proof.

## See also

`ASM_MESON_TAC`, `GEN_MESON_TAC`, `MESON_TAC`.

# meson_prefine

```
meson_prefine : bool ref
```

## Synopsis

Makes `MESON` apply Plaisted's positive refinement.

## Description

This is one of several parameters determining the behavior of `MESON`, `MESON_TAC` and related rules and tactics. When the flag `meson_prefine` is `true`, as it is by default, Plaisted's "positive refinement" is used in proof search; this limits the search space at the cost of sometimes requiring longer proofs. When `meson_prefine` is false, this refinement is not applied.

## Failure

Not applicable.

## Uses
For users requiring fine control over the algorithms used in MESON's first-order proof search.

## Comments
For more details, see Plaisted's article "A Sequent-Style Model Elimination Strategy and a Positive Refinement", Journal of Automated Reasoning volume 6, 1990.

## See also
meson_brand, meson_chatty, meson_dcutin, meson_depth, meson_skew, meson_split_limit,

# meson_skew

```
meson_skew : int ref
```

## Synopsis
Determines skew in MESON proof tree search limits.

## Description
This is one of several parameters determining the behavior of MESON, MESON_TAC and related rules and tactics. During search, MESON successively searches for proofs of larger and larger 'size'. The "skew" value determines what proportion of the entire proof size is permitted in the left-hand half of the list of subgoals. The symmetrical value is 2 (meaning one half), the default setting of 3 (one third) seems generally better because it can cut down on redundancy in proofs.

## Failure
Not applicable.

## Uses
For users requiring fine control over the algorithms used in MESON's first-order proof search.

## Comments
For more details of MESON's search strategy, see Harrison's paper "Optimizing Proof Search in Model Elimination", CADE-13, 1996.

## See also
meson_brand, meson_chatty, meson_dcutin, meson_depth, meson_prefine, meson_split_limit,

# meson_split_limit

```
meson_split_limit : int ref
```

## Synopsis
Limit initial case splits before `MESON` proper is applied.

## Description
This is one of several parameters determining the behavior of `MESON`, `MESON_TAC` and related rules and tactics. Before these rules or tactics are applied, the formula to be proved is often decomposed by splitting, for example an equivalence `p <=> q` to two separate implications `p ==> q` and `q ==> p`. This often makes the eventual proof much easier for `MESON`. On the other hand, if splitting is applied too many times, it can become inefficient. The value `meson_split_limit` (default `8`) is the maximum number of times that splitting can be applied before `MESON` proper.

## Failure
Not applicable.

## Uses
For users requiring fine control over the algorithms used in `MESON`'s first-order proof search.

## See also
`meson_brand`, `meson_chatty`, `meson_dcutin`, `meson_depth`, `meson_prefine`, `meson_skew`.

# MESON_TAC

```
MESON_TAC : thm list -> tactic
```

## Synopsis
Automated first-order proof search tactic.

## Description
A call to `MESON_TAC[theorems]` will attempt to establish the current goal using pure first-order reasining, taking `theorems` as the starting-point. (It does not take the assumptions of the goal into account, but the similar function `ASM_MESON_TAC` does.) It will usually either solve the goal completely or run for an infeasible length of time before terminating, but it may sometimes fail quickly.

Although `MESON_TAC` is capable of some fairly non-obvious pieces of first-orderreasoning, and will handle equality adequately, it does purely logical reasoning. It will exploit no special properties of the constants in the goal, other than equality and logical primitives. Any properties that are needed must be supplied explicitly in the theorem list, e.g. `LE_REFL` to tell it that `<=` on natural numbers is reflexive, or `REAL_ADD_SYM` to tell it that addition on real numbers is symmetric.

## Failure
Fails if the goal is unprovable within the search bounds, though not necessarily in a feasible amount of time.

## Example

Here is a simple logical property taken from Dijstra's EWD 1062-1, which we set as our goal:

```
# g `(!x. x <= x) /\
     (!x y z. x <= y /\ y <= z ==> x <= z) /\
     (!x y. f(x) <= y <=> x <= g(y))
     ==> (!x y. x <= y ==> f(x) <= f(y))`;;
```

It is solved quickly by:

```
# e(MESON_TAC[]);;
0..0..1..3..8..17..solved at 25
CPU time (user): 0.
val it : goalstack = No subgoals
```

Note however that the proof did not rely on any special features of '<='; any binary relation symbol would have worked. Even simple proofs that rely on special properties of the constants need to have those properties supplied in the list. Note also that MESON is limited to essentially first-order reasoning, meaning that it cannot invent higher-order quantifier instantiations. Thus, it cannot prove the following, which involves a quantification over a function g:

```
# g `!f:A->B s.
       (!x y. x IN s /\ y IN s /\ (f x = f y) ==> (x = y)) <=>
       (?g. !x. x IN s ==> (g(f(x)) = x))`;;
```

However, we can manually reduce it to two subgoals:

```
# e(REPEAT GEN_TAC THEN MATCH_MP_TAC EQ_TRANS THEN
    EXISTS_TAC `?g:B->A. !y x. x IN s /\ y = f x ==> g y = x` THEN
    CONJ_TAC THENL
     [REWRITE_TAC[GSYM SKOLEM_THM]; AP_TERM_TAC THEN ABS_TAC]);;
val it : goalstack = 2 subgoals (2 total)

`(!y x. x IN s /\ y = f x ==> g y = x) <=> (!x. x IN s ==> g (f x) = x)`

`(!x y. x IN s /\ y IN s /\ f x = f y ==> x = y) <=>
 (!y. ?g. !x. x IN s /\ y = f x ==> g = x)`
```

and both of those are solvable directly by MESON_TAC[].

## See also

ASM_MESON_TAC, GEN_MESON_TAC, MESON.

# META_EXISTS_TAC

```
META_EXISTS_TAC : (string * thm) list * term -> goalstate
```

## Synopsis

Changes existentially quantified variable to metavariable.

## Description

Given a goal of the form `A ?- ?x. t[x]`, the tactic `X_META_EXISTS_TAC` gives the new goal `A ?- t[x]` where `x` is a new metavariable. In the resulting proof, it is as if the variable has been assigned here to the later choice for this metavariable, which can be made through for example `UNIFY_ACCEPT_TAC`.

## Failure

Never fails.

## Example

See `UNIFY_ACCEPT_TAC` for an example of using metavariables.

## Uses

Delaying instantiations until the correct term becomes clearer.

## Comments

Users should probably steer clear of using metavariables if possible. Note that the metavariable instantiations apply across the whole fringe of goals, not just the current goal, and can lead to confusion.

## See also

`EXISTS_TAC`, `META_SPEC_TAC`, `UNIFY_ACCEPT_TAC`, `X_META_EXISTS_TAC`.

# META_SPEC_TAC

```
META_SPEC_TAC : term -> thm -> tactic
```

## Synopsis

Replaces universally quantified variable in theorem with metavariable.

## Description

Given a variable `v` and a theorem `th` of the form `A |- !x. p[x]`, the tactic `META_SPEC_TAC ‘v‘ th` is a tactic that adds the theorem `A |- p[v]` to the assumptions of the goal, with `v` a new metavariable. This can later be instantiated, e.g. by `UNIFY_ACCEPT_TAC`, and it is as if the instantiation were done at this point.

## Failure

Fails if `v` is not a variable.

## Example

See `UNIFY_ACCEPT_TAC` for an example of using metavariables.

## Uses
Delaying instantiations until the right choice becomes clearer.

## Comments
Users should probably steer clear of using metavariables if possible. Note that the metavariable instantiations apply across the whole fringe of goals, not just the current goal, and can lead to confusion.

## See also
EXISTS_TAC, EXISTS_TAC, UNIFY_ACCEPT_TAC, X_META_EXISTS_TAC.

# mk_abs

```
mk_abs : term * term -> term
```

## Synopsis
Constructs an abstraction.

## Description
If `v` is a variable and `t` any term, then `mk_abs(v,t)` produces the abstraction term `\v. t`. It is not necessary that `v` should occur free in `t`.

## Failure
Fails if `v` is not a variable. See `mk_gabs` for constructing generalized abstraction terms.

## Example

```
# mk_abs(`x:num`,`x + 1`);;
val it : term = `\x. x + 1`
```

## See also
dest_abs, is_abs, mk_gabs.

# mk_binary

```
mk_binary : string -> term * term -> term
```

## Synopsis
Constructs an instance of a named monomorphic binary operator.

## Description

The call `mk_binary s (l,r)` constructs a binary application `(op l) r` where `op` is the monomorphic constant with name `s`. Note that it will in general `\em not` work if the constant is polymorphic.

## Failure

If there is no constant at all with name `s`, or if the constant is polymorphic and the terms do not match its most general type.

## Example

This case works fine:

```
# mk_binary "+" (`1`,`2`);;
val it : term = `1 + 2`
```

but here we hit the monomorphism restriction:

```
# mk_binary "=" (`a:A`,`b:A`);;
val it : term = `a = b`
# mk_binary "=" (`1`,`2`);;
Exception: Failure "mk_binary".
```

## See also

`dest_binary, is_binary, mk_binop.`

# mk_binder

```
mk_binder : string -> term * term -> term
```

## Synopsis

Constructs a term with a named constant applied to an abstraction.

## Description

The call `mk_binder "c" (x,t)` returns the term `c (\x. t)` where `c` is a constant with the given name appropriately type-instantiated. Note that the binder parsing status of `c` is irrelevant, though only if it is parsed as a binder will the resulting term be printed and parseable as `c x. t`.

## Failure

Failus if `x` is not a variable, if there is no constant `c` or if the type of that constant cannot be instantiated to match the abstraction.

## Example

```
# mk_binder "!" (`x:num`,`x + 1 > 0`);;
val it : term = `!x. x + 1 > 0`
```

## See also

dest_binder, is_binder.

# mk_binop

```
mk_binop : term -> term -> term -> term
```

## Synopsis

The call `mk_binop op l r` returns the term `(op l) r`.

## Description

The call `mk_binop op l r` returns the term `(op l) r` provided that is well-typed. Otherwise it fails. The term `op` need not be a constant nor parsed as infix, but that is the usual case. Note that type variables in `op` are not instantiated, so it needs to be the correct instance for the terms `l` and `r`.

## Failure

Fails if the types are incompatible.

## Example

```
# mk_binop `(+):num->num->num` `1` `2`;;
val it : term = `1 + 2`
```

## See also

dest_binop, is_binop, mk_binary.

# MK_BINOP

```
MK_BINOP : term -> thm * thm -> thm
```

## Synopsis

Compose equational theorems with binary operator.

## Description

Given a term `op` and the pair of theorems `(|- l = l'),(|- r = r')`, the function `MK_BINOP` returns the theorem `|- op l r = op l' r'`, provided the types are compatible.

## Failure

Fails if the types are incompatible for the term `op l r`.

## Example

```
# let th1 = NUM_REDUCE_CONV `2 * 2`
  and th2 = NUM_REDUCE_CONV `2 EXP 2`;;
val th1 : thm = |- 2 * 2 = 4
val th2 : thm = |- 2 EXP 2 = 4
# MK_BINOP `(+):num->num->num` (th1,th2);;
val it : thm = |- 2 * 2 + 2 EXP 2 = 4 + 4
```

## See also

`BINOP_CONV`, `DEPTH_BINOP_CONV`, `MK_COMB`.

# mk_comb

```
mk_comb : term * term -> term
```

## Synopsis

Constructs a combination.

## Description

Given two terms `f` and `x`, the call `mk_comb(f,x)` returns the combination or application `f x`. It is necessary that `f` has a function type with domain type the same as `x`'s type.

## Failure

Fails if the types of the terms are not compatible as specified above.

## Example

```
# mk_comb(`SUC`,`0`);;
val it : term = `SUC 0`

#  mk_comb(`SUC`,`T`);;
Exception: Failure "mk_comb: types do not agree".
```

## See also

`dest_comb`, `is_comb`, `list_mk_comb`, `list_mk_icomb`, `mk_icomb`.

# MK_COMB_TAC

`MK_COMB_TAC : tactic`

## Synopsis
Breaks down a goal between function applications into equality of functions and arguments.

## Description
Given a goal whose conclusion is an equation between function applications `A ?- f x = g y`, the tactic `MK_COMB_TAC` breaks it down to two subgoals expressing equality of the corresponding rators and rands:

```
          A ?- f x = g y
   ================================  MK_COMB_TAC
      A ?- f = g      A ?- x = y
```

## Failure
Fails if the conclusion of the goal is not an equation between applications.

## See also
`ABS_TAC`, `AP_TERM_TAC`, `AP_THM_TAC`, `BINOP_TAC`, `MK_COMB`.

# MK_COMB

`MK_COMB : thm * thm -> thm`

## Synopsis
Proves equality of combinations constructed from equal functions and operands.

## Description
When applied to theorems `A1 |- f = g` and `A2 |- x = y`, the inference rule `MK_COMB` returns the theorem `A1 u A2 |- f x = g y`.

```
    A1 |- f = g   A2 |- x = y
   ---------------------------  MK_COMB
       A1 u A2 |- f x = g y
```

## Failure
Fails unless both theorems are equational and `f` and `g` are functions whose domain types are the same as the types of `x` and `y` respectively.

## Example

```
# TRANS (ASSUME `0 = 1`) (ASSUME `1 = 2`);;
val it : thm = 0 = 1, 1 = 2 |- 0 = 2
```

## Comments

This is one of HOL Light's 10 primitive inference rules. It underlies, among other things, the replacement of subterms in rewriting.

## See also

`AP_TERM`, `AP_THM`.

# mk_cond

```
mk_cond : term * term * term -> term
```

## Synopsis

Constructs a conditional term.

## Description

`mk_cond(`t`,`t1`,`t2`)` returns `` `if t then t1 else t2` ``.

## Failure

Fails with `mk_cond` if `t` is not of type `` `:bool` `` or if `t1` and `t2` are of different types.

## See also

`dest_cond`, `is_cond`.

# mk_conj

```
mk_conj : term * term -> term
```

## Synopsis

Constructs a conjunction.

## Description

`mk_conj(`t1`,`t2`)` returns `` `t1 /\ t2` ``.

## Failure

Fails with `mk_conj` if either `t1` or `t2` are not of type `` `:bool` ``.

## Example

```
# mk_conj(`1 + 1 = 2`,`2 + 2 = 4`);;
val it : term = `1 + 1 = 2 /\ 2 + 2 = 4`
```

## See also

dest_conj, is_conj, list_mk_conj.

# MK_CONJ

```
MK_CONJ : thm -> thm -> thm
```

## Synopsis

Conjoin both sides of two equational theorems.

## Description

Given two theorems, each with a Boolean equation as conclusion, MK_CONJ returns the equation resulting from conjoining their respective sides:

```
      A |- p <=> p'   B |- q <=> q'
   ----------------------------------- MK_CONJ
        A u B |- p /\ q <=> p' /\ q'
```

## Failure

Fails unless both input theorems are Boolean equations (iff).

## Example

```
# let th1 = ARITH_RULE `0 < n <=> ~(n = 0)`
  and th2 = ARITH_RULE `1 <= n <=> ~(n = 0)`;;
val th1 : thm = |- 0 < n <=> ~(n = 0)
val th2 : thm = |- 1 <= n <=> ~(n = 0)

# MK_CONJ th1 th2;;
val it : thm = |- 0 < n /\ 1 <= n <=> ~(n = 0) /\ ~(n = 0)
```

## See also

AP_TERM, AP_THM, MK_BINOP, MK_COMB, MK_DISJ, MK_EXISTS, MK_FORALL.

# mk_cons

```
mk_cons : term -> term -> term
```

## Synopsis
Constructs a `CONS` pair.

## Description
`mk_cons `h` `t`` returns ``CONS h t``.

## Failure
Fails if second term is not of list type or if the first term is not of the same type as the elements of the list.

## Example

```
# mk_cons `1` `l:num list`;;
val it : term = `CONS 1 l`

# mk_cons `1` `[2;3;4]`;;
val it : term = `[1; 2; 3; 4]`
```

## See also
`dest_cons`, `dest_list`, `is_cons`, `is_list`, `mk_flist`, `mk_list`.

# mk_const

```
mk_const : string * (hol_type * hol_type) list -> term
```

## Synopsis
Produce constant term by applying an instantiation to its generic type.

## Description
This is the basic way of constructing a constant term in HOL Light, applying a specific instantiation (by `type_subst`) to its generic type. It may sometimes be more convenient to use `mk_mconst`, which just takes the desired type for the constant and finds the instantiation itself; that is also a natural inverse for `dest_const`. However, `mk_const` is likely to be significantly faster.

## Failure
Fails if there is no constant of the given type.

## Example

```
# get_const_type "=";;
val it : hol_type = `:A->A->bool`

# mk_const("=",[`:num`,`:A`]);;
val it : term = `(=)`
# type_of it;;
val it : hol_type = `:num->num->bool`

# mk_const("=",[`:num`,`:A`]) = mk_mconst("=",`:num->num->bool`);;
val it : bool = true
```

## See also

dest_const, is_const, mk_mconst, type_subst.

# mk_disj

mk_disj : term * term -> term

## Synopsis

Constructs a disjunction.

## Description

mk_disj(`t1`,`t2`) returns `t1 \/ t2`.

## Failure

Fails with mk_disj if either t1 or t2 are not of type `:bool`.

## Example

```
# mk_disj(`x = 1`,`y <= 2`);;
val it : term = `x = 1 \/ y <= 2`
```

## See also

dest_disj, is_disj, list_mk_disj.

# MK_DISJ

MK_DISJ : thm -> thm -> thm

## Synopsis
Disjoin both sides of two equational theorems.

## Description
Given two theorems, each with a Boolean equation as conclusion, MK_DISJ returns the equation resulting from disjoining their respective sides:

```
     A |- p <=> p'   B |- q <=> q'
  ----------------------------------- MK_DISJ
       A u B |- p \/ q <=> p' \/ q'
```

## Failure
Fails unless both input theorems are Boolean equations (iff).

## Example

```
# let th1 = ARITH_RULE `1 < x <=> 1 <= x - 1`
  and th2 = ARITH_RULE `~(1 < x) <=> x = 0 \/ x = 1`;;
val th1 : thm = |- 1 < x <=> 1 <= x - 1
val th2 : thm = |- ~(1 < x) <=> x = 0 \/ x = 1

# MK_DISJ th1 th2;;
val it : thm = |- 1 < x \/ ~(1 < x) <=> 1 <= x - 1 \/ x = 0 \/ x = 1
```

## See also
AP_TERM, AP_THM, MK_BINOP, MK_COMB, MK_CONJ, MK_EXISTS, MK_FORALL.

# mk_eq

```
mk_eq : term * term -> term
```

## Synopsis
Constructs an equation.

## Description
mk_eq(`t1`,`t2`) returns `t1 = t2`.

## Failure
Fails with mk_eq if t1 and t2 have different types.

### Example

```
# mk_eq(`1`,`2`);;
val it : term = `1 = 2`
```

### See also

dest_eq, is_eq.

## mk_exists

mk_exists : term * term -> term

### Synopsis

Term constructor for existential quantification.

### Description

mk_exists(`v`,`t`) returns `?v. t`.

### Failure

Fails with if first term is not a variable or if t is not of type `:bool`.

### Example

```
# mk_exists(`x:num`,`x + 1 = 1 + x`);;
val it : term = `?x. x + 1 = 1 + x`
```

### See also

dest_exists, is_exists, list_mk_exists.

## MK_EXISTS

MK_EXISTS : term -> thm -> thm

### Synopsis

Existentially quantifies both sides of equational theorem.

## Description

Given a theorem `th` whose conclusion is a Boolean equation (iff), the rule `MK_EXISTS ‘v‘ th` existentially quantifies both sides of `th` over the variable `v`, provided it is not free in the hypotheses

```
            A |- p <=> q
     --------------------------- MK_EXISTS ‘v‘ [where v not free in A]
       A |- (?v. p) <=> (?v. q)
```

## Failure

Fails if the term is not a variable or is free in the hypotheses of the theorem, or if the theorem does not have a Boolean equation for its conclusion.

## Example

```
# let th = ARITH_RULE ‘f(x:A) >= 1 <=> ~(f(x) = 0)‘;;
val th : thm = |- f x >= 1 <=> ~(f x = 0)

# MK_EXISTS ‘x:A‘ th;;
val it : thm = |- (?x. f x >= 1) <=> (?x. ~(f x = 0))
```

## See also

`AP_TERM`, `AP_THM`, `MK_BINOP`, `MK_COMB`, `MK_CONJ`, `MK_DISJ`, `MK_FORALL`.

# mk_flist

```
mk_flist : term list -> term
```

## Synopsis

Constructs object-level list from nonempty list of terms.

## Description

`mk_flist [‘t1‘;...;‘tn‘]` returns `‘[t1;...;tn]‘`. The list must be nonempty, since the type could not be inferred for that case. For cases where you may need to construct an empty list, use `mk_list`.

## Failure

Fails if the list is empty or if the types of any elements differ from each other.

## Example

```
# mk_flist(map mk_small_numeral (1--10));;
val it : term = ‘[1; 2; 3; 4; 5; 6; 7; 8; 9; 10]‘
```

## See also

`dest_cons`, `dest_list`, `is_cons`, `is_list`, `mk_cons`, `mk_list`.

# mk_forall

mk_forall : term * term -> term

## Synopsis
Term constructor for universal quantification.

## Description
mk_forall(‘v‘,‘t‘) returns ‘!v. t‘.

## Failure
Fails with if first term is not a variable or if t is not of type ‘:bool‘.

## Example

```
# mk_forall(‘x:num‘,‘x + 1 = 1 + x‘);;
val it : term = ‘!x. x + 1 = 1 + x‘
```

## See also
dest_forall, is_forall, list_mk_forall.

# MK_FORALL

MK_FORALL : term -> thm -> thm

## Synopsis
Universally quantifies both sides of equational theorem.

## Description
Given a theorem th whose conclusion is a Boolean equation (iff), the rule MK_FORALL ‘v‘ th universally quantifies both sides of th over the variable v, provided it is not free in the hypotheses

```
          A |- p <=> q
   ---------------------------- MK_FORALL ‘v‘ [where v not free in A]
      A |- (!v. p) <=> (!v. q)
```

## Failure
Fails if the term is not a variable or is free in the hypotheses of the theorem, or if the theorem does not have a Boolean equation for its conclusion.

## Example

```
# let th = ARITH_RULE `f(x:A) >= 1 <=> ~(f(x) = 0)`;;
val th : thm = |- f x >= 1 <=> ~(f x = 0)

# MK_FORALL `x:A` th;;
val it : thm = |- (!x. f x >= 1) <=> (!x. ~(f x = 0))
```

## See also

AP_TERM, AP_THM, MK_BINOP, MK_COMB, MK_CONJ, MK_DISJ, MK_EXISTS.

# mk_fset

```
mk_fset : term list -> term
```

## Synopsis

Constructs an explicit set enumeration from a nonempty list of elements.

## Description

When applied to a list of terms [`t1`; ...; `tn`] of the same type, the function mk_fset constructs an explicit set enumeration term `{t1, ..., tn}`. Note that duplicated elements are maintained in the resulting term, though this is logically the same as the set without them. If you need to generate enumerations for empty sets, use mk_setenum; in this case the type also needs to be specified.

## Failure

Fails if there are terms of more than one type in the list, or if the list is empty.

## Example

```
# mk_fset (map mk_small_numeral (0--7));;
val it : term = `{\small\verb%0, 1, 2, 3, 4, 5, 6, 7%}`
```

## See also

dest_setenum, is_setenum, mk_flist, mk_setenum.

# mk_fthm

```
mk_fthm : term list * term -> thm
```

## Synopsis
Create arbitrary theorem by adding additional 'false' assumption.

## Description
The call `mk_fthm(asl,c)` returns a theorem with conclusion `c` and assumption list `asl` together with the special assumption `_FALSITY_`, which is defined to be logically equivalent to `F` (false). This is the closest approach to `mk_thm` that does not involve adding a new axiom and so potentially compromising soundness.

## Failure
Fails if any of the given terms does not have Boolean type.

## Example

```
# mk_fthm([],`1 = 2`);;
val it : thm = _FALSITY_ |- 1 = 2
```

## Uses
Used for validity-checking of justification functions as a sanity check in tactic applications: see `VALID`.

## See also
`CHEAT_TAC`, `mk_thm`, `new_axiom`, `VALID`.

# mk_fun_ty

```
mk_fun_ty : hol_type -> hol_type -> hol_type
```

## Synopsis
Construct a function type.

## Description
The call `mk_fun_ty ty1 ty2` gives the function type `ty1->ty2`. This is an exact synonym of `mk_type("fun",[ty1; ty2])`, but a little more convenient.

## Failure
Never fails.

## Example

```
# mk_fun_ty `:num` `:num`;;
val it : hol_type = `:num->num`

# itlist mk_fun_ty [`:A`; `:B`; `:C`] `:bool`;;
val it : hol_type = `:A->B->C->bool`
```

## See also

dest_type, mk_type.

# mk_gabs

```
mk_gabs : term * term -> term
```

## Synopsis

Constructs a generalized abstraction.

## Description

Given a pair of terms `s` and `t`, the call `mk_gabs(s,t)` constructs a canonical 'generalized abstraction' that is thought of as 'some function that always maps `s` to `t`'. In the case where `s` is a variable, the result is an ordinary abstraction as constructed by `mk_abs`. In other cases, the canonical composite structure is created. Note that the logical construct is welldefined even if there is no function mapping `s` to `t`, and this function will always succeed, even if the resulting structure is not really useful.

## Failure

Never fails.

## Example

Here is a simple abstraction:

```
# mk_gabs(`x:bool`,`~x`);;
val it : term = `\x. ~x`
```

and here are a couple of potentially useful generalized ones:

```
# mk_gabs(`(x:num,y:num)`,`x + y + 1`);;
val it : term = `\(x,y). x + y + 1`

# mk_gabs(`CONS (h:num) t`,`if h = 0 then t else CONS h t`);;
val it : term = `\CONS h t. if h = 0 then t else CONS h t`
```

while here is a vacuous one about which nothing interesting will be proved, because there is no welldefined function that always maps `x + y` to `x`:

```
# mk_gabs(`x + y:num`,`x:num`);;
val it : term = `\(x + y). x`
```

## See also
`dest_gabs, GEN_BETA_CONV, is_gabs, list_mk_gabs.`

# mk_goalstate

`mk_goalstate : goal -> goalstate`

## Synopsis
Converts a goal into a 1-element goalstate.

## Description
Given a goal `g`, the call `mk_goalstate g` converts it into a goalstate with that goal as its only member. (A goalstate consists of a list of subgoals as well as justification and metavariable information.)

## Failure
Never fails.

## See also
`g, set_goal, TAC_PROOF.`

# mk_icomb

`mk_icomb : term * term -> term`

## Synopsis
Makes a combination, instantiating types in rator if necessary.

## Description
The call `mk_icomb(f,x)` makes the combination `f x`, just as with `mk_comb`, but if necessary to ensure the types are compatible it will instantiate type variables in `f` first.

## Failure
Fails if the rator type is impossible to instantiate comparibly.

## Example

The analogous call to the following using plain `mk_const` would fail:

```
# mk_icomb(`(!)`,`\x. x = 1`);;
Warning: inventing type variables
val it : term = `!x. x = 1`
```

## Uses

A handy way of making combinations involving polymorphic constants, without needing a manual instantiation of the generic type.

## See also

`list_mk_icomb, mk_comb, type_match.`

# mk_iff

```
mk_iff : term * term -> term
```

## Synopsis

Constructs a logical equivalence (Boolean equation).

## Description

`mk_iff(`t1`,`t2`)` returns `` `t1 <=> t2` ``.

## Failure

Fails with unless `t1` and `t2` both have Boolean type.

## Example

```
# mk_iff(`x = 1`,`1 = x`);;
val it : term = `x = 1 <=> 1 = x`
```

## Comments

Simply `mk_eq` has the same effect on successful calls. However `mk_iff` is slightly more efficient, and will fail if the terms do not have Boolean type.

## See also

`dest_iff, is_iff,mk_eq.`

# mk_imp

```
mk_imp : term * term -> term
```

## Synopsis
Constructs an implication.

## Description
`mk_imp(‘t1‘,‘t2‘)` returns `‘t1 ==> t2‘`.

## Failure
Fails with `mk_imp` if either `t1` or `t2` are not of type `‘:bool‘`.

## Example

```
# mk_imp(‘p /\ q‘,‘r:bool‘);;
val it : term = ‘p /\ q ==> r‘
```

## See also
`dest_imp, is_imp, list_mk_imp.`

# mk_intconst

`mk_intconst : num -> term`

## Synopsis
Converts an OCaml number to a canonical integer literal of type `:int`.

## Description
The call `mk_intconst n` where `n` is an OCaml number (type `num`) produces the canonical integer literal of type `:int` representing the integer `n`. This will be of the form ‘`&m`’ for a numeral `m` (when `n` is nonnegative) or ‘`-- &m`’ for a nonzero numeral `m` (when `n` is negative).

## Failure
Fails if applied to a number that is not an integer (type `num` also includes rational numbers).

## Example

```
# mk_intconst (Int (-101));;
val it : term = ‘-- &101‘

# type_of it;;
val it : hol_type = ‘:int‘
```

## See also
`dest_intconst, is_intconst, mk_realintconst.`

# mk_list

```
mk_list : term list * hol_type -> term
```

## Synopsis
Constructs object-level list from list of terms.

## Description
`mk_list([`t1`;...;`tn`],`:ty`)` returns ``[t1;...;tn]:(ty)list``. The type argument is required so that empty lists can be constructed. If you know the list is nonempty, you can just use `mk_flist` where this is not required.

## Failure
Fails with if any term in the list is not of the type specified as the second argument.

## Example

```
# mk_list([`1`; `2`],`:num`);;
val it : term = `[1; 2]`

# mk_list([],`:num`);;
val it : term = `[]`

# type_of it;;
val it : hol_type = `:(num)list`
```

## See also
`dest_cons`, `dest_list`, `is_cons`, `is_list`, `mk_cons`, `mk_flist`.

# mk_mconst

```
mk_mconst : string * hol_type -> term
```

## Synopsis
Constructs a constant with type matching.

## Description
`mk_mconst("const",`:ty`)` returns the constant ``const:ty``.

## Failure

Fails with `mk_mconst: ...` if the string supplied is not the name of a known constant, or if it is known but the type supplied is not the correct type for the constant.

## Example

```
# mk_mconst ("T",`:bool`);;
val it : term = `T`

# mk_mconst ("T",`:num`);;
Exception: Failure "mk_const: generic type cannot be instantiated".
```

## Comments

The primitive HOL Light facility for making constants is `mk_const`, which takes a type instantiation to apply to the constant's generic type. The function `mk_mconst` requires type matching and so is in general somewhat less efficient. However it is sometimes more convenient, and a natural inverse for `dest_const`.

## See also

`mk_const, dest_const, is_const, mk_var, mk_comb, mk_abs.`

# mk_neg

`mk_neg : term -> term`

## Synopsis

Constructs a logical negation.

## Description

`mk_neg `t`` returns ``~t``.

## Failure

Fails with `mk_neg` unless `t` is of type `bool`.

## Example

```
# mk_neg `p /\ q`;;
val it : term = `~(p /\ q)`

# mk_neg `~p`;;
val it : term = `~ ~p`
```

## See also

`dest_neg, is_neg.`

# mk_numeral

```
mk_numeral : num -> term
```

## Synopsis
Maps a nonnegative integer to corresponding numeral term.

## Description
The call `mk_numeral n` where `n` is a nonnegative integer of type `num` (this is OCaml's type of unlimited-precision numbers) returns the HOL numeral representation of `n`.

## Failure
Fails if the argument is negative or not integral (type `num` can include rationals).

## Example

```
# mk_numeral (Int 10);;
val it : term = `10`

# mk_numeral(pow2 64);;
val it : term = `18446744073709551616`
```

## Comments
The similar function `mk_small_numeral` works from a regular machine integer, Ocaml type `int`. If that suffices, it may be simpler.

## See also
`dest_numeral`, `dest_small_numeral`, `is_numeral`, `mk_small_numeral`, `term_of_rat`.

# mk_pair

```
mk_pair : term * term -> term
```

## Synopsis
Constructs object-level pair from a pair of terms.

## Description
`` mk_pair(`t1`,`t2`) `` returns `` `(t1,t2)` ``.

## Failure
Never fails.

## Example

```
# mk_pair(`x:real`,`T`);;
val it : term = `x,T`
```

## See also

dest_pair, is_pair, mk_cons.

# mk_primed_var

```
mk_primed_var : term list -> term -> term
```

## Synopsis

Rename variable to avoid specified names and constant names.

## Description

The call mk_primed_var avoid v will return a renamed variant of v, by adding primes, so that its name is not the same as any of the variables in the list avoid, nor the same as any constant name. It is a more conservative version of the renaming function variant.

## Failure

Fails if one of the items in the list avoids is not a variable, or if v itself is not.

## Example

This shows how the effect is more conservative than variant because it even avoids variables of the same name and different type:

```
# variant [`x:bool`] `x:num`;;
val it : term = `x`
# mk_primed_var [`x:bool`] `x:num`;;
val it : term = `x'`
```

and this shows how it also avoids constant names:

```
# mk_primed_var [] (mk_var("T",`:num`));;
val it : term = `T'`
```

## See also

variant, variants.

# mk_prover

```
mk_prover : ('a -> conv) -> ('a -> thm list -> 'a) -> 'a -> prover
```

## Synopsis
Construct a prover from applicator and state augmentation function.

## Description
The HOL Light simplifier (e.g. as invoked by `SIMP_TAC`) allows provers of type `prover` to be installed into simpsets, to automatically dispose of side-conditions. These may maintain a state dynamically and augment it as more theorems become available (e.g. a theorem `p |- p` becomes available when simplifying the consequenct of an implication `` `p ==> q` ``). In order to allow maximal flexibility in the data structure used to maintain state, provers are set up in an 'object-oriented' style, where the context is part of the prover function itself. A call `mk_prover app aug` where `app: 'a -> conv` is an application operation to prove a term using the context of type `'a` and `aug : 'a -> thm list -> 'a` is an augmentation operation to add whatever representation of the theorem list in the state of the prover is chosen, gives a canonical prover of this form. The crucial point is that the type `'a` is invisible in the resulting prover, so different provers can maintain their state in different ways. (In the trivial case, one might just use `thm list` for the state, and appending for the augmentation.)

## Failure
Does not normally fail unless the functions provided are abnormal.

## Uses
This is mostly for experts wishing to customize the simplifier.

## Comments
I learned of this ingenious trick for maintaining context from Don Syme, who discovered it by reading some code written by Richard Boulton. I was told by Simon Finn that there are similar ideas in the functional language literature for simulating existential types.

## See also
`apply_prover, augment, SIMP_CONV, SIMP_RULE, SIMP_TAC.`

# mk_realintconst

```
mk_realintconst : num -> term
```

## Synopsis
Converts an OCaml number to a canonical integer literal of type `:real`.

## Description

The call `mk_realintconst n` where `n` is an OCaml number (type `num`) produces the canonical literal of type `:real` representing the integer `n`. This will be of the form '`&m`' for a numeral `m` (when `n` is nonnegative) or '`-- &m`' for a nonzero numeral `m` (when `n` is negative).

## Failure

Fails if applied to a number that is not an integer (type `num` also includes rational numbers).

## Example

```
# mk_realintconst (Int (-101));;
val it : term = `-- &101`

# type_of it;;
val it : hol_type = `:real`
```

## See also

`dest_realintconst`, `is_realintconst`, `mk_intconst`, `rat_of_term`.

# mk_rewrites

```
mk_rewrites : bool -> thm -> thm list -> thm list
```

## Synopsis

Turn theorem into list of (conditional) rewrites.

## Description

Given a Boolean flag `b`, a theorem `th` and a list of theorems `thl`, the call `mk_rewrites b th thl` breaks `th` down into a collection of rewrites (for example, splitting conjunctions up into several sub-theorems) and appends them to the front of `thl` (which are normally theorems already processed in this way). Non-equational theorems `|- p` are converted to `|- p <=> T`. If the flag `b` is true, then implicational theorems `|- p ==> s = t` are used as conditional rewrites; otherwise they are converted to `|- (p ==> s = t) <=> T`. This function is applied inside `extend_basic_rewrites` and `set_basic_rewrites`.

## Failure

Never fails.

## Example

```
# ADD_CLAUSES;;
val it : thm =
  |- (!n. 0 + n = n) /\
     (!m. m + 0 = m) /\
     (!m n. SUC m + n = SUC (m + n)) /\
     (!m n. m + SUC n = SUC (m + n))

# mk_rewrites false ADD_CLAUSES [];;
val it : thm list =
  [|- 0 + n = n; |- m + 0 = m; |- SUC m + n = SUC (m + n);
   |- m + SUC n = SUC (m + n)]
```

## See also

extend_basic_rewrites, GEN_REWRITE_CONV, REWRITE_CONV, set_basic_rewrites, SIMP_CONV.

# mk_select

```
mk_select : term * term -> term
```

## Synopsis

Constructs a choice binding.

## Description

The call `mk_select(‘v‘,‘t‘)` returns the choice term `‘@v. t‘`.

## Failure

Fails if `v` is not a variable.

## See also

is_slect, mk_select.

# mk_setenum

```
mk_setenum : term list * hol_type -> term
```

## Synopsis

Constructs an explicit set enumeration from a list of elements.

## Description

When applied to a list of terms [`t1`; ...; `tn`] and a type ty, where each term in the list has type ty, the function mk_setenum constructs an explicit set enumeration term `{t1, ..., tn}`. Note that duplicated elements are maintained in the resulting term, though this is logically the same as the set without them. The type is needed so that the empty set can be constructed; if you know that the list is nonempty, you can use mk_fset instead.

## Failure

Fails if some term in the list has the wrong type, i.e. not ty.

## Example

```
# mk_setenum([`1`; `2`; `3`],`:num`);;
val it : term = `{1, 2, 3}`
```

## See also

dest_setenum, is_setenum, mk_fset, mk_list.

# mk_small_numeral

mk_small_numeral : int -> term

## Synopsis

Maps a nonnegative integer to corresponding numeral term.

## Description

The call mk_small_numeral n where n is a nonnegative OCaml machine integer returns the HOL numeral representation of n.

## Failure

Fails if the argument is negative.

## Example

```
# mk_small_numeral 12;;
val it : term = `12`
```

## Comments

The similar function mk_numeral works from an unlimited precision integer, OCaml type num. However, none of HOL's inference rules depend on the behaviour of machine integers, so logical soundness is not an issue.

## See also

dest_numeral, dest_small_numeral, is_numeral, mk_numeral, term_of_rat.

# mk_thm

```
mk_thm : term list * term -> thm
```

## Synopsis
Creates an arbitrary theorem as an axiom (dangerous!)

## Description
The function `mk_thm` can be used to construct an arbitrary theorem. It is applied to a pair consisting of the desired assumption list (possibly empty) and conclusion. All the terms therein should be of type `bool`.

```
mk_thm([`a1`;...;`an`],`c`) = ({a1,...,an} |- c)
```

## Failure
Fails unless all the terms provided for assumptions and conclusion are of type `bool`.

## Example
The following shows how to create a simple contradiction:

```
#mk_thm([],`F`);;
|- F
```

## Comments
Although `mk_thm` can be useful for experimentation or temporarily plugging gaps, its use should be avoided if at all possible in important proofs, because it can be used to create theorems leading to contradictions. You can check whether any axioms have been asserted by `mk_thm` or `new_axiom` by the call `axioms()`.

## See also
`CHEAT_TAC`, `mk_fthm`, `new_axiom`.

# mk_type

```
mk_type : string * hol_type list -> hol_type
```

## Synopsis
Constructs a type (other than a variable type).

## Description

mk_type("op",[`:ty1`;...;`:tyn`]) returns `:(ty1,...,tyn)op` where op is the name of a known n-ary type constructor.

## Failure

Fails with mk_type if the string is not the name of a known type, or if the type is known but the length of the list of argument types is not equal to the arity of the type constructor.

## Example

```
# mk_type ("bool",[]);;
val it : hol_type = `:bool`

# mk_type ("list",[`:bool`]);;
val it : hol_type = `:(bool)list`

# mk_type ("fun",[`:num`;`:bool`]);;
val it : hol_type = `:num->bool`
```

## See also

dest_type, mk_vartype.

# mk_uexists

mk_uexists : term * term -> term

## Synopsis

Term constructor for unique existence.

## Description

mk_uexists(`v`,`t`) returns `?!v. t`.

## Failure

Fails with if first term is not a variable or if t is not of type `:bool`.

## Example

```
# mk_uexists(`n:num`,`prime(n) /\ EVEN(n)`);;
val it : term = `?!n. prime n /\ EVEN n`
```

## See also

dest_uexists, is_uexists, mk_exists.

# mk_var

`mk_var : string * hol_type -> term`

## Synopsis
Constructs a variable of given name and type.

## Description
`mk_var("var",`:ty`)` returns the variable `` `var:ty` ``.

## Failure
Never fails.

## Comments
`mk_var` can be used to construct variables with names which are not acceptable to the term parser. In particular, a variable with the name of a known constant can be constructed using `mk_var`.

## See also
`dest_var, is_var, mk_const, mk_comb, mk_abs.`

# mk_vartype

`mk_vartype : string -> hol_type`

## Synopsis
Constructs a type variable of the given name.

## Description
`mk_vartype "A")` returns a type variable `` `:A` ``.

## Failure
Never fails.

## Example

```
# mk_vartype "Test";;
val it : hol_type = `:Test`

# mk_vartype "bool";;
val it : hol_type = `:bool`
```

Note that the second type is `\em not` the inbuilt type of Booleans, even though it prints like it.

## Comments

`mk_vartype` can be used to create type variables with names which will not parse, i.e. they cannot be entered by quotation. Using such type variables is probably bad practice. HOL Light convention is to start type variables with an uppercase letter.

## See also

`dest_vartype, is_vartype, mk_type.`

# MONO_TAC

`MONO_TAC : tactic`

## Synopsis

Attempt to prove monotonicity theorem automatically.

## Description

## Failure

Never fails but may have no effect.

## Example

We set up the following goal:

```
# g `(!x. P x ==> Q x) ==> (?y. P y /\ ~Q y) ==> (?y. Q y /\ ~P y)`;;
...
```

and after breaking it down, we reach the standard form expected for monotonicity goals:

```
# e STRIP_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [`!x. P x ==> Q x`]

`(?y. P y /\ ~Q y) ==> (?y. Q y /\ ~P y)`
```

Indeed, it is solved automatically:

```
# e MONO_TAC;;
val it : goalstack = No subgoals
```

## Comments

Normally, this kind of reasoning is automated by the inductive definitions package, so explicit use of this tactic is rare.

## See also

`monotonicity_theorems`, `new_inductive_definition`,
`prove_inductive_relations_exist`, `prove_monotonicity_hyps`.

# monotonicity_theorems

```
monotonicity_theorems : thm list ref
```

## Synopsis

List of monotonicity theorems for inductive definitions package.

## Description

The various tools for making inductive definitions, such as `new_inductive_definition`, need to prove certain 'monotonicity' side-conditions. They attempt to do so automatically by using various pre-proved theorems asserting the monotonicity of certain opertators. Normally, all this happens smoothly without user intervention, but if the inductive definition involves new operators, you may need to augment this list with corresponding monotonicity theorems.

## Failure

Not applicable.

## Example

Suppose we define a 'lexical order' construct:

```
# let LEX = define
   `(LEX(<<) [] l <=> F) /\
    (LEX(<<) l [] <=> F) /\
    (LEX(<<) (CONS h1 t1) (CONS h2 t2) <=>
          if h1 << h2 then LENGTH t1 = LENGTH t2
          else (h1 = h2) /\ LEX(<<) t1 t2)`;;
```

If we want to make an inductive definition that uses this — for example a lexicographic path order on a representation of first-order terms — we need to add a theorem asserting that this operation is monotonic. To prove it, we first establish a lemma:

```
# let LEX_LENGTH = prove
   (`!l1 l2 R. LEX(R) l1 l2 ==> (LENGTH l1 = LENGTH l2)`,
    REPEAT(LIST_INDUCT_TAC THEN SIMP_TAC[LEX]) THEN ASM_MESON_TAC[LENGTH]);;
```

and hence derive monotonicity:

```
# let MONO_LEX = prove
   (`(!x:A y:A. R x y ==> S x y) ==> LEX R x y ==> LEX S x y`,
    DISCH_TAC THEN
    MAP_EVERY (fun t -> SPEC_TAC(t,t)) [`x:A list`; `y:A list`] THEN
    REPEAT(LIST_INDUCT_TAC THEN REWRITE_TAC[LEX]) THEN
    ASM_MESON_TAC[LEX_LENGTH]);;
```

We can now make the inductive definitions package aware of it by:

```
# monotonicity_theorems := MONO_LEX::(!monotonicity_theorems);;
```

## See also

`new_inductive_definition`.

# MP_CONV

```
MP_CONV : conv -> thm -> thm
```

## Synopsis

Removes antecedent of implication theorem by solving it with a conversion.

## Description

The call `MP_CONV conv th`, where the theorem `th` has the form `A |- p ==> q`, attempts to solve the antecedent `p` by applying the conversion `conv` to it. If this conversion returns either `|- p` or `|- p <=> T`, then `MP_CONV` returns `A |- q`. Otherwise it fails.

## Failure

Fails if the conclusion of the theorem is not implicational or if the conversion fails to prove its antecedent.

## Example

Suppose we generate this 'epsilon-delta' theorem:

```
# let th = MESON[LE_REFL]
   `(!e. &0 < e / &2 <=> &0 < e) /\
    (!a x y e. abs(x - a) < e / &2 /\ abs(y - a) < e / &2 ==> abs(x - y) < e)
    ==> (!e. &0 < e ==> ?n. !m. n <= m ==> abs(x m - a) < e)
        ==> (!e. &0 < e ==> ?n. !m. n <= m ==> abs(x m - x n) < e)`;;
```

We can eliminate the antecedent:

```
# MP_CONV REAL_ARITH th;;
val it : thm =
  |- (!e. &0 < e ==> (?n. !m. n <= m ==> abs (x m - a) < e))
     ==> (!e. &0 < e ==> (?n. !m. n <= m ==> abs (x m - x n) < e))
```

## See also

MP, MATCH_MP.

# MP

```
MP : thm -> thm -> thm
```

## Synopsis

Implements the Modus Ponens inference rule.

## Description

When applied to theorems `A1 |- t1 ==> t2` and `A2 |- t1`, the inference rule `MP` returns the theorem `A1 u A2 |- t2`.

```
    A1 |- t1 ==> t2   A2 |- t1
   ----------------------------  MP
          A1 u A2 |- t2
```

## Failure

Fails unless the first theorem is an implication whose antecedent is the same as the conclusion of the second theorem (up to alpha-conversion).

## Example

```
# let th1 = TAUT `p ==> p \/ q`
  and th2 = ASSUME `p:bool`;;
val th1 : thm = |- p ==> p \/ q
val th2 : thm = p |- p

# MP th1 th2;;
val it : thm = p |- p \/ q
```

## See also

EQ_MP, MATCH_MP, MATCH_MP_TAC, MP_TAC.

# MP_TAC

MP_TAC : thm_tactic

## Synopsis

Adds a theorem as an entecedent to the conclusion of the goal.

## Description

When applied to the theorem A' |- s and the goal A ?- t, the tactic MP_TAC reduces the goal to A ?- s ==> t. Unless A' is a subset of A, this is an invalid tactic.

```
       A ?- t
   ==============  MP_TAC (A' |- s)
    A ?- s ==> t
```

## Failure

Never fails.

## Uses

For moving assumptions into the conclusion of the goal, which often makes it easier to manipulate via REWRITE_TAC or decompose by ANTS_TAC.

## See also

MATCH_MP_TAC, MP, UNDISCH_TAC.

# name_of

name_of : term -> string

## Synopsis
Gets the name of a constant or variable.

## Description
When applied to a term that is either a constant or a variable, `name_of` returns its name (its true name, even if there is an interface mapping for it in effect). When applied to any other term, it returns the empty string.

## Failure
Never fails.

## Example

```
# name_of `x:int`;;
val it : string = "x"

# name_of `SUC`;;
val it : string = "SUC"

# name_of `x + 1`;;
val it : string = ""
```

## See also
`dest_const`, `dest_var`.

# needs

```
needs : string -> unit
```

## Synopsis
Load a file if not already loaded.

## Description
The given file is loaded from the path as for `loadt`, unless it has already been loaded into the current session (by `loads`, `loadt` or `needs`) and has apparently (based on an MD5 checksum) not changed since then.

## Failure
Fails if the file is not found or generates a failure on loading.

### Example

If a proof relies on more number theory, you might start it with

```
needs "Examples/prime.ml";;
needs "Examples/pocklington.ml";;
```

If necessary, these files will be loaded as for `loadt`. However, if they have already been loaded (e.g. if the current proof is a component of a larger proof that has already used them), they will not be reloaded.

### Uses

The `needs` function gives a simple form of dependency management. It is good practice to start every file with a `needs` declaration for any library that it depends on.

### See also

`loads, loadt`

## net_of_cong

```
net_of_cong : thm -> (int * (term -> thm)) net -> (int * (term -> thm)) net
```

### Synopsis

Add a congruence rule to a net.

### Description

The underlying machinery in rewriting and simplification assembles (conditional) rewrite rules and other conversions into a net, including a priority number so that, for example, pure rewrites get applied before conditional rewrites. The congruence rules used by the simplifier to establish context (see `extend_basic_congs`) are also stored in this structure, with the lowest priority 4. A call `net_of_cong th net` adds `th` as a new congruence rule to `net` to yield an updated net.

### Failure

Fails unless the congruence is of the appropriate implicational form.

### See also

`extend_basic_congs, net_of_conv, net_of_thm.`

## net_of_conv

```
net_of_conv : term -> 'a -> (int * 'a) net -> (int * 'a) net
```

## Synopsis

The underlying machinery in rewriting and simplification assembles (conditional) rewrite rules and other conversions into a net, including a priority number so that, for example, pure rewrites get applied before conditional rewrites. A call `net_of_conv 'pat' cnv net` will add `cnv` to `net` with priority 2 (lower than pure rewrites but higher than conditional ones) to give a new net; this net can be used by `REWRITES_CONV`, for example. The term `pat` is a pattern used inside the net to place `conv` appropriately (see `enter` for more details). This means that `cnv` will never even be tried on terms that clearly cannot be instances of `pat`.

## Failure

Never fails.

## See also

`enter`, `net_of_cong`, `lookup`, `net_of_thm`, `REWRITES_CONV`.

# net_of_thm

```
net_of_thm : bool -> thm -> (int * (term -> thm)) net -> (int * (term -> thm)) net
```

## Synopsis

Insert a theorem into a net as a (conditional) rewrite.

## Description

The underlying machinery in rewriting and simplification assembles (conditional) rewrite rules and other conversions into a net, including a priority number so that, for example, pure rewrites get applied before conditional rewrites. Such a net can then be used by `REWRITES_CONV`. A call `net_of_thm rf th net` where `th` is a pure or conditional equation (as constructed by `mk_rewrites`, for example) will insert that rewrite rule into the net with priority 1 (the highest) for a pure rewrite or 3 for a conditional rewrite, to yield an updated net.

If `rf` is `true`, it indicates that this net will be used for repeated rewriting (e.g. as in `REWRITE_CONV` rather than `ONCE_REWRITE_CONV`), and therefore equations are simply discarded without changing the net if the LHS occurs free in the RHS. This does not exclude more complicated looping situations, but is still useful.

## Failure

Fails on a theorem that is neither a pure nor conditional equation.

## See also

`mk_rewrites`, `net_of_cong`, `net_of_conv`, `REWRITES_CONV`.

# new_axiom

```
new_axiom : term -> thm
```

## Synopsis
Sets up a new axiom.

## Description
If tm is a term of type bool, a call new_axiom tm) creates a theorem

```
|- tm
```

## Failure
Fails if the given term does not have type bool.

## Example

```
# let ax = new_axiom `x = 1`;;
val ax : thm = |- x = 1
```

Note that as with all theorems, variables are implicitly universally quantified, so this axiom asserts that all numbers are equal to 1. Of course, we can then derive a contradiction:

```
CONV_RULE NUM_REDUCE_CONV (INST [`0`,`x:num`] ax);;
val it : thm = |- F
```

Normal use of HOL Light should avoid asserting axioms. They can lead to inconsistency, albeit not in such an obvious way. Provided theories are extended by definitions, consistency is preserved.

## Comments
For most purposes, it is unnecessary to declare new axioms: all of classical mathematics can be derived by definitional extension alone. Proceeding by definition is not only more elegant, but also guarantees the consistency of the deductions made. However, there are certain entities which cannot be modelled in simple type theory without further axioms, such as higher transfinite ordinals.

## See also
mk_thm, new_definition.

# new_basic_definition

```
new_basic_definition : term -> thm
```

## Synopsis
Makes a simple new definition of the form c = t.

## Description

If `t` is a closed term and `c` a variable whose name has not been used as a constant, then `new_basic_definition ‘c = t‘` will define a new constant `c` and return the theorem `|- c = t` for that new constant (not the variable in the given term). There is an additional restriction that all type variables involved in `t` must occur in the constant's type.

## Failure

Fails if `c` is already a constant.

## Example

Here is a simple example

```
# let googolplex = new_basic_definition
    ‘googolplex = 10 EXP (10 EXP 100)‘;;
val googolplex : thm = |- googolplex = 10 EXP (10 EXP 100)
```

and of course we can equally well use logical equivalence:

```
# let true_def = new_basic_definition ‘true <=> T‘;;
val true_def : thm = |- true <=> T
```

The following example helps to explain why the restriction on type variables is present:

```
# new_basic_definition ‘trivial <=> !x:A. x = x‘;;
Exception:
Failure "new_definition: Type variables not reflected in constant".
```

If we had been allowed to get back a definitional theorem, we could separately type-instantiate it to the 1-element type `1` and the 2-element type `bool`. In one case the RHS is true, and in the other it is false, yet both are asserted equal to the constant `trivial`.

## Comments

There are simpler or more convenient ways of making definitions, such as `define` and `new_definition`, but this is the primitive principle underlying them all.

## See also

`define`, `new_definition`, `new_inductive_definition`, `new_recursive_definition`, `new_specification`.

# new_basic_type_definition

```
new_basic_type_definition : string -> string * string -> thm -> thm * thm
```

## Synopsis

Introduces a new type in bijection with a nonempty subset of an existing type.

## Description

The call `new_basic_type_definition "ty" ("mk","dest") th` where `th` is a theorem of the form `|- P x` (say `x` has type `rep`) will introduce a new type called `ty` plus two new constants `mk:rep->ty` and `dest:ty->rep`, and return two theorems that together assert that `mk` and `dest` establish a bijection between the universe of the new type `ty` and the subset of the type `rep` identified by the predicate `P`: `|- mk(dest a) = a` and `|- P r <=> dest(mk r) = r`. If the theorem involves type variables `A1,...,An` then the new type will be an $n$-ary type constructor rather than a basic type. The theorem is needed to ensure that that set is nonempty; all types in HOL are nonempty.

## Failure

Fails if any of the type or constant names is already in use, if the conclusion of the theorem is not a combination, or if its rator `P` contains free variables.

## Example

Here we define a basic type with 32 elements:

```
# let th = ARITH_RULE `(\x. x < 32) 0`;;
val th : thm = |- (\x. x < 32) 0
# let absth,repth = new_basic_type_definition "32" ("mk_32","dest_32") th;;
val absth : thm = |- mk_32 (dest_32 a) = a
val repth : thm = |- (\x. x < 32) r <=> dest_32 (mk_32 r) = r
```

and here is a declaration of a type of finite sets over a base type, a unary type constructor:

```
# let th = CONJUNCT1 FINITE_RULES;;
val th : thm = |- FINITE {\small\verb%%}

# let tybij = new_basic_type_definition "fin" ("mk_fin","dest_fin") th;;
val tybij : thm * thm =
  (|- mk_fin (dest_fin a) = a, |- FINITE r <=> dest_fin (mk_fin r) = r)
```

so now types like `:(num)fin` make sense.

## Comments

This is the primitive principle of type definition in HOL Light, but other functions like `define_type` or `new_type_definition` are usually more convenient.

## See also

`define_type, new_type_definition.`

# new_constant

```
new_constant : string * hol_type -> unit
```

## Synopsis
Declares a new constant.

## Description
A call `new_constant("c",`:ty`)` makes `c` a constant with most general type `ty`.

## Failure
Fails if there is already a constant of that name in the current theory.

## Example

```
#new_constant("graham's_number",`:num`);;
val it : unit = ()
```

## Uses
Can be useful for declaring some arbitrary parameter, but more usually a prelude to some new axioms about the constant introduced. Take care when using `new_axiom`!

## See also
`constants, new_axiom, new_definition.`

# new_definition

```
new_definition : term -> thm
```

## Synopsis
Declare a new constant and a definitional axiom.

## Description
The function `new_definition` provides a facility for definitional extensions. It takes a term giving the desired definition. The value returned by `new_definition` is a theorem stating the definition requested by the user.

Let `v_1`,...,`v_n` be tuples of distinct variables, containing the variables `x_1,...,x_m`. Evaluating `new_definition `c v_1 ... v_n = t``, where `c` is a variable whose name is not already used as a constant, declares `c` to be a new constant and returns the theorem:

```
|- !x_1 ... x_m. c v_1 ... v_n = t
```

Optionally, the definitional term argument may have any of its variables universally quantified.

## Failure
`new_definition` fails if `c` is already a constant or if the definition does not have the right form.

## Example

A NAND relation on signals indexed by 'time' can be defined as follows.

```
# new_definition
    `NAND2  (in_1,in_2) out <=> !t:num. out t <=> ~(in_1 t /\ in_2 t)`;;
val it : thm =
  |- !out in_1 in_2.
        NAND2 (in_1,in_2) out <=> (!t. out t <=> ~(in_1 t /\ in_2 t))
```

## Comments

Note that the conclusion of the theorem returned is essentially the same as the term input by the user, except that `c` was a variable in the original term but is a constant in the returned theorem. The function `define` is significantly more flexible in the kinds of definition it allows, but for some purposes this more basic principle is fine.

## See also

`define`, `new_basic_definition`, `new_inductive_definition`, `new_recursive_definition`, `new_specification`.

# new_inductive_definition

```
new_inductive_definition : term -> thm * thm * thm
```

## Synopsis

Define a relation or family of relations inductively.

## Description

The function `new_inductive_definition` is applied to a conjunction of "rules" of the form `!x1...xn. Pi ==> Ri t1 ... tk`. This conjunction is interpreted as an inductive definition of a set of relations `Ri` (however many appear in the consequents of the rules). That is, the relations are defined to be the smallest ones closed under the rules. The function `new_inductive_definition` will convert this into explicit definitions, define a new constant for each `Ri`, and return a triple of theorems. The first one will be the "rule" theorem, which essentially matches the input clauses except that the `Ri` are now the new constants; this simply says that the new relations are indeed closed under the rules. The second theorem is an induction theorem, asserting that the relations are the least ones closed under the rules. Finally, the cases theorem gives a case analysis theorem showing how each set of values satisfying the relation may be composed.

## Failure

Fails if the clauses are malformed, if the constants are already in use, or if there are unproven monotonicity hypotheses. In the last case, you can try `prove_inductive_relations_exist` to examine these hypotheses, and either try to prove them manually or extend `monotonicity_theorems` to let HOL do it.

## Example

A classic example where we have mutual induction is the set of even and odd numbers:

```
# let eo_RULES,eo_INDUCT, eo_CASES = new_inductive_definition
   `even(0) /\ odd(1) /\
    (!n. even(n) ==> odd(n + 1)) /\
    (!n. odd(n) ==> even(n + 1))`;;
val eo_RULES : thm =
  |- even 0 /\
     odd 1 /\
     (!n. even n ==> odd (n + 1)) /\
     (!n. odd n ==> even (n + 1))
val eo_INDUCT : thm =
  |- !odd' even'.
         even' 0 /\
         odd' 1 /\
         (!n. even' n ==> odd' (n + 1)) /\
         (!n. odd' n ==> even' (n + 1))
         ==> (!a0. odd a0 ==> odd' a0) /\ (!a1. even a1 ==> even' a1)
val eo_CASES : thm =
  |- (!a0. odd a0 <=> a0 = 1 \/ (?n. a0 = n + 1 /\ even n)) /\
     (!a1. even a1 <=> a1 = 0 \/ (?n. a1 = n + 1 /\ odd n))
```

Note that the 'rules' theorem corresponds exactly to the input, and says that indeed the relations do satisfy the rules. The 'induction' theorem says that the relations are the minimal ones satisfying the rules. You can use this to prove properties by induction, e.g. the relationship with the pre-defined concepts of odd and even:

```
# g `(!n. odd(n) ==> ODD(n)) /\ (!n. even(n) ==> EVEN(n))`;;
```

applying the induction theorem:

```
# e(MATCH_MP_TAC eo_INDUCT);;
val it : goalstack = 1 subgoal (1 total)

`EVEN 0 /\
 ODD 1 /\
 (!n. EVEN n ==> ODD (n + 1)) /\
 (!n. ODD n ==> EVEN (n + 1))`
```

This is easily finished off by, for example:

```
# e(REWRITE_TAC[GSYM NOT_EVEN; EVEN_ADD; ARITH]);;
val it : goalstack = No subgoals
```

For another example, consider defining a simple propositional logic:

```
# parse_as_infix("-->",(13,"right"));;
val it : unit = ()
# let form_tybij = define_type "form = False | --> form form";;
val form_tybij : thm * thm =
  (|- !P. P False /\ (!a0 a1. P a0 /\ P a1 ==> P (a0 --> a1)) ==> (!x. P x),
   |- !f0 f1.
          ?fn. fn False = f0 /\
               (!a0 a1. fn (a0 --> a1) = f1 a0 a1 (fn a0) (fn a1)))
```

and making an inductive definition of the provability relation:

```
# parse_as_infix("|--",(11,"right"));;
val it : unit = ()

# let provable_RULES,provable_INDUCT,provable_CASES = new_inductive_definition
 `(!p. p IN A ==> A |-- p) /\
  (!p q. A |-- p --> (q --> p)) /\
  (!p q r. A |-- (p --> q --> r) --> (p --> q) --> (p --> r)) /\
  (!p. A |-- ((p --> False) --> False) --> p) /\
  (!p q. A |-- p --> q /\ A |-- p ==> A |-- q)`;;
val provable_RULES : thm =
  |- !A. (!p. p IN A ==> A |-- p) /\
         (!p q. A |-- p --> q --> p) /\
         (!p q r. A |-- (p --> q --> r) --> (p --> q) --> p --> r) /\
         (!p. A |-- ((p --> False) --> False) --> p) /\
         (!p q. A |-- p --> q /\ A |-- p ==> A |-- q)
val provable_INDUCT : thm =
  |- !A |--'.
         (!p. p IN A ==> |--' p) /\
         (!p q. |--' (p --> q --> p)) /\
         (!p q r. |--' ((p --> q --> r) --> (p --> q) --> p --> r)) /\
         (!p. |--' (((p --> False) --> False) --> p)) /\
         (!p q. |--' (p --> q) /\ |--' p ==> |--' q)
         ==> (!a. A |-- a ==> |--' a)
val provable_CASES : thm =
```

## See also

prove_inductive_relations_exist.

# new_recursive_definition

```
new_recursive_definition : thm -> term -> thm
```

## Synopsis

Define recursive function over inductive type.

## Description

new_recursive_definition provides the facility for defining primitive recursive functions on arbitrary inductive types. The first argument is the primitive recursion theorem for the concrete type in question; this is normally the second theorem obtained from define_type. The second argument is a term giving the desired primitive recursive function definition. The value returned by new_recursive_definition is a theorem stating the primitive recursive definition requested by the user. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

Let C1, ..., Cn be the constructors of the type, and let '(Ci vs)' represent a (curried) application of the ith constructor to a sequence of variables. Then a curried primitive recursive function fn over ty can be specified by a conjunction of (optionally universally-quantified) clauses of the form:

```
   fn v1 ... (C1 vs1) ... vm  =  body1   /\
   fn v1 ... (C2 vs2) ... vm  =  body2   /\
                             .
                             .
   fn v1 ... (Cn vsn) ... vm  =  bodyn
```

where the variables v1, ..., vm, vs are distinct in each clause, and where in the ith clause fn appears (free) in bodyi only as part of an application of the form:

```
   `fn t1 ... v ... tm`
```

in which the variable v of type ty also occurs among the variables vsi.

If <definition> is a conjunction of clauses, as described above, then evaluating:

```
   new_recursive_definition th `<definition>`;;
```

automatically proves the existence of a function fn that satisfies the defining equations, and then declares a new constant with this definition as its specification.

new_recursive_definition also allows the supplied definition to omit clauses for any number of constructors. If a defining equation for the ith constructor is omitted, then the value of fn at that constructor:

```
fn v1 ... (Ci vsi) ... vn
```

is left unspecified (fn, however, is still a total function).

## Failure

Fails if the definition cannot be matched up with the recursion theorem provided (you may find that define still works in such cases), or if there is already a constant of the given name.

## Example

The following defines a function to produce the union of a list of sets:

```
# let LIST_UNION = new_recursive_definition list_RECURSION
  `(LIST_UNION [] = {}) /\
   (LIST_UNION (CONS h t) = h UNION (LIST_UNION t))`;;
    Warning: inventing type variables
val ( LIST_UNION ) : thm =
  |- LIST_UNION [] = {} /\ LIST_UNION (CONS h t) = h UNION LIST_UNION t
```

## Comments

For many purposes, define is a simpler way of defining recursive types; it has a simpler interface (no need to specify the recursion theorem to use) and it is more general. However, for suitably constrained definitions new_recursive_definition works well and is much more efficient.

## See also

define, prove_inductive_relations_exist, prove_recursive_functions_exist.

# new_specification

```
new_specification : string list -> thm -> thm
```

## Synopsis

Introduces a constant or constants satisfying a given property.

## Description

The ML function new_specification implements the primitive rule of constant specification for the HOL logic. Evaluating:

```
new_specification ["c1";...;"cn"] |- ?x1...xn. t
```

simultaneously introduces new constants named `c1`, ..., `cn` satisfying the property:

```
|- t[c1,...,cn/x1,...,xn]
```

This theorem is returned by the call to `new_specification`.

## Failure

`new_specification` fails if any one of `‘c1‘`, ..., `‘cn‘` is already a constant.

## Uses

`new_specification` can be used to introduce constants that satisfy a given property without having to make explicit equational constant definitions for them. For example, the built-in constants `MOD` and `DIV` are defined in the system by first proving the theorem:

```
# DIVMOD_EXIST_0;;
val it : thm =
  |- !m n. ?q r. if n = 0 then q = 0 /\ r = 0 else m = q * n + r /\ r < n
```

Skolemizing it to made the parametrization explicit:

```
# let th = REWRITE_RULE[SKOLEM_THM] DIVMOD_EXIST_0;;
val th : thm =
  |- ?q r.
         !m n.
             if n = 0
             then q m n = 0 /\ r m n = 0
             else m = q m n * n + r m n /\ r m n < n
```

and then making the constant specification:

```
# new_specification ["DIV"; "MOD"] th;;
```

giving the theorem:

```
# DIVISION_0;;
val it : thm =
  |- !m n.
         if n = 0
         then m DIV n = 0 /\ m MOD n = 0
         else m = m DIV n * n + m MOD n /\ m MOD n < n
```

## See also

`define`, `new_definition`.

# new_type_abbrev

```
new_type_abbrev : string * hol_type -> unit
```

## Synopsis
Sets up a new type abbreviation.

## Description
A call `new_type_abbrev("ab",`:ty`` creates a new type abbreviation `ab` for the type `ty`. In future, `` `:ab` `` may be used rather than the perhaps complicated expresion `` `:ty` ``. Note that the association is purely an abbreviation for parsing. Type abbreviations have no logical significance; types are internally represented after the abbreviations have been fully expanded. At present, type abbreviations are not reversed when printing types, mainly because this may contract abbreviations where it is unwanted.

## Failure
Never fails.

## Example

```
# new_type_abbrev("bitvector",`:bool list`);;
val it : unit = ()

# `LENGTH(x:bitvector)`;;
val it : term = `LENGTH x`

# type_of (rand it);;
val it : hol_type = `:(bool)list`
```

## See also
`define_type`, `new_type_definition`, `remove_type_abbrev`, `type_abbrevs`.

# new_type_definition

```
new_type_definition : string -> string * string -> thm -> thm
```

## Synopsis
Introduces a new type in bijection with a nonempty subset of an existing type.

## Description

The call `new_basic_type_definition "ty" ("mk","dest") th` where `th` is a theorem of the form `|- ?x. P x` (say `x` has type `rep`) will introduce a new type called `ty` plus two new constants `mk:rep->ty` and `dest:ty->rep`, and return a theorem asserting that `mk` and `dest` establish a bijection between the universe of the new type `ty` and the subset of the type `rep` identified by the predicate `P`:

```
|- (!a. mk(dest a) = a) /\ (!r. P r <=> dest(mk r) = r)
```

If the theorem involves type variables `A1,...,An` then the new type will be an $n$-ary type constructor rather than a basic type. The theorem is needed to ensure that that set is nonempty; all types in HOL are nonempty.

## Example

Here we define a basic type with 7 elements:

```
# let th = prove(`?x. x < 7`,EXISTS_TAC `0` THEN ARITH_TAC);;
val th : thm = |- ?x. x < 7

# let tybij = new_type_definition "7" ("mk_7","dest_7") th;;
val tybij : thm =
  |- (!a. mk_7 (dest_7 a) = a) /\ (!r. r < 7 <=> dest_7 (mk_7 r) = r)
```

and here is a declaration of a type of finite sets over a base type, a unary type constructor:

```
# let th = MESON[FINITE_RULES] `?s:A->bool. FINITE s`;;
 0..0..solved at 2
CPU time (user): 0.
val th : thm = |- ?s. FINITE s

# let tybij = new_type_definition "finiteset" ("mk_fin","dest_fin") th;;
val tybij : thm =
  |- (!a. mk_fin (dest_fin a) = a) /\
     (!r. FINITE r <=> dest_fin (mk_fin r) = r)
```

so now types like `:(num)finiteset` make sense.

## Failure

Fails if any of the type or constant names is already in use, if the conclusion of the theorem is not an existentially quantified term, or if the rator `P` of its body contains free variables.

## See also

`define_type, new_basic_type_definition, new_type_abbrev.`

# new_type

```
new_type : string * int -> unit
```

## Synopsis
Declares a new type or type constructor.

## Description
A call `new_type("t",n)` declares a new `n`-ary type constructor called `t`; if `n` is zero, this is just a new base type.

## Failure
Fails if HOL is there is already a type operator of that name in the current theory.

## Example
A version of ZF set theory might declare a new type `set` and start using it as follows:

```
# new_type("set",0);;
val it : unit = ()
# new_constant("mem",`:set->set->bool`);;
val it : unit = ()
# parse_as_infix("mem",(11,"right"));;
val it : unit = ()
# let ZF_EXT = new_axiom `(!z. z mem x <=> z mem y) ==> (x = y)`;;
val ( ZF_EXT ) : thm = |- (!z. z mem x <=> z mem y) ==> x = y
```

## Comments
As usual, asserting new concepts is discouraged; if possible it is better to use type definitions; see `new_type_definition` and `define_type`.

## See also
`define_type`, `new_axiom`, `new_constant`, `new_definition`, `new_type_definition`.

# NNFC_CONV

```
NNFC_CONV : conv
```

## Synopsis
Convert a term to negation normal form.

## Description
The conversion `NNFC_CONV` proves a term equal to an equivalent in 'negation normal form' (NNF). This means that other propositional connectives are eliminated in favour of conjunction ('`/\`'), disjunction ('`\/`') and negation ('`~`'), and the negations are pushed down to the level of atomic formulas, also through universal and existential quantifiers, with double negations eliminated.

## Failure
Never fails; on non-Boolean terms it just returns a reflexive theorem.

## Example

```
# NNFC_CONV `(!x. p(x) <=> q(x)) ==> ~ ?y. p(y) /\ ~q(y)`;;
Warning: inventing type variables
val it : thm =
  |- (!x. p x <=> q x) ==> ~(?y. p y /\ ~q y) <=>
     (?x. (p x \/ q x) /\ (~p x \/ ~q x)) \/ (!y. ~p y \/ q y)
```

## Uses

Mostly useful as a prelude to automated proof procedures, but users may sometimes find it useful.

## Comments

A toplevel equivalence `p <=> q` is converted to `(p \/ ~q) /\ (~p \/ q)`. In general this "splitting" of equivalences is done with the expectation that the final formula may be put into conjunctive normal form (CNF), as a prelude to a proof (rather than refutation) procedure. An otherwise similar conversion `NNC_CONV` prefers a 'disjjunctive' splitting and is better suited for a term that will later be translated to DNF for refutation.

## See also

`GEN_NNF_CONV`, `NNF_CONV`.

# NNF_CONV

```
NNF_CONV : conv
```

## Synopsis

Convert a term to negation normal form.

## Description

The conversion `NNF_CONV` proves a term equal to an equivalent in 'negation normal form' (NNF). This means that other propositional connectives are eliminated in favour of conjunction ('`/\`'), disjunction ('`\/`') and negation ('`~`'), and the negations are pushed down to the level of atomic formulas, also through universal and existential quantifiers, with double negations eliminated.

## Failure

Never fails; on non-Boolean terms it just returns a reflexive theorem.

## Example

```
# NNF_CONV `(!x. p(x) <=> q(x)) ==> ~ ?y. p(y) /\ ~q(y)`;;
Warning: inventing type variables
val it : thm =
  |- (!x. p x <=> q x) ==> ~(?y. p y /\ ~q y) <=>
     (?x. p x /\ ~q x \/ ~p x /\ q x) \/ (!y. ~p y \/ q y)
```

## Uses

Mostly useful as a prelude to automated proof procedures, but users may sometimes find it useful.

## Comments

A toplevel equivalence `p <=> q` is converted to `(p /\ q) \/ (~p /\ ~q)`. In general this "splitting" of equivalences is done with the expectation that the final formula may be put into disjunctive normal form (DNF), as a prelude to a refutation procedure. An otherwise similar conversion `NNFC_CONV` prefers a 'conjunctive' splitting and is better suited for a term that will later be translated to CNF.

## See also

`GEN_NNF_CONV`, `NNFC_CONV`.

# NO_CONV

`NO_CONV : conv`

## Synopsis

Conversion that always fails.

## Failure

`NO_CONV` always fails.

## See also

`ALL_CONV`.

# NO_TAC

`NO_TAC : tactic`

## Synopsis
Tactic that always fails.

## Description
Whatever goal it is applied to, `NO_TAC` always fails with `Failure "NO_TAC"`.

## Failure
Always fails.

## Example
However trivial the goal, `NO_TAC` always fails:

```
# g `T`;;
val it : goalstack = 1 subgoal (1 total)

`T`

# e NO_TAC;;
Exception: Failure "NO_TAC".
```

however, `tac THEN NO_TAC` will never reach `NO_TAC` if `tac` leaves no subgoals:

```
# e(REWRITE_TAC[] THEN NO_TAC);;
val it : goalstack = No subgoals
```

## Uses
Can be useful in forcing certain "speculative" tactics to fail unless they solve the goal completely. For example, you might wish to break down a huge conjunction of goals and attempt to solve as many conjuncts as possible by just rewriting with a list of theorems `[thl]`. You could do:

```
REPEAT CONJ_TAC THEN REWRITE_TAC[thl]
```

However, if you don't want to apply the rewrites unless they result in an immediate solution, you can do instead:

```
REPEAT CONJ_TAC THEN TRY(REWRITE_TAC[thl] THEN NO_TAC)
```

## See also
`ALL_TAC`, `ALL_THEN`, `FAIL_TAC`, `NO_THEN`.

# NOT_ELIM

```
NOT_ELIM : thm -> thm
```

## Synopsis
Transforms `|- ~t` into `|- t ==> F`.

## Description
When applied to a theorem `A |- ~t`, the inference rule `NOT_ELIM` returns the theorem `A |- t ==> F`.

```
     A |- ~t
  --------------  NOT_ELIM
   A |- t ==> F
```

## Failure
Fails unless the theorem has a negated conclusion.

## Example

```
  # let th = UNDISCH(TAUT `p ==> ~ ~p`);;
  val th : thm = p |- ~ ~p

  # NOT_ELIM th;;
  val it : thm = p |- ~p ==> F
```

## See also
`EQF_ELIM, EQF_INTRO, NOT_INTRO.`

# NO_THEN

`NO_THEN : thm_tactical`

## Synopsis
Theorem-tactical which always fails.

## Description
When applied to a theorem-tactic and a theorem, the theorem-tactical `NO_THEN` always fails with `Failwith "NO_THEN"`.

## Failure
Always fails when applied to a theorem-tactic and a theorem (note that it never gets as far as being applied to a goal!)

## Uses
Writing compound tactics or tacticals.

## See also
`ALL_TAC, ALL_THEN, FAIL_TAC, NO_TAC.`

# nothing

`nothing : 'a -> 'b list * 'a`

## Synopsis
Trivial parser that parses nothing.

## Description
The parser `nothing` parses nothing: it returns the empty list as its parsed item and all the input as its unparsed input.

## Failure
Never fails.

## Uses
This can be useful in alternations (‘`||`’) with other parsers producing a list of items.

## Comments
This is one of a suite of combinators for manipulating “parsers”. A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also
`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `possibly`, `rightbin`, `some`.

# NOT_INTRO

`NOT_INTRO : thm -> thm`

## Synopsis
Transforms `|- t ==> F` into `|- ~t`.

## Description

When applied to a theorem `A |- t ==> F`, the inference rule `NOT_INTRO` returns the theorem `A |- ~t`.

```
    A |- t ==> F
   --------------  NOT_INTRO
      A |- ~t
```

## Failure

Fails unless the theorem has an implicative conclusion with `F` as the consequent.

## Example

```
  # let th = TAUT `F ==> F`;;
  val th : thm = |- F ==> F

  # NOT_INTRO th;;
  val it : thm = |- ~F
```

## See also

`EQF_ELIM`, `EQF_INTRO`, `NOT_ELIM`.

# nsplit

```
nsplit : ('a -> 'b * 'a) -> 'c list -> 'a -> 'b list * 'a
```

## Synopsis

Applies a destructor in right-associative mode a specified number of times.

## Description

If `d` is an inverse to a binary constructor `f`, then

```
  nsplit d l (f(x1,f(x2,...f(xn,y))))
```

where the list `l` has length `k`, returns

```
  ([x1;...;xk],f(x(k+1),...f(xn,y))
```

## Failure

Never fails.

## Example

```
# nsplit dest_conj [1;2;3] `a /\ b /\ c /\ d /\ e /\ f`;;
val it : term list * term = ([`a`; `b`; `c`], `d /\ e /\ f`)
```

## See also
`splitlist`, `rev_splitlist`, `striplist`.

# null_inst

`null_inst : instantiation`

## Synopsis
Empty instantiation.

## Description
Several functions use objects of type `instantiation`, consisting of type and term instantiations and higher-order matching information. This instantiation `null_inst` is the trivial instantiation that does nothing.

## Failure
Not applicable.

## Example
Instantiating a term with it has no effect:

```
# instantiate null_inst `x + 1 = 2`;;
val it : term = `x + 1 = 2`
```

## See also
`instantiate`, `INSTANTIATE`, `INSTANTIATE_ALL`, `term_match`.

# null_meta

`null_meta : term list * instantiation`

## Synopsis
Empty metavariable information.

## Description

This is a pair consisting of an empty list of terms and a null instantiation (see `null_inst`). It is used inside most tactics to indicate that they do nothing interesting with metavariables.

## Failure

Not applicable.

## Comments

This is not intended for general use, but readers writing custom tactics from scratch may find it convenient.

## See also

`null_inst`.

# num_0

```
num_0 : num
```

## Synopsis

Constant zero in unlimited-size integers.

## Description

The constant `num_0` is bound to the integer constant 0 in the unlimited-precision numbers provided by the OCaml `Num` library.

## Failure

Not applicable.

## Uses

Exactly the same as `Int 0`, but may save recreation of a cons cell each time.

## See also

`num_1`, `num_2`, `num_10`.

# num_10

```
num_10 : num
```

## Synopsis

Constant zero in unlimited-size integers.

## Description
The constant `num_10` is bound to the integer constant 10 in the unlimited-precision numbers provided by the OCaml `Num` library.

## Failure
Not applicable.

## Uses
Exactly the same as `Int 10`, but may save recreation of a cons cell each time.

## See also
`num_0, num_1, num_2.`

# num_1

```
num_1 : num
```

## Synopsis
Constant zero in unlimited-size integers.

## Description
The constant `num_1` is bound to the integer constant 1 in the unlimited-precision numbers provided by the OCaml `Num` library.

## Failure
Not applicable.

## Uses
Exactly the same as `Int 1`, but may save recreation of a cons cell each time.

## See also
`num_0, num_2, num_10.`

# num_2

```
num_2 : num
```

## Synopsis
Constant zero in unlimited-size integers.

## Description

The constant `num_2` is bound to the integer constant 2 in the unlimited-precision numbers provided by the OCaml `Num` library.

## Failure

Not applicable.

## Uses

Exactly the same as `Int 2`, but may save recreation of a cons cell each time.

## See also

`num_0`, `num_1`, `num_10`.

# NUM_ADD_CONV

```
NUM_ADD_CONV : term -> thm
```

## Synopsis

Proves what the sum of two natural number numerals is.

## Description

If `n` and `m` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_ADD_CONV ‘n + m‘` returns the theorem:

```
|- n + m = s
```

where `s` is the numeral that denotes the sum of the natural numbers denoted by `n` and `m`.

## Failure

`NUM_ADD_CONV tm` fails if `tm` is not of the form `‘n + m‘`, where `n` and `m` are numerals.

## Example

```
# NUM_ADD_CONV ‘75 + 25‘;;
val it : thm = |- 75 + 25 = 100
```

## See also

`NUM_DIV_CONV`, `NUM_EQ_CONV`, `NUM_EVEN_CONV`, `NUM_EXP_CONV`, `NUM_FACT_CONV`, `NUM_GE_CONV`, `NUM_GT_CONV`, `NUM_LE_CONV`, `NUM_LT_CONV`, `NUM_MOD_CONV`, `NUM_MULT_CONV`, `NUM_ODD_CONV`, `NUM_PRE_CONV`, `NUM_REDUCE_CONV`, `NUM_RED_CONV`, `NUM_REL_CONV`, `NUM_SUB_CONV`, `NUM_SUC_CONV`.

# NUMBER_RULE

`NUMBER_RULE : term -> thm`

## Synopsis

Automatically prove elementary divisibility property over the natural numbers.

## Description

`NUMBER_RULE` is a partly heuristic rule that can often automatically prove elementary "divisibility" properties of the natural numbers. The precise subset that is dealt with is difficult to describe rigorously, but many universally quantified combinations of `divides`, `coprime`, `gcd` and congruences `(x == y) (mod n)` can be proved automatically, as well as some existentially quantified goals. See a similar rule `INTEGER_RULE` for the integers for a representative set of examples.

## Failure

Fails if the goal is not accessible to the methods used.

## Example

Here is a typical example, which would be rather tedious to prove manually:

```
# NUMBER_RULE
   `!a b a' b'. ~(gcd(a,b) = 0) /\ a = a' * gcd(a,b) /\ b = b' * gcd(a,b)
                ==> coprime(a',b')`;;
...
val it : thm =
|- !a b a' b'.
      ~(gcd (a,b) = 0) /\ a = a' * gcd (a,b) /\ b = b' * gcd (a,b)
      ==> coprime (a',b')
```

## See also

`ARITH_RULE`, `INTEGER_RULE`, `NUMBER_TAC`, `NUM_RING`.

# NUMBER_TAC

`NUMBER_TAC : tactic`

## Synopsis

Automated tactic for elementary divisibility properties over the natural numbers.

## Description

The tactic `NUMBER_TAC` is a partly heuristic tactic that can often automatically prove elementary "divisibility" properties of the natural numbers. The precise subset that is dealt with is difficult to describe rigorously, but many universally quantified combinations of `divides`, `coprime`, `gcd` and congruences `(x == y) (mod n)` can be proved automatically, as well as some existentially quantified goals. See the documentation for `INTEGER_RULE` for a larger set of representative examples.

## Failure

Fails if the goal is not accessible to the methods used.

## Example

A typical elementary divisibility property is that if two numbers are congruent with respect to two coprime (without non-trivial common factors) moduli, then they are congruent with respect to their product:

```
# g `!m n x y:num. (x == y) (mod m) /\ (x == y) (mod n) /\ coprime(m,n)
                   ==> (x == y) (mod (m * n))`;;
...
```

It can be solved automatically using `NUMBER_TAC`:

```
# e NUMBER_TAC;;
...
val it : goalstack = No subgoals
```

The analogous goal without the coprimality assumption will fail, and indeed the goal would be false without it.

## See also

`ARITH_TAC`, `INTEGER_TAC`, `NUMBER_RULE`, `NUM_RING`.

# NUM_CANCEL_CONV

```
NUM_CANCEL_CONV : term -> thm
```

## Synopsis

Cancels identical terms from both sides of natural number equation.

## Description

Given an equational term `` `t1 + ... + tn = s1 + ... + sm` `` (with arbitrary association of the additions) where both sides have natural number type, the conversion identifies common elements among the `ti` and `si`, and cancels them from both sides, returning a theorem:

```
|- t1 + ... + tn = s1 + ... + sm <=> u1 + ... + uk = v1 + ... + vl
```

where the `ui` and `vi` are the remaining elements of the `ti` and `si` respectively, in some order.

## Failure

Fails if applied to a term that is not an equation between natural number terms.

## Example

```
# NUM_CANCEL_CONV `(a + b + x * y + SUC c) + d = SUC c + d + y * z`;;
val it : thm =
  |- (a + b + x * y + SUC c) + d = SUC c + d + y * z <=>
     x * y + b + a = y * z
```

## Uses

Simplifying equations where explicitly directing the cancellation would be tedious. However, this is mostly intended for "bootstrapping", before more powerful rules like `ARITH_RULE` and `NUM_RING` are available.

## See also

`ARITH_CONV`, `ARITH_RULE`, `ARITH_TAC`, `NUM_RING`.

# num_CONV

```
num_CONV : term -> thm
```

## Synopsis

Provides definitional axiom for a nonzero numeral.

## Description

`num_CONV` is an axiom-scheme from which one may obtain a defining equation for any numeral not equal to `0` (i.e. `1`, `2`, `3`,...). If `` `n` `` is such a constant, then `` num_CONV `n` `` returns the theorem:

```
  |- n = SUC m
```

where `m` is the numeral that denotes the predecessor of the number denoted by `n`.

## Failure

`num_CONV tm` fails if `tm` is `` `0` `` or if not `tm` is not a numeral.

## Example

```
# num_CONV `3`;;
val it : thm = |- 3 = SUC 2
```

# NUM_DIV_CONV

```
NUM_DIV_CONV : term -> thm
```

## Synopsis
Proves what the truncated quotient of two natural number numerals is.

## Description
If `n` and `m` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_DIV_CONV ‘n DIV m‘` returns the theorem:

```
|- n DIV m = s
```

where `s` is the numeral that denotes the truncated quotient of the numbers denoted by `n` and `m`.

## Failure
`NUM_DIV_CONV tm` fails if `tm` is not of the form `‘n DIV m‘`, where `n` and `m` are numerals, or if the second numeral `m` is zero.

## Example

```
# NUM_DIV_CONV ‘99 DIV 9‘;;
val it : thm = |- 99 DIV 9 = 11

# NUM_DIV_CONV ‘334 DIV 3‘;;
val it : thm = |- 334 DIV 3 = 111

# NUM_DIV_CONV ‘11 DIV 0‘;;
Exception: Failure "NUM_DIV_CONV".
```

## Comments
For definiteness, quotients with zero denominator are in fact designed to be zero. However, it is perhaps bad style to rely on this fact, so the conversion just fails in this case.

## See also
`NUM_ADD_CONV`, `NUM_DIVMOD_CONV`, `NUM_EQ_CONV`, `NUM_EVEN_CONV`, `NUM_EXP_CONV`, `NUM_FACT_CONV`, `NUM_GE_CONV`, `NUM_GT_CONV`, `NUM_LE_CONV`, `NUM_LT_CONV`, `NUM_MOD_CONV`, `NUM_MULT_CONV`, `NUM_ODD_CONV`, `NUM_PRE_CONV`, `NUM_REDUCE_CONV`, `NUM_RED_CONV`, `NUM_REL_CONV`, `NUM_SUB_CONV`, `NUM_SUC_CONV`.

# numdom

```
numdom : num -> num * num
```

## Synopsis
Returns numerator and denominator of normalized fraction.

## Description

Given a rational number as supported by the `Num` library, `numdom` returns a numerator-denominator pair corresponding to that rational number cancelled down to its reduced form, $p/q$ where $q > 0$ and $p$ and $q$ have no common factor.

## Failure

Never fails.

## Example

```
# numdom(Int 22 // Int 7);;
val it : num * num = (22, 7)
# numdom(Int 0);;
val it : num * num = (0, 1)
# numdom(Int 100);;
val it : num * num = (100, 1)
# numdom(Int 4 // Int(-2));;
val it : num * num = (-2, 1)
```

## See also

`denominator`, `numerator`.

# NUM_EQ_CONV

```
NUM_EQ_CONV : conv
```

## Synopsis

Proves equality or inequality of two numerals.

## Description

If `n` and `m` are two numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_EQ_CONV `n = m`` returns:

```
|- (n = m) <=> T        or        |- (n = m) <=> F
```

depending on whether the natural numbers represented by `n` and `m` are equal or not equal, respectively.

## Failure

`NUM_EQ_CONV tm` fails if `tm` is not of the form `` `n = m` ``, where `n` and `m` are numerals.

## Example

```
# NUM_EQ_CONV `1 = 2`;;
val it : thm = |- 1 = 2 <=> F

# NUM_EQ_CONV `12 = 12`;;
val it : thm = |- 12 = 12 <=> T
```

## Uses
Performing basic arithmetic reasoning while producing a proof.

## See also
NUM_ADD_CONV, NUM_DIV_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# numerator

```
numerator : num -> num
```

## Synopsis
Returns numerator of rational number in canonical form.

## Description
Given a rational number as supported by the `Num` library, `numerator` returns the numerator $p$ from the rational number cancelled to its reduced form, $p/q$ where $q > 0$ and $p$ and $q$ have no common factor.

## Failure
Never fails.

## Example

```
# numerator(Int 22 // Int 7);;
val it : num = 22
# numerator(Int 0);;
val it : num = 0
# numerator(Int 100);;
val it : num = 100
# numerator(Int 4 // Int(-2));;
val it : num = -2
```

## See also
denominator, numdom.

# NUM_EVEN_CONV

```
NUM_EVEN_CONV : conv
```

## Synopsis
Proves whether a natural number numeral is even.

## Description
If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `NUM_EVEN_CONV ‘n‘` returns one of the theorems:

```
|- EVEN(n) <=> T
```

or

```
|- EVEN(n) <=> F
```

according to whether the number denoted by `n` is even.

## Failure
Fails if applied to a term that is not of the form `‘EVEN n‘` with `n` a numeral.

## Example

```
# NUM_EVEN_CONV ‘EVEN 99‘;;
val it : thm = |- EVEN 99 <=> F
# NUM_EVEN_CONV ‘EVEN 123456‘;;
val it : thm = |- EVEN 123456 <=> T
```

## See also
`NUM_ADD_CONV`, `NUM_DIV_CONV`, `NUM_EQ_CONV`, `NUM_EXP_CONV`, `NUM_FACT_CONV`, `NUM_GE_CONV`, `NUM_GT_CONV`, `NUM_LE_CONV`, `NUM_LT_CONV`, `NUM_MOD_CONV`, `NUM_MULT_CONV`, `NUM_ODD_CONV`, `NUM_PRE_CONV`, `NUM_REDUCE_CONV`, `NUM_RED_CONV`, `NUM_REL_CONV`, `NUM_SUB_CONV`, `NUM_SUC_CONV`.

# NUM_EXP_CONV

```
NUM_EXP_CONV : term -> thm
```

## Synopsis
Proves what the exponential of two natural number numerals is.

## Description

If `n` and `m` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_EXP_CONV ‘n EXP m‘` returns the theorem:

```
|- n EXP m = s
```

where `s` is the numeral that denotes the natural number denoted by `n` raised to the power of the one denoted by `m`.

## Failure

`NUM_EXP_CONV tm` fails if `tm` is not of the form `‘n EXP m‘`, where `n` and `m` are numerals.

## Example

```
# NUM_EXP_CONV ‘2 EXP 64‘;;
val it : thm = |- 2 EXP 64 = 18446744073709551616

# NUM_EXP_CONV ‘1 EXP 99‘;;
val it : thm = |- 1 EXP 99 = 1

# NUM_EXP_CONV ‘0 EXP 0‘;;
val it : thm = |- 0 EXP 0 = 1

# NUM_EXP_CONV ‘0 EXP 10000‘;;
val it : thm = |- 0 EXP 10000 = 0
```

## See also

`NUM_ADD_CONV`, `NUM_DIV_CONV`, `NUM_EQ_CONV`, `NUM_EVEN_CONV`, `NUM_FACT_CONV`, `NUM_GE_CONV`, `NUM_GT_CONV`, `NUM_LE_CONV`, `NUM_LT_CONV`, `NUM_MOD_CONV`, `NUM_MULT_CONV`, `NUM_ODD_CONV`, `NUM_PRE_CONV`, `NUM_REDUCE_CONV`, `NUM_RED_CONV`, `NUM_REL_CONV`, `NUM_SUB_CONV`, `NUM_SUC_CONV`.

# NUM_FACT_CONV

```
NUM_FACT_CONV : term -> thm
```

## Synopsis

Proves what the factorial of a natural number numeral is.

## Description

If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `NUM_FACT_CONV ‘FACT n‘` returns the theorem:

```
|- FACT n = s
```

where `s` is the numeral that denotes the factorial of the natural number denoted by `n`.

## Failure

NUM_FACT_CONV tm fails if tm is not of the form ‘FACT n‘, where n is a numeral.

## Example

```
# NUM_FACT_CONV ‘FACT 0‘;;
val it : thm = |- FACT 0 = 1

# NUM_FACT_CONV ‘FACT 6‘;;
val it : thm = |- FACT 6 = 720

# NUM_FACT_CONV ‘FACT 30‘;;
val it : thm = |- FACT 30 = 265252859812191058636308480000000
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_GE_CONV

```
NUM_GE_CONV : conv
```

## Synopsis

Proves whether one numeral is greater than or equal to another.

## Description

If n and m are two numerals (e.g. 0, 1, 2, 3,...), then NUM_GE_CONV ‘n >= m‘ returns:

```
|- n >= m <=> T        or        |- n >= m <=> F
```

depending on whether the natural number represented by n is greater than or equal to the one represented by m.

## Failure

NUM_GE_CONV tm fails if tm is not of the form ‘n >= m‘, where n and m are numerals.

## Example

```
# NUM_GE_CONV `1 >= 0`;;
val it : thm = |- 1 >= 0 <=> T

# NUM_GE_CONV `181 >= 211`;;
val it : thm = |- 181 >= 211 <=> F
```

## Uses

Performing basic arithmetic reasoning while producing a proof.

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_GT_CONV

NUM_GT_CONV : conv

## Synopsis

Proves whether one numeral is greater than another.

## Description

If n and m are two numerals (e.g. 0, 1, 2, 3,...), then NUM_GT_CONV `n > m` returns:

```
|- n > m <=> T        or        |- n > m <=> F
```

depending on whether the natural number represented by n is greater than the one represented by m.

## Failure

NUM_GT_CONV tm fails if tm is not of the form `n > m`, where n and m are numerals.

## Example

```
# NUM_GT_CONV `3 > 2`;;
val it : thm = |- 3 > 2 <=> T

# NUM_GT_CONV `77 > 77`;;
val it : thm = |- 77 > 77 <=> F
```

## Uses

Performing basic arithmetic reasoning while producing a proof.

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_LE_CONV

`NUM_LE_CONV : conv`

## Synopsis

Proves whether one numeral is less than or equal to another.

## Description

If `n` and `m` are two numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_LE_CONV ‘n <= m‘` returns:

```
|- n <= m <=> T       or       |- n <= m <=> F
```

depending on whether the natural number represented by `n` is less than or equal to the one represented by `m`.

## Failure

`NUM_LE_CONV tm` fails if `tm` is not of the form `‘n <= m‘`, where `n` and `m` are numerals.

## Example

```
# NUM_LE_CONV ‘12 <= 19‘;;
val it : thm = |- 12 <= 19 <=> T

# NUM_LE_CONV ‘12345 <= 12344‘;;
val it : thm = |- 12345 <= 12344 <=> F
```

## Uses

Performing basic arithmetic reasoning while producing a proof.

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_LT_CONV

`NUM_LT_CONV : conv`

## Synopsis

Proves whether one numeral is less than another.

## Description

If `n` and `m` are two numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_LT_CONV ‘n < m‘` returns:

```
|- n < m <=> T        or        |- n < m <=> F
```

depending on whether the natural number represented by `n` is less than the one represented by `m`.

## Failure

`NUM_LT_CONV tm` fails if `tm` is not of the form `‘n < m‘`, where `n` and `m` are numerals.

## Example

```
# NUM_LT_CONV ‘42 < 42‘;;
val it : thm = |- 42 < 42 <=> F

# NUM_LT_CONV ‘11 < 19‘;;
val it : thm = |- 11 < 19 <=> T
```

## Uses

Performing basic arithmetic reasoning while producing a proof.

## See also

`NUM_ADD_CONV`, `NUM_DIV_CONV`, `NUM_EQ_CONV`, `NUM_EVEN_CONV`, `NUM_EXP_CONV`, `NUM_FACT_CONV`, `NUM_GE_CONV`, `NUM_GT_CONV`, `NUM_LE_CONV`, `NUM_MOD_CONV`, `NUM_MULT_CONV`, `NUM_ODD_CONV`, `NUM_PRE_CONV`, `NUM_REDUCE_CONV`, `NUM_RED_CONV`, `NUM_REL_CONV`, `NUM_SUB_CONV`, `NUM_SUC_CONV`.

# NUM_MOD_CONV

`NUM_MOD_CONV : term -> thm`

## Synopsis

Proves what the remainder on dividing one natural number numeral by another is.

## Description

If n and m are numerals (e.g. 0, 1, 2, 3,...), then NUM_MOD_CONV `n MOD m` returns the theorem:

```
|- n MOD m = s
```

where s is the numeral that denotes the remainder on dividing the number denoted by n by the one denoted by m.

## Failure

NUM_MOD_CONV tm fails if tm is not of the form `n MOD m`, where n and m are numerals, or if the second numeral m is zero.

## Example

```
# NUM_MOD_CONV `1089 MOD 9`;;
val it : thm = |- 1089 MOD 9 = 0

# NUM_MOD_CONV `1234 MOD 3`;;
val it : thm = |- 1234 MOD 3 = 1

# NUM_MOD_CONV `11 MOD 0`;;
Exception: Failure "NUM_MOD_CONV".
```

## Comments

For definiteness, remainders with zero denominator are in fact designed to be zero. However, it is perhaps bad style to rely on this fact, so the conversion just fails in this case.

## See also

NUM_ADD_CONV, NUM_DIVMOD_CONV, NUM_MOD_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_MULT_CONV

```
NUM_MULT_CONV : term -> thm
```

## Synopsis

Proves what the product of two natural number numerals is.

## Description

If n and m are numerals (e.g. 0, 1, 2, 3,...), then NUM_MULT_CONV `n * m` returns the theorem:

```
|- n * m = s
```

where s is the numeral that denotes the product of the natural numbers denoted by n and m.

## Failure
NUM_MULT_CONV tm fails if tm is not of the form ‘n * m‘, where n and m are numerals.

## Example

```
# NUM_MULT_CONV ‘12345 * 12345‘;;
val it : thm = |- 12345 * 12345 = 152399025
```

## See also
NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV,NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_NORMALIZE_CONV

```
NUM_NORMALIZE_CONV : term -> thm
```

## Synopsis
Puts natural number expressions built using addition, multiplication and powers in canonical polynomial form.

## Description
Given a term t of natural number type built up from other “atomic” components (not necessarily simple variables) and numeral constants by addition, multiplication and exponentiation by constant exponents, NUM_NORMALIZE_CONV t will return |- t = t' where t' is the result of putting the term into a normalized form, essentially a multiplied-out polynomial with a specific ordering of and within monomials.

## Failure
Should never fail.

## Example

```
# NUM_NORMALIZE_CONV ‘1 + (1 + x + x EXP 2) * (x + (x * x) EXP 2)‘;;
val it : thm =
  |- 1 + (1 + x + x EXP 2) * (x + (x * x) EXP 2) =
     x EXP 6 + x EXP 5 + x EXP 4 + x EXP 3 + x EXP 2 + x + 1
```

## Comments
This can be used to prove simple algebraic equations, but NUM_RING or ARITH_RULE are generally more powerful and convenient for that. In particular, this function does not handle cutoff subtraction or other such operations.

## See also
ARITH_RULE, NUM_REDUCE_CONV, NUM_RING, REAL_POLY_CONV, SEMIRING_NORMALIZERS_CONV.

# NUM_ODD_CONV

```
NUM_ODD_CONV : conv
```

## Synopsis
Proves whether a natural number numeral is odd.

## Description
If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `NUM_ODD_CONV ‘n‘` returns one of the theorems:

```
|- ODD(n) <=> T
```

or

```
|- ODD(n) <=> F
```

according to whether the number denoted by `n` is odd.

## Failure
Fails if applied to a term that is not of the form `‘ODD n‘` with `n` a numeral.

## Example

```
# NUM_ODD_CONV ‘ODD 123‘;;
val it : thm = |- ODD 123 <=> T

# NUM_ODD_CONV ‘ODD 1234‘;;
val it : thm = |- ODD 1234 <=> F
```

## See also
NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# num_of_string

```
num_of_string : string -> num
```

## Synopsis
Converts decimal, hex or binary string representation into number.

## Description
The call `num_of_string "n"` converts the string `"n"` into an OCaml unlimited-precision number (type `num`). The string may be simply a sequence of decimal digits (e.g. `"123"`), or a hexadecimal representation starting with `0x` as in C (e.g. `"0xFF"`), or a binary number starting with `0b` (e.g. `"0b101"`).

## Failure
Fails unless the string is a valid representation of one of these forms.

## Example

```
# num_of_string "0b11000000";;
val it : num = 192
```

## See also
`dest_numeral`, `mk_numeral`.

# NUM_PRE_CONV

```
NUM_PRE_CONV : term -> thm
```

## Synopsis
Proves what the cutoff predecessor of a natural number numeral is.

## Description
If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `NUM_PRE_CONV ‘PRE n‘` returns the theorem:

```
|- PRE n = s
```

where `s` is the numeral that denotes the cutoff predecessor of the natural number denoted by `n` (that is, the result of subtracting 1 from it, or zero if it is already zero).

## Failure
`NUM_PRE_CONV tm` fails if `tm` is not of the form `‘PRE n‘`, where `n` is a numeral.

## Example

```
# NUM_PRE_CONV `PRE 0`;;
val it : thm = |- PRE 0 = 0

# NUM_PRE_CONV `PRE 12345`;;
val it : thm = |- PRE 12345 = 12344
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV.

# NUM_RED_CONV

```
NUM_RED_CONV : term -> thm
```

## Synopsis

Performs one arithmetic or relational operation on natural number numerals by proof.

## Description

When applied to a term that is either a unary operator application `` `SUC n` ``, `` `PRE n` `` or `` `FACT n` `` for a numeral `n`, or a relational operator application `` `m < n` ``, `` `m <= n` ``, `` `m > n` ``, `` `m >= n` `` or `` `m = n` ``, or a binary arithmetic operation `` `m + n` ``, `` `m - n` ``, `` `m * n` ``, `` `m EXP n` ``, `` `m DIV n` `` or `` `m MOD n` `` applied to numerals `m` and `n`, the conversion `NUM_RED_CONV` will 'reduce' it and return a theorem asserting its equality to the reduced form.

## Failure

`NUM_RED_CONV tm` fails if `tm` is not of one of the forms specified.

## Example

```
# NUM_RED_CONV `2 + 2`;;
val it : thm = |- 2 + 2 = 4

# NUM_RED_CONV `1089 < 2231`;;
val it : thm = |- 1089 < 2231 <=> T

# NUM_RED_CONV `FACT 11`;;
val it : thm = |- FACT 11 = 39916800
```

Note that the immediate operands must be numerals. For deeper reduction of combinations of numerals, use NUM_REDUCE_CONV:

```
# NUM_RED_CONV `(432 - 234) + 198`;;
Exception: Failure "REWRITES_CONV".

# NUM_REDUCE_CONV `(432 - 234) + 198`;;
val it : thm = |- 432 - 234 + 198 = 396
```

## Uses

Access to this 'one-step' reduction is not usually especially useful, but if you want to add a conversion conv for some other operator on numbers, you can conveniently incorporate it into NUM_REDUCE_CONV with

```
# let NUM_REDUCE_CONV' = DEPTH_CONV(REAL_RAT_RED_CONV ORELSEC conv);;
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV, REAL_RAT_RED_CONV.

# NUM_REDUCE_CONV

```
NUM_REDUCE_CONV : term -> thm
```

## Synopsis

Evaluate subexpressions built up from natural number numerals, by proof.

## Description

When applied to a term, NUM_REDUCE_CONV performs a recursive bottom-up evaluation by proof of subterms built from numerals using the unary operators 'SUC', 'PRE' and 'FACT' and the binary arithmetic ('+', '-', '*', 'EXP', 'DIV', 'MOD') and relational ('<', '<=', '>', '>=', '=') operators, as well as propagating constants through logical operations, e.g. `T /\ x <=> x`, returning a theorem that the original and reduced terms are equal.

## Failure

Never fails, but may have no effect.

## Example

```
# NUM_REDUCE_CONV `(432 - 234) + 198`;;
val it : thm = |- 432 - 234 + 198 = 396

# NUM_REDUCE_CONV
    `if 100 < 200 then 2 EXP (8 DIV 2) else 3 EXP ((26 EXP 0) * 3)`;;
val it : thm =
 |- (if 100 < 200 then 2 EXP (8 DIV 2) else 3 EXP (26 EXP 0 * 3)) = 16

# NUM_REDUCE_CONV `(!x. f(x + 2 + 2) < f(x + 0)) ==> f(12 * x) = f(12 * 12)`;;
val it : thm =
  |- (!x. f (x + 2 + 2) < f (x + 0)) ==> f (12 * x) = f (12 * 12) <=>
     (!x. f (x + 4) < f (x + 0)) ==> f (12 * x) = f 144
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_TAC, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV, REAL_RAT_REDUCE_CONV.

# NUM_REDUCE_TAC

```
NUM_REDUCE_TAC : tactic
```

## Synopsis

Evaluate subexpressions of goal built up from natural number numerals.

## Description

When applied to a goal, `NUM_REDUCE_TAC` performs a recursive bottom-up evaluation by proof of subterms of the conclusion built from numerals using the unary operators '`SUC`', '`PRE`' and '`FACT`' and the binary arithmetic ('`+`', '`-`', '`*`', '`EXP`', '`DIV`', '`MOD`') and relational ('`<`', '`<=`', '`>`', '`>=`', '`=`') operators, as well as propagating constants through logical operations, e.g. `T /\ x <=> x`, returning a new subgoal where all these subexpressions are reduced.

## Failure

Never fails, but may have no effect.

## Example

```
# g `1 EXP 3 + 12 EXP 3 = 1729 /\ 9 EXP 3 + 10 EXP 3 = 1729`;;
val it : goalstack = 1 subgoal (1 total)

`1 EXP 3 + 12 EXP 3 = 1729 /\ 9 EXP 3 + 10 EXP 3 = 1729`

# e NUM_REDUCE_TAC;;
val it : goalstack = No subgoals
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV, NUM_SUC_CONV, REAL_RAT_REDUCE_CONV.

# NUM_REL_CONV

```
NUM_REL_CONV : term -> thm
```

## Synopsis

Performs relational operation on natural number numerals by proof.

## Description

When applied to a term that is a relational operator application `` `m < n` ``, `` `m <= n` ``, `` `m > n` ``, `` `m >= n` `` or `` `m = n` `` applied to numerals `m` and `n`, the conversion `NUM_REL_CONV` will 'reduce' it and return a theorem asserting its equality to `` `T` `` or `` `F` `` as appropriate.

## Failure

`NUM_REL_CONV tm` fails if `tm` is not of one of the forms specified.

## Example

```
# NUM_REL_CONV `1089 < 2231`;;
val it : thm = |- 1089 < 2231 <=> T

# NUM_REL_CONV `1089 >= 2231`;;
val it : thm = |- 1089 >= 2231 <=> F
```

Note that the immediate operands must be numerals. For deeper reduction of combinations of numerals, use NUM_REDUCE_CONV.

```
# NUM_REL_CONV `2 + 2 = 4`;;
Exception: Failure "REWRITES_CONV".

# NUM_REDUCE_CONV `2 + 2 = 4`;;
val it : thm = |- 2 + 2 = 4 <=> T
```

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_SUB_CONV, NUM_SUC_CONV, REAL_RAT_RED_CONV.

# NUM_RING

```
NUM_RING : term -> thm
```

## Synopsis

Ring decision procedure instantiated to natural numbers.

## Description

The rule NUM_RING should be applied to a formula that, after suitable normalization, can be considered a universally quantified Boolean combination of equations and inequations between terms of type :num. If that formula holds in all integral domains, NUM_RING will prove it. Any "alien" atomic formulas that are not natural number equations will not contribute to the proof but will not in themselves cause an error. The function is a particular instantiation of RING, which is a more generic procedure for ring and semiring structures.

## Failure

Fails if the formula is unprovable by the methods employed. This does not necessarily mean that it is not valid for :num, but rather that it is not valid on all integral domains (see below).

## Example

The following formula is proved because it holds in all integral domains:

```
# NUM_RING `(x + y) EXP 2 = x EXP 2 ==> y = 0 \/ y + 2 * x = 0`;;
1 basis elements and 0 critical pairs
Translating certificate to HOL inferences
val it : thm = |- (x + y) EXP 2 = x EXP 2 ==> y = 0 \/ y + 2 * x = 0
```

but the following isn't, even though over `:num` it is equivalent:

```
# NUM_RING `(x + y) EXP 2 = x EXP 2 ==> y = 0 \/ x = 0`;;
2 basis elements and 1 critical pairs
3 basis elements and 2 critical pairs
3 basis elements and 1 critical pairs
4 basis elements and 1 critical pairs
4 basis elements and 0 critical pairs
Exception: Failure "find".
```

## Comments

Note that since we are working over `:num`, which is not really a ring, cutoff subtraction is not true ring subtraction and the ability of `NUM_RING` to handle it is limited. Instantiations of `RING` to actual rings, such as `REAL_RING`, have no such problems.

## See also

`ARITH_RULE, ARITH_TAC, ideal_cofactors, NUM_NORMALIZE_CONV, REAL_RING, RING.`

# NUM_SIMPLIFY_CONV

`NUM_SIMPLIFY_CONV : conv`

## Synopsis

Eliminates predecessor, cutoff subtraction, even and odd, division and modulus.

## Description

When applied to a term, `NUM_SIMPLIFY_CONV` tries to get rid of instances of the natural number operators `PRE`, `DIV`, `MOD` and `-` (which is cutoff subtraction), as well as the `EVEN` and `ODD` predicates, by rephrasing properties in terms of multiplication and addition, adding new variables if necessary. Some attempt is made to introduce quantifiers so that they are effectively universally quantified. However, the input formula should be in NNF for this aspect to be completely reliable.

## Failure

Should never fail, but in obscure situations may leave some instance of the troublesome operators (for example, if they are mapped over a list instead of simply applied).

## Example

```
# NUM_SIMPLIFY_CONV `~(n = 0) ==> PRE(n) + 1 = n`;;
val it : thm =
  |- ~(n = 0) ==> PRE n + 1 = n <=>
     (!m. ~(n = SUC m) /\ (~(m = 0) \/ ~(n = 0)) \/ n = 0 \/ m + 1 = n)
```

## Uses

Not really intended for most users, but a prelude inside several automated routines such as `ARITH_RULE`. It is because of this preprocessing step that such rules can handle these troublesome operators to some extent, e.g.

```
# ARITH_RULE `~(n = 0) ==> n DIV 3 < n`;;
val it : thm = |- ~(n = 0) ==> n DIV 3 < n
```

## See also

`ARITH_CONV`, `ARITH_RULE`, `ARITH_TAC`, `NUM_RING`.

# NUM_SUB_CONV

```
NUM_SUB_CONV : term -> thm
```

## Synopsis

Proves what the cutoff difference of two natural number numerals is.

## Description

If `n` and `m` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `NUM_SUB_CONV `n - m`` returns the theorem:

```
|- n - m = s
```

where `s` is the numeral that denotes the result of subtracting the natural number denoted by `m` from the one denoted by `n`, returning zero for all cases where `m` is greater than `n` (cutoff subtraction over the natural numbers).

## Failure

`NUM_SUB_CONV tm` fails if `tm` is not of the form `` `n - m` ``, where `n` and `m` are numerals.

## Example

```
# NUM_SUB_CONV `4321 - 1234`;;
val it : thm = |- 4321 - 1234 = 3087

# NUM_SUB_CONV `77 - 88`;;
val it : thm = |- 77 - 88 = 0
```

## Comments

Note that subtraction over type `:num` is defined as this cutoff subtraction. If you want a number system with negative numbers, use `:int` or `:real`.

## See also

NUM_ADD_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV, NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUC_CONV.

# NUM_SUC_CONV

```
NUM_SUC_CONV : term -> thm
```

## Synopsis

Proves what the successor of a natural number numeral is.

## Description

If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `NUM_SUC_CONV `SUC n`` returns the theorem:

```
|- SUC n = s
```

where `s` is the numeral that denotes the successor of the natural number denoted by `n` (that is, the result of adding 1 to it).

## Failure

`NUM_SUC_CONV tm` fails if `tm` is not of the form `` `SUC n` ``, where `n` is a numeral.

## Example

```
# NUM_SUC_CONV `SUC 0`;;
val it : thm = |- SUC 0 = 1

# NUM_SUC_CONV `SUC 12345`;;
val it : thm = |- SUC 12345 = 12346
```

## See also

NUM_ADD_CONV, num_CONV, NUM_DIV_CONV, NUM_EQ_CONV, NUM_EVEN_CONV, NUM_EXP_CONV, NUM_FACT_CONV, NUM_GE_CONV, NUM_GT_CONV, NUM_LE_CONV, NUM_LT_CONV,

NUM_MOD_CONV, NUM_MULT_CONV, NUM_ODD_CONV, NUM_PRE_CONV, NUM_REDUCE_CONV, NUM_RED_CONV, NUM_REL_CONV, NUM_SUB_CONV.

# NUM_TO_INT_CONV

```
NUM_TO_INT_CONV : conv
```

## Synopsis
Maps an assertion over natural numbers to equivalent over reals.

## Description
Given a term, with arbitrary quantifier alternations over the natural numbers, NUM_TO_INT_CONV proves its equivalence to a term involving integer operations and quantifiers. Some preprocessing removes certain natural-specific operations such as PRE and cutoff subtraction, quantifiers are systematically relativized to the set of positive integers.

## Failure
Never fails.

## Example

```
# NUM_TO_INT_CONV `n - m <= n`;;
val it : thm =
  |- n - m <= n <=>
     (!i. ~(&0 <= i) \/
          (~(&m = &n + i) \/ &0 <= &n) /\ (~(&n = &m + i) \/ i <= &n))
```

## Uses
Mostly intended as a preprocessing step to allow rules for the integers to deduce facts about natural numbers too.

## See also
ARITH_RULE, INT_ARITH, INT_OF_REAL_THM, NUM_SIMPLIFY_CONV.

# occurs_in

```
occurs_in : hol_type -> hol_type -> bool
```

## Synopsis
Tests if one type occurs in another.

## Description

The call `occurs_in ty1 ty2` returns `true` if `ty1` occurs as a subtype of `ty2`, including the case where `ty1` and `ty2` are the same. If returns `false` otherwise. The type `ty1` does not have to be a type variable.

## Failure

Never fails.

## Example

```
# occurs_in `:A` `:(A)list->bool`;;
val it : bool = true
# occurs_in `:num->num` `:num->num->bool`;;
val it : bool = false
# occurs_in `:num->bool` `:num->num->bool`;;
val it : bool = true
```

## See also

`free_in, tyvars, vfree_in.`

# o

```
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

## Synopsis

Composes two functions: `(f o g) x` = `f (g x)`.

## Failure

Never fails.

## See also

`C, F_F, I, K, W.`

# ONCE_ASM_REWRITE_RULE

```
ONCE_ASM_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis

Rewrites a theorem once including built-in rewrites and the theorem's assumptions.

## Description

ONCE_ASM_REWRITE_RULE applies all possible rewrites in one step over the subterms in the conclusion of the theorem, but stops after rewriting at most once at each subterm. This strategy is specified as for ONCE_DEPTH_CONV. For more details see ASM_REWRITE_RULE, which does search recursively (to any depth) for matching subterms. The general strategy for rewriting theorems is described under GEN_REWRITE_RULE.

## Failure

Never fails.

## Uses

This tactic is used when rewriting with the hypotheses of a theorem (as well as a given list of theorems and basic_rewrites), when more than one pass is not required or would result in divergence.

## See also

ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_DEPTH_CONV, ONCE_REWRITE_RULE, PURE_ASM_REWRITE_RULE, PURE_ONCE_ASM_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

# ONCE_ASM_REWRITE_TAC

```
ONCE_ASM_REWRITE_TAC : thm list -> tactic
```

## Synopsis

Rewrites a goal once including built-in rewrites and the goal's assumptions.

## Description

ONCE_ASM_REWRITE_TAC behaves in the same way as ASM_REWRITE_TAC, but makes one pass only through the term of the goal. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. See GEN_REWRITE_TAC for more information on rewriting a goal in HOL.

## Failure

ONCE_ASM_REWRITE_TAC does not fail and, unlike ASM_REWRITE_TAC, does not diverge. The resulting tactic may not be valid, if the rewrites performed add new assumptions to the theorem eventually proved.

## Example

The use of ONCE_ASM_REWRITE_TAC to control the amount of rewriting performed is illustrated on this goal:

```
# g `a = b /\ b = c ==> (P a b <=> P c a)`;;
Warning: inventing type variables
Warning: Free variables in goal: P, a, b, c
val it : goalstack = 1 subgoal (1 total)

`a = b /\ b = c ==> (P a b <=> P c a)`

# e STRIP_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [`a = b`]
 1 [`b = c`]

`P a b <=> P c a`
```

The application of `ONCE_ASM_REWRITE_TAC` rewrites each applicable subterm just once:

```
# e(ONCE_ASM_REWRITE_TAC[]);;
val it : goalstack = 1 subgoal (1 total)

 0 [`a = b`]
 1 [`b = c`]

`P b c <=> P c b`
```

## Uses

`ONCE_ASM_REWRITE_TAC` can be applied once or iterated as required to give the effect of `ASM_REWRITE_TAC`, either to avoid divergence or to save inference steps.

## See also

`basic_rewrites`, `ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

# ONCE_ASM_SIMP_TAC

```
ONCE_ASM_SIMP_TAC : thm list -> tactic
```

## Synopsis

Simplify toplevel applicable terms in goal using assumptions and context.

## Description

A call to `ONCE_ASM_SIMP_TAC[theorems]` will apply conditional contextual rewriting with `theorems` and the current assumptions of the goal to the goal's conclusion. The `ONCE` prefix means that the toplevel simplification is only applied once to the toplevel terms, though any conditional subgoals generated are then simplfied repeatedly. For more details on this kind of rewriting, see `SIMP_CONV`. If the extra generality of contextual conditional rewriting is not needed, `ONCE_ASM_REWRITE_TAC` is usually more efficient.

## Failure

Never fails, but may loop indefinitely.

## See also

`ASM_SIMP_TAC, ONCE_ASM_REWRITE_TAC, SIMP_CONV, SIMP_TAC, REWRITE_TAC.`

# ONCE_DEPTH_CONV

`ONCE_DEPTH_CONV : conv -> conv`

## Synopsis

Applies a conversion once to the first suitable sub-term(s) encountered in top-down order.

## Description

`ONCE_DEPTH_CONV c tm` applies the conversion `c` once to the first subterm or subterms encountered in a top-down 'parallel' search of the term `tm` for which `c` succeeds. If the conversion `c` fails on all subterms of `tm`, the theorem returned is `|- tm = tm`.

## Failure

Never fails.

## Example

The following example shows how `ONCE_DEPTH_CONV` applies a conversion to only the first suitable subterm(s) found in a top-down search:

```
# ONCE_DEPTH_CONV BETA_CONV `(\x. (\y. y + x) 1) 2`;;
val it : thm = |- (\x. (\y. y + x) 1) 2 = (\y. y + 2) 1
```

Here, there are two beta-redexes in the input term. One of these occurs within the other, so `BETA_CONV` is applied only to the outermost one.

Note that the supplied conversion is applied by `ONCE_DEPTH_CONV` to all independent subterms at which it succeeds. That is, the conversion is applied to every suitable subterm not contained in some other subterm for which the conversions also succeeds, as illustrated by the following example:

```
# ONCE_DEPTH_CONV num_CONV `(\x. (\y. y + x) 1) 2`;;
val it : thm = |- (\x. (\y. y + x) 1) 2 = (\x. (\y. y + x) (SUC 0)) (SUC 1)
```

Here `num_CONV` is applied to both `1` and `2`, since neither term occurs within a larger subterm for which the conversion `num_CONV` succeeds.

### Uses

ONCE_DEPTH_CONV is frequently used when there is only one subterm to which the desired conversion applies. This can be much faster than using other functions that attempt to apply a conversion to all subterms of a term (e.g. DEPTH_CONV). If, for example, the current goal in a goal-directed proof contains only one beta-redex, and one wishes to apply BETA_CONV to it, then the tactic

```
CONV_TAC (ONCE_DEPTH_CONV BETA_CONV)
```

may, depending on where the beta-redex occurs, be much faster than

```
CONV_TAC (TOP_DEPTH_CONV BETA_CONV)
```

ONCE_DEPTH_CONV c may also be used when the supplied conversion c never fails, in which case using a conversion such as DEPTH_CONV c, which applies c repeatedly would never terminate.

### See also

DEPTH_BINOP_CONV, DEPTH_CONV, PROP_ATOM_CONV, REDEPTH_CONV, TOP_DEPTH_CONV, TOP_SWEEP_CONV.

## ONCE_DEPTH_SQCONV

```
ONCE_DEPTH_SQCONV : strategy
```

### Synopsis

Applies simplification to the first suitable sub-term(s) encountered in top-down order.

### Description

HOL Light's simplification functions (e.g. SIMP_TAC) have their traversal algorithm controlled by a "strategy". ONCE_DEPTH_SQCONV is a strategy corresponding to ONCE_DEPTH_CONV for ordinary conversions: simplification is applied to the first suitable subterm(s) encountered in top-down order.

### Failure

Not applicable.

### See also

DEPTH_SQCONV, ONCE_DEPTH_CONV, REDEPTH_SQCONV, TOP_DEPTH_SQCONV, TOP_SWEEP_SQCONV.

## ONCE_REWRITE_CONV

```
ONCE_REWRITE_CONV : thm list -> conv
```

## Synopsis
Rewrites a term, including built-in tautologies in the list of rewrites.

## Description
ONCE_REWRITE_CONV searches for matching subterms and applies rewrites once at each subterm, in the manner specified for ONCE_DEPTH_CONV. The rewrites which are used are obtained from the given list of theorems and the set of tautologies stored in basic_rewrites. See GEN_REWRITE_CONV for the general method of using theorems to rewrite a term.

## Failure
ONCE_REWRITE_CONV does not fail; it does not diverge.

## Uses
ONCE_REWRITE_CONV can be used to rewrite a term when recursive rewriting is not desired.

## See also
GEN_REWRITE_CONV, PURE_ONCE_REWRITE_CONV, PURE_REWRITE_CONV, REWRITE_CONV.

# ONCE_REWRITE_RULE

```
ONCE_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis
Rewrites a theorem, including built-in tautologies in the list of rewrites.

## Description
ONCE_REWRITE_RULE searches for matching subterms and applies rewrites once at each subterm, in the manner specified for ONCE_DEPTH_CONV. The rewrites which are used are obtained from the given list of theorems and the set of tautologies stored in basic_rewrites. See GEN_REWRITE_RULE for the general method of using theorems to rewrite an object theorem.

## Failure
ONCE_REWRITE_RULE does not fail; it does not diverge.

## Uses
ONCE_REWRITE_RULE can be used to rewrite a theorem when recursive rewriting is not desired.

## See also
ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_ASM_REWRITE_RULE, PURE_ONCE_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

# ONCE_REWRITE_TAC

```
ONCE_REWRITE_TAC : thm list -> tactic
```

## Synopsis
Rewrites a goal only once with `basic_rewrites` and the supplied list of theorems.

## Description
A set of equational rewrites is generated from the theorems supplied by the user and the set of basic tautologies, and these are used to rewrite the goal at all subterms at which a match is found in one pass over the term part of the goal. The result is returned without recursively applying the rewrite theorems to it. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. More details about rewriting can be found under `GEN_REWRITE_TAC`.

## Failure
`ONCE_REWRITE_TAC` does not fail and does not diverge. It results in an invalid tactic if any of the applied rewrites introduces new assumptions to the theorem eventually proved.

## Example
Given a theorem list:

```
# let thl = map (num_CONV o mk_small_numeral) (1--3);;
val thl : thm list = [|- 1 = SUC 0; |- 2 = SUC 1; |- 3 = SUC 2]
```

and the following goal:

```
# g `0 < 3`;;
val it : goalstack = 1 subgoal (1 total)

`0 < 3`
```

the tactic `ONCE_REWRITE_TAC thl` performs a single rewrite

```
# e(ONCE_REWRITE_TAC thl);;
val it : goalstack = 1 subgoal (1 total)

`0 < SUC 2`
```

in contrast to `REWRITE_TAC thl` which would rewrite the goal repeatedly into this form:

```
# e(REWRITE_TAC thl);;
val it : goalstack = 1 subgoal (1 total)

`0 < SUC (SUC (SUC 0))`
```

## Uses
`ONCE_REWRITE_TAC` can be used iteratively to rewrite when recursive rewriting would diverge. It can also be used to save inference steps.

## See also
`ASM_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

# ONCE_SIMP_CONV

```
ONCE_SIMP_CONV : thm list -> conv
```

## Synopsis
Simplify a term once by conditional contextual rewriting.

## Description
A call `ONCE_SIMP_CONV thl tm` will return `|- tm = tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of conditional rewriting, see `SIMP_TAC`. The `ONCE` prefix indicates that the first applicable terms in a toplevel term will be simplified once only, though conditional subgoals generated will be simplified repeatedly.

## Failure
Never fails, but may return a reflexive theorem `|- tm = tm` if no simplifications can be made.

## See also
`ASM_SIMP_TAC`, `SIMP_RULE`, `SIMP_TAC`.

# ONCE_SIMPLIFY_CONV

```
ONCE_SIMPLIFY_CONV : simpset -> thm list -> conv
```

## Synopsis
General top-level simplification with arbitrary simpset.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset'. Given a simpset `ss` and an additional list of theorems `thl` to be used as (conditional or unconditional) rewrite rules, `SIMPLIFY_CONV ss thl` gives a simplification conversion with a top-down single simplification traversal strategy (`ONCE_DEPTH_SQCONV`) and a nesting limit of 1 for the recursive solution of subconditions by further simplification.

## Failure
Never fails.

## Uses
Usually some other interface to the simplifier is more convenient, but you may want to use this to employ a customized simpset.

## See also

GEN_SIMPLIFY_CONV, ONCE_DEPTH_SQCONV, SIMPLIFY_CONV, SIMP_CONV, SIMP_RULE, SIMP_TAC.

# ONCE_SIMP_RULE

```
ONCE_SIMP_RULE : thm list -> thm -> thm
```

## Synopsis

Simplify conclusion of a theorem once by conditional contextual rewriting.

## Description

A call `ONCE_SIMP_RULE thl (|- tm)` will return `|- tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of conditional rewriting, see `SIMP_CONV`. The `ONCE` prefix indicates that the first applicable terms in a toplevel term will be simplified once only, though conditional subgoals generated will be simplified repeatedly.

## Failure

Never fails, but may return the initial theorem unchanged.

## See also

ASM_SIMP_TAC, SIMP_CONV, SIMP_RULE, SIMP_TAC.

# ONCE_SIMP_TAC

```
ONCE_SIMP_TAC : thm list -> tactic
```

## Synopsis

Simplify conclusion of goal once by conditional contextual rewriting.

## Description

When applied to a goal `A ?- g`, the tactic `ONCE_SIMP_TAC thl` returns a new goal `A ?- g'` where `g'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of conditional rewriting, see `SIMP_CONV`. The `ONCE` prefix indicates that thefirst applicable terms in a toplevel term will be simplified once only, though conditional subgoals generated will be simplified repeatedly.

## Failure

Never fails, though may not change the goal if no simplifications are applicable.

### See also
ONCE_SIMP_CONV, ONCE_SIMP_RULE, SIMP_CONV, SIMP_TAC.

# ORDERED_IMP_REWR_CONV

```
ORDERED_IMP_REWR_CONV : (term -> term -> bool) -> thm -> term -> thm
```

### Synopsis
Basic conditional rewriting conversion restricted by term order.

### Description
Given an ordering relation `ord`, an equational theorem `A |- !x1...xn. p ==> s = t` that expresses a conditional rewrite rule, the conversion `ORDERED_IMP_REWR_CONV` gives a conversion that applied to any term `s'` will attempt to match the left-hand side of the equation `s = t` to `s'`, and return the corresponding theorem `A |- p' ==> s' = t'`, but only if `ord ‘s'‘ ‘t'‘`, i.e. if the left-hand side is “greater” in the ordering than the right-hand side, after instantiation. If the ordering condition is violated, it will fail, even if the match is fine.

### Failure
Fails if the theorem is not of the right form or the two terms cannot be matched, for example because the variables that need to be instantiated are free in the hypotheses `A`, or if the ordering requirement fails.

### Example

### Uses
Applying conditional rewrite rules that are permutative and would loop without some ordering restriction. Applied automatically to some permutative rewrite rules in the simplifier, e.g. in `SIMP_CONV`.

### See also
IMP_REWR_CONV, ORDERED_REWR_CONV, REWR_CONV, SIMP_CONV, term_order.

# ORDERED_REWR_CONV

```
ORDERED_REWR_CONV : (term -> term -> bool) -> thm -> term -> thm
```

### Synopsis
Basic rewriting conversion restricted by term order.

## Description

Given an ordering relation `ord`, an equational theorem `A |- !x1...xn. s = t` that expresses a rewrite rule, the conversion `ORDERED_REWR_CONV` gives a conversion that applied to any term `s'` will attempt to match the left-hand side of the equation `s = t` to `s'`, and return the corresponding theorem `A |- s' = t'`, but only if `ord ‘s'‘ ‘t'‘`, i.e. if the left-hand side is "greater" in the ordering than the right-hand side, after instantiation. If the ordering condition is violated, it will fail, even if the match is fine.

## Failure

Fails if the theorem is not of the right form or the two terms cannot be matched, for example because the variables that need to be instantiated are free in the hypotheses `A`, or if the ordering requirement fails.

## Example

We apply the permutative rewrite:

```
# ADD_SYM;;
val it : thm = |- !m n. m + n = n + m
```

with the default term ordering `term_order` designed for this kind of application. Note that it applies in one direction:

```
# ORDERED_REWR_CONV term_order ADD_SYM ‘1 + 2‘;;
val it : thm = |- 1 + 2 = 2 + 1
```

but not the other:

```
# ORDERED_REWR_CONV term_order ADD_SYM ‘2 + 1‘;;
Exception: Failure "ORDERED_REWR_CONV: wrong orientation".
```

## Uses

Applying conditional rewrite rules that are permutative and would loop without some restriction. Thanks to the fact that higher-level rewriting operations like `REWRITE_CONV` and `REWRITE_TAC` have ordering built in for permutative rewrite rules, rewriting with theorem like `ADD_AC` will effectively normalize terms.

## See also

`IMP_REWR_CONV`, `ORDERED_IMP_REWR_CONV`, `REWR_CONV`, `SIMP_CONV`, `term_order`.

# orelsec_

```
orelsec_ : conv -> conv -> conv
```

## Synopsis
Non-infix version of ORELSEC.

## See also
ORELSEC.

# ORELSEC

```
(ORELSEC) : conv -> conv -> conv
```

## Synopsis
Applies the first of two conversions that succeeds.

## Description
(c1 ORELSEC c2) `t` returns the result of applying the conversion c1 to the term `t` if this succeeds. Otherwise (c1 ORELSEC c2) `t` returns the result of applying the conversion c2 to the term `t`.

## Failure
(c1 ORELSEC c2) `t` fails both c1 and c2 fail when applied to `t`.

## Example

```
# (NUM_ADD_CONV ORELSEC NUM_MULT_CONV) `2 + 2`;;
val it : thm = |- 2 + 2 = 4

#  (NUM_ADD_CONV ORELSEC NUM_MULT_CONV) `1 * 1`;;
val it : thm = |- 1 * 1 = 1
```

## See also
FIRST_CONV, THENC.

# orelse_

```
orelse_ : tactic -> tactic -> tactic
```

## Synopsis
Non-infix version of ORELSE.

## See also
ORELSE.

# ORELSE

```
(ORELSE) : tactic -> tactic -> tactic
```

## Synopsis
Applies first tactic, and iff it fails, applies the second instead.

## Description
If `t1` and `t2` are tactics, `t1 ORELSE t2` is a tactic which applies `t1` to a goal, and iff it fails, applies `t2` to the goal instead.

## Failure
The application of `ORELSE` to a pair of tactics never fails. The resulting tactic fails if both `t1` and `t2` fail when applied to the relevant goal.

## Example
The tactic `STRIP_TAC` breaks down the logical structure of a goal in various ways, e.g. stripping off universal quantifiers and putting the antecedent of implicational conclusions into the assumptions. However it does not break down equivalences into two implications, as `EQ_TAC` does. So you might start breaking down a goal corresponding to the inbuilt theorem `MOD_EQ_0`

```
  # g `!m n. ~(n = 0) ==> ((m MOD n = 0) <=> (?q. m = q * n))`;;
  ...
```

as follows

```
  # e(REPEAT(STRIP_TAC ORELSE EQ_TAC));;
  val it : goalstack = 2 subgoals (2 total)

   0 [`~(n = 0)`]
   1 [`m = q * n`]

  `m MOD n = 0`

   0 [`~(n = 0)`]
   1 [`m MOD n = 0`]

  `?q. m = q * n`
```

## See also
EVERY, FIRST, THEN.

# orelse_tcl_

```
orelse_tcl_ : thm_tactical -> thm_tactical -> thm_tactical
```

## Synopsis
Non-infix version of `ORELSE_TCL`.

## See also
`ORELSE_TCL`.

# ORELSE_TCL

```
(ORELSE_TCL) : thm_tactical -> thm_tactical -> thm_tactical
```

## Synopsis
Applies a theorem-tactical, and if it fails, tries a second.

## Description
When applied to two theorem-tacticals, `ttl1` and `ttl2`, a theorem-tactic `ttac`, and a theorem `th`, if `ttl1 ttac th` succeeds, that gives the result. If it fails, the result is `ttl2 ttac th`, which may itself fail.

## Failure
`ORELSE_TCL` fails if both the theorem-tacticals fail when applied to the given theorem-tactic and theorem.

## See also
`EVERY_TCL`, `FIRST_TCL`, `THEN_TCL`.

# ||

```
(||) : ('a -> 'b) -> ('a -> 'b) -> 'a -> 'b
```

## Synopsis
Produce alternative composition of two parsers.

## Description

If p1 and p2 are two parsers, p1 || p2 is a new parser that first tries to parse the input using p1, and if that fails with exception Noparse, tries p2 instead. The output is whatever parse result was achieved together with the unparsed input.

## Failure

Never fails.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of :('a)list -> 'b * ('a)list. The function should take a list of objects of type :'a (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

++, >>, a, atleast, elistof, finished, fix, leftbin, listof, many, nothing, possibly, rightbin, some.

# overload_interface

```
overload_interface : string * term -> unit
```

## Synopsis

Overload a symbol so it may denote a particular underlying constant.

## Description

HOL Light allows the same identifier to denote several different underlying constants. A call to overload_interface("ident",`cname`), where cname is either a constant to be denoted or a variable with the same name and type (if the constant is not yet defined) will include cname as one of the possible overload resolutions of the symbol ident. Moreover, when the resolution is not possible from type information, cname will now be the default. However, before any calls to overload_interface, the constant must have been declared overloadable with make_overloadable, and the term `cname` must have a type that is an instance of the most general "type skeleton" specified there.

## Failure

Fails if the identifier has not been declared overloadble, if the term is not a constant or variable, or it its type is not an instance of the declared type skeleton.

## Example

The symbol '+' has an overload skeleton of type `:A->A->A`. Here we overload it on type :bool to denote logical 'or'. (This is just for illustration; it's strongly recommended that you don't do this, since you will typically need to add more type annotations in terms to compensate for the ambiguity.)

```
# overload_interface("+",`(\/)`);;
val it : unit = ()
```

Now we can use the symbol '+' with multiple meanings in the same terms; the underlying constants are still the original ones, though:

```
# `(x = 1) + (1 + 1 = 2)`;;
val it : term = `(x = 1) + (1 + 1 = 2)`
```

You can also overload polymorphic symbols, e.g. overload '+' so that it maps to list append:

```
# overload_interface("+",`APPEND`);;
Warning: inventing type variables
val it : unit = ()

# APPEND;;
val it : thm = |- (!l. [] + l = l) /\ (!h t l. CONS h t + l = CONS h (t + l))
```

## See also

`make_overloadable`, `override_interface`, `prioritize_overload`, `reduce_interface`, `remove_interface`, `the_interface`, `the_overload_skeletons`.

# override_interface

```
override_interface : string * term -> unit
```

## Synopsis

Map identifier to specific underlying constant.

## Description

A call to `override_interface("name",`cname`)` makes the parser map instances of identifier `name` to whatever constant is called `cname`. Note that the term `` `cname` `` in the call may either be that constant or a variable of the appropriate type. This contrasts with `overload_interface`, which can make the same identifier map to several underlying constants, depending on type. A call to `override_interface` removes all other overloadings of the identifier, if any.

## Failure

Fails unless the term is a constant or variable.

## Example

You might want to make the exponentiation operation `EXP` on natural numbers parse and print as '^'. You can do this with

```
# override_interface("^",`(EXP)`);;
val it : unit = ()
```

Note that the special parse status (infix in this case) is based on the interface identifier, not the underlying constant, so that does not make '^' parse as infix:

```
# EXP;;
val it : thm = |- (!m. ^ m 0 = 1) /\ (!m n. ^ m (SUC n) = m * ^ m n)
```

but you can do that with a separate `parse_as_infix` call. It is also possible to override polymorphic constants, and all instances will be handled. For example, HOL Light's built-in list operations don't look much like OCaml:

```
# APPEND;;
val it : thm =
  |- (!l. APPEND [] l = l) /\
     (!h t l. APPEND (CONS h t) l = CONS h (APPEND t l))
```

but after a few interface modifications:

```
# parse_as_infix("::",(25,"right"));;
# parse_as_infix("@",(16,"right"));;
# override_interface("::",`CONS`);;
# override_interface("@",`APPEND`);;
```

it looks closer (you can remove the spaces round `::` using `unspaced_binops`):

```
# APPEND;;
val it : thm = |- (!l. [] @ l = l) /\ (!h t l. h :: t @ l = h :: (t @ l))
```

## See also

`overload_interface`, `parse_as_infix`, `reduce_interface`, `remove_interface`, `the_interface`, `the_overload_skeletons`.

# PAIRED_BETA_CONV

```
PAIRED_BETA_CONV : term -> thm
```

## Synopsis

Performs generalized beta conversion for tupled beta-redexes.

## Description

The conversion PAIRED_BETA_CONV implements beta-reduction for certain applications of tupled lambda abstractions called ‘tupled beta-redexes’. Tupled lambda abstractions have the form ‘\<vs>.tm‘, where <vs> is an arbitrarily-nested tuple of variables called a ‘varstruct’. For the purposes of PAIRED_BETA_CONV, the syntax of varstructs is given by:

```
<vs>  ::=   (v1,v2)  |  (<vs>,v)  |  (v,<vs>)  |  (<vs>,<vs>)
```

where v, v1, and v2 range over variables. A tupled beta-redex is an application of the form ‘(\<vs>.tm) t‘, where the term ‘t‘ is a nested tuple of values having the same structure as the varstruct <vs>. For example, the term:

```
‘(\((a,b),(c,d)). a + b + c + d)  ((1,2),(3,4))‘
```

is a tupled beta-redex, but the term:

```
‘(\((a,b),(c,d)). a + b + c + d)  ((1,2),p)‘
```

is not, since p is not a pair of terms.

Given a tupled beta-redex ‘(\<vs>.tm) t‘, the conversion PAIRED_BETA_CONV performs generalized beta-reduction and returns the theorem

```
|-  (\<vs>.tm) t = t[t1,...,tn/v1,...,vn]
```

where ti is the subterm of the tuple t that corresponds to the variable vi in the varstruct <vs>. In the simplest case, the varstruct <vs> is flat, as in the term:

```
‘(\(v1,...,vn).t) (t1,...,tn)‘
```

When applied to a term of this form, PAIRED_BETA_CONV returns:

```
|- (\(v1, ... ,vn).t) (t1, ... ,tn) = t[t1,...,tn/v1,...,vn]
```

As with ordinary beta-conversion, bound variables may be renamed to prevent free variable capture. That is, the term t[t1,...,tn/v1,...,vn] in this theorem is the result of substituting ti for vi in parallel in t, with suitable renaming of variables to prevent free variables in t1, ..., tn becoming bound in the result.

## Failure

PAIRED_BETA_CONV tm fails if tm is not a tupled beta-redex, as described above. Note that ordinary beta-redexes are specifically excluded: PAIRED_BETA_CONV fails when applied to ‘(\v.t)u‘. For these beta-redexes, use BETA_CONV.

## Example

The following is a typical use of the conversion:

```
# PAIRED_BETA_CONV ‘(\((a,b),(c,d)). a + b + c + d)  ((1,2),(3,4))‘;;
val it : thm = |- (\((a,b),c,d). a + b + c + d) ((1,2),3,4) = 1 + 2 + 3 + 4
```

Note that the term to which the tupled lambda abstraction is applied must have the same structure as the varstruct. For example, the following fail:

```
# PAIRED_BETA_CONV `(\((a,b),p). a + b)  ((1,2),(3+5,4))`;;

# PAIRED_BETA_CONV `(\((a,b),(c,d)). a + b + c + d)  ((1,2),p)`;;
```

because `p` is not a pair.

### See also

BETA_CONV, BETA_RULE, BETA_TAC, GEN_BETA_CONV.

## parse_as_binder

```
parse_as_binder : string -> unit
```

### Synopsis

Makes the quotation parser treat a name as a binder.

### Description

The call `parse_as_binder "c"` will make the quotation parser treat `c` as a binder, that is, allow the syntactic sugaring `` `c x. y` `` as a shorthand for `` `c (\x. y)` ``. As with normal binders, e.g. the universal quantifier, the special syntactic status may be suppressed by enclosing `c` in parentheses: `(c)`.

### Failure

Never fails.

### Example

```
# parse_as_binder "infinitely_many";;
val it : unit = ()
# `infinitely_many p:num. prime(p)`;;
`infinitely_many p. prime(p)`;;
```

### See also

binders, parses_as_binder, unparse_as_binder.

## parse_as_infix

```
parse_as_infix : string * (int * string) -> unit
```

## Synopsis
Adds identifier to list of infixes, with given precedence and associativity.

## Description
Certain identifiers are treated as infix operators with a given precedence and associativity (left or right). The call `parse_as_infix("op",(p,a))` adds `op` to the infix operators with precedence `p` and associativity `a` (it should be one of the two strings `"left"` or `"right"`). Note that the infix status is based purely on the name, which can be alphanumeric or symbolic, and does not depend on whether the name denotes a constant.

## Failure
Never fails; if the given string was already an infix, its precedence and associativity are changed to the new values.

## Example

```
# strip_comb `n choose k`;;
Warning: inventing type variables
val it : term * term list = (`n`, [`choose`; `k`])

# parse_as_infix("choose",(22,"right"));;
val it : unit = ()
# strip_comb `n choose k`;;
Warning: inventing type variables
val it : term * term list = (`(choose)`, [`n`; `k`])
```

## Uses
Adding user-defined binary operators.

## See also
`get_infix_status, infixes, unparse_as_infix.`

# parse_as_prefix

`parse_as_prefix : string -> unit`

## Synopsis
Gives an identifier prefix status.

## Description
Certain identifiers `c` have prefix status, meaning that combinations of the form `c f x` will be parsed as `c (f x)` rather than the usual `(c f) x`. The call `parse_as_prefix "c"` adds `c` to the list of such identifiers.

## Failure
Never fails, even if the string already has prefix status.

## See also
`is_prefix`, `prefixes`, `unparse_as_prefix`.

# parse_inductive_type_specification

```
parse_inductive_type_specification : string -> (hol_type * (string * hol_type list) list) list
```

## Synopsis
Parses the specification for an inductive type into a structured format.

## Description
The underlying function `define_type_raw` used inside `define_type` expects the inductive type specification in a more structured format. The function `parse_inductive_type_specification` parses the usual string form as handed to `define_type` and yields this structured form. In fact, `define_type` is just the composition of `define_type_raw` and `parse_inductive_type_specification`.

## Failure
Fails if there is a parsing error in the inductive type specification.

## See also
`define_type`, `define_type_raw`.

# parse_preterm

```
parse_preterm : lexcode list -> preterm * lexcode list
```

## Synopsis
Parses a preterm.

## Description
The call `parse_preterm t`, where `t` is a list of lexical tokens (as produced by `lex`), parses the tokens and returns a preterm as well as the unparsed tokens.

## Failure
Fails if there is a syntax error in the token list.

## Uses
This is mostly an internal function; pretypes and preterms are used as an intermediate representation for typechecking and overload resolution and are not normally of concern to users.

## See also
`lex`, `parse_pretype`, `parse_term`, `parse_type`.

# parse_pretype

```
parse_pretype : lexcode list -> pretype * lexcode list
```

## Synopsis
Parses a pretype.

## Description
The call `parse_pretype t`, where `t` is a list of lexical tokens (as produced by `lex`), parses the tokens and returns a pretype as well as the unparsed tokens.

## Failure
Fails if there is a syntax error in the token list.

## Uses
This is mostly an internal function; pretypes and preterms are used as an intermediate representation for typechecking and overload resolution and are not normally of concern to users.

## See also
`lex`, `parse_preterm`, `parse_term`, `parse_type`.

# parses_as_binder

```
parses_as_binder : string -> bool
```

## Synopsis
Tests if a string has binder status in the parser.

## Description
Certain identifiers `c` have binder status, meaning that `` `c x. y` `` is parsed as a shirthand for `` `(c) (\x. y)` ``. The call `parses_as_binder "c"` tests if `c` is one of the identifiers with binder status.

## Failure
Never fails.

## Example

```
# parses_as_binder "!";;
val it : bool = true
# parses_as_binder "==>";;
val it : bool = false
```

## See also
binders, parses_as_binder, unparse_as_binder.

# parse_term

`parse_term : string -> term`

## Synopsis
Parses a string into a HOL term.

## Description
The call `parse_term "s"` parses the string `s` into a HOL term. This is the function that is invoked automatically when a term is written in quotations `‘s‘`.

## Failure
Fails in the event of a syntax error or unparsed input.

## Example

```
# parse_term "p /\\ q ==> r";;
val it : term = ‘p /\ q ==> r‘
```

## Comments
Note that backslash characters should be doubled up when entering OCaml strings, as in the example above, since they are the string escape character. This is handled automatically by the quotation parser, so one doesn't need to do it (indeed shouldn't do it) when entering quotations between backquotes.

## See also
lex, parse_type.

# parse_type

```
parse_type : string -> hol_type
```

## Synopsis
Parses a string into a HOL type.

## Description
The call `parse_type "s"` parses the string `s` into a HOL type. This is the function that is invoked automatically when a type is written in quotations with an initial colon `‘:s‘`.

## Failure
Fails in the event of a syntax error or unparsed input.

## Example

```
# parse_type "num->bool";;
val it : hol_type = ‘:num->bool‘
```

## See also
`lex`, `parse_term`.

# partition

```
partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

## Synopsis
Separates a list into two lists using a predicate.

## Description
`partition p l` returns a pair of lists. The first list contains the elements which satisfy `p`. The second list contains all the other elements.

## Failure
Never fails.

## Example

```
# partition (fun x -> x mod 2 = 0) (1--10);;
val it : int list * int list = ([2; 4; 6; 8; 10], [1; 3; 5; 7; 9])
```

## See also
`chop_list`, `remove`, `filter`.

# PART_MATCH

```
PART_MATCH : (term -> term) -> thm -> term -> thm
```

## Synopsis
Instantiates a theorem by matching part of it to a term.

## Description
When applied to a 'selector' function of type `term -> term`, a theorem and a term:

```
PART_MATCH fn (A |- !x1...xn. t) tm
```

the function `PART_MATCH` applies `fn` to `t'` (the result of specializing universally quantified variables in the conclusion of the theorem), and attempts to match the resulting term to the argument term `tm`. If it succeeds, the appropriately instantiated version of the theorem is returned. Limited higher-order matching is supported, and some attempt is made to maintain bound variable names in higher-order matching.

## Failure
Fails if the selector function `fn` fails when applied to the instantiated theorem, or if the match fails with the term it has provided.

## Example
Suppose that we have the following theorem:

```
th = |- !x. x ==> x
```

then the following:

```
PART_MATCH (fst o dest_imp) th `T`
```

results in the theorem:

```
|- T ==> T
```

because the selector function picks the antecedent of the implication (the inbuilt specialization gets rid of the universal quantifier), and matches it to `T`. For a higher-order case rather similiar to what goes on inside HOL's `INDUCT_TAC`:

```
# num_INDUCTION;;
val it : thm = |- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)

# PART_MATCH rand it `!n. n <= n * n`;;
val it : thm =
  |- 0 <= 0 * 0 /\ (!n. n <= n * n ==> SUC n <= SUC n * SUC n)
     ==> (!n. n <= n * n)
```

To show a more interesting case with higher-order matching, where the pattern is not quite a higher-order pattern in the usual sense, consider the theorem:

```
# let th = MESON[num_CASES; NOT_SUC]
    `(!n. P(SUC n)) <=> !n. ~(n = 0) ==> P n`;;
...
val th : thm = |- (!n. P (SUC n)) <=> (!n. ~(n = 0) ==> P n)
```

and instantiate it as follows:

```
# PART_MATCH lhs th `!n. 1 <= SUC n`;;
val it : thm = |- (!n. 1 <= SUC n) <=> (!n. ~(n = 0) ==> 1 <= n)
```

## See also

INST_TYPE, INST_TY_TERM, match.

# PAT_CONV

```
PAT_CONV : term -> conv -> conv
```

## Synopsis

Apply a conversion at subterms identified by a "pattern" lambda-abstraction.

## Description

The call `PAT_CONV `\x1 ... xn. t[x1,...,xn]` cnv` gives a new conversion that applies `cnv` to subterms of the target term corresponding to the free instances of any `xi` in the pattern `t[x1,...,xn]`. The fact that the pattern is a function has no logical significance; it is just used as a convenient format for the pattern.

## Failure

Never fails until applied to a term, but then it may fail if the core conversion does on the chosen subterms.

## Example

Here we choose to evaluate just two subterms:

```
# PAT_CONV `\x. x + a + x` NUM_ADD_CONV `(1 + 2) + (3 + 4) + (5 + 6)`;;
val it : thm = |- (1 + 2) + (3 + 4) + 5 + 6 = 3 + (3 + 4) + 11
```

while here we swap two particular quantifiers in a long chain:

```
#  PAT_CONV `\x. !x1 x2 x3 x4 x5. x` (REWR_CONV SWAP_FORALL_THM)
     `!a b c d e f g h. something`;;
Warning: inventing type variables
Warning: inventing type variables
val it : thm =
  |- (!a b c d e f g h. something) <=> (!a b c d e g f h. something)
```

## See also

ABS_CONV, BINDER_CONV, BINOP_CONV, PATH_CONV, RAND_CONV, RATOR_CONV.

# PATH_CONV

```
PATH_CONV : string -> conv -> conv
```

## Synopsis

Applies a conversion to the subterm indicated by a path string.

## Description

The call `PATH_CONV p cnv` gives a new conversion that applies `cnv` to the subterm of a term identified by the path string `p`. This path string is interpreted as a sequence of direction indications:

- `"b"`: take the body of an abstraction
- `"l"`: take the left (rator) path in an application
- `"r"`: take the right (rand) path in an application

## Failure

The basic call to the path string and conversion never fails, but when applied to the term it may, if the path is not meaningful or if the conversion itself fails on the indicated subterm.

## Uses

More concise indication of sub-conversion application than by composing `RATOR_CONV`, `RAND_CONV` and `ABS_CONV`.

## Example

```
# PATH_CONV "rlr" NUM_ADD_CONV `(1 + 2) + (3 + 4) + (5 + 6)`;;
val it : thm = |- (1 + 2) + (3 + 4) + 5 + 6 = (1 + 2) + 7 + 5 + 6
```

## See also

find_path, follow_path.

# p

```
p : unit -> goalstack
```

## Synopsis

Prints the top level of the subgoal package goal stack.

## Description

The function p is part of the subgoal package, and prints the current goalstate.

## Failure

Never fails.

## Uses

Examining the proof state during an interactive proof session.

## Comments

Strictly speaking this function is side-effect-free. It simply \em returns the current goalstate. However, automatic printing will normally then print it, so that is the net effect.

## See also

b, e, g, r.

# PINST

```
PINST : (hol_type * hol_type) list -> (term * term) list -> thm -> thm
```

## Synopsis

Instantiate types and terms in a theorem.

## Description

The call PINST [ty1,tv1; ...; tyn,tvn] [tm1,v1; ...; tmk,vk] th instantiates both types and terms in the theorem th using the two instantiation lists. The tyi should be types, the `tvi` type variables, the `tmi` terms and the `vi` term variables. Note carefully that the `vi` refer to variables in the theorem *before* type instantiation, but the `tmi` should be replacements for the type-instantiated ones. More explicitly, the behaviour is as follows. First, the type variables in `th` are instantiated according to the list `[ty1,tv1; ...; tyn,tvn]`, exactly as for `INST_TYPE`. Moreover the same type instantiation is applied to the variables in the second list, to give `[tm1,v1'; ...; tmk,vk']`. This is then used to instantiate the already type-instantiated theorem.

## Failure

Fails if the instantiation lists are ill-formed, as with `INST` and `INST_TYPE`, for example if some `tvi` is not a type variable.

## Example

```
# let th = MESON[] `(x:A = y) <=> (y = x)`;;
...
val th : thm = |- x = y <=> y = x

# PINST [`:num`,`:A`] [`2 + 2`,`x:A`; `4`,`y:A`] th;;
val it : thm = |- 2 + 2 = 4 <=> 4 = 2 + 2
```

## See also

`INST, INST_TYPE.`

# >>

```
(>>) : ('a -> 'b * 'c) -> ('b -> 'd) -> 'a -> 'd * 'c
```

## Synopsis

Apply function to parser result.

## Description

If `p` is a parser and `f` a function from the parse result type, `p >> f` gives a new parser that 'pipes the original parser output through f', i.e. applies `f` to the result of the parse.

## Failure

Never fails.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

++, ||, a, atleast, elistof, finished, fix, leftbin, listof, many, nothing, possibly, rightbin, some.

# POP_ASSUM

```
POP_ASSUM : thm_tactic -> tactic
```

## Synopsis

Applies tactic generated from the first element of a goal's assumption list.

## Description

When applied to a theorem-tactic and a goal, POP_ASSUM applies the theorem-tactic to the first element of the assumption list, and applies the resulting tactic to the goal without the first assumption in its assumption list:

```
POP_ASSUM f ({A1;...;An} ?- t) = f (... |- A1) ({A2;...;An} ?- t)
```

## Failure

Fails if the assumption list of the goal is empty, or the theorem-tactic fails when applied to the popped assumption, or if the resulting tactic fails when applied to the goal (with depleted assumption list).

## Comments

It is possible simply to use the theorem ASSUME `A1` as required rather than use POP_ASSUM; this will also maintain A1 in the assumption list, which is generally useful. In addition, this approach can equally well be applied to assumptions other than the first.

There are admittedly times when POP_ASSUM is convenient, but it is unwise to use it if there is more than one assumption in the assumption list, since this introduces a dependency on the ordering and makes proofs somewhat brittle with respect to changes.

Another point to consider is that if the relevant assumption has been obtained by DISCH_TAC, it is often cleaner to use DISCH_THEN with a theorem-tactic. For example, instead of:

```
DISCH_TAC THEN POP_ASSUM (fun th -> SUBST1_TAC (SYM th))
```

one might use

```
DISCH_THEN (SUBST1_TAC o SYM)
```

## Example

Starting with the goal:

```
# g `!f x. 0 = x ==> f(x * f(x)) = f(x)`;;
```

and breaking it down:

```
# e(REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`0 = x`]

`f (x * f x) = f x`
```

we might use the equation to substitute backwards:

```
# e(POP_ASSUM(SUBST1_TAC o SYM) THEN REWRITE_TAC[MULT_CLAUSES]);;
```

but another alternative would have been:

```
# e(REWRITE_TAC[MULT_CLAUSES; SYM(ASSUME `0 = x`)]);;
```

and we could even have avoided putting the equation in the assumptions at all by from the beginning doing:

```
# e(REPEAT GEN_TAC THEN DISCH_THEN(SUBST1_TAC o SYM) THEN
    REWRITE_TAC[MULT_CLAUSES]);;
```

## Uses

Making more delicate use of an assumption than rewriting or resolution using it.

## See also

ASSUME, ASSUM_LIST, EVERY_ASSUM, POP_ASSUM_LIST, REWRITE_TAC.

# POP_ASSUM_LIST

```
POP_ASSUM_LIST : (thm list -> tactic) -> tactic
```

## Synopsis
Generates a tactic from the assumptions, discards the assumptions and applies the tactic.

## Description
When applied to a function and a goal, POP_ASSUM_LIST applies the function to a list of theorems corresponding to the assumptions of the goal, then applies the resulting tactic to the goal with an empty assumption list.

```
POP_ASSUM_LIST f ({A1;...;An} ?- t) = f [.. |- A1; ... ; .. |- An] (?- t)
```

## Failure
Fails if the function fails when applied to the list of assumptions, or if the resulting tactic fails when applied to the goal with no assumptions.

## Comments
There is nothing magical about POP_ASSUM_LIST: the same effect can be achieved by using ASSUME a explicitly wherever the assumption a is used. If POP_ASSUM_LIST is used, it is unwise to select elements by number from the ASSUMEd-assumption list, since this introduces a dependency on ordering.

## Example
We can collect all the assumptions of a goal into a conjunction and make them a new antecedent by:

```
POP_ASSUM_LIST(MP_TAC o end_itlist CONJ)
```

## Uses
Making more delicate use of the assumption list than simply rewriting etc.

## See also
ASSUM_LIST, EVERY_ASSUM, POP_ASSUM, REWRITE_TAC.

# possibly

```
possibly : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
```

## Synopsis
Attempts to parse, returning empty list of items in case of failure.

## Description
If p is a parser, then possibly p is another parser that attempts to parse with p and if successful returns the result as a singleton list, but will return the empty list instead if the core parser p raises Noparse.

## Failure

Never fails.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `rightbin`, `some`.

# pow10

`pow10 : int -> num`

## Synopsis

Returns power of 10 as umlimited-size integer.

## Description

When applied to an integer `n` (type `int`), `pow10` returns $10^n$ as an unlimited-precision integer (type `num`). The argument may be negative.

## Failure

Never fails.

## Example

```
# pow10(-1);;
val it : num = 1/10
# pow10(16);;
val it : num = 10000000000000000
```

## See also

`pow2`.

# pow2

`pow2 : int -> num`

## Synopsis
Returns power of 2 as umlimited-size integer.

## Description
When applied to an integer `n` (type `int`), `pow2` returns $2^n$ as an unlimited-precision integer (type `num`). The argument may be negative.

## Failure
Never fails.

## Example

```
# pow2(-2);;
val it : num = 1/4
# pow2(64);;
val it : num = 18446744073709551616
```

## See also
`pow10`.

# pp_print_qterm

`pp_print_qterm : formatter -> term -> unit`

## Synopsis
Prints a term with surrounding quotes to formatter.

## Description
The call `pp_print_term fmt tm` prints the usual textual representation of the term `tm` to the formatter `fmt`, in the form `` `tm` ``.

## Failure
Should never fail unless the formatter does.

## Comments
The usual case where the formatter is the standard output is `print_qterm`.

## See also
`pp_print_term`, `print_qterm`, `print_term`.

# pp_print_qtype

`pp_print_qtype : formatter -> hol_type -> unit`

## Synopsis
Prints a type with initial colon and surrounding quotes to formatter.

## Description
The call `pp_print_type fmt ty` prints the usual textual representation of the type `ty` to the formatter `fmt`, in the form `‘:ty‘`.

## Failure
Should never fail unless the formatter does.

## Comments
The usual case where the formatter is the standard output is `print_qtype`.

## See also
`pp_print_type`, `print_qtype`, `print_type`.

# pp_print_term

```
pp_print_term : formatter -> term -> unit
```

## Synopsis
Prints a term (without quotes) to formatter.

## Description
The call `pp_print_term fmt tm` prints the usual textual representation of the term `tm` to the formatter `fmt`. The string is just `tm` not `‘tm‘`.

## Failure
Should never fail unless the formatter does.

## Comments
The usual case where the formatter is the standard output is `print_term`.

## See also
`pp_print_qterm`, `print_qterm`, `print_term`.

# pp_print_thm

```
pp_print_thm : formatter -> thm -> unit
```

## Synopsis
Prints a theorem to formatter.

## Description
The call `pp_print_thm fmt th` prints the usual textual representation of the theorem `th` to the formatter `fmt`.

## Failure
Should never fail unless the formatter does.

## Comments
The usual case where the formatter is the standard output is `print_thm`.

## See also
`print_thm`.

# pp_print_type

```
pp_print_type : formatter -> hol_type -> unit
```

## Synopsis
Prints a type (without colon or quotes) to formatter.

## Description
The call `pp_print_type fmt ty` prints the usual textual representation of the type `ty` to the formatter `fmt`. The string is just `ty` not `` `:ty` ``.

## Failure
Should never fail unless the formatter does.

## Comments
The usual case where the formatter is the standard output is `print_type`.

## See also
`pp_print_qtype`, `print_qtype`, `print_type`.

# prebroken_binops

```
prebroken_binops : string list ref
```

## Synopsis
Determines which binary operators are line-broken to the left

## Description
The reference variable `prebroken_binops` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. It holds a list of the names of binary operators that, when a line break is needed, will be printed after the line break rather than before it. By default it contains just implication.

## Failure
Not applicable.

## Comments
Putting more operators such as conjunction in this list gives an output format closer to the one advocated in Lamport's "How to write a large formula" paper.

## See also
`pp_print_term`, `print_all_thm`, `print_unambiguous_comprehensions`, `reverse_interface_mapping`, `typify_universal_set`, `unspaced_binops`.

# prefixes

```
prefixes : unit -> string list
```

## Synopsis
Certain identifiers `c` have prefix status, meaning that combinations of the form `c f x` will be parsed as `c (f x)` rather than the usual `(c f) x`. The call `prefixes()` returns the list of all such identifiers.

## Failure
Never fails.

## Example
In the default HOL state:

```
# prefixes();;
val it : string list = ["~"; "--"; "mod"]
```

This explains, for example, why '`~ ~ p`' parses as '`~(~p)`' rather than parsing as '`(~ ~) p`' and generating a typechecking error.

## See also
`is_prefix`, `parse_as_prefix`, `unparse_as_prefix`.

# PRENEX_CONV

```
PRENEX_CONV : conv
```

## Synopsis
Puts a term already in NNF into prenex form.

## Description
When applied to a term already in negation normal form (see `NNF_CONV`, for example), the conversion `PRENEX_CONV` proves it equal to an equivalent in prenex form, with all quantifiers at the top level and a propositional body.

## Failure
Never fails; even on non-Boolean terms it will just produce a reflexive theorem.

## Example

```
# PRENEX_CONV `(!x. ?y. P x y) \/ (?u. !v. ?w. Q u v w)`;;
Warning: inventing type variables
val it : thm =
  |- (!x. ?y. P x y) \/ (?u. !v. ?w. Q u v w) <=>
     (!x. ?y u. !v. ?w. P x y \/ Q u v w)
```

## See also
`CNF_CONV`, `DNF_CONV`, `NNFC_CONV`, `NNF_CONV`, `SKOLEM_CONV`, `WEAK_CNF_CONV`, `WEAK_DNF_CONV`.

# PRESIMP_CONV

```
PRESIMP_CONV : conv
```

## Synopsis
Applies basic propositional simplifications and some miniscoping.

## Description
The conversion `PRESIMP_CONV` applies various routine simplifications to Boolean terms involving constants, e.g. `p /\ T <=> p`. It also tries to push universal quantifiers through conjunctions and existential quantifiers through disjunctions, e.g. `(?x. p[x] \/ q[x]) <=> (?x. p[x]) \/ (?x. q[x` ("miniscoping") but does not transform away other connectives like implication that would allow it do do this more completely.

## Failure
Never fails.

## Example

```
# PRESIMP_CONV `?x. x = 1 /\ y = 1 \/ F \/ T /\ y = 2`;;
val it : thm =
  |- (?x. x = 1 /\ y = 1 \/ F \/ T /\ y = 2) <=>
     (?x. x = 1) /\ y = 1 \/ y = 2
```

## Uses
Useful as an initial simplification before more substantial normal form conversions.

## See also
CNF_CONV, DNF_CONV, MINISCOPE_CONV, NNF_CONV, PRENEX_CONV, SKOLEM_CONV.

# preterm_of_term

`preterm_of_term : term -> preterm`

## Synopsis
Converts a term into a preterm.

## Description
HOL Light uses "pretypes" and "preterms" as intermediate structures for parsing and type-checking, which are later converted to types and terms. A call `preterm_of_term `tm`` converts in the other direction, from a normal HOL term back to a preterm.

## Failure
Never fails.

## Uses
User manipulation of preterms is not usually necessary, unless you seek to radically change aspects of parsing and typechecking.

## See also
pretype_of_type, term_of_preterm.

# pretype_of_type

`pretype_of_type : hol_type -> pretype`

## Synopsis
Converts a type into a pretype.

## Description
HOL Light uses "pretypes" and "preterms" as intermediate structures for parsing and typechecking, which are later converted to types and terms. A call `preterm_of_term ‘tm‘` converts in the other direction, from a normal HOL term back to a preterm.

## Failure
Never fails.

## Uses
User manipulation of pretypes is not usually necessary, unless you seek to radically change aspects of parsing and typechecking.

## See also
`preterm_of_term`, `type_of_pretype`.

# print_all_thm

```
print_all_thm : bool ref
```

## Synopsis
Flag determining whether the assumptions of theorems are printed explicitly.

## Description
The reference variable `print_all_thm` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. When it is `true`, as it is by default, all assumptions of theorems are printed. When it is `false`, they are abbreviated by dots.

## Failure
Not applicable.

## Example

```
# let th = ADD_ASSUM ‘1 + 1 = 2‘ (ASSUME ‘2 + 2 = 4‘);;
val th : thm = 2 + 2 = 4, 1 + 1 = 2 |- 2 + 2 = 4
# print_all_thm := false;;
val it : unit = ()
# th;;
val it : thm = ... |- 2 + 2 = 4
```

## See also
`pp_print_term`, `prebroken_binops`, `print_unambiguous_comprehensions`, `reverse_interface_mapping`, `typify_universal_set`, `unspaced_binops`.

# print_fpf

```
print_fpf : ('a, 'b) func -> unit
```

## Synopsis
Print a finite partial function.

## Description
This prints a finite partial function but only as a trivial string '`<func>`'. Installed automatically at the top level and probably not useful for most users.

## Failure
Never fails.

## See also
`|->`, `|=>`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# print_goal

```
print_goal : goal -> unit
```

## Synopsis
Print a goal.

## Description
`print_goalstack g` prints the goal `g` to standard output, with no following newline.

## Failure
Never fails.

## Comments
This is invoked automatically when something of type `goal` is produced at the top level, so manual invocation is not normally needed.

## See also
`print_goalstack`, `print_term`.

# print_goalstack

`print_goalstack : goalstack -> unit`

## Synopsis
Print a goalstack.

## Description
`print_goalstack gs` prints the goalstack `gs` to standard output, with no following newline.

## Failure
Never fails.

## Comments
This is invoked automatically when something of type `goalstack` is produced at the top level, so manual invocation is not normally needed.

## See also
`print_goal`, `print_term`.

# print_num

`print_num : num -> unit`

## Synopsis
Print an arbitrary-precision number to the terminal.

## Description
This function prints an arbitrary-precision (type `num`) number to the terminal. It is automatically invoked on anything of type `num` at the toplevel anyway, but it may sometimes be useful to issue it under user control.

## Failure
Never fails.

# print_qterm

`print_qterm : term -> unit`

## Synopsis

Prints a HOL term with surrounding quotes to standard output.

## Description

The call `print_term tm` prints the usual textual representation of the term `tm` to the standard output, that is `` `:tm` ``.

## Failure

Never fails.

## Comments

This is the function that is invoked automatically in the toplevel when printing terms.

## See also

`pp_print_qterm`, `pp_print_term`, `print_term`.

# print_qtype

```
print_qtype : hol_type -> unit
```

## Synopsis

Prints a type with colon and surrounding quotes to standard output.

## Description

The call `print_type ty` prints the usual textual representation of the type `ty` to the standard output, that is `` `:ty` ``.

## Failure

Never fails.

## Comments

This is the function that is invoked automatically in the toplevel when printing types.

## See also

`pp_print_qtype`, `pp_print_type`, `print_type`.

# print_term

```
print_term : term -> unit
```

## Synopsis
Prints a HOL term (without quotes) to the standard output.

## Description
The call `print_term tm` prints the usual textual representation of the term `tm` to the standard output. The string is just `tm` not `‘tm‘`.

## Failure
Never fails.

## Uses
Producing debugging output in complex rules. Note that terms are already printed at the toplevel anyway, so it is not needed to examine results interactively.

## See also
`pp_print_qterm`, `pp_print_term`, `print_qterm`.

# print_thm

```
print_thm : thm -> unit
```

## Synopsis
Prints a HOL theorem to the standard output.

## Description
The call `print_thm th` prints the usual textual representation of the theorem `th` to the standard output.

## Comments
This is invoked automatically at the toplevel when theorems are printed.

## See also
`print_type`, `print_term`.

# print_to_string

```
print_to_string : (formatter -> 'a -> 'b) -> 'a -> string
```

## Synopsis
Modifies a formatting printing function to return its output as a string.

### Description
If p is a printing function whose first argument is a formatter (a standard OCaml datatype indicating an output for printing functions), `print_to_string P` gives a function that invokes it and collects and returns its output as a string.

### Failure
Fails only if the core printing function fails.

### Example
The standard function `string_of_term` is defined as:

```
# let string_of_term = print_to_string pp_print_term;;
```

### Uses
Converting a general printing function to a 'convert to string' function, as in the example above.

### See also
`pp_print_term`, `pp_print_thm`, `pp_print_type`.

# print_type

```
print_type : hol_type -> unit
```

### Synopsis
Prints a type (without colon or quotes) to standard output.

### Description
The call `print_type ty` prints the usual textual representation of the type `ty` to the standard output. The string is just `ty` not `` `:ty` ``.

### Failure
Never fails.

### Uses
Producing debugging output in complex rules. Note that terms are already printed at the toplevel anyway, so it is not needed to examine results interactively.

### See also
`pp_print_qtype`, `pp_print_type`, `print_qtype`.

# print_unambiguous_comprehensions

```
print_unambiguous_comprehensions : bool ref
```

## Synopsis
Determines whether bound variables in set abstractions are made explicit.

## Description
The reference variable `print_unambiguous_comprehensions` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. When it is `true`, all set comprehensions are printed with an explicit indication of the bound variables in the middle: `` `{t | vs | p}` ``. When it is `false`, as it is by default, this printing of the set of bound variables is only done when the term would otherwise fail to match the default parsing behaviour on input, and otherwise just printed as `` `{t | p}` ``. The parsing behaviour for such a term is to take the bound variables to be those free in both `t` and `p`, unless there is just one variable free in `t` (in which case that variable is the only bound one) or there are none free in `p` (in which case all free variables of `t` are taken).

## Failure
Not applicable.

## Example

```
# print_unambiguous_comprehensions := false;;
val it : unit = ()
# `{x + y | x | EVEN(x)}`;;
val it : term = `{x + y | EVEN x}`

# print_unambiguous_comprehensions := true;;
val it : unit = ()
# `{x + y | x | EVEN(x)}`;;
val it : term = `{x + y | x | EVEN x}`
```

## See also
`pp_print_term`, `prebroken_binops`, `print_all_thm`, `reverse_interface_mapping`, `typify_universal_set`, `unspaced_binops`.

# prioritize_int

```
prioritize_int : unit -> unit
```

## Synopsis
Give integer type `int` priority in operator overloading.

## Description
Symbols for several arithmetical (‘`+`’, ‘`-`’, ...) and relational (‘`<`’, ‘`>=`’, ...) operators are overloaded so that they may denote the operators for several different number systems, particularly

num (natural numbers), int (integers) and real (real numbers). The choice is normally made based on some known types, or the presence of operators that are not overloaded for the number systems. (For example, numerals like 42 are always assumed to be of type num, while the division operator '/' is only defined for real.) In the absence of any such indication, a default choice will be made. The effect of prioritize_int() is to make int, the integer type, the default.

## Failure
Never fails.

## Example
With integer priority, most things are interpreted as type int

```
# prioritize_int();;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:int`
```

except that numerals are always of type num, and so:

```
# type_of `x + 1`;;
val it : hol_type = `:num`
```

and any explicit type information is used before using the defaults:

```
# type_of `(x:real) + y`;;
val it : hol_type = `:real`
```

## Comments
It is perhaps better practice to insert types explicitly to avoid dependence on such defaults, otherwise proofs can become context-dependent. However it is often very convenient.

## See also
make_overloadable, overload_interface, prioritize_num, prioritize_overload, prioritize_real, the_overload_skeletons.

# prioritize_num

```
prioritize_num : unit -> unit
```

## Synopsis
Give natural number type num priority in operator overloading.

## Description

Symbols for several arithmetical ('`+`', '`-`', ...) and relational ('`<`', '`>=`', ...) operators are overloaded so that they may denote the operators for several different number systems, particularly `num` (natural numbers), `int` (integers) and `real` (real numbers). The choice is normally made based on some known types, or the presence of operators that are not overloaded for the number systems. (For example, numerals like `42` are always assumed to be of type `num`, while the division operator '`/`' is only defined for `real`.) In the absence of any such indication, a default choice will be made. The effect of `prioritize_num()` is to make `num`, the natural number type, the default.

## Failure

Never fails.

## Example

With real priority, most things are interpreted as type `real`:

```
# prioritize_real();;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:real`
```

except that numerals are always of type `num`, and so:

```
# type_of `x + 1`;;
val it : hol_type = `:num`
```

By making `num` the priority, everything is interpreted as `num`:

```
# prioritize_num();;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:num`
```

unless there is some explicit type information to the contrary:

```
# type_of `(x:real) + y`;;
val it : hol_type = `:real`
```

## Comments

It is perhaps better practice to insert types explicitly to avoid dependence on such defaults, otherwise proofs can become context-dependent. However it is often very convenient.

## See also

`make_overloadable`, `overload_interface`, `prioritize_int`, `prioritize_overload`, `prioritize_real`, `the_overload_skeletons`.

# prioritize_overload

```
prioritize_overload : hol_type -> unit
```

## Synopsis
Give overloaded constants involving a given type priority in operator overloading.

## Description
In general, overloaded operators in the concrete syntax, such as '+', are ambigous, referring to one of several underlying constants. The choice is normally made based on some known types, or the presence of operators that are not overloaded for the number systems. (For example, numerals like 42 are always assumed to be of type `num`, while the division operator '/' is only defined for `real`.) In the absence of any such indication, a default choice will be made. The effect of `prioritize_overload `:ty`` is to run through the overloaded symbols making the first instance of each where the generic type variables in the type skeleton are replaced by type ``:ty`` the first priority when no other indication is made.

## Failure
Never fails.

## Example
With real priority, most things are interpreted as type `real`:

```
# prioritize_overload `:real`;;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:real`
```

By making `int` the priority, everything is interpreted as `int`:

```
# prioritize_overload `:int`;;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:int`
```

unless there is some explicit type information to the contrary:

```
# type_of `(x:real) + y`;;
val it : hol_type = `:real`
```

## Comments
It is perhaps better practice to insert types explicitly to avoid dependence on such defaults, otherwise proofs can become context-dependent. However it is often very convenient.

## See also

`make_overloadable`, `overload_interface`, `prioritize_int`, `prioritize_num`, `prioritize_real`, `the_overload_skeletons`.

# prioritize_real

```
prioritize_real : unit -> unit
```

## Synopsis

Give real number type `real` priority in operator overloading.

## Description

Symbols for several arithmetical (‘`+`’, ‘`-`’, ...) and relational (‘`<`’, ‘`>=`’, ...) operators are overloaded so that they may denote the operators for several different number systems, particularly `num` (natural numbers), `int` (integers) and `real` (real numbers). The choice is normally made based on some known types, or the presence of operators that are not overloaded for the number systems. (For example, numerals like `42` are always assumed to be of type `num`, while the division operator ‘`/`’ is only defined for `real`.) In the absence of any such indication, a default choice will be made. The effect of `prioritize_real()` is to make `real`, the real number type, the default.

## Failure

Never fails.

## Example

With real priority, most things are interpreted as type `real`:

```
# prioritize_real();;
val it : unit = ()

# type_of `x + y`;;
val it : hol_type = `:real`
```

except that numerals are always of type `num`, and so:

```
# type_of `x + 1`;;
val it : hol_type = `:num`
```

and any explicit type information is used before using the defaults:

```
# type_of `(x:int) + y`;;
val it : hol_type = `:int`
```

## Comments

It is perhaps better practice to insert types explicitly to avoid dependence on such defaults, otherwise proofs can become context-dependent. However it is often very convenient.

## See also

`make_overloadable`, `overload_interface`, `prioritize_int`, `prioritize_num`, `prioritize_overload`, `the_overload_skeletons`.

# PROP_ATOM_CONV

```
PROP_ATOM_CONV : conv -> conv
```

## Synopsis

Applies a conversion to the 'atomic subformulas' of a formula.

## Description

When applied to a Boolean term, `PROP_ATOM_CONV conv` descends recursively through any number of the core propositional connectives '`~`', '`/\`', '`\/`', '`==>`' and '`<=>`', as well as the quantifiers '`!x. p[x]`', '`?x. p[x]`' and '`?!x. p[x]`'. When it reaches a subterm that can no longer be decomposed into any of those items (e.g. the starting term if it is not of Boolean type), the conversion `conv` is tried, with a reflexive theorem returned in case of failure. That is, the conversion is applied to the "atomic subformulas" in the usual sense of first-order logic.

## Failure

Never fails.

## Example

Here we swap all equations in a formula, but not any logical equivalences that are part of its logical structure:

```
# PROP_ATOM_CONV(ONCE_DEPTH_CONV SYM_CONV)
    `(!x. x = y ==> x = z) <=> (y = z <=> 1 + z = z + 1)`;;
  val it : thm =
    |- ((!x. x = y ==> x = z) <=> y = z <=> 1 + z = z + 1) <=>
       (!x. y = x ==> z = x) <=>
       z = y <=>
       z + 1 = 1 + z
```

By contrast, just `ONCE_DEPTH_CONV SYM_CONV` would just swap the top-level logical equivalence.

## Uses

Carefully constraining the application of conversions.

## See also

`DEPTH_BINOP_CONV`, `ONCE_DEPTH_CONV`.

# prove_cases_thm

```
prove_cases_thm : thm -> thm
```

## Synopsis

Proves a structural cases theorem for an automatically-defined concrete type.

## Description

`prove_cases_thm` takes as its argument a structural induction theorem, in the form returned by `prove_induction_thm` for an automatically-defined concrete type. When applied to such a theorem, `prove_cases_thm` automatically proves and returns a theorem which states that every value the concrete type in question is denoted by the value returned by some constructor of the type.

## Failure

Fails if the argument is not a theorem of the form returned by `prove_induction_thm`

## Example

The following type definition for labelled binary trees:

```
# let ith,rth = define_type "tree = LEAF num | NODE tree tree";;
val ith : thm =
  |- !P. (!a. P (LEAF a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (NODE a0 a1))
         ==> (!x. P x)
val rth : thm =
  |- !f0 f1.
         ?fn. (!a. fn (LEAF a) = f0 a) /\
              (!a0 a1. fn (NODE a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

returns an induction theorem `ith` that can then be fed to `prove_cases_thm`:

```
# prove_cases_thm ith;;
val it : thm = |- !x. (?a. x = LEAF a) \/ (?a0 a1. x = NODE a0 a1)
```

## Comments

An easier interface is `cases "tree"`. This function is mainly intended to generate the cases theorems for that function.

## See also

`cases`, `define_type`, `INDUCT_THEN`, `new_recursive_definition`, `prove_constructors_distinct`, `prove_constructors_one_one`, `prove_induction_thm`.

# prove_constructors_distinct

```
prove_constructors_distinct : thm -> thm
```

## Synopsis

Proves that the constructors of an automatically-defined concrete type yield distinct values.

## Description

`prove_constructors_distinct` takes as its argument a primitive recursion theorem, in the form returned by `define_type` for an automatically-defined concrete type. When applied to such a theorem, `prove_constructors_distinct` automatically proves and returns a theorem which states that distinct constructors of the concrete type in question yield distinct values of this type.

## Failure

Fails if the argument is not a theorem of the form returned by `define_type`, or if the concrete type in question has only one constructor.

## Example

The following type definition for labelled binary trees:

```
# let ith,rth = define_type "tree = LEAF num | NODE tree tree";;
val ith : thm =
  |- !P. (!a. P (LEAF a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (NODE a0 a1))
         ==> (!x. P x)
val rth : thm =
  |- !f0 f1.
         ?fn. (!a. fn (LEAF a) = f0 a) /\
              (!a0 a1. fn (NODE a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

returns a recursion theorem `rth` that can then be fed to `prove_constructors_distinct`:

```
# prove_constructors_distinct rth;;
val it : thm = |- !a a0' a1'. ~(LEAF a = NODE a0' a1')
```

This states that leaf nodes are different from internal nodes. When the concrete type in question has more than two constructors, the resulting theorem is just conjunction of inequalities of this kind.

## Comments

An easier interface is `distinctness "tree"`; this function is mainly intended to generate that theorem internally.

## See also

`define_type`, `distinctness`, `INDUCT_THEN`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_one_one`, `prove_induction_thm`, `prove_rec_fn_exists`.

# prove_constructors_injective

```
prove_constructors_injective : thm -> thm
```

## Synopsis
Proves that the constructors of an automatically-defined concrete type are injective.

## Description
`prove_constructors_one_one` takes as its argument a primitive recursion theorem, in the form returned by `define_type` for an automatically-defined concrete type. When applied to such a theorem, `prove_constructors_one_one` automatically proves and returns a theorem which states that the constructors of the concrete type in question are injective (one-to-one). The resulting theorem covers only those constructors that take arguments (i.e. that are not just constant values).

## Failure
Fails if the argument is not a theorem of the form returned by `define_type`, or if all the constructors of the concrete type in question are simply constants of that type.

## Example
The following type definition for labelled binary trees:

```
# let ith,rth = define_type "tree = LEAF num | NODE tree tree";;
val ith : thm =
  |- !P. (!a. P (LEAF a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (NODE a0 a1))
         ==> (!x. P x)
val rth : thm =
  |- !f0 f1.
         ?fn. (!a. fn (LEAF a) = f0 a) /\
              (!a0 a1. fn (NODE a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

returns a recursion theorem `rth` that can then be fed to `prove_constructors_injective`:

```
# prove_constructors_injective rth;;
val it : thm =
  |- (!a a'. LEAF a = LEAF a' <=> a = a') /\
     (!a0 a1 a0' a1'. NODE a0 a1 = NODE a0' a1' <=> a0 = a0' /\ a1 = a1')
```

This states that the constructors `LEAF` and `NODE` are both injective.

## Comments
An easier interface is `injectivity "tree"`; the present function is mainly intended to generate that theorem internally.

### See also

`define_type`, `INDUCT_THEN`, `injectivity`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_distinct`, `prove_induction_thm`, `prove_rec_fn_exists`.

# prove

```
prove : term * tactic -> thm
```

### Synopsis

Attempts to prove a boolean term using the supplied tactic.

### Description

When applied to a term-tactic pair `(tm,tac)`, the function `prove` attempts to prove the goal `?- tm`, that is, the term `tm` with no assumptions, using the tactic `tac`. If `prove` succeeds, it returns the corresponding theorem `A |- tm`, where the assumption list `A` may not be empty if the tactic is invalid; `prove` has no inbuilt validity-checking.

### Failure

Fails if the term is not of type `bool` (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal. In the latter case `prove` will list the unsolved goals to help the user.

### See also

`TAC_PROOF`, `VALID`.

# prove_general_recursive_function_exists

```
prove_general_recursive_function_exists : term -> thm
```

### Synopsis

Proves existence of general recursive function.

### Description

The function `prove_general_recursive_function_exists` should be applied to an existentially quantified term `` `?f. def_1[f] /\ ... /\ def_n[f]` ``, where each clause `def_i` is a universally quantified equation with an application of `f` to arguments on the left-hand side. The idea is that these clauses define the action of `f` on arguments of various kinds, for example on an empty list and nonempty list:

```
?f. (f [] = a) /\ (!h t. CONS h t = k[f,h,t])
```

or on even numbers and odd numbers:

```
?f. (!n. f(2 * n) = a[f,n]) /\ (!n. f(2 * n + 1) = b[f,n])
```

The returned value is a theorem whose conclusion matches the input term, with zero, one or two assumptions, depending on what conditions had been proven automatically. Roughly, one assumption states that the clauses are not mutually contradictory, as in

```
?f. (!n. f(n + 1) = 1) /\ (!n. f(n + 2) = 2)
```

and the other states that there is some wellfounded order making any recursion admissible.

## Failure

Fails only if the definition is malformed. However it is possible that for an inadmissible definition the assumptions of the theorem may not hold.

## Example

In the definition of the Fibonacci numbers, the function successfully eliminates all the hypotheses and just proves the claimed existence assertion:

```
# prove_general_recursive_function_exists
   `?fib. fib 0 = 1 /\ fib 1 = 1 /\
          !n. fib(n + 2) = fib(n) + fib(n + 1)`;;
val it : thm =
  |- ?fib. fib 0 = 1 /\ fib 1 = 1 /\ (!n. fib (n + 2) = fib n + fib (n + 1))
```

whereas in the following case, the function cannot automatically discover the appropriate ordering to make the recursion admissible, so an assumption is included:

```
# let eth = prove_general_recursive_function_exists
   `?upto. !m n. upto m n =
                 if n < m then []
                 else if m = n then [n]
                 else  CONS m (upto (m + 1) n)`;;
val eth : thm =
  ?(<<). WF (<<) /\ (!m n. (T /\ ~(n < m)) /\ ~(m = n) ==> m + 1,n << m,n)
  |- ?upto. !m n.
                upto m n =
                (if n < m
                 then []
                 else if m = n then [n] else CONS m (upto (m + 1) n))
```

You can prove the condition by supplying an appropriate ordering, e.g.

```
# let wfth = prove(hd(hyp eth),
                   EXISTS_TAC `measure (\(m:num,n:num). n - m)` THEN
                   REWRITE_TAC[WF_MEASURE; measure] THEN ARITH_TAC);;
val wfth : thm =
  |- ?(<<). WF (<<) /\ (!m n. (T /\ ~(n < m)) /\ ~(m = n) ==> m + 1,n << m,n)
```

and so get the pure existence theorem with `PROVE_HYP wfth eth`.

## Uses

To prove existence of a recursive function defined by clauses without actually defining it. In order to define it, use `define`. To further forestall attempts to prove conditions automatically, consider `pure_prove_recursive_function_exists` or even `instantiate_casewise_recursion`.

## See also

`define`, `instantiate_casewise_recursion`, `pure_prove_recursive_function_exists`.

# PROVE_HYP

```
PROVE_HYP : thm -> thm -> thm
```

## Synopsis

Eliminates a provable assumption from a theorem.

## Description

When applied to two theorems, `PROVE_HYP` gives a new theorem with the conclusion of the second and the union of the assumption list minus the conclusion of the first theorem.

```
     A1 |- t1     A2 |- t2
   ------------------------  PROVE_HYP
    (A1 u A2) - {t1} |- t2
```

## Failure

Never fails.

## Example

```
# let th1 = CONJUNCT2(ASSUME `p /\ q /\ r`)
  and th2 = CONJUNCT2(ASSUME `q /\ r`);;
val th1 : thm = p /\ q /\ r |- q /\ r
val th2 : thm = q /\ r |- r

# PROVE_HYP th1 th2;;
val it : thm = p /\ q /\ r |- r
```

## Comments

This is sometimes known as the Cut rule. It is not necessary for the conclusion of the first theorem to be the same as an assumption of the second, but `PROVE_HYP` is otherwise of doubtful value.

## See also

`DEDUCT_ANTISYM_RULE`, `DISCH`, `MP`, `UNDISCH`.

# prove_inductive_relations_exist

```
prove_inductive_relations_exist : term -> thm
```

## Synopsis

Prove existence of inductively defined relations without defining them.

## Description

The function `prove_inductive_relations_exist` should be given a specification for an inductively defined relation `R`, or more generally a family `R1,...,Rn` of mutually inductive relations; the required format is explained further in the entry for `new_inductive_definition`. It returns an existential theorem `A |- ?R1 ... Rn. rules /\ induction /\ cases`, where `rules`, `induction` and `cases` are the rule, induction and cases theorems, explained further in the entry for `new_inductive_definition`. In contrast with `new_inductive_definition`, no actual definitions are made. The assumption list `A` is normally empty, but will include any monotonicity hypotheses that were not proven automatically.

## Failure

Fails if the form of the rules is wrong.

## Example

The traditional example of even and odd numbers:

```
# prove_inductive_relations_exist
   `even(0) /\ odd(1) /\
    (!n. even(n) ==> odd(n + 1)) /\
    (!n. odd(n) ==> even(n + 1))`;;
val it : thm =
  |- ?even odd.
         (even 0 /\
          odd 1 /\
          (!n. even n ==> odd (n + 1)) /\
          (!n. odd n ==> even (n + 1))) /\
         (!odd' even'.
              even' 0 /\
              odd' 1 /\
              (!n. even' n ==> odd' (n + 1)) /\
              (!n. odd' n ==> even' (n + 1))
              ==> (!a0. odd a0 ==> odd' a0) /\ (!a1. even a1 ==> even' a1)) /\
         (!a0. odd a0 <=> a0 = 1 \/ (?n. a0 = n + 1 /\ even n)) /\
         (!a1. even a1 <=> a1 = 0 \/ (?n. a1 = n + 1 /\ odd n))
```

Here is a example where we get a nonempty list of hypotheses because HOL cannot prove monotonicity (and indeed, it doesn't hold).

```
# prove_inductive_relations_exist `!x. ~P(x) ==> P(x+1)`;;
val it : thm =
  !P P'.
      (!a. P a ==> P' a)
      ==> (!a. (?x. a = x + 1 /\ ~P x) ==> (?x. a = x + 1 /\ ~P' x))
  |- ?P. (!x. ~P x ==> P (x + 1)) /\
         (!P'. (!x. ~P' x ==> P' (x + 1)) ==> (!a. P a ==> P' a)) /\
         (!a. P a <=> (?x. a = x + 1 /\ ~P x))
```

## Uses

Using existence of inductive relations as an auxiliary device inside a proof.

## See also

`new_inductive_definition`.

# prove_monotonicity_hyps

```
prove_monotonicity_hyps : thm -> thm
```

## Synopsis
Attempt to prove monotonicity hypotheses of theorem automatically.

## Description
Given a theorem `A |- t`, the rule `prove_monotonicity_hyps` attempts to prove and remove all hypotheses that are not equations, by breaking them down and repeatedly using `MONO_TAC`. Any that are equations or are not automatically provable will be left as they are.

## Failure
Never fails but may have no effect.

## Comments
Normally, this kind of reasoning is automated by the inductive definitions package, so explicit use of this tactic is rare.

## See also
`MONO_TAC`, `monotonicity_theorems`, `new_inductive_definition`, `prove_inductive_relations_exist`.

# prove_recursive_functions_exist

```
prove_recursive_functions_exist : thm -> term -> thm
```

## Synopsis
Prove existence of recursive function over inductive type.

## Description
This function has essentially the same interface and functionality as `new_recursive_definition`, but it merely proves the existence of the function rather than defining it.

The first argument to `prove_recursive_functions_exist` is the primitive recursion theorem for the concrete type in question; this is normally the second theorem obtained from `define_type`. The second argument is a term giving the desired primitive recursive function definition. The value returned by `prove_recursive_functions_exist` is a theorem stating the existence of a function satisfying the 'definition' clauses. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

Let `C1`, ..., `Cn` be the constructors of this type, and let '`(Ci vs)`' represent a (curried) application of the `i`th constructor to a sequence of variables. Then a curried primitive recursive function fn over ty can be specified by a conjunction of (optionally universally-quantified) clauses of the form:

```
fn v1 ... (C1 vs1) ... vm  =  body1   /\
fn v1 ... (C2 vs2) ... vm  =  body2   /\
                              .
                              .
fn v1 ... (Cn vsn) ... vm  =  bodyn
```

where the variables v1, ..., vm, vs are distinct in each clause, and where in the ith clause fn appears (free) in bodyi only as part of an application of the form:

```
`fn t1 ... v ... tm`
```

in which the variable v of type ty also occurs among the variables vsi.

If <definition> is a conjunction of clauses, as described above, then evaluating:

```
prove_recursive_functions_exist th `<definition>`;;
```

automatically proves the existence of a function fn that satisfies the defining equations supplied, and returns a theorem:

```
|- ?fn. <definition>
```

prove_recursive_functions_exist also allows the supplied definition to omit clauses for any number of constructors. If a defining equation for the ith constructor is omitted, then the value of fn at that constructor:

```
fn v1 ... (Ci vsi) ... vn
```

is left unspecified (fn, however, is still a total function).

## Failure

Fails if the clauses cannot be matched up with the recursion theorem. You may find that prove_general_recursive_function_exists still works in such cases.

## Example

Here we show that there exists a product function:

```
prove_recursive_functions_exist num_RECURSION
 `(prod f 0 = 1) /\ (!n. prod f (SUC n) = f(SUC n) * prod f n)`;;
val it : thm =
  |- ?prod. prod f 0 = 1 /\ (!n. prod f (SUC n) = f (SUC n) * prod f n)
```

## Comments

Often prove_general_recursive_function_exists is an easier route to the same goal. Its interface is simpler (no need to specify the recursion theorem) and it is more powerful. However, for suitably constrained definitions `prove_recursive_functions_exist` works well and is much more efficient.

## Uses

It is more usual to want to actually make definitions of recursive functions. However, if a recursive function is needed in the middle of a proof, and seems to ad-hoc for general use, you may just use `prove_recursive_functions_exist`, perhaps adding the "definition" as an assumption of the goal with `CHOOSE_TAC`.

## See also

`new_inductive_definition`, `new_recursive_definition`,
`prove_general_recursive_function_exists`.

# PURE_ASM_REWRITE_RULE

```
PURE_ASM_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis

Rewrites a theorem including the theorem's assumptions as rewrites.

## Description

The list of theorems supplied by the user and the assumptions of the object theorem are used to generate a set of rewrites, without adding implicitly the basic tautologies stored under `basic_rewrites`. The rule searches for matching subterms in a top-down recursive fashion, stopping only when no more rewrites apply. For a general description of rewriting strategies see `GEN_REWRITE_RULE`.

## Failure

Rewriting with `PURE_ASM_REWRITE_RULE` does not result in failure. It may diverge, in which case `PURE_ONCE_ASM_REWRITE_RULE` may be used.

## See also

`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`,
`PURE_ONCE_ASM_REWRITE_RULE`.

# PURE_ASM_REWRITE_TAC

```
PURE_ASM_REWRITE_TAC : thm list -> tactic
```

## Synopsis

Rewrites a goal including the goal's assumptions as rewrites.

## Description

PURE_ASM_REWRITE_TAC generates a set of rewrites from the supplied theorems and the assumptions of the goal, and applies these in a top-down recursive manner until no match is found. See GEN_REWRITE_TAC for more information on the group of rewriting tactics.

## Failure

PURE_ASM_REWRITE_TAC does not fail, but it can diverge in certain situations. For limited depth rewriting, see PURE_ONCE_ASM_REWRITE_TAC. It can also result in an invalid tactic.

## Uses

To advance or solve a goal when the current assumptions are expected to be useful in reducing the goal.

## See also

ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, SUBST_TAC.

# PURE_ASM_SIMP_TAC

```
PURE_ASM_SIMP_TAC : thm list -> tactic
```

## Synopsis

Perform simplification of goal by conditional contextual rewriting using assumptions.

## Description

A call to PURE_ASM_SIMP_TAC[theorems] will apply conditional contextual rewriting with theorems and the current assumptions of the goal to the goal's conclusion, but not the default simplfiications (see basic_rewrites and basic_convs). For more details on this kind of rewriting, see SIMP_CONV. If the extra generality of contextual conditional rewriting is not needed, REWRITE_TAC is usually more efficient.

## Failure

Never fails, but may loop indefinitely.

## See also

ASM_REWRITE_TAC, ASM_SIMP_TAC, SIMP_CONV, SIMP_TAC, REWRITE_TAC.

# PURE_ONCE_ASM_REWRITE_RULE

```
PURE_ONCE_ASM_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis
Rewrites a theorem once, including the theorem's assumptions as rewrites.

## Description
`PURE_ONCE_ASM_REWRITE_RULE` excludes the basic tautologies in `basic_rewrites` from the theorems used for rewriting. It searches for matching subterms once only, without recursing over already rewritten subterms. For a general introduction to rewriting tools see `GEN_REWRITE_RULE`.

## Failure
`PURE_ONCE_ASM_REWRITE_RULE` does not fail and does not diverge.

## See also
`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_ASM_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_ASM_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

# PURE_ONCE_ASM_REWRITE_TAC

```
PURE_ONCE_ASM_REWRITE_TAC : thm list -> tactic
```

## Synopsis
Rewrites a goal once, including the goal's assumptions as rewrites.

## Description
A set of rewrites generated from the assumptions of the goal and the supplied theorems is used to rewrite the term part of the goal, making only one pass over the goal. The basic tautologies are not included as rewrite theorems. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. See `GEN_REWRITE_TAC` for more information on rewriting tactics in general.

## Failure
`PURE_ONCE_ASM_REWRITE_TAC` does not fail and does not diverge.

## Uses
Manipulation of the goal by rewriting with its assumptions, in instances where rewriting with tautologies and recursive rewriting is undesirable.

## See also
`ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

# PURE_ONCE_REWRITE_CONV

```
PURE_ONCE_REWRITE_CONV : thm list -> conv
```

### Synopsis
Rewrites a term once with only the given list of rewrites.

### Description
PURE_ONCE_REWRITE_CONV generates rewrites from the list of theorems supplied by the user, without including the tautologies given in basic_rewrites. The applicable rewrites are employeded once, without entailing in a recursive search for matches over the term. See GEN_REWRITE_CONV for more details about rewriting strategies in HOL.

### Failure
This rule does not fail, and it does not diverge.

### See also
GEN_REWRITE_CONV, ONCE_DEPTH_CONV, ONCE_REWRITE_CONV, PURE_REWRITE_CONV, REWRITE_CONV.

## PURE_ONCE_REWRITE_RULE

```
PURE_ONCE_REWRITE_RULE : thm list -> thm -> thm
```

### Synopsis
Rewrites a theorem once with only the given list of rewrites.

### Description
PURE_ONCE_REWRITE_RULE generates rewrites from the list of theorems supplied by the user, without including the tautologies given in basic_rewrites. The applicable rewrites are employeded once, without entailing in a recursive search for matches over the theorem. See GEN_REWRITE_RULE for more details about rewriting strategies in HOL.

### Failure
This rule does not fail, and it does not diverge.

### See also
ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_DEPTH_CONV, ONCE_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

## PURE_ONCE_REWRITE_TAC

```
PURE_ONCE_REWRITE_TAC : thm list -> tactic
```

## Synopsis
Rewrites a goal using a supplied list of theorems, making one rewriting pass over the goal.

## Description
`PURE_ONCE_REWRITE_TAC` generates a set of rewrites from the given list of theorems, and applies them at every match found through searching once over the term part of the goal, without recursing. It does not include the basic tautologies as rewrite theorems. The order in which the rewrites are applied is unspecified. For more information on rewriting tactics see `GEN_REWRITE_TAC`.

## Failure
`PURE_ONCE_REWRITE_TAC` does not fail and does not diverge.

## Uses
This tactic is useful when the built-in tautologies are not required as rewrite equations and recursive rewriting is not desired.

## See also
`ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_ASM_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

# pure_prove_recursive_function_exists

```
pure_prove_recursive_function_exists : term -> thm
```

## Synopsis
Proves existence of general recursive function but leaves unproven assumptions.

## Description
The function `pure_prove_recursive_function_exists` should be applied to an existentially quantified term `` `?f. def_1[f] /\ ... /\ def_n[f]` ``, where each clause `def_i` is a universally quantified equation with an application of `f` to arguments on the left-hand side. The idea is that these clauses define the action of `f` on arguments of various kinds, for example on an empty list and nonempty list:

```
?f. (f [] = a) /\ (!h t. CONS h t = k[f,h,t])
```

or on even numbers and odd numbers:

```
?f. (!n. f(2 * n) = a[f,n]) /\ (!n. f(2 * n + 1) = b[f,n])
```

The returned value is a theorem whose conclusion matches the input term, with in general one or two assumptions stating what properties must hold so that the existence of such a function to be deduced. Roughly, one assumption states that the clauses are not mutually contradictory, as in

```
?f. (!n. f(n + 1) = 1) /\ (!n. f(n + 2) = 2)
```

and the other states that there is some wellfounded order making any recursion admissible. This rule attempts to eliminate any hypotheses of the first kind, but does not attempt to guess a wellfounded ordering as `prove_general_recursive_function_exists` does.

## Failure

Fails only if the definition is malformed. However it is possible that for an inadmissible definition the assumptions of the theorem may not hold.

## Example

In the definition of the Fibonacci numbers, the function successfully eliminates the mutual consistency hypotheses:

```
# pure_prove_recursive_function_exists
   `?fib. fib 0 = 1 /\ fib 1 = 1 /\
          !n. fib(n + 2) = fib(n) + fib(n + 1)`;;
val it : thm =
  ?(<<). WF (<<) /\ (!n. T ==> n << n + 2) /\ (!n. T ==> n + 1 << n + 2)
  |- ?fib. fib 0 = 1 /\ fib 1 = 1 /\ (!n. fib (n + 2) = fib n + fib (n + 1))
```

but leaves a wellfounded ordering to be given. (By contrast, `prove_general_recursive_function_exists` will automatically eliminate it.)

## Uses

Normally, use `prove_general_recursive_function_exists` for this operation. Use the present function only when the attempt by `prove_general_recursive_function_exists` to discharge the proof obligations is not successful and merely wastes time.

## See also

`define`, `instantiate_casewise_recursion`,
`prove_general_recursive_function_exists`.

# PURE_REWRITE_CONV

```
PURE_REWRITE_CONV : thm list -> conv
```

## Synopsis

Rewrites a term with only the given list of rewrites.

## Description

This conversion provides a method for rewriting a term with the theorems given, and excluding simplification with tautologies in `basic_rewrites`. Matching subterms are found recursively, until no more matches are found. For more details on rewriting see `GEN_REWRITE_CONV`.

## Uses

`PURE_REWRITE_CONV` is useful when the simplifications that arise by rewriting a theorem with `basic_rewrites` are not wanted.

## Failure

Does not fail. May result in divergence, in which case `PURE_ONCE_REWRITE_CONV` can be used.

## See also

`GEN_REWRITE_CONV, ONCE_REWRITE_CONV, PURE_ONCE_REWRITE_CONV, REWRITE_CONV.`

# PURE_REWRITE_RULE

```
PURE_REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis

Rewrites a theorem with only the given list of rewrites.

## Description

This rule provides a method for rewriting a theorem with the theorems given, and excluding simplification with tautologies in `basic_rewrites`. Matching subterms are found recursively starting from the term in the conclusion part of the theorem, until no more matches are found. For more details on rewriting see `GEN_REWRITE_RULE`.

## Uses

`PURE_REWRITE_RULE` is useful when the simplifications that arise by rewriting a theorem with `basic_rewrites` are not wanted.

## Failure

Does not fail. May result in divergence, in which case `PURE_ONCE_REWRITE_RULE` can be used.

## See also

`ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_REWRITE_RULE, PURE_ASM_REWRITE_RULE, PURE_ONCE_ASM_REWRITE_RULE, PURE_ONCE_REWRITE_RULE, REWRITE_RULE.`

# PURE_REWRITE_TAC

```
PURE_REWRITE_TAC : thm list -> tactic
```

## Synopsis
Rewrites a goal with only the given list of rewrites.

## Description
PURE_REWRITE_TAC behaves in the same way as REWRITE_TAC, but without the effects of the built-in tautologies. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. For more information on rewriting strategies see GEN_REWRITE_TAC.

## Failure
PURE_REWRITE_TAC does not fail, but it can diverge in certain situations; in such cases PURE_ONCE_REWRITE_TAC may be used.

## Uses
This tactic is useful when the built-in tautologies are not required as rewrite equations. It is sometimes useful in making more time-efficient replacements according to equations for which it is clear that no extra reduction via tautology will be needed. (The difference in efficiency is only apparent, however, in quite large examples.)

PURE_REWRITE_TAC advances goals but solves them less frequently than REWRITE_TAC; to be precise, PURE_REWRITE_TAC only solves goals which are rewritten to `T` (i.e. TRUTH) without recourse to any other tautologies.

## Example
It might be necessary, say for subsequent application of an induction hypothesis, to resist reducing a term `b = T` to `b`.

```
# g `b <=> T`;;
Warning: Free variables in goal: b
val it : goalstack = 1 subgoal (1 total)

`b <=> T`

# e(PURE_REWRITE_TAC[]);;
val it : goalstack = 1 subgoal (1 total)

`b <=> T`

# e(REWRITE_TAC[]);;
val it : goalstack = 1 subgoal (1 total)

`b`
```

## See also
ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, REWRITE_TAC, SUBST_TAC.

# PURE_SIMP_CONV

```
PURE_SIMP_CONV : thm list -> conv
```

## Synopsis
Simplify a term repeatedly by conditional contextual rewriting, not using default simplifications.

## Description
A call `SIMP_CONV thl tm` will return `|- tm = tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules. This is similar to `SIMP_CONV`, and the documentation for that contains more details. The `PURE` prefix means that the usual built-in simplifications (see `basic_rewrites` and `basic_convs`) are not applied.

## Failure
Never fails, but may return a reflexive theorem `|- tm = tm` if no simplifications can be made.

## See also
`PURE_REWRITE_CONV`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# PURE_SIMP_RULE

```
PURE_SIMP_RULE : thm list -> thm -> thm
```

## Synopsis
Simplify conclusion of a theorem repeatedly by conditional contextual rewriting, not using default simplifications.

## Description
A call `SIMP_CONV thl (|- tm)` will return `|- tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules. Howver, the `PURE` prefix indicates that it will not automatically include the usual built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of conditional rewriting, see `SIMP_CONV`.

## Failure
Never fails, but may return the input theorem unchanged if no simplifications were applicable.

## See also
`ONCE_SIMP_RULE`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# PURE_SIMP_TAC

```
PURE_SIMP_TAC : thm list -> tactic
```

## Synopsis
Simplify a goal repeatedly by conditional contextual rewriting without default simplifications.

## Description
When applied to a goal `A ?- g`, the tactic `PURE_SIMP_TAC thl` returns a new goal `A ?- g'` where `g'` results from applying the theorems in `thl` as (conditional) rewrite rules. The `PURE` prefix means that it does not apply the built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details, see `SIMP_CONV`.

## Failure
Never fails, though may not change the goal if no simplifications are applicable.

## Comments
To add the assumptions of the goal to the rewrites, use `PURE_ASM_SIMP_TAC` (or just `ASM PURE_SIMP_TAC`).

## See also
`ASM, ASM_SIMP_TAC, mk_rewrites, ONCE_SIMP_CONV, REWRITE_TAC, SIMP_CONV, SIMP_RULE.`

# qmap

```
qmap : ('a -> 'a) -> 'a list -> 'a list
```

## Synopsis
Maps a function of type `'a -> 'a` over a list, optimizing the unchanged case.

## Description
The call `qmap f [x1;...;xn]` returns the list `[f(x1);...;f(xn)]`. In this respect it behaves like `map`. However with `qmap`, the function `f` must have the same domain and codomain type, and in cases where the function returns the argument unchanged (actually pointer-equal, tested by '=='), the implementation often avoids rebuilding an equal copy of the list, so can be much more efficient.

## Failure
Fails if one of the embedded evaluations of `f` fails, but not otherwise.

## Example

Let us map the identity function over a million numbers:

```
# let million = 1--1000000;;
val million : int list =
  [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
   ...]
```

First we use ordinary `map`; the computation takes some time because the list is traversed and reconstructed, giving a fresh copy:

```
# time (map I) million == million;;
CPU time (user): 2.95
val it : bool = false
```

But `qmap` is markedly faster, uses no extra heap memory, and the result is pointer-equal to the input:

```
# time (qmap I) million == million;;
CPU time (user): 0.13
val it : bool = true
```

## Uses

Many logical operations, such as substitution, may in common cases return their arguments unchanged. In this case it is very useful to optimize the traversal in this way. Several internal logical manipulations like `vsubst` use this technique.

## Comments

## See also

`map`.

# quotexpander

```
quotexpander : string -> string
```

## Synopsis

Quotation expander.

## Description

This function determines how anything in `` `backquotes` `` is expanded on input.

## Failure

Never fails.

## Example

```
# quotexpander "1 + 1";;
val it : string = "parse_term \"1 + 1\""
# quotexpander ":num";;
val it : string = "parse_type \"num\""
```

## Comments

Not intended for general use, but automatically invoked when anything is typed in backquotes `` `like this` ``. May be of some interest for users wishing to change the behavior of the quotation parser.

# RAND_CONV

```
RAND_CONV : conv -> conv
```

## Synopsis

Applies a conversion to the operand of an application.

## Description

If `c` is a conversion that maps a term `` `t2` `` to the theorem `|- t2 = t2'`, then the conversion `RAND_CONV c` maps applications of the form `` `t1 t2` `` to theorems of the form:

```
|- (t1 t2) = (t1 t2')
```

That is, `` RAND_CONV c `t1 t2` `` applies `c` to the operand of the application `` `t1 t2` ``.

## Failure

`RAND_CONV c tm` fails if `tm` is not an application or if `tm` has the form `` `t1 t2` `` but the conversion `c` fails when applied to the term `t2`. The function returned by `RAND_CONV c` may also fail if the ML function `c` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

## Example

```
# RAND_CONV num_CONV `SUC 2`;;
val it : thm = |- SUC 2 = SUC (SUC 1)
```

## See also

`ABS_CONV`, `COMB_CONV`, `COMB_CONV2`, `LAND_CONV`, `RATOR_CONV`, `SUB_CONV`.

# rand

`rand : term -> term`

## Synopsis
Returns the operand from a combination (function application).

## Description
`rand `t1 t2`` returns ``t2``.

## Failure
Fails with `rand` if term is not a combination.

## Example

```
# rand `SUC 0`;;
val it : term = `0`
# rand `x + y`;;
val it : term = `y`
```

## See also
`rator, lhand, dest_comb.`

# ran

`ran : ('a, 'b) func -> 'b list`

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The `ran` operation returns the domain of such a function, i.e. the set of result values for the points on which it is defined.

## Failure
Attempts to `setify` the resulting list, so may fail if the range type does not admit comparisons.

## Example

```
# ran (1 |=> "1");;
val it : string list = ["1"]
# ran(itlist I [2|->4; 3|->6] undefined);;
val it : int list = [4; 6]
```

## See also
`|->, |=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefine, undefined.`

# rat_of_term

`rat_of_term : term -> num`

## Synopsis
Converts a canonical rational literal of type `:real` to an OCaml number.

## Description
The call `rat_of_term t` where term `t` is a canonical rational literal of type `:real` returns the corresponding OCaml rational number (type `num`). The canonical literals are integer literals `&n` for numeral `n`, `-- &n` for a nonzero numeral `n`, or ratios `&p / &q` or `-- &p / &q` where `p` is nonzero, `q > 1` and `p` and `q` share no common factor.

## Failure
Fails when applied to a term that is not a canonical rational literal.

## Example

```
# rat_of_term `-- &22 / &7`;;
val it : num = -22/7
```

## See also
`is_ratconst, mk_realintconst, REAL_RAT_REDUCE_CONV, term_of_rat.`

# RATOR_CONV

`RATOR_CONV : conv -> conv`

## Synopsis
Applies a conversion to the operator of an application.

## Description
If `c` is a conversion that maps a term `` `t1` `` to the theorem `|- t1 = t1'`, then the conversion `RATOR_CONV c` maps applications of the form `` `t1 t2` `` to theorems of the form:

```
|- (t1 t2) = (t1' t2)
```

That is, `` RATOR_CONV c `t1 t2` `` applies `c` to the operand of the application `` `t1 t2` ``.

## Failure
`RATOR_CONV c tm` fails if `tm` is not an application or if `tm` has the form `` `t1 t2` `` but the conversion `c` fails when applied to the term `t1`. The function returned by `RATOR_CONV c` may also fail if

the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

## Example

```
# RATOR_CONV BETA_CONV `(\x y. x + y) 1 2`;;
val it : thm = |- (\x y. x + y) 1 2 = (\y. 1 + y) 2
```

## See also

`ABS_CONV`, `COMB_CONV`, `COMB2_CONV`, `RAND_CONV`, `SUB_CONV`.

# rator

```
rator : term -> term
```

## Synopsis

Returns the operator from a combination (function application).

## Description

`rator(`t1 t2`)` returns `` `t1` ``.

## Failure

Fails with `rator` if term is not a combination.

## Example

```
# rator `f(x)`;;
Warning: inventing type variables
val it : term = `f`

# rator `~p`;;
val it : term = `(~)`

# rator `x + y`;;
val it : term = `(+) x`
```

## See also

`dest_comb`, `lhand`, `lhs`, `rand`.

# r

```
r : int -> goalstack
```

## Synopsis

Reorders the subgoals on top of the subgoal package goal stack.

## Description

The function `r` is part of the subgoal package. It 'rotates' the current list of goals by the given number, which may be positive or negative. For a description of the subgoal package, see `set_goal`.

## Failure

If there are no goals.

## Example

```
# g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3]) /\ (HD (TL[1;2;3]) = 2)`;;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3] /\ HD (TL [1; 2; 3]) = 2`

# e (REPEAT CONJ_TAC);;
val it : goalstack = 3 subgoals (3 total)

`HD (TL [1; 2; 3]) = 2`

`TL [1; 2; 3] = [2; 3]`

`HD [1; 2; 3] = 1`

# r 1;;
val it : goalstack = 1 subgoal (3 total)

`TL [1; 2; 3] = [2; 3]`

# r 1;;
val it : goalstack = 1 subgoal (3 total)

`HD (TL [1; 2; 3]) = 2`
```

## Uses

Proving subgoals in a different order from that generated by the subgoal package.

## See also

`b, e, g,p, set_goal, top_thm.`

# REAL_ARITH

`REAL_ARITH : term -> thm`

## Synopsis

Attempt to prove term using basic algebra and linear arithmetic over the reals.

## Description

`REAL_ARITH` is the basic tool for proving elementary lemmas about real equations and inequalities. Given a term, it first applies various normalizations, eliminating constructs such as `max`, `min` and `abs` by introducing case splits, splitting over the arms of conditionals and putting any equations and inequalities into a form `p(x) <><> 0` where `<><>` is an equality or inequality function and `p(x)` is in a normal form for polynomials as produced by `REAL_POLY_CONV`. The problem is split into the refutation of various conjunctions of such subformulas. A refutation of each is attempted using simple linear inequality reasoning (essentially Fourier-Motzkin elimination). Note that no non-trivial nonlinear inequality reasoning is performed (see below).

## Failure

Fails if the term is not provable using the algorithm sketched above.

## Example

Here is some simple inequality reasoning, showing how constructs like `abs`, `max` and `min` can be handled:

```
# REAL_ARITH
    `abs(x) < min e d / &2 /\ abs(y) < min e d / &2 ==> abs(x + y) < d + e`;;
val it : thm =
  |- abs x < min e d / &2 /\ abs y < min e d / &2 ==> abs (x + y) < d + e
```

The following example also involves inequality reasoning, but the initial algebraic normalization is critical to make the pieces match up:

```
# REAL_ARITH `(&1 + x) * (&1 - x) * (&1 + x pow 2) < &1 ==> &0 < x pow 4`;;
val it : thm = |- (&1 + x) * (&1 - x) * (&1 + x pow 2) < &1 ==> &0 < x pow 4
```

## Uses

Very convenient for providing elementary lemmas that would otherwise be painful to prove manually.

## Comments

For nonlinear equational reasoning, use `REAL_RING` or `REAL_FIELD`. For nonlinear inequality reasoning, there are no powerful rules built into HOL Light, but the additional derived rules defined in `Examples/sos.ml` and `Rqe/make.ml` may be useful.

## See also

`ARITH_TAC`, `INT_ARITH_TAC`, `REAL_ARITH_TAC`, `REAL_FIELD`, `REAL_RING`.

# REAL_ARITH_TAC

```
REAL_ARITH_TAC : tactic
```

## Synopsis
Attempt to prove goal using basic algebra and linear arithmetic over the reals.

## Description
The tactic REAL_ARITH_TAC is the tactic form of REAL_ARITH. Roughly speaking, it will automatically prove any formulas over the reals that are efectively universally quantified and can be proved valid by algebraic normalization and linear equational and inequality reasoning. See REAL_ARITH for more information about the algorithm used and its scope.

## Failure
Fails if the goal is not in the subset solvable by these means, or is not valid.

## Example
Here is a goal that holds by virtue of pure algebraic normalization:

```
# g `(x1 pow 2 + x2 pow 2 + x3 pow 2 + x4 pow 2) pow 2 =
     ((x1 + x2) pow 4 + (x1 + x3) pow 4 + (x1 + x4) pow 4 +
      (x2 + x3) pow 4 + (x2 + x4) pow 4 + (x3 + x4) pow 4 +
      (x1 - x2) pow 4 + (x1 - x3) pow 4 + (x1 - x4) pow 4 +
      (x2 - x3) pow 4 + (x2 - x4) pow 4 + (x3 - x4) pow 4) / &6`;;
```

and here is one that holds by linear inequality reasoning:

```
# g `&26 < x / &2 ==> abs(x / &4 + &1) < abs(x / &3)`;;
```

so either goal is solved simply by:

```
# e REAL_ARITH_TAC;;
val it : goalstack = No subgoals
```

## Comments
For nonlinear equational reasoning, use CONV_TAC REAL_RING or CONV_TAC REAL_FIELD. For nonlinear inequality reasoning, there are no powerful rules built into HOL Light, but the additional derived rules defined in Examples/sos.ml and Rqe/make.ml may be useful.

## See also
ARITH_TAC, INT_ARITH_TAC, REAL_ARITH, REAL_FIELD, REAL_RING.

# REAL_FIELD

```
REAL_FIELD : term -> thm
```

## Synopsis
Prove basic ‘field’ facts over the reals.

## Description

Most of the built-in HOL arithmetic decision procedures have limited ability to deal with inversion or division. `REAL_FIELD` is an enhancement of `REAL_RING` that has the same underlying method but first performs various case-splits, reducing a goal involving the inverse `inv(t)` of a term `t` to the cases where `t = 0` where `t * inv(t) = &1`, repeatedly for all such `t`. After subsequently splitting the goal into normal form, `REAL_RING` (for algebraic reasoning) is applied; if this fails then `REAL_ARITH` is also tried, since this allows some `t = 0` cases to be excluded by simple linear reasoning.

## Failure

Fails if the term is not provable using the methods described.

## Example

Here we do some simple algebraic simplification, ruling out the degenerate `x = &0` case using the inequality in the antecedent.

```
# REAL_FIELD `!x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))`;;
...
val it : thm = |- !x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))
```

## Comments

Except for the discharge of conditions using linear reasoning, this rule is essentially equational. For nonlinear inequality reasoning, there are no powerful rules built into HOL Light, but the additional derived rules defined in `Examples/sos.ml` and `Rqe/make.ml` may be useful.

## See also

`ARITH_TAC`, `INT_ARITH_TAC`, `REAL_ARITH`, `REAL_ARITH_TAC`, `REAL_RING`.

# real_ideal_cofactors

```
real_ideal_cofactors : term list -> term -> term list
```

## Synopsis

Produces cofactors proving that one real polynomial is in the ideal generated by others.

## Description

The call `` real_ideal_cofactors [`p1`; ...; `pn`] `p` ``, where all the terms have type `:real` and can be considered as polynomials, will test whether `p` is in the ideal generated by the `p1,...,pn`. If so, it will return a corresponding list `` [`q1`; ...; `qn`] `` of 'cofactors' such that the following is an algebraic identity (provable by `REAL_RING` or a slight elaboration of `REAL_POLY_CONV`, for example):

```
p = p1 * q1 + ... + pn * qn
```

hence providing an explicit certificate for the ideal membership. If ideal membership does not hold, `real_ideal_cofactors` fails. The test is performed using a Gröbner basis procedure.

## Failure

Fails if the terms are ill-typed, or if ideal membership fails.

## Example

Here is a fairly simple example:

```
# prioritize_real();;
val it : unit = ()

# real_ideal_cofactors
   [`y1 * y3 + x1 * x3`;
    `y3 * (y2 - y3) + (x2 - x3) * x3`]
   `x3 * y3 * (y1 * (x2 - x3) - x1 * (y2 - y3))`;;
...
val it : term list = [`&1 * y3 pow 2 + -- &1 * y2 * y3`; `&1 * y1 * y3`]
```

and we can confirm the identity as follows (note that REAL_IDEAL_CONV already does this directly):

```
# REAL_RING `(&1 * y3 pow 2 + -- &1 * y2 * y3) * (y1 * y3 + x1 * x3) +
             (&1 * y1 * y3) * (y3 * (y2 - y3) + (x2 - x3) * x3) =
             x3 * y3 * (y1 * (x2 - x3) - x1 * (y2 - y3))`;;
```

## Comments

When we say that terms can be 'considered as polynomials', we mean that initial normalization, essentially in the style of REAL_POLY_CONV, will be applied, but some complex constructs such as conditional expressions will be treated as atomic.

## See also

ideal_cofactors, int_ideal_cofactors, REAL_IDEAL_CONV, REAL_RING, RING, RING_AND_IDEAL_CONV.

# REAL_IDEAL_CONV

```
REAL_IDEAL_CONV : term list -> term -> thm
```

## Synopsis

Produces identity proving ideal membership over the reals.

## Description

The call REAL_IDEAL_CONV [`p1`; ...; `pn`] `p`, where all the terms have type :real and can be considered as polynomials, will test whether p is in the ideal generated by the p1,...,pn. If so, it will return a corresponding theorem |- p = q1 * p1 + ... + qn * pn showing how to express p in terms of the other polynomials via some 'cofactors' qi.

## Failure
Fails if the terms are ill-typed, or if ideal membership fails.

## Example
In the case of a singleton list, this just corresponds to dividing one multivariate polynomial by another, e.g.

```
# REAL_IDEAL_CONV [`x - &1`] `x pow 4 - &1`;;
1 basis elements and 0 critical pairs
val it : thm =
  |- x pow 4 - &1 = (&1 * x pow 3 + &1 * x pow 2 + &1 * x + &1) * (x - &1)
```

## See also
ideal_cofactors, real_ideal_cofactors, REAL_RING, RING, RING_AND_IDEAL_CONV.

# REAL_INT_ABS_CONV

```
REAL_INT_ABS_CONV : conv
```

## Synopsis
Conversion to produce absolute value of an integer literal of type :real.

## Description
The call REAL_INT_ABS_CONV `abs c`, where c is an integer literal of type :real, returns the theorem |- abs c = d where d is the canonical integer literal that is equal to c's absolute value. The literal c may be of the form &n or -- &n (with nonzero n in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not the negation of one of the permitted forms of integer literal of type :real.

## Example

```
# REAL_INT_ABS_CONV `abs(-- &42)`;;
val it : thm = |- abs (-- &42) = &42
```

## Comments
The related function REAL_RAT_ABS_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
INT_ABS_CONV, REAL_RAT_ABS_CONV, REAL_INT_REDUCE_CONV.

# REAL_INT_ADD_CONV

```
REAL_INT_ADD_CONV : conv
```

## Synopsis
Conversion to perform addition on two integer literals of type :real.

## Description
The call REAL_INT_ADD_CONV ‘c1 + c2‘ where c1 and c2 are integer literals of type :real, returns |- c1 + c2 = d where d is the canonical integer literal that is equal to c1 + c2. The literals c1 and c2 may be of the form &n or -- &n (with nonzero n in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not the sum of two permitted integer literals of type :real.

## Example

```
# REAL_INT_ADD_CONV ‘-- &17 + &25‘;;
val it : thm = |- -- &17 + &25 = &8
```

## Comments
The related function REAL_RAT_ADD_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
INT_ADD_CONV, REAL_RAT_ADD_CONV, REAL_INT_REDUCE_CONV.

# REAL_INT_EQ_CONV

```
REAL_INT_EQ_CONV : conv
```

## Synopsis
Conversion to prove whether one integer literal of type :real is equal to another.

## Description

The call `REAL_INT_EQ_CONV ‘c1 < c2‘` where `c1` and `c2` are integer litersls of type `:real`, returns whichever of `|- c1 = c2 <=> T` or `|- c1 = c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

## Failure

Fails if applied to a term that is not an equality comparison on two permitted integer literals of type `:real`.

## Example

```
# REAL_INT_EQ_CONV ‘&1 = &2‘;;
val it : thm = |- &1 = &2 <=> F

# REAL_INT_EQ_CONV ‘-- &1 = -- &1‘;;
val it : thm = |- -- &1 = -- &1 <=> T
```

## Comments

The related function `REAL_RAT_EQ_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

`INT_EQ_CONV`, `REAL_RAT_EQ_CONV`, `REAL_RAT_REDUCE_CONV`.

# REAL_INT_GE_CONV

```
REAL_INT_GE_CONV : conv
```

## Synopsis

Conversion to prove whether one integer literal of type `:real` is `>=` another.

## Description

The call `REAL_INT_GE_CONV ‘c1 >= c2‘` where `c1` and `c2` are integer litersls of type `:real`, returns whichever of `|- c1 >= c2 <=> T` or `|- c1 >= c2 <=> F` is true. By an integer literal we mean either `&n` or `-- &n` where `n` is a numeral.

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type `:real`.

## Example

```
# REAL_INT_GE_CONV `&7 >= &6`;;
val it : thm = |- &7 >= &6 <=> T
```

## Comments

The related function REAL_RAT_GE_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

INT_GE_CONV, REAL_RAT_GE_CONV, REAL_RAT_REDUCE_CONV.

# REAL_INT_GT_CONV

REAL_INT_GT_CONV : conv

## Synopsis

Conversion to prove whether one integer literal of type :real is < another.

## Description

The call REAL_INT_GT_CONV `c1 > c2` where c1 and c2 are integer litersls of type :real, returns whichever of |- c1 > c2 <=> T or |- c1 > c2 <=> F is true. By an integer literal we mean either &n or -- &n where n is a numeral.

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type :real.

## Example

```
# REAL_INT_GT_CONV `&1 > &2`;;
val it : thm = |- &1 > &2 <=> F
```

## Comments

The related function REAL_RAT_GT_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

INT_GT_CONV, REAL_RAT_GT_CONV, REAL_RAT_REDUCE_CONV.

# REAL_INT_LE_CONV

REAL_INT_LE_CONV : conv

## Synopsis
Conversion to prove whether one integer literal of type :real is <= another.

## Description
The call REAL_INT_LE_CONV ‘c1 <= c2‘ where c1 and c2 are integer litersls of type :real, returns whichever of |- c1 <= c2 <=> T or |- c1 <= c2 <=> F is true. By an integer literal we mean either &n or -- &n where n is a numeral.

## Failure
Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type :real.

## Example

```
# REAL_INT_LE_CONV ‘&11 <= &77‘;;
val it : thm = |- &11 <= &77 <=> T
```

## Comments
The related function REAL_RAT_LE_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
INT_LE_CONV, REAL_RAT_LE_CONV, REAL_RAT_REDUCE_CONV.

# REAL_INT_LT_CONV

REAL_INT_LT_CONV : conv

## Synopsis
Conversion to prove whether one integer literal of type :real is < another.

## Description
The call REAL_INT_LT_CONV ‘c1 < c2‘ where c1 and c2 are integer litersls of type :real, returns whichever of |- c1 < c2 <=> T or |- c1 < c2 <=> F is true. By an integer literal we mean either &n or -- &n where n is a numeral.

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted integer literals of type `:real`.

## Example

```
# REAL_INT_LT_CONV `-- &18 < &64`;;
val it : thm = |- -- &18 < &64 <=> T
```

## Comments

The related function `REAL_RAT_LT_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

`INT_LT_CONV`, `REAL_RAT_LT_CONV`, `REAL_RAT_REDUCE_CONV`.

# REAL_INT_MUL_CONV

```
REAL_INT_MUL_CONV : conv
```

## Synopsis

Conversion to perform multiplication on two integer literals of type `:real`.

## Description

The call `REAL_INT_MUL_CONV `c1 * c2`` where `c1` and `c2` are integer literals of type `:real`, returns `|- c1 * c2 = d` where `d` is the canonical integer literal that is equal to `c1 * c2`. The literals `c1` and `c2` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not the product of two permitted integer literals of type `:real`.

## Example

```
# REAL_INT_MUL_CONV `&6 * -- &9`;;
val it : thm = |- &6 * -- &9 = -- &54
```

## Comments

The related function `REAL_RAT_MUL_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
INT_MUL_CONV, REAL_RAT_MUL_CONV, REAL_INT_REDUCE_CONV.

# REAL_INT_NEG_CONV

REAL_INT_NEG_CONV : conv

## Synopsis
Conversion to negate an integer literal of type :real.

## Description
The call REAL_INT_NEG_CONV `--c`, where c is an integer literal of type :real, returns the theorem |- --c = d where d is the canonical integer literal that is equal to c's negation. The literal c may be of the form &n or -- &n (with nonzero n in the latter case) and the result will be of the same form.

## Failure
Fails if applied to a term that is not the negation of one of the permitted forms of integer literal of type :real.

## Example

```
# REAL_INT_NEG_CONV `-- (-- &3 / &2)`;;
val it : thm = |- --(-- &3 / &2) = &3 / &2
```

## Comments
The related function REAL_RAT_NEG_CONV subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also
INT_NEG_CONV, REAL_RAT_NEG_CONV, REAL_INT_REDUCE_CONV.

# REAL_INT_POW_CONV

REAL_INT_POW_CONV : conv

## Synopsis
Conversion to perform exponentiation on a integer literal of type :real.

## Description

The call `REAL_INT_POW_CONV ‘c pow n‘` where `c` is an integer literal of type `:real` and `n` is a numeral of type `:num`, returns `|- c pow n = d` where `d` is the canonical integer literal that is equal to `c` raised to the `n`th power. The literal `c` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not a permitted integer literal of type `:real` raised to a numeral power.

## Example

```
# REAL_INT_POW_CONV ‘(-- &2) pow 77‘;;
val it : thm = |- -- &2 pow 77 = -- &151115727451828646838272
```

## Comments

The related function `REAL_RAT_POW_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

`INT_POW_CONV`, `REAL_INT_POW_CONV`, `REAL_INT_REDUCE_CONV`.

# REAL_INT_RAT_CONV

```
REAL_INT_RAT_CONV : conv
```

## Synopsis

Convert basic rational constant of real type to canonical form.

## Description

When applied to a term that is a rational constant of type `:real`, `REAL_INT_RAT_CONV` converts it to an explicit ratio `&p / &q` or `-- &p / &q`; `q` is always there, even if it is `1`.

## Failure

Never fails; simply has no effect if it is not applied to a suitable constant.

## Example

```
# REAL_INT_RAT_CONV `&22 / &7`;;
val it : thm = |- &22 / &7 = &22 / &7

# REAL_INT_RAT_CONV `&42`;;
val it : thm = |- &42 = &42 / &1

# REAL_INT_RAT_CONV `#3.1415926`;;
val it : thm = |- #3.1415926 = &31415926 / &10000000
```

## Uses

Mainly for internal use as a preprocessing step in rational-number calculations.

## See also

REAL_RAT_REDUCE_CONV.

# REAL_INT_RED_CONV

```
REAL_INT_RED_CONV : term -> thm
```

## Synopsis

Performs one arithmetic or relational operation on integer literals of type `:real`.

## Description

When applied to any of the terms `` `--c` ``, `` `abs c` ``, `` `c1 + c2` ``, `` `c1 - c2` ``, `` `c1 * c2` ``, `` `c pow n` ``, `` `c1 <= c2` ``, `` `c1 < c2` ``, `` `c1 >= c2` ``, `` `c1 > c2` ``, `` `c1 = c2` ``, where `c`, `c1` and `c2` are integer literals of type `:real` and `n` is a numeral of type `:num`, `REAL_INT_RED_CONV` returns a theorem asserting the equivalence of the term to a canonical integer (for the arithmetic operators) or a truth-value (for the relational operators). The integer literals are terms of the form `&n` or `-- &n` (with nonzero `n` in the latter case).

## Failure

Fails if applied to an inappropriate term.

## Uses

More convenient for most purposes is `REAL_INT_REDUCE_CONV`, which applies these evaluation conversions recursively at depth, or still more generally `REAL_RAT_REDUCE_CONV` which applies to any rational numbers, not just integers. Still, access to this 'one-step' reduction can be handy if you want to add a conversion conv for some other operator on real number literals, which you can conveniently incorporate it into REAL_INT_REDUCE_CONV with

```
# let REAL_INT_REDUCE_CONV' =
    DEPTH_CONV(REAL_INT_RED_CONV ORELSEC conv);;
```

## See also

INT_RED_CONV, REAL_INT_REDUCE_CONV, REAL_RAT_RED_CONV.

# REAL_INT_REDUCE_CONV

```
REAL_INT_REDUCE_CONV : conv
```

## Synopsis

Evaluate subexpressions built up from integer literals of type :real, by proof.

## Description

When applied to a term, REAL_INT_REDUCE_CONV performs a recursive bottom-up evaluation by proof of subterms built from integer literals of type :real using the unary operators '--', 'inv' and 'abs', and the binary arithmetic ('+', '-', '*', '/', 'pow') and relational ('<', '<=', '>', '>=', '=') operators, as well as propagating literals through logical operations, e.g. T /\ x <=> x, returning a theorem that the original and reduced terms are equal. The permissible integer literals are of the form &n or -- &n for numeral n, nonzero in the negative case.

## Failure

Never fails, but may have no effect.

## Example

```
# REAL_INT_REDUCE_CONV
   `if &5 pow 4 < &4 pow 5 then (&2 pow 3 - &1) pow 2 + &1 else &99`;;
val it : thm =
  |- (if &5 pow 4 < &4 pow 5 then (&2 pow 3 - &1) pow 2 + &1 else &99) = &50
```

## Comments

The corresponding INT_REDUCE_CONV works for the type of integers. The more general function REAL_RAT_REDUCE_CONV works similarly over :real but for arbitrary rational literals.

## See also

NUM_REDUCE_CONV, INT_REDUCE_CONV, REAL_RAT_REDUCE_CONV.

# REAL_INT_SUB_CONV

`REAL_INT_SUB_CONV : conv`

## Synopsis

Conversion to perform subtraction on two integer literals of type `:real`.

## Description

The call `REAL_INT_SUB_CONV ‘c1 - c2‘` where `c1` and `c2` are integer literals of type `:real`, returns `|- c1 - c2 = d` where `d` is the canonical integer literal that is equal to `c1 - c2`. The literals `c1` and `c2` may be of the form `&n` or `-- &n` (with nonzero `n` in the latter case) and the result will be of the same form.

## Failure

Fails if applied to a term that is not the difference of two permitted integer literals of type `:real`.

## Example

```
# REAL_INT_SUB_CONV ‘&33 - &77‘;;
val it : thm = |- &33 - &77 = -- &44
```

## Comments

The related function `REAL_RAT_SUB_CONV` subsumes this functionality, also applying to rational literals. Unless the restriction to integers is desired or a tiny efficiency difference matters, it should be used in preference.

## See also

`INT_SUB_CONV, REAL_RAT_SUB_CONV, REAL_INT_REDUCE_CONV.`

# REAL_LE_IMP

`REAL_LE_IMP : thm -> thm`

## Synopsis

Perform transitivity chaining for non-strict real number inequality.

## Description

When applied to a theorem `A |- s <= t` where `s` and `t` have type `real`, the rule `REAL_LE_IMP` returns `A |- !x1...xn z. t <= z ==> s <= z`, where `z` is some variable and the `x1,...,xn` are free variables in `s` and `t`.

## Failure

Fails if applied to a theorem whose conclusion is not of the form ‘s <= t‘ for some real number terms s and t.

## Example

```
# REAL_LE_IMP (REAL_ARITH ‘x:real <= abs(x)‘);;
val it : thm = |- !x z. abs x <= z ==> x <= z
```

## Uses

Can make transitivity chaining in goals easier, e.g. by FIRST_ASSUM(MATCH_MP_TAC o REAL_LE_IMP).

## See also

LE_IMP, REAL_ARITH, REAL_LET_IMP.

# REAL_LET_IMP

```
REAL_LET_IMP : thm -> thm
```

## Synopsis

Perform transitivity chaining for mixed strict/non-strict real number inequality.

## Description

When applied to a theorem A |- s <= t where s and t have type real, the rule REAL_LE_IMP returns A |- !x1...xn z. t < z ==> s < z, where z is some variable and the x1,...,xn are free variables in s and t.

## Failure

Fails if applied to a theorem whose conclusion is not of the form ‘s <= t‘ for some real number terms s and t.

## Example

```
# REAL_LET_IMP (REAL_ARITH ‘abs(x + y) <= abs(x) + abs(y)‘);;
val it : thm = |- !x y z. abs x + abs y < z ==> abs (x + y) < z
```

## Uses

Can make transitivity chaining in goals easier, e.g. by FIRST_ASSUM(MATCH_MP_TAC o REAL_LE_IMP).

## See also

LE_IMP, REAL_ARITH, REAL_LE_IMP.

# REAL_LINEAR_PROVER

```
REAL_LINEAR_PROVER : (thm list * thm list * thm list -> positivstellensatz -> thm) -> th
```

## Synopsis

Refute real equations and inequations by linear reasoning (not intended for general use).

## Description

The `REAL_LINEAR_PROVER` function should be given two arguments. The first is a proof translator that constructs a contradiction from a tuple of three theorem lists using a Positivstellensatz refutation, which is essentially a representation of how to add and multiply equalities or inequalities chosen from the list to reach a trivially false equation or inequality such as `&0 > &0`. The second argument is a triple of theorem lists, respectively a list of equations of the form `A_i |- p_i = &0`, a list of non-strict inequalities of the form `B_j |- q_i >= &0`, and a list of strict inequalities of the form `C_k |- r_k > &0`, with both sides being real in each case. Any theorems not of that form will not lead to an error, but will be ignored and cannot contribute to the proof. The prover attempts to construct a Positivstellensatz refutation by normalization as in `REAL_POLY_CONV` then linear inequality reasoning, and if successful will apply the translator function to this refutation to obtain `D |- F` where all assumptions `D` occurs among the `A_i`, `B_j` or `C_k`. Otherwise, or if the translator itself fails, the call fails.

## Failure

Fails if there is no refutation using linear reasoning or if an improper form inhibits proof for other reasons, or if the translator fails.

## Uses

This is not intended for general use. The core real inequality reasoner `REAL_ARITH` is implemented by:

```
# let REAL_ARITH = GEN_REAL_ARITH REAL_LINEAR_PROVER;;
```

In this way, all specifically linear functionality is isolated in `REAL_LINEAR_PROVER`, and the rest of the infrastructure of Positivstellensatz proof translation and initial normalization (including elimination of `abs`, `max`, `min`, conditional expressions etc.) is available for use with more advanced nonlinear provers. Such a prover, based on semidefinite programming and requiring support of an external SDP solver to find Positivstellensatz refutations, can be found in `Examples/sos.ml`.

## See also

`GEN_REAL_ARITH`, `REAL_ARITH`, `REAL_POLY_CONV`.

# REAL_POLY_ADD_CONV

```
REAL_POLY_ADD_CONV : term -> thm
```

## Synopsis
Adds two real polynomials while retaining canonical form.

## Description
For many purposes it is useful to retain polynomials in a canonical form. For more information on the usual normal form in HOL Light, see the function REAL_POLY_CONV, which converts a polynomial to normal form while proving the equivalence of the original and normalized forms. The function REAL_POLY_ADD_CONV is a more delicate conversion that, given a term p1 + p2 where p1 and p2 are real polynomials in normal form, returns a theorem |- p1 + p2 = p where p is in normal form.

## Failure
Fails if applied to a term that is not the sum of two real terms. If these subterms are not polynomials in normal form, the overall normalization is not guaranteed.

## Example

```
# REAL_POLY_ADD_CONV `(x pow 2 + x) + (x pow 2 + -- &1 * x + &1)`;;
val it : thm =
  |- (x pow 2 + x) + x pow 2 + -- &1 * x + &1 = &2 * x pow 2 + &1
```

## Uses
More delicate polynomial operations that simply the direct normalization with REAL_POLY_CONV.

## See also
REAL_ARITH, REAL_POLY_CONV, REAL_POLY_MUL_CONV, REAL_POLY_NEG_CONV, REAL_POLY_POW_CONV, REAL_POLY_SUB_CONV, REAL_RING.

# REAL_POLY_CONV

```
REAL_POLY_CONV : term -> thm
```

## Synopsis
Converts a real polynomial into canonical form.

## Description
Given a term of type :real that is built up using addition, subtraction, negation, multiplication, and inversion and division of constants, REAL_POLY_CONV converts it into a canonical polynomial form and returns a theorem asserting the equivalence of the original and canonical terms. The basic elements need not simply be variables or constants; anything not built up using the operators given above will be considered 'atomic' for the purposes of this conversion, for example inv(x} where {\small\verbxpolynomial form is a 'multiplied out' sum of products, with the monomials (product terms) ordered according to the canonical OCaml order on terms. In particular, it is just &0 if the polynomial is identically zero.

## Failure

Never fails, even if the term has the wrong type; in this case it merely returns a reflexive theorem.

## Example

This illustrates how terms are 'multiplied out':

```
# REAL_POLY_CONV
   `(x + &1) * (x pow 2 + &1) * (x pow 4 + &1)`;;
val it : thm =
  |- (x + &1) * (x pow 2 + &1) * (x pow 4 + &1) =
     x pow 7 + x pow 6 + x pow 5 + x pow 4 + x pow 3 + x pow 2 + x + &1
```

and the following is an example of how a complicated algebraic identity (due to Liouville?) simplifies to zero. Note that division is permissible because it is only by constants.

```
# REAL_POLY_CONV
   `((x1 + x2) pow 4 + (x1 + x3) pow 4 + (x1 + x4) pow 4 +
     (x2 + x3) pow 4 + (x2 + x4) pow 4 + (x3 + x4) pow 4) / &6 +
    ((x1 - x2) pow 4 + (x1 - x3) pow 4 + (x1 - x4) pow 4 +
     (x2 - x3) pow 4 + (x2 - x4) pow 4 + (x3 - x4) pow 4) / &6 -
    (x1 pow 2 + x2 pow 2 + x3 pow 2 + x4 pow 2) pow 2`;;
val it : thm =
  |- ((x1 + x2) pow 4 +
      (x1 + x3) pow 4 +
      (x1 + x4) pow 4 +
      (x2 + x3) pow 4 +
      (x2 + x4) pow 4 +
      (x3 + x4) pow 4) /
     &6 +
     ((x1 - x2) pow 4 +
      (x1 - x3) pow 4 +
      (x1 - x4) pow 4 +
      (x2 - x3) pow 4 +
      (x2 - x4) pow 4 +
      (x3 - x4) pow 4) /
     &6 -
     (x1 pow 2 + x2 pow 2 + x3 pow 2 + x4 pow 2) pow 2 =
     &0
```

## Uses

Keeping terms in normal form. For simply proving equalities, `REAL_RING` is more powerful and usually more convenient.

## See also

`INT_POLY_CONV`, `REAL_ARITH`, `REAL_RING`, `SEMIRING_NORMALIZERS_CONV`.

# REAL_POLY_MUL_CONV

```
REAL_POLY_MUL_CONV : term -> thm
```

## Synopsis
Multiplies two real polynomials while retaining canonical form.

## Description
For many purposes it is useful to retain polynomials in a canonical form. For more information on the usual normal form in HOL Light, see the function `REAL_POLY_CONV`, which converts a polynomial to normal form while proving the equivalence of the original and normalized forms. The function `REAL_POLY_MUL_CONV` is a more delicate conversion that, given a term `p1 * p2` where `p1` and `p2` are real polynomials in normal form, returns a theorem `|- p1 * p2 = p` where `p` is in normal form.

## Failure
Fails if applied to a term that is not the product of two real terms. If these subterms are not polynomials in normal form, the overall normalization is not guaranteed.

## Example

```
# REAL_POLY_MUL_CONV `(x pow 2 + x) * (x pow 2 + -- &1 * x + &1)`;;
val it : thm = |- (x pow 2 + x) * (x pow 2 + -- &1 * x + &1) = x pow 4 + x
```

## Uses
More delicate polynomial operations that simply the direct normalization with `REAL_POLY_CONV`.

## See also
`REAL_ARITH`, `REAL_POLY_ADD_CONV`, `REAL_POLY_CONV`, `REAL_POLY_NEG_CONV`, `REAL_POLY_POW_CONV`, `REAL_POLY_SUB_CONV`, `REAL_RING`.

# REAL_POLY_NEG_CONV

```
REAL_POLY_NEG_CONV : term -> thm
```

## Synopsis
Negates real polynomial while retaining canonical form.

## Description
For many purposes it is useful to retain polynomials in a canonical form. For more information on the usual normal form in HOL Light, see the function `REAL_POLY_CONV`, which converts a

polynomial to normal form while proving the equivalence of the original and normalized forms. The function `REAL_POLY_NEG_CONV` is a more delicate conversion that, given a term `--p` where `p` is a real polynomial in normal form, returns a theorem `|- --p = p'` where `p'` is in normal form.

## Failure

Fails if applied to a term that is not the negation of a real term. If negation is applied to a polynomial in non-normal form, the overall normalization is not guaranteed.

## Example

```
# REAL_POLY_NEG_CONV `--(x pow 2 + -- &1)`;;
val it : thm = |- --(x pow 2 + -- &1) = -- &1 * x pow 2 + &1
```

## Uses

More delicate polynomial operations than simply the direct normalization with `REAL_POLY_CONV`.

## See also

`REAL_ARITH`, `REAL_POLY_ADD_CONV`, `REAL_POLY_CONV`, `REAL_POLY_MUL_CONV`, `REAL_POLY_POW_CONV`, `REAL_POLY_SUB_CONV`, `REAL_RING`.

# REAL_POLY_POW_CONV

```
REAL_POLY_POW_CONV : term -> thm
```

## Synopsis

Raise real polynomial to numeral power while retaining canonical form.

## Description

For many purposes it is useful to retain polynomials in a canonical form. For more information on the usual normal form in HOL Light, see the function `REAL_POLY_CONV`, which converts a polynomial to normal form while proving the equivalence of the original and normalized forms. The function `REAL_POLY_POW_CONV` is a more delicate conversion that, given a term `p1 pow n` where `p` is a real polynomial in normal form and `n` a numeral, returns a theorem `|- p pow n = p'` where `p'` is in normal form.

## Failure

Fails if applied to a term that is not a real term raised to a numeral power. If the subterm is not a polynomial in normal form, the overall normalization is not guaranteed.

## Example

```
# REAL_POLY_POW_CONV `(x + &1) pow 4`;;
val it : thm =
  |- (x + &1) pow 4 = x pow 4 + &4 * x pow 3 + &6 * x pow 2 + &4 * x + &1
```

## Uses

More delicate polynomial operations that simply the direct normalization with REAL_POLY_CONV.

## See also

REAL_ARITH, REAL_POLY_ADD_CONV, REAL_POLY_CONV, REAL_POLY_MUL_CONV, REAL_POLY_NEG_CONV, REAL_POLY_SUB_CONV, REAL_RING.

# REAL_POLY_SUB_CONV

```
REAL_POLY_SUB_CONV : term -> thm
```

## Synopsis

Subtracts two real polynomials while retaining canonical form.

## Description

For many purposes it is useful to retain polynomials in a canonical form. For more information on the usual normal form in HOL Light, see the function REAL_POLY_CONV, which converts a polynomial to normal form while proving the equivalence of the original and normalized forms. The function REAL_POLY_SUB_CONV is a more delicate conversion that, given a term `p1 - p2` where `p1` and `p2` are real polynomials in normal form, returns a theorem `|- p1 - p2 = p` where `p` is in normal form.

## Failure

Fails if applied to a term that is not the difference of two real terms. If these subterms are not polynomials in normal form, the overall normalization is not guaranteed.

## Example

```
# REAL_POLY_SUB_CONV `(x pow 2 + x) - (x pow 2 + -- &1 * x + &1)`;;
val it : thm = |- (x pow 2 + x) - (x pow 2 + -- &1 * x + &1) = &2 * x + -- &1
```

## Uses

More delicate polynomial operations that simply the direct normalization with REAL_POLY_CONV.

## See also

REAL_ARITH, REAL_POLY_SUB_CONV, REAL_POLY_CONV, REAL_POLY_MUL_CONV, REAL_POLY_NEG_CONV, REAL_POLY_POW_CONV, REAL_RING.

# REAL_RAT_ABS_CONV

REAL_RAT_ABS_CONV : term -> thm

## Synopsis

Conversion to produce absolute value of a rational literal of type :real.

## Description

The call REAL_RAT_ABS_CONV ‘abs c‘, where c is a rational literal of type :real, returns the theorem |- abs c = d where d is the canonical rational literal that is equal to c's absolute value. The literal c may be an integer literal (&n or -- &n), a ratio (&p / &q or -- &p / &q), or a decimal (#DDD.DDDD or #DDD.DDDDeNN). The returned value d is always put in the form &p / &q or -- &p / &q with q > 1 and p and q sharing no common factor, or simply &p or -- &p when that is impossible.

## Failure

Fails if applied to a term that is not the absolute value function applied to one of the permitted forms of rational literal of type :real.

## Example

```
# REAL_RAT_ABS_CONV ‘abs(-- &3 / &2)‘;;
val it : thm = |- abs (-- &3 / &2) = &3 / &2
```

## See also

REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_ADD_CONV

REAL_RAT_ADD_CONV : conv

## Synopsis

Conversion to perform addition on two rational literals of type :real.

## Description

The call REAL_RAT_ADD_CONV ‘c1 + c2‘ where c1 and c2 are rational literals of type :real, returns |- c1 + c2 = d where d is the canonical rational literal that is equal to c1 + c2. The literals c1 and c2 may be integer literals (&n or -- &n), ratios (&p / &q or -- &p / &q), or decimals (#DDD.DDDD or #DDD.DDDDeNN). The result d is always put in the form &p / &q or -- &p / &q with q > 1 and p and q sharing no common factor, or simply &p or -- &p when that is impossible.

## Failure
Fails if applied to a term that is not the sum of two permitted rational literals of type :real.

## Example

```
# REAL_RAT_ADD_CONV `-- &11 / &12 + #0.09`;;
val it : thm = |- -- &11 / &12 + #0.09 = -- &62 / &75
```

## See also
REAL_RAT_ABS_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_DIV_CONV

REAL_RAT_DIV_CONV : conv

## Synopsis
Conversion to perform division on two rational literals of type :real.

## Description
The call REAL_RAT_DIV_CONV `c1 / c2` where c1 and c2 are rational literals of type :real, returns |- c1 / c2 = d where d is the canonical rational literal that is equal to c1 / c2. The literals c1 and c2 may be integer literals (&n or -- &n), ratios (&p / &q or -- &p / &q), or decimals (#DDD.DDDD or #DDD.DDDDeNN). The result d is always put in the form &p / &q or -- &p / &q with q > 1 and p and q sharing no common factor, or simply &p or -- &p when that is impossible.

## Failure
Fails if applied to a term that is not the quotient of two permitted rational literals of type :real, or if the divisor is zero.

## Example

```
# REAL_RAT_DIV_CONV `&2000 / (-- &40 / &12)`;;
val it : thm = |- &2000 / (-- &40 / &12) = -- &600
```

## See also
REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV,

REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_EQ_CONV

```
REAL_RAT_EQ_CONV : conv
```

## Synopsis
Conversion to prove whether one rational constant of type `:real` is equal to another.

## Description
The call `REAL_RAT_EQ_CONV ‘c1 = c2‘` where `c1` and `c2` are rational constants of type `:real`, returns whichever of `|- c1 = c2 <=> T` or `|- c1 = c2 <=> F` is true. The constants `c1` and `c2` may be integer constants (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`).

## Failure
Fails if applied to a term that is not an equality comparison on two permitted rational constants of type `:real`.

## Example

```
# REAL_RAT_EQ_CONV ‘#0.40 = &2 / &5‘;;
val it : thm = |- #0.40 = &2 / &5 <=> T

# REAL_RAT_EQ_CONV ‘#3.14 = &22 / &7‘;;
val it : thm = |- #3.14 = &22 / &7 <=> F
```

## See also
REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_GE_CONV

```
REAL_RAT_GE_CONV : conv
```

## Synopsis
Conversion to prove whether one rational literal of type `:real` is `>=` another.

## Description

The call `REAL_RAT_GE_CONV ‘c1 >= c2‘` where `c1` and `c2` are rational literals of type `:real`, returns whichever of `|- c1 >= c2 <=> T` or `|- c1 >= c2 <=> F` is true. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`).

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted rational literals of type `:real`.

## Example

```
# REAL_RAT_GE_CONV ‘#3.1415926 >= &22 / &7‘;;
val it : thm = |- #3.1415926 >= &22 / &7 <=> F
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_GT_CONV

```
REAL_RAT_GT_CONV : conv
```

## Synopsis

Conversion to prove whether one rational literal of type `:real` is `>` another.

## Description

The call `REAL_RAT_GT_CONV ‘c1 > c2‘` where `c1` and `c2` are rational literals of type `:real`, returns whichever of `|- c1 > c2 <=> T` or `|- c1 > c2 <=> F` is true. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`).

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted rational literals of type `:real`.

## Example

```
# REAL_RAT_GT_CONV ‘&3 / &2 > #1.11‘;;
val it : thm = |- &3 / &2 > #1.11 <=> T
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV,

REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_INV_CONV

```
REAL_RAT_INV_CONV : term -> thm
```

## Synopsis
Conversion to invert a rational constant of type `:real`.

## Description
The call `REAL_RAT_INV_CONV ‘inv c‘`, where `c` is a rational constant of type `:real`, returns the theorem `|- inv c = d` where `d` is the canonical rational constant that is equal to `c`'s multiplicative inverse (reciprocal). The constant `c` may be an integer constant (`&n` or `-- &n`), a ratio (`&p / &q` or `-- &p / &q`), or a decimal (`#DDD.DDDD` or `#DDD.DDDDeNN`). The returned value `d` is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure
Fails if applied to a term that is not the multiplicative inverse function applied to one of the permitted forms of rational constant of type `:real`, or if the constant is zero.

## Example

```
# REAL_RAT_INV_CONV ‘inv(-- &5 / &9)‘;;
val it : thm = |- inv (-- &5 / &9) = -- &9 / &5
```

## See also
REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_LE_CONV

```
REAL_RAT_LE_CONV : conv
```

## Synopsis
Conversion to prove whether one rational literal of type `:real` is `<=` another.

## Description

The call `REAL_RAT_LE_CONV ‘c1 <= c2‘` where `c1` and `c2` are rational literals of type `:real`, returns whichever of `|- c1 <= c2 <=> T` or `|- c1 <= c2 <=> F` is true. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`).

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted rational literals of type `:real`.

## Example

```
# REAL_RAT_LE_CONV ‘#3.1415926 <= &22 / &7‘;;
val it : thm = |- #3.1415926 <= &22 / &7 <=> T
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_LT_CONV

```
REAL_RAT_LT_CONV : conv
```

## Synopsis

Conversion to prove whether one rational literal of type `:real` is `<` another.

## Description

The call `REAL_RAT_LT_CONV ‘c1 < c2‘` where `c1` and `c2` are rational literals of type `:real`, returns whichever of `|- c1 < c2 <=> T` or `|- c1 < c2 <=> F` is true. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`).

## Failure

Fails if applied to a term that is not the appropriate inequality comparison on two permitted rational literals of type `:real`.

## Example

```
# REAL_RAT_LT_CONV ‘#3.1415926 < &355 / &113‘;;
val it : thm = |- #3.1415926 < &355 / &113 <=> T
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV,

REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_MUL_CONV

```
REAL_RAT_MUL_CONV : conv
```

## Synopsis
Conversion to perform multiplication on two rational literals of type `:real`.

## Description
The call `REAL_RAT_MUL_CONV ‘c1 * c2‘` where `c1` and `c2` are rational literals of type `:real`, returns `|- c1 * c2 = d` where `d` is the canonical rational literal that is equal to `c1 * c2`. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`). The result `d` is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure
Fails if applied to a term that is not the product of two permitted rational literals of type `:real`.

## Example
```
# REAL_RAT_MUL_CONV ‘#3.16227766016837952 * #3.16227766016837952‘;;
val it : thm =
  |- #3.16227766016837952 * #3.16227766016837952 =
     &244140625000000002902889155426649 / &24414062500000000000000000000000
```

## See also
REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_NEG_CONV

```
REAL_RAT_NEG_CONV : term -> thm
```

## Synopsis
Conversion to negate a rational literal of type `:real`.

## Description

The call `REAL_RAT_NEG_CONV ‘--c‘`, where `c` is a rational literal of type `:real`, returns the theorem `|- --c = d` where `d` is the canonical rational literal that is equal to `c`'s negation. The literal `c` may be an integer literal (`&n` or `-- &n`), a ratio (`&p / &q` or `-- &p / &q`), or a decimal (`#DDD.DDDD` or `#DDD.DDDDeNN`). The returned value `d` is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure

Fails if applied to a term that is not the negation of one of the permitted forms of rational literal of type `:real`.

## Example

```
# REAL_RAT_NEG_CONV ‘-- (-- &3 / &2)‘;;
val it : thm = |- --(-- &3 / &2) = &3 / &2
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_POW_CONV

```
REAL_RAT_POW_CONV : conv
```

## Synopsis

Conversion to perform exponentiation on a rational literal of type `:real`.

## Description

The call `REAL_RAT_POW_CONV ‘c pow n‘` where `c` is a rational literal of type `:real` and `n` is a numeral of type `:num`, returns `|- c pow n = d` where `d` is the canonical rational literal that is equal to `c` raised to the `n`th power. The literal `c` may be an integer literal (`&n` or `-- &n`), a ratios (`&p / &q` or `-- &p / &q`), or a decimal (`#DDD.DDDD` or `#DDD.DDDDeNN`). The result `d` is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure

Fails if applied to a term that is not a permitted rational literal of type `:real` raised to a numeral power.

## Example

```
# REAL_RAT_POW_CONV `#1.414 pow 2`;;
val it : thm = |- #1.414 pow 2 = &1999396 / &1000000
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_RED_CONV

```
REAL_RAT_RED_CONV : term -> thm
```

## Synopsis

Performs one arithmetic or relational operation on rational literals of type `:real`.

## Description

When applied to any of the terms `` `--c` ``, `` `inv c` ``, `` `abs c` ``, `` `c1 + c2` ``, `` `c1 - c2` ``, `` `c1 * c2` ``, `` `c1 / c2` ``, `` `c pow n` ``, `` `c1 <= c2` ``, `` `c1 < c2` ``, `` `c1 >= c2` ``, `` `c1 > c2` ``, `` `c1 = c2` ``, where `c`, `c1` and `c2` are rational literals of type `:real` and `n` is a numeral of type `:num`, `REAL_RAT_RED_CONV` returns a theorem asserting the equivalence of the term to a canonical rational (for the arithmetic operators) or a truth-value (for the relational operators).

The permissible rational literals are integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`). Any numeric result is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure

Fails if applied to an inappropriate term, or if `c` is zero in `` `inv c` ``, or if `c2` is zero in `c1 / c2`.

## Uses

More convenient for most purposes is `REAL_RAT_REDUCE_CONV`, which applies these evaluation conversions recursively at depth. But access to this 'one-step' reduction can be handy if you want to add a conversion `conv` for some other operator on real number literals, which you can conveniently incorporate it into `REAL_RAT_REDUCE_CONV` with

```
# let REAL_RAT_REDUCE_CONV' =
    DEPTH_CONV(REAL_RAT_RED_CONV ORELSEC conv);;
```

## See also

REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV,

REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_REDUCE_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_REDUCE_CONV

```
REAL_RAT_REDUCE_CONV : conv
```

## Synopsis
Evaluate subexpressions built up from rational literals of type `:real`, by proof.

## Description
When applied to a term, `REAL_RAT_REDUCE_CONV` performs a recursive bottom-up evaluation by proof of subterms built from rational literals of type `:real` using the unary operators ‘`--`’, ‘`inv`’ and ‘`abs`’, and the binary arithmetic (‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’, ‘`pow`’) and relational (‘`<`’, ‘`<=`’, ‘`>`’, ‘`>=`’, ‘`=`’) operators, as well as propagating literals through logical operations, e.g. `T /\ x <=> x`, returning a theorem that the original and reduced terms are equal.

The permissible rational literals are integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`). Any numeric result is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure
Never fails, but may have no effect.

## Example

```
# REAL_RAT_REDUCE_CONV
    `#3.1415926535 < &355 / &113 /\ &355 / &113 < &3 + &1 / &7`;;
val it : thm =
  |- #3.1415926535 < &355 / &113 /\ &355 / &113 < &3 + &1 / &7 <=> T
```

## See also
NUM_REDUCE_CONV, REAL_RAT_ABS_CONV, REAL_RAT_ADD_CONV, REAL_RAT_DIV_CONV, REAL_RAT_EQ_CONV, REAL_RAT_GE_CONV, REAL_RAT_GT_CONV, REAL_RAT_INV_CONV, REAL_RAT_LE_CONV, REAL_RAT_LT_CONV, REAL_RAT_MUL_CONV, REAL_RAT_NEG_CONV, REAL_RAT_POW_CONV, REAL_RAT_RED_CONV, REAL_RAT_SUB_CONV.

# REAL_RAT_SUB_CONV

```
REAL_RAT_SUB_CONV : conv
```

## Synopsis

Conversion to perform subtraction on two rational literals of type `:real`.

## Description

The call `REAL_RAT_SUB_CONV ‘c1 - c2‘` where `c1` and `c2` are rational literals of type `:real`, returns `|- c1 - c2 = d` where `d` is the canonical rational literal that is equal to `c1 - c2`. The literals `c1` and `c2` may be integer literals (`&n` or `-- &n`), ratios (`&p / &q` or `-- &p / &q`), or decimals (`#DDD.DDDD` or `#DDD.DDDDeNN`). The result `d` is always put in the form `&p / &q` or `-- &p / &q` with `q > 1` and `p` and `q` sharing no common factor, or simply `&p` or `-- &p` when that is impossible.

## Failure

Fails if applied to a term that is not the subtraction function applied to two permitted rational literals of type `:real`.

## Example

```
# REAL_RAT_SUB_CONV ‘&355 / &113 - #3.1415926‘;;
val it : thm = |- &355 / &113 - #3.1415926 = &181 / &565000000
```

## See also

`REAL_RAT_ABS_CONV`, `REAL_RAT_ADD_CONV`, `REAL_RAT_DIV_CONV`, `REAL_RAT_EQ_CONV`, `REAL_RAT_GE_CONV`, `REAL_RAT_GT_CONV`, `REAL_RAT_INV_CONV`, `REAL_RAT_LE_CONV`, `REAL_RAT_LT_CONV`, `REAL_RAT_MUL_CONV`, `REAL_RAT_NEG_CONV`, `REAL_RAT_POW_CONV`, `REAL_RAT_REDUCE_CONV`, `REAL_RAT_RED_CONV`.

# REAL_RING

```
REAL_RING : term -> thm
```

## Synopsis

Ring decision procedure instantiated to real numbers.

## Description

The rule `REAL_RING` should be applied to a formula that, after suitable normalization, can be considered a universally quantified Boolean combination of equations and inequations between terms of type `:real`. If that formula holds in all integral domains, `REAL_RING` will prove it. Any “alien” atomic formulas that are not real number equations will not contribute to the proof but will not in themselves cause an error. The function is a particular instantiation of `RING`, which is a more generic procedure for ring and semiring structures.

## Failure

Fails if the formula is unprovable by the methods employed. This does not necessarily mean that it is not valid for `:real`, but rather that it is not valid on all integral domains (see below).

## Example

This simple example is based on the inversion of a homographic function (from Gosper's notes on continued fractions):

```
# REAL_RING
   `y * (c * x + d) = a * x + b ==> x * (c * y - a) = b - d * y`;;
2 basis elements and 0 critical pairs
val it : thm = |- y * (c * x + d) = a * x + b ==> x * (c * y - a) = b - d * y
```

The following more complicated example verifies a classic Cardano reduction formula for cubic equations:

```
# REAL_RING
   `p = (&3 * a1 - a2 pow 2) / &3 /\
    q = (&9 * a1 * a2 - &27 * a0 - &2 * a2 pow 3) / &27 /\
    z = x - a2 / &3 /\
    x * w = w pow 2 - p / &3 /\
    ~(p = &0)
    ==> (z pow 3 + a2 * z pow 2 + a1 * z + a0 = &0 <=>
        (w pow 3) pow 2 - q * (w pow 3) - p pow 3 / &27 = &0)`;;
...
```

Note that formulas depending on specific features of the real numbers are not always provable by this generic ring procedure. For example we can prove:

```
# REAL_RING
   `s pow 2 = &2
    ==> (x pow 4 + &1 = &0 <=>
         x pow 2 + s * x + &1 = &0 \/ x pow 2 - s * x + &1 = &0)`;;
...
```

but not the much simpler real-specific fact:

```
# REAL_RING `x pow 4 + 1 = &0 ==> F`;;
Exception: Failure "tryfind".
```

To support real-specific nonlinear reasoning, you may like to investigate the experimental decision procedure in `Examples/sos.ml`. For general support for division (fields) see `REAL_FIELD`.

## Uses

Often useful for generating non-trivial algebraic lemmas. Even when it is not capable of solving the whole problem, it can often deal with the most tedious algebraic parts. For example after loading in the definitions of trig functions:

```
# needs "Examples/transc.ml";;
```

you may wish to prove a tedious trig identity such as:

```
# g `(--((&7 * cos x pow 6) * sin x) * &7) / &49 -
     (--((&5 * cos x pow 4) * sin x) * &5) / &25 * &3 +
      --((&3 * cos x pow 2) * sin x) + sin x =
     sin x pow 7`;;
```

which can be done by REAL_RING together with one simple lemma:

```
# SIN_CIRCLE;;
val it : thm = |- !x. sin x pow 2 + cos x pow 2 = &1
```

as follows:

```
# e(MP_TAC(SPEC `x:real` SIN_CIRCLE) THEN CONV_TAC REAL_RING);;
2 basis elements and 0 critical pairs
val it : goalstack = No subgoals
```

## See also

ARITH_RULE, ARITH_TAC, INT_RING, NUM_RING, real_ideal_cofactors, REAL_ARITH, REAL_FIELD, RING.

# RECALL_ACCEPT_TAC

```
RECALL_ACCEPT_TAC : ('a -> thm) -> 'a -> goal -> goalstate
```

## Synopsis

Delay evaluation of theorem-producing function till needed.

## Description

Given a theorem-producing inference rule `f` and its argument `a`, the tactic `RECALL_ACCEPT_TAC f a` will evaluate `th = f a` and do `ACCEPT_TAC th`, but only when the tactic is applied to a goal.

## Failure

Never fails until subsequently applied to a goal, but then may fail if the theorem-producing function does.

## Example

You might for example do

```
RECALL_ACCEPT_TAC (EQT_ELIM o NUM_REDUCE_CONV) `16 EXP 53 < 15 EXP 55`;;
```

and the call

```
(EQT_ELIM o NUM_REDUCE_CONV) `16 EXP 53 < 15 EXP 55`
```

will be delayed until the tactic is applied.

## Uses

Delaying a time-consuming compound inference rule in a tactic script until it is actually used.

# REDEPTH_CONV

```
REDEPTH_CONV : conv -> conv
```

## Synopsis

Applies a conversion bottom-up to all subterms, retraversing changed ones.

## Description

`REDEPTH_CONV c tm` applies the conversion `c` repeatedly to all subterms of the term `tm` and recursively applies `REDEPTH_CONV c` to each subterm at which `c` succeeds, until there is no subterm remaining for which application of `c` succeeds.

More precisely, `REDEPTH_CONV c tm` repeatedly applies the conversion `c` to all the subterms of the term `tm`, including the term `tm` itself. The supplied conversion `c` is applied to the subterms of `tm` in bottom-up order and is applied repeatedly (zero or more times, as is done by `REPEATC`) to each subterm until it fails. If `c` is successfully applied at least once to a subterm, `t` say, then the term into which `t` is transformed is retraversed by applying `REDEPTH_CONV c` to it.

## Failure

`REDEPTH_CONV c tm` never fails but can diverge if the conversion `c` can be applied repeatedly to some subterm of `tm` without failing.

## Example

The following example shows how `REDEPTH_CONV` retraverses subterms:

```
# REDEPTH_CONV BETA_CONV `(\f x. (f x) + 1) (\y.y) 2`;;
val it : thm = |- (\f x. f x + 1) (\y. y) 2 = 2 + 1
```

Here, `BETA_CONV` is first applied successfully to the (beta-redex) subterm:

```
`(\f x. (f x) + 1) (\y.y)`
```

This application reduces this subterm to:

```
`(\x. ((\y.y) x) + 1)`
```

`REDEPTH_CONV BETA_CONV` is then recursively applied to this transformed subterm, eventually reducing it to `` `(\x. x + 1)` ``. Finally, a beta-reduction of the top-level term, now the simplified beta-redex `` `(\x. x + 1) 2` ``, produces `` `2 + 1` ``.

## See also

`DEPTH_CONV`, `ONCE_DEPTH_CONV`, `TOP_DEPTH_CONV`, `TOP_SWEEP_CONV`.

# REDEPTH_SQCONV

```
REDEPTH_SQCONV : strategy
```

## Synopsis

Applies simplification bottom-up to all subterms, retraversing changed ones.

## Description

HOL Light's simplification functions (e.g. `SIMP_TAC`) have their traversal algorithm controlled by a "strategy". `REDEPTH_SQCONV` is a strategy corresponding to `REDEPTH_CONV` for ordinary conversions: simplification is applied bottom-up to all subterms, retraversing changed ones.

## Failure

Not applicable.

## See also

`DEPTH_SQCONV`, `ONCE_DEPTH_SQCONV`, `REDEPTH_CONV`, `TOP_DEPTH_SQCONV`, `TOP_SWEEP_SQCONV`.

# reduce_interface

```
reduce_interface : string * term -> unit
```

## Synopsis
Remove a specific overload/interface mapping for an identifier.

## Description
HOL Light allows an identifier to map to a specific constant (see `override_interface`) or be overloaded to several depending on type (see `overload_interface`). A call to `remove_interface "ident"` removes all such mappings for the identifier `ident`.

## Failure
Never fails, whether or not there were any interface mappings in effect.

## See also
`overload_interface`, `override_interface`, `remove_interface`, `the_interface`.

# refine

```
refine : refinement -> goalstack
```

## Synopsis
Applies a refinement to the current goalstack.

## Description
The call `refine r` applies the refinement `r` to the current goalstate, adding the resulting goalstate at the head of the current goalstack list. (A goalstate consists of a list of subgoals as well as justification and metavariable information.)

## Failure
Fails if the refinement fails.

## Comments
Most users will not want to handle refinements explicitly. Usually one just applies a tactic to the first goal in a goalstate.

# REFL

```
REFL : term -> thm
```

## Synopsis
Returns theorem expressing reflexivity of equality.

## Description

REFL maps any term `t` to the corresponding theorem |- t = t.

## Failure

Never fails.

## Example

```
# REFL `2`;;
val it : thm = |- 2 = 2

# REFL `p:bool`;;
val it : thm = |- p <=> p
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

ALL_CONV, REFL_TAC.

# REFL_TAC

```
REFL_TAC : tactic
```

## Synopsis

Solves a goal that is an equation between alpha-equivalent terms.

## Description

When applied to a goal A ?- t = t', where t and t' are alpha-equivalent, REFL_TAC completely solves it.

```
  A ?- t = t'
=============  REFL_TAC
```

## Failure

Fails unless the goal is an equation between alpha-equivalent terms.

## See also

ACCEPT_TAC, MATCH_ACCEPT_TAC, REWRITE_TAC.

# REFUTE_THEN

```
REFUTE_THEN : thm_tactic -> goal -> goalstate
```

## Synopsis
Assume the negation of the goal and apply theorem-tactic to it.

## Description
The tactic `REFUTE_THEN ttac` applied to a goal `g`, assumes the negation of the goal and applies `ttac` to it and a similar goal with a conclusion of `F`. More precisely, if the original goal `A ?- u` is unnegated and `ttac`'s action is

```
 A ?- F
========  ttac (ASSUME `~u`)
 B ?- v
```

then we have

```
  A ?- u
==============  REFUTE_THEN ttac
  B ?- v
```

For example, if `ttac` is just `ASSUME_TAC`, this corresponds to a classic 'proof by contradiction':

```
      A ?- u
=================  REFUTE_THEN ASSUME_TAC
 A u {~u} ?- F
```

Whatever `ttac` may be, if the conclusion `u` of the goal is negated, the effect is the same except that the assumed theorem will not be double-negated, so the effect is the same as `DISCH_THEN`.

## Failure
Never fails unless the underlying theorem-tactic `ttac` does.

## Uses
Classical 'proof by contradiction'.

## Comments
When applied to an unnegated goal, this tactic embodies implicitly the classical principle of 'proof by contradiction', but for negated goals the tactic is also intuitionistically valid.

## See also
`BOOL_CASES_TAC`, `DISCH_THEN`.

# remark

`remark : string -> unit`

## Synopsis

Output a string and newline if and only if `verbose` flag is set.

## Description

If the `verbose` flag is set to `true`, then the call `remark s` prints the string `s` and a following newline. If the `verbose` flag is set to `false`, this call does nothing. This function is used for informative output in several automated rules such as `MESON`.

## Failure

Never fails.

## Example

```
# remark "Proof is going OK so far";;
Proof is going OK so far
val it : unit = ()
# verbose := false;;
val it : unit = ()
#  remark "Proof is going OK so far";;
val it : unit = ()
```

## See also

`report`, `verbose`.

# remove

`remove : ('a -> bool) -> 'a list -> 'a * 'a list`

## Synopsis

Separates the first element of a list to satisfy a predicate from the rest of the list.

## Failure

Fails if no element satisfes the predicate. This will always be the case for an empty list.

## Example

```
# remove (fun x -> x >= 3) [1;2;3;4;5;6];;
val it : int * int list = (3, [1; 2; 4; 5; 6])
```

## See also

partition, filter.

# remove_interface

remove_interface : string -> unit

## Synopsis

Remove all overload/interface mappings for an identifier.

## Description

HOL Light allows an identifier to map to a specific constant (see override_interface) or be overloaded to several depending on type (see overload_interface). A call to remove_interface "ident" removes all such mappings for the identifier ident.

## Failure

Never fails, whether or not there were any interface mappings in effect.

## See also

overload_interface, override_interface, reduce_interface, the_interface.

# REMOVE_THEN

REMOVE_THEN : string -> thm_tactic -> tactic

## Synopsis

Apply a theorem tactic to named assumption, removing the assumption.

## Description

The tactic USE_THEN "name" ttac applies the theorem-tactic ttac to the assumption labelled name (or the first in the list if there is more than one), removing the assumption from the goal.

## Failure

Fails if there is no assumption of that name or if the theorem-tactic fails when applied to it.

## Example
See `LABEL_TAC` for a relevant example.

## Uses
Using an assumption identified by name that will not be needed again in the proof.

## See also
`ASSUME`, `FIND_ASSUM`, `LABEL_TAC`, `REMOVE_THEN`, `USE_THEN`

# REPEATC

```
REPEATC : conv -> conv
```

## Synopsis
Repeatedly apply a conversion (zero or more times) until it fails.

## Description
If `c` is a conversion effects a transformation of a term `t` to a term `t'`, that is if `c` maps `t` to the theorem `|- t = t'`, then `REPEATC c` is the conversion that repeats this transformation as often as possible. More exactly, if `c` maps the term `` `ti` `` to `|- ti=t(i+1)` for `i` from `1` to `n`, but fails when applied to the `n`+1th term `` `t(n+1)` ``, then `` REPEATC c `t1` `` returns `|- t1 = t(n+1)`. And if `` c `t` `` fails, them `` REPEATC c `t` `` returns `|- t = t`.

## Failure
Never fails, but can diverge if the supplied conversion never fails.

## Example

```
# BETA_CONV `(\x. (\y. x + y) (x + 1)) 1`;;
val it : thm = |- (\x. (\y. x + y) (x + 1)) 1 = (\y. 1 + y) (1 + 1)

# REPEATC BETA_CONV `(\x. (\y. x + y) (x + 1)) 1`;;
val it : thm = |- (\x. (\y. x + y) (x + 1)) 1 = 1 + 1 + 1
```

# repeat

```
repeat : ('a -> 'a) -> 'a -> 'a
```

## Synopsis
Repeatedly apply a function until it fails.

## Description
The call `repeat f x` successively applies `f` over and over again starting with `x`, and stops at the first point when a `Failure _` exception occurs.

## Failure
Never fails. If `f` fails at once it returns `x`.

## Example

```
# repeat (snd o dest_forall) `!x y z. x + y + z < 1`;;
val it : term = `x + y + z < 1`
```

## Comments
If you know exactly how many times you want to apply it, you may prefer `funpow`.

## See also
`funpow`, `fail`.

# REPEAT_GTCL

```
REPEAT_GTCL : thm_tactical -> thm_tactical
```

## Synopsis
Applies a theorem-tactical until it fails when applied to a goal.

## Description
When applied to a theorem-tactical, a theorem-tactic, a theorem and a goal:

```
REPEAT_GTCL ttl ttac th goal
```

`REPEAT_GTCL` repeatedly modifies the theorem according to `ttl` till the result of handing it to `ttac` and applying it to the goal fails (this may be no times at all).

## Failure
Fails iff the theorem-tactic fails immediately when applied to the theorem and the goal.

## Example
The following tactic matches `th`'s antecedents against the assumptions of the goal until it can do so no longer, then puts the resolvents onto the assumption list:

```
REPEAT_GTCL (IMP_RES_THEN ASSUME_TAC) th
```

## See also
`REPEAT_TCL`, `THEN_TCL`.

# REPEAT_TCL

```
REPEAT_TCL : thm_tactical -> thm_tactical
```

## Synopsis
Repeatedly applies a theorem-tactical until it fails when applied to the theorem.

## Description
When applied to a theorem-tactical, a theorem-tactic and a theorem:

```
REPEAT_TCL ttl ttac th
```

`REPEAT_TCL` repeatedly modifies the theorem according to `ttl` until it fails when given to the theorem-tactic `ttac`.

## Failure
Fails iff the theorem-tactic fails immediately when applied to the theorem.

## Example
It is often desirable to repeat the action of basic theorem-tactics. For example `CHOOSE_THEN` strips off a single existential quantification, so one might use `REPEAT_TCL CHOOSE_THEN` to get rid of them all.

Alternatively, one might want to repeatedly break apart a theorem which is a nested conjunction and apply the same theorem-tactic to each conjunct. For example the following goal:

```
# g `(0 = w /\ 0 = x) /\ 0 = y /\ 0 = z ==> w + x + y + z = 0`;;
Warning: Free variables in goal: w, x, y, z
val it : goalstack = 1 subgoal (1 total)

`(0 = w /\ 0 = x) /\ 0 = y /\ 0 = z ==> w + x + y + z = 0`
```

might be solved by

```
# e(DISCH_THEN (REPEAT_TCL CONJUNCTS_THEN (SUBST1_TAC o SYM)) THEN
    REWRITE_TAC[ADD_CLAUSES]);;
```

## See also
REPEAT_GTCL, THEN_TCL.

# REPEAT

```
REPEAT : tactic -> tactic
```

## Synopsis
Repeatedly applies a tactic until it fails.

## Description
The tactic `REPEAT t` is a tactic which applies `t` to a goal, and while it succeeds, continues applying it to all subgoals generated.

## Failure
The application of `REPEAT` to a tactic never fails, and neither does the composite tactic, even if the basic tactic fails immediately, unless it raises an exception other that `Failure ...`.

## Example
If we start with a goal having many universal quantifiers:

```
# g `!w x y z. w < z /\ x < y ==> w * x + 1 <= y * z`;;
```

then `GEN_TAC` will strip them off one at a time:

```
# e GEN_TAC;;
val it : goalstack = 1 subgoal (1 total)

`!x y z. w < z /\ x < y ==> w * x + 1 <= y * z`
```

and `REPEAT GEN_TAC` will strip them off as far as possible:

```
# e(REPEAT GEN_TAC);;
val it : goalstack = 1 subgoal (1 total)

`w < z /\ x < y ==> w * x + 1 <= y * z`
```

Similarly, `REPEAT COND_CASES_TAC` will eliminate all free conditionals in the goal instead of just one.

## See also
`EVERY`, `FIRST`, `ORELSE`, `THEN`, `THENL`.

# replicate

```
replicate : 'a -> int -> 'a list
```

## Synopsis
Makes a list consisting of a value replicated a specified number of times.

## Description
`replicate x n` returns `[x;...;x]`, a list of length `n`.

## Failure

Fails if number of replications is less than zero.

# REPLICATE_TAC

```
REPLICATE_TAC : int -> tactic -> tactic
```

## Synopsis

Apply a tactic a specific number of times.

## Description

The call `REPLICATE n tac` gives a new tactic that it equivalent to an `n`-fold repetition of `tac`, i.e. `tac THEN tac THEN ... THEN tac`.

## Failure

The call `REPLICATE n tac` never fails, but when applied to a goal it will fail if the tactic does.

## Example

We might conceivably want to strip off exactly three universal quantifiers from a goal that contains more than three. We can use `REPLICATE_TAC 3 GEN_TAC` to do that.

## See also

`EVERY`, `MAP_EVERY`, `THEN`.

# report

```
report : string -> unit
```

## Synopsis

Prints a string and a following line break.

## Description

The call `report s` prints the string `s` to the terminal and then a following newline.

## Failure

Never fails.

## Example

```
# report "Proof completed OK";;
Proof completed OK
val it : unit = ()
```

## See also
remark, warn.

# report_timing

report_timing : bool ref

## Synopsis
Flag to determine whether time function outputs CPU time measure.

## Description
When report_timing is true, a call time f x will evaluate f x as usual but also as a side-effect print out the CPU time taken. If report_timing is false, nothing will be printed. Times are already printed in this way automatically as informative output in some rules like MESON, so this can be used to silence them.

## Failure
Not applicable.

## Example

```
# time NUM_REDUCE_CONV `2 EXP 300 < 2 EXP 200`;;
CPU time (user): 0.13
val it : thm = |- 2 EXP 300 < 2 EXP 200 <=> F
# report_timing := false;;
val it : unit = ()
# time NUM_REDUCE_CONV `2 EXP 300 < 2 EXP 200`;;
val it : thm = |- 2 EXP 300 < 2 EXP 200 <=> F
```

## See also
time.

# reserved_words

reserved_words : unit -> string list

## Synopsis
Returns the list of reserved words.

## Description
Certain identifiers in HOL are reserved, e.g. ‘`if`’, ‘`let`’ and ‘`|`’, meaning that they are special to the parser and cannot be used as ordinary identifiers. The call `reserved_words()` returns a list of such identifiers.

## Failure
Never fails.

## Example
In the default HOL state:

```
  # reserved_words();;
  val it : string list =
    ["("; ")"; "["; "]"; "{\small\verb%"; "%}"; ":"; ";"; "."; "|"; "let"; "in"; "and";
     "if"; "then"; "else"; "//"]
```

## See also
`is_reserved_word, reserve_words, unreserve_words.`

# reserve_words

```
reserve_words : string list -> unit
```

## Synopsis
Add given strings to the set of reserved words.

## Description
Certain identifiers in HOL are reserved, e.g. ‘`if`’, ‘`let`’ and ‘`|`’, meaning that they are special to the parser and cannot be used as ordinary identifiers. A call `reserve_words l` adds all strings in `l` to the list of reserved identifiers.

## Failure
Never fails, regardless of whether the given strings were already reserved.

## See also
`is_reserved_word, reserved_words, unreserve_words.`

# retypecheck

```
retypecheck : (string * pretype) list -> preterm -> preterm
```

## Synopsis
Typecheck a term, iterating over possible overload resolutions.

## Description
This is the main HOL Light typechecking function. Given an environment `env` of pretype assignments for variables, it assigns a pretype to all variables and constants, including performing resolution of overloaded constants based on what type information there is. Normally, this happens implicitly when a term is entered in the quotation parser.

## Failure
Fails if some terms cannot be consistently assigned a type.

## Comments
Only users seeking to change HOL's parser and typechecker quite radically need to use this function.

## See also
`term_of_preterm`.

# rev_assocd

```
rev_assocd : 'a -> ('b * 'a) list -> 'b -> 'b
```

## Synopsis
Looks up item in association list taking default in case of failure.

## Description
The call `rev_assocd y [x1,y1; ...; xn,yn] x` returns the first `xi` in the list where the corresponding `yi` is the same as `y`. If there is no such item, it returns the value `x`. This is similar to `rev_assoc` except that the latter will fail rather than take a default.

## Failure
Never fails.

## Example

```
# rev_assocd 6 [1,2; 2,4; 3,6] (-1);;
val it : int = 3
# rev_assocd 8 [1,2; 2,4; 3,6] (-1);;
val it : int = -1
```

## Uses
Simple lookup without exception handling.

## See also

assocd, rev_assoc.

# rev_assoc

rev_assoc : 'a -> ('b * 'a) list -> 'b

## Synopsis

Searches a list of pairs for a pair whose second component equals a specified value.

## Description

rev_assoc y [(x1,y1);...;(xn,yn)] returns the first xi in the list such that yi equals y.

## Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

## Example

```
# rev_assoc 2 [(1,4);(3,2);(2,5);(2,6)];;
val it : int = 3
```

## See also

assoc, find, mem, tryfind, exists, forall.

# rev

rev : 'a list -> 'a list

## Synopsis

Reverses a list.

## Description

rev [x1;...;xn] returns [xn;...;x1].

## Failure

Never fails.

# reverse_interface_mapping

```
reverse_interface_mapping : bool ref
```

## Synopsis
Determines whether interface map is printed on output (default `true`).

## Description
The reference variable `reverse_interface_mapping` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. When `reverse_interface_mapping` is `true` (as it is by default), the front-end interface map for a constant or variable is printed. When it is `false`, the core constant or variable is printed.

## Failure
Not applicable.

## Example
Here is a simple library theorem about real numbers as it usually appears:

```
# reverse_interface_mapping := true;;
val it : unit = ()
# REAL_EQ_SUB_LADD;;
val it : thm = |- !x y z. x = y - z <=> x + z = y
```

but with another setting of `reverse_interface_mapping` we see that the usual symbol '`+`' is an interface for `real_add`, while the 'iff' sign is just an interface for Boolean equality:

```
# reverse_interface_mapping := false;;
val it : unit = ()
# REAL_EQ_SUB_LADD;;
val it : thm = |- !x y z. (x = real_sub y z) = real_add x z = y
```

## See also
`pp_print_term`, `prebroken_binops`, `print_all_thm`, `print_unambiguous_comprehensions`, `the_interface`, `typify_universal_set`, `unspaced_binops`.

# rev_itlist2

```
rev_itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

## Synopsis
Applies a paired function between adjacent elements of 2 lists.

## Description
`itlist2 f ([x1;...;xn],[y1;...;yn]) z` returns

```
f xn yn ( ... (f x2 y2 (f x1 y1 z))...)%}.
```

It returns `z` if both lists are empty.

## Failure
Fails if the two lists are of different lengths.

## Example
This takes a 'dot product' of two vectors of integers:

```
# let dot v w = rev_itlist2 (fun x y z -> x * y + z) v w 0;;
val dot : int list -> int list -> int = <fun>
# dot [1;2;3] [4;5;6];;
val it : int = 32
```

## See also
`itlist, rev_itlist, rev_itlist2, end_itlist, uncurry.`

# rev_itlist

```
rev_itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

## Synopsis
Applies a binary function between adjacent elements of the reverse of a list.

## Description
`rev_itlist f [x1;...;xn] y` returns `f xn ( ... (f x2 (f x1 y))...)`. It returns `y` if the list is empty.

## Failure
Never fails.

## Example

```
# rev_itlist (fun x y -> x * y) [1;2;3;4] 1;;
val it : int = 24
```

## See also
`itlist, end_itlist.`

# rev_splitlist

```
rev_splitlist : ('a -> 'a * 'b) -> 'a -> 'a * 'b list
```

## Synopsis

Applies a binary destructor repeatedly in right-associative mode.

## Description

If a destructor function `d` inverts a binary constructor `f`, for example `dest_comb` for `mk_comb`, and fails when applied to `y`, then:

```
rev_splitlist d f(...(f(f(w,x1),x2),...xn)
```

returns

```
(w,[x1; ... ; xn])
```

## Failure

Never fails.

## Example

The function `strip_comb` is actually just defined as `rev_splitlist dest_comb`, which acts as follows:

```
# rev_splitlist dest_comb `x + 1 + 2`;;
val it : term * term list = (`(+)`, [`x`; `1 + 2`])
```

## See also

`itlist`, `splitlist`, `striplist`.

# REWR_CONV

```
REWR_CONV : thm -> term -> thm
```

## Synopsis

Uses an instance of a given equation to rewrite a term.

## Description

`REWR_CONV` is one of the basic building blocks for the implementation of rewriting in the HOL system. In particular, the term replacement or rewriting done by all the built-in rewriting rules and tactics is ultimately done by applications of `REWR_CONV` to appropriate subterms. The description given here for `REWR_CONV` may therefore be taken as a specification of the atomic action of replacing equals by equals that is used in all these higher level rewriting tools.

The first argument to `REWR_CONV` is expected to be an equational theorem which is to be used as a left-to-right rewrite rule. The general form of this theorem is:

```
A |- t[x1,...,xn] = u[x1,...,xn]
```

where `x1`, ..., `xn` are all the variables that occur free in the left-hand side of the conclusion of the theorem but do not occur free in the assumptions. Any of these variables may also be universally quantified at the outermost level of the equation, as for example in:

```
A |- !x1...xn. t[x1,...,xn] = u[x1,...,xn]
```

Note that `REWR_CONV` will also work, but will give a generally undesirable result (see below), if the right-hand side of the equation contains free variables that do not also occur free on the left-hand side, as for example in:

```
A |- t[x1,...,xn] = u[x1,...,xn,y1,...,ym]
```

where the variables `y1`, ..., `ym` do not occur free in `t[x1,...,xn]`.

If `th` is an equational theorem of the kind shown above, then `REWR_CONV th` returns a conversion that maps terms of the form `t[e1,...,en/x1,...,xn]`, in which the terms `e1`, ..., `en` are free for `x1`, ..., `xn` in `t`, to theorems of the form:

```
A |- t[e1,...,en/x1,...,xn] = u[e1,...,en/x1,...,xn]
```

That is, `REWR_CONV th tm` attempts to match the left-hand side of the rewrite rule `th` to the term `tm`. If such a match is possible, then `REWR_CONV` returns the corresponding substitution instance of `th`.

If `REWR_CONV` is given a theorem `th`:

```
A |- t[x1,...,xn] = u[x1,...,xn,y1,...,ym]
```

where the variables `y1`, ..., `ym` do not occur free in the left-hand side, then the result of applying the conversion `REWR_CONV th` to a term `t[e1,...,en/x1,...,xn]` will be:

```
A |- t[e1,...,en/x1,...,xn] = u[e1,...,en,v1,...,vm/x1,...,xn,y1,...,ym]
```

where `v1`, ..., `vm` are variables chosen so as to be free nowhere in `th` or in the input term. The user has no control over the choice of the variables `v1`, ..., `vm`, and the variables actually chosen may well be inconvenient for other purposes. This situation is, however, relatively rare; in most equations the free variables on the right-hand side are a subset of the free variables on the left-hand side.

In addition to doing substitution for free variables in the supplied equational theorem (or 'rewrite rule'), `REWR_CONV th tm` also does type instantiation, if this is necessary in order to match the left-hand side of the given rewrite rule `th` to the term argument `tm`. If, for example, `th` is the theorem:

```
A |- t[x1,...,xn] = u[x1,...,xn]
```

and the input term `tm` is (a substitution instance of) an instance of `t[x1,...,xn]` in which the types `ty1`, ..., `tyi` are substituted for the type variables `vty1`, ..., `vtyi`, that is if:

```
tm = t[ty1,...,tyn/vty1,...,vtyn][e1,...,en/x1,...,xn]
```

then `REWR_CONV th tm` returns:

```
A |- (t = u)[ty1,...,tyn/vty1,...,vtyn][e1,...,en/x1,...,xn]
```

Note that, in this case, the type variables `vty1`, ..., `vtyi` must not occur anywhere in the hypotheses `A`. Otherwise, the conversion will fail.

## Failure

`REWR_CONV th` fails if `th` is not an equation or an equation universally quantified at the outermost level. If `th` is such an equation:

```
th = A |- !v1....vi. t[x1,...,xn] = u[x1,...,xn,y1,...,yn]
```

then `REWR_CONV th tm` fails unless the term `tm` is alpha-equivalent to an instance of the left-hand side `t[x1,...,xn]` which can be obtained by instantiation of free type variables (i.e. type variables not occurring in the assumptions `A`) and substitution for the free variables `x1`, ..., `xn`.

## Example

The following example illustrates a straightforward use of `REWR_CONV`. The supplied rewrite rule is polymorphic, and both substitution for free variables and type instantiation may take place. `EQ_SYM_EQ` is the theorem:

```
|- !x y:A. x = y <=> y = x
```

and `REWR_CONV EQ_SYM` behaves as follows:

```
# REWR_CONV EQ_SYM_EQ `1 = 2`;;
val it : thm = |- 1 = 2 <=> 2 = 1
# REWR_CONV EQ_SYM_EQ `1 < 2`;;
Exception: Failure "term_pmatch".
```

The second application fails because the left-hand side `` `x = y` `` of the rewrite rule does not match the term to be rewritten, namely `` `1 < 2` ``.

In the following example, one might expect the result to be the theorem `A |- f 2 = 2`, where `A` is the assumption of the supplied rewrite rule:

```
# REWR_CONV (ASSUME `!x:A. f x = x`) `f 2:num`;;
Exception: Failure "term_pmatch: can't instantiate local constant".
```

The application fails, however, because the type variable `A` appears in the assumption of the theorem returned by `` ASSUME `!x:A. f x = x` ``.

Failure will also occur in situations like:

```
# REWR_CONV (ASSUME `f (n:num) = n`) `f 2:num`;;
Exception: Failure "term_pmatch: can't instantiate local constant".
```

where the left-hand side of the supplied equation contains a free variable (in this case n) which is also free in the assumptions, but which must be instantiated in order to match the input term.

### See also
IMP_REWR_CONV, ORDERED_REWR_CONV, REWRITE_CONV.

## REWRITE_CONV

```
REWRITE_CONV : thm list -> conv
```

### Synopsis
Rewrites a term including built-in tautologies in the list of rewrites.

### Description
Rewriting a term using REWRITE_CONV utilizes as rewrites two sets of theorems: the tautologies in the ML list basic_rewrites and the ones supplied by the user. The rule searches top-down and recursively for subterms which match the left-hand side of any of the possible rewrites, until none of the transformations are applicable. There is no ordering specified among the set of rewrites.

Variants of this conversion allow changes in the set of equations used: PURE_REWRITE_CONV and others in its family do not rewrite with the theorems in basic_rewrites.

The top-down recursive search for matches may not be desirable, as this may increase the number of inferences being made or may result in divergence. In this case other rewriting tools such as ONCE_REWRITE_CONV and GEN_REWRITE_CONV can be used, or the set of theorems given may be reduced.

See GEN_REWRITE_CONV for the general strategy for simplifying theorems in HOL using equational theorems.

### Failure
Does not fail, but may diverge if the sequence of rewrites is non-terminating.

### Uses
Used to manipulate terms by rewriting them with theorems. While resulting in high degree of automation, REWRITE_CONV can spawn a large number of inference steps. Thus, variants such as PURE_REWRITE_CONV, or other rules such as SUBST_CONV, may be used instead to improve efficiency.

### See also
basic_rewrites, GEN_REWRITE_CONV, ONCE_REWRITE_CONV, PURE_REWRITE_CONV, REWR_CONV, SUBST_CONV.

# REWRITE_RULE

```
REWRITE_RULE : thm list -> thm -> thm
```

## Synopsis
Rewrites a theorem including built-in tautologies in the list of rewrites.

## Description
Rewriting a theorem using `REWRITE_RULE` utilizes as rewrites two sets of theorems: the tautologies in the ML list `basic_rewrites` and the ones supplied by the user. The rule searches top-down and recursively for subterms which match the left-hand side of any of the possible rewrites, until none of the transformations are applicable. There is no ordering specified among the set of rewrites.

Variants of this rule allow changes in the set of equations used: `PURE_REWRITE_RULE` and others in its family do not rewrite with the theorems in `basic_rewrites`. Rules such as `ASM_REWRITE_RULE` add the assumptions of the object theorem (or a specified subset of these assumptions) to the set of possible rewrites.

The top-down recursive search for matches may not be desirable, as this may increase the number of inferences being made or may result in divergence. In this case other rewriting tools such as `ONCE_REWRITE_RULE` and `GEN_REWRITE_RULE` can be used, or the set of theorems given may be reduced.

See `GEN_REWRITE_RULE` for the general strategy for simplifying theorems in HOL using equational theorems.

## Failure
Does not fail, but may diverge if the sequence of rewrites is non-terminating.

## Uses
Used to manipulate theorems by rewriting them with other theorems. While resulting in high degree of automation, `REWRITE_RULE` can spawn a large number of inference steps. Thus, variants such as `PURE_REWRITE_RULE`, or other rules such as `SUBST`, may be used instead to improve efficiency.

## See also
`ASM_REWRITE_RULE`, `basic_rewrites`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWR_CONV`, `REWRITE_CONV`, `SUBST`.

# REWRITES_CONV

```
REWRITES_CONV : ('a * (term -> 'b)) net -> term -> 'b
```

## Synopsis
Apply a prioritized conversion net to the term at the top level.

## Description
The underlying machinery in rewriting and simplification assembles (conditional) rewrite rules and other conversions into a net, including a priority number so that, for example, pure rewrites get applied before conditional rewrites. If `net` is such a net (for example, constructed using `mk_rewrites` and `net_of_thm`), then `REWRITES_CONV net` is a conversion that uses all those conversions at the toplevel to attempt to rewrite the term. If a conditional rewrite is applied, the resulting theorem will have an assumption. This is the primitive operation that performs HOL Light rewrite steps.

## Failure
Fails when applied to the term if none of the conversions in the net are applicable.

## See also
`GENERAL_REWRITE_CONV`, `GEN_REWRITE_CONV`, `mk_rewrites`, `net_of_conv`, `net_of_thm`, `REWRITE_CONV`.

# REWRITE_TAC

```
REWRITE_TAC : thm list -> tactic
```

## Synopsis
Rewrites a goal including built-in tautologies in the list of rewrites.

## Description
Rewriting tactics in HOL provide a recursive left-to-right matching and rewriting facility that automatically decomposes subgoals and justifies segments of proof in which equational theorems are used, singly or collectively. These include the unfolding of definitions, and the substitution of equals for equals. Rewriting is used either to advance or to complete the decomposition of subgoals.

`REWRITE_TAC` transforms (or solves) a goal by using as rewrite rules (i.e. as left-to-right replacement rules) the conclusions of the given list of (equational) theorems, as well as a set of built-in theorems (common tautologies) held in the ML variable `basic_rewrites`. Recognition of a tautology often terminates the subgoaling process (i.e. solves the goal).

The equational rewrites generated are applied recursively and to arbitrary depth, with matching and instantiation of variables and type variables. A list of rewrites can set off an infinite rewriting process, and it is not, of course, decidable in general whether a rewrite set has that property. The order in which the rewrite theorems are applied is unspecified, and the user should not depend on any ordering.

See `GEN_REWRITE_TAC` for more details on the rewriting process. Variants of `REWRITE_TAC` allow the use of a different set of rewrites. Some of them, such as `PURE_REWRITE_TAC`, exclude the basic tautologies from the possible transformations. ASM_REWRITE_TAC and others include the assumptions at the goal in the set of possible rewrites.

Still other tactics allow greater control over the search for rewritable subterms. Several of them such as ONCE_REWRITE_TAC do not apply rewrites recursively. GEN_REWRITE_TAC allows a rewrite to be applied at a particular subterm.

## Failure

REWRITE_TAC does not fail. Certain sets of rewriting theorems on certain goals may cause a non-terminating sequence of rewrites. Divergent rewriting behaviour results from a term t being immediately or eventually rewritten to a term containing t as a sub-term. The exact behaviour depends on the HOL implementation; it may be possible (unfortunately) to fall into Lisp in this event.

## Example

The arithmetic theorem GT, |- !n m. m > n <=> n < m, is used below to advance a goal:

```
# g ‘4 < 5‘;;
val it : goalstack = 1 subgoal (1 total)

‘4 < 5‘

# e(REWRITE_TAC[GT]);;
val it : goalstack = 1 subgoal (1 total)

‘4 < 5‘
```

It is used below with the theorem LT_0, |- !n. 0 < SUC n, to solve a goal:

```
# g ‘SUC n > 0‘;;
Warning: Free variables in goal: n
val it : goalstack = 1 subgoal (1 total)

‘SUC n > 0‘

# e(REWRITE_TAC[GT; LT_0]);;
val it : goalstack = No subgoals
```

## Uses

Rewriting is a powerful and general mechanism in HOL, and an important part of many proofs. It relieves the user of the burden of directing and justifying a large number of minor proof steps. REWRITE_TAC fits a forward proof sequence smoothly into the general goal-oriented framework. That is, (within one subgoaling step) it produces and justifies certain forward inferences, none of which are necessarily on a direct path to the desired goal.

REWRITE_TAC may be more powerful a tactic than is needed in certain situations; if efficiency is at stake, alternatives might be considered.

## See also

ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWR_CONV, REWRITE_CONV, SUBST_TAC.

# rhs

```
rhs : term -> term
```

## Synopsis

Returns the right-hand side of an equation.

## Description

`rhs ‘t1 = t2‘` returns `‘t2‘`.

## Failure

Fails with `rhs` if term is not an equality.

## Example

```
# rhs ‘2 + 2 = 4‘;;
val it : term = ‘4‘
```

## See also

dest_eq, lhs, rand.

# RIGHT_BETAS

```
RIGHT_BETAS : term list -> thm -> thm
```

## Synopsis

Apply and beta-reduce equational theorem with abstraction on RHS.

## Description

Given a list of arguments `[‘a1‘; ...; ‘an‘]` and a theorem of the form `A |- f = \x1 ... xn. t[x1,...` the rule `RIGHT_BETAS` returns `A |- f a1 ... an = t[a1,...,an]`. That is, it applies the theorem to the list of arguments and beta-reduces the right-hand side.

## Failure

Fails if the argument types are wrong or the RHS has too few abstractions.

## Example

```
# RIGHT_BETAS [`x:num`; `y:num`] (ASSUME `f = \a b c. a + b + c + 1`);;
val it : thm = f = (\a b c. a + b + c + 1) |- f x y = (\c. x + y + c + 1)
```

## See also

`BETA_CONV`, `BETAS_CONV`.

# rightbin

```
rightbin : ('a -> 'b * 'c) -> ('c -> 'd * 'a) -> ('d -> 'b -> 'b -> 'b) -> string -> 'a -> 'b *
```

## Synopsis

Parses iterated right-associated binary operator.

## Description

If `p` is a parser for "items" of some kind, `s` is a parser for some "separator", `c` is a 'constructor' function taking an element as parsed by `s` and two other elements as parsed by `p` and giving a new such element, and `e` is an error message, then `leftbin p s c e` will parse an iterated sequence of items by `p` and separated by something parsed with `s`. It will repeatedly apply the constructor function `c` to compose these elements into one, associating to the right. For example, the input:

```
<p1> <s1> <p2> <s2> <p3> <s3> <p4>
```

meaning successive segments `pi` that are parsed by `p` and `sj` that are parsed by `s`, will result in

```
c s1 c1 (c s2 p2 (c s3 p3 p4))
```

## Failure

The call `rightbin p s c e` never fails, though the resulting parser may.

## Comments

This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also

`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `some`.

# RING_AND_IDEAL_CONV

```
RING_AND_IDEAL_CONV : (term -> num) * (num -> term) * conv * term * term * term * term *
```

## Synopsis

Returns a pair giving a ring proof procedure and an ideal membership routine.

## Description

This function combines the functionality of `RING` and `ideal_cofactors`. Each of these requires the same rather lengthy input. When you want to apply both to the same set of parameters, you can do so using `RING_AND_IDEAL_CONV`. That is:

```
RING_AND_IDEAL_CONV parms
```

is functionally equivalent to:

```
RING parms,ideal_cofactors parms
```

For more information, see the documentation for those two functions.

## Failure

Fails if the parameters are wrong.

## See also

`ideal_cofactors`, `RING`.

# RING

```
RING : (term -> num) * (num -> term) * conv * term * term * term * term * term * term *
```

## Synopsis

Generic ring procedure.

## Description

The `RING` function takes a number of arguments specifying a ring structure and giving operations for computing and proving over it. Specifically the call is:

```
RING(toterm,tonum,EQ_CONV,
     neg,add,sub,inv,mul,div,pow,
     INTEGRAL_TH,FIELD_TH,POLY_CONV)
```

where `toterm` is a conversion from constant terms in the structure to rational numbers (e.g. `rat_of_term` for the reals), `tonum` is the opposite (e.g. `term_of_rat` for the reals), `EQ_CONV` is an equality test conversion (e.g. REAL_RAT_EQ_CONV), neg is negation, add is addition, sub is subtraction, inv is multiplicative inverse, div is division, pow is power, INTEGRAL_TH is an integrality theorem and FIELD_TH is a field theorem (see below) and POLY_CONV is a polynomial normalization theorem for the structure as returned by SEMIRING_NORMALIZERS_CONV (e.g. REAL_POLY_CONV for the reals).

The integrality theorem essentially states that if a product is zero, so is one of the factors (i.e. the structure is an integral domain), but this is stated in an unnatural way to allow application to structures without negation. It is permissible in this case to use boolean variables instead of operators such as negation and subtraction. The precise form of the theorem (notation for natural numbers, but this is supposed to be over the same structure):

```
|- (!x. 0 * x = 0) /\
   (!x y z. x + y = x + z <=> y = z) /\
   (!w x y z. w * y + x * z = w * z + x * y <=> w = x \/ y = z)
```

The field theorem is of the following form. It is not logically necessary, and if the structure is not a field you can just pass in TRUTH instead. However, it is usually beneficial for performance to include it.

```
|- !x y. ~(x = y) <=> ?z. (x - y) * z = 1
```

It returns a proof procedure that will attempt to prove a formula that, after suitable normalization, can be considered a universally quantified Boolean combination of equations and inequations between terms of the right type. If that formula holds in all integral domains, it will prove it. Any “alien” atomic formulas that are not natural number equations will not contribute to the proof.

## Failure

Fails if the theorems are malformed.

## Example

The instantiation for the real numbers (in fact this is already available under the name REAL_RING) could be coded as:

```
let REAL_RING =
  let REAL_INTEGRAL = prove
   (`(!x. &0 * x = &0) /\
     (!x y z. (x + y = x + z) <=> (y = z)) /\
     (!w x y z. (w * y + x * z = w * z + x * y) <=> (w = x) \/ (y = z))`,
    REWRITE_TAC[MULT_CLAUSES; EQ_ADD_LCANCEL] THEN
    REWRITE_TAC[GSYM REAL_OF_NUM_EQ;
                GSYM REAL_OF_NUM_ADD; GSYM REAL_OF_NUM_MUL] THEN
    ONCE_REWRITE_TAC[GSYM REAL_SUB_0] THEN
    REWRITE_TAC[GSYM REAL_ENTIRE] THEN REAL_ARITH_TAC)
  and REAL_INVERSE = prove
   (`!x y:real. ~(x = y) <=> ?z. (x - y) * z = &1`,
    REPEAT GEN_TAC THEN
    GEN_REWRITE_TAC (LAND_CONV o RAND_CONV) [GSYM REAL_SUB_0] THEN
    MESON_TAC[REAL_MUL_RINV; REAL_MUL_LZERO; REAL_ARITH `~(&1 = &0)`]) in
  RING(rat_of_term,term_of_rat,REAL_RAT_EQ_CONV,
       `(--):real->real`,`(+):real->real->real`,`(-):real->real->real`,
       `(inv):real->real`,`(*):real->real->real`,`(/):real->real->real`,
       `(pow):real->num->real`,
       REAL_INTEGRAL,REAL_INVERSE,REAL_POLY_CONV);;
```

after which, for example, we can verify a reduction for cubic equations to quadratics entirely automatically:

```
# REAL_RING
   `p = (&3 * a1 - a2 pow 2) / &3 /\
    q = (&9 * a1 * a2 - &27 * a0 - &2 * a2 pow 3) / &27 /\
    z = x - a2 / &3 /\
    x * w = w pow 2 - p / &3 /\
    ~(p = &0)
    ==> (z pow 3 + a2 * z pow 2 + a1 * z + a0 = &0 <=>
        (w pow 3) pow 2 - q * (w pow 3) - p pow 3 / &27 = &0)`;;
```

## See also

ideal_cofactors, NUM_RING, REAL_FIELD, REAL_RING, real_ideal_cofactors, RING_AND_IDEAL_CONV.

# rotate

```
rotate : int -> refinement
```

## Synopsis
Rotate a goalstate.

## Description
The function `rotate n gl` rotates a list `gl` of subgoals by `n` places. The function `r` is the special case where this modification is applied to the imperative variable of unproven subgoals.

## Failure
Fails only if the list of goals is empty.

## See also
`r`.

# RULE_ASSUM_TAC

```
RULE_ASSUM_TAC : (thm -> thm) -> tactic
```

## Synopsis
Maps an inference rule over the assumptions of a goal.

## Description
When applied to an inference rule `f` and a goal `({A1;...;An} ?- t)`, the `RULE_ASSUM_TAC` tactical applies the inference rule to each of the assumptions to give a new goal.

```
             {A1,...,An} ?- t
   ====================================  RULE_ASSUM_TAC f
    {f(.. |- A1),...,f(.. |- An)} ?- t
```

## Failure
The application of `RULE_ASSUM_TAC f` to a goal fails iff `f` fails when applied to any of the assumptions of the goal.

## Comments
It does not matter if the goal has no assumptions, but in this case `RULE_ASSUM_TAC` has no effect.

## See also
`ASSUM_LIST`, `MAP_EVERY`, `MAP_FIRST`, `POP_ASSUM_LIST`.

# SELECT_CONV

```
SELECT_CONV : term -> thm
```

## Synopsis
Eliminates an epsilon term by introducing an existential quantifier.

## Description
The conversion `SELECT_CONV` expects a boolean term of the form `` `P[@x.P[x]/x]` ``, which asserts that the epsilon term `@x.P[x]` denotes a value, `x` say, for which `P[x]` holds. This assertion is equivalent to saying that there exists such a value, and `SELECT_CONV` applied to a term of this form returns the theorem `|- P[@x.P[x]/x] = ?x. P[x]`.

## Failure
Fails if applied to a term that is not of the form `` `P[@x.P[x]/x]` ``.

## Example

```
# SELECT_CONV `(@n. n < m) < m`;;
val it : thm = |- (@n. n < m) < m <=> (?n. n < m)
```

## Uses
Particularly useful in conjunction with `CONV_TAC` for proving properties of values denoted by epsilon terms. For example, suppose that one wishes to prove the goal

```
# g `!m. 0 < m ==> (@n. n < m) < SUC m`;;
```

We start off:

```
# e(REPEAT STRIP_TAC THEN
    MATCH_MP_TAC(ARITH_RULE `!m n. m < n ==> m < SUC n`));;
val it : goalstack = 1 subgoal (1 total)

 0 [`0 < m`]

`(@n. n < m) < m`
```

This is now in the correct form for using `SELECT_CONV`:

```
# e(CONV_TAC SELECT_CONV);;
val it : goalstack = 1 subgoal (1 total)

 0 [`0 < m`]

`?n. n < m`
```

and the resulting subgoal is straightforward to prove, e.g. by `ASM_MESON_TAC[]` or `` EXISTS_TAC `0` THEN A ``

## See also
`SELECT_ELIM`, `SELECT_INTRO`, `SELECT_RULE`.

# SELECT_ELIM_TAC

```
SELECT_ELIM_TAC : tactic
```

## Synopsis
Eliminate select terms from a goal.

## Description
The tactic `SELECT_ELIM_TAC` attempts to remove from a goal any select terms, i.e. instances of the Hilbert choice operator `@x. P[x]`. First, any instances that occur inside their own predicate, i.e. `P[@x. P[x]]`, are replaced simply by `?x. P[x]`, as with `SELECT_CONV`. Other select-terms are eliminated by replacing each on with a new variable `v` and adding a corresponding instance of the axiom `SELECT_AX`, of the form `!x. P[x] ==> P[v]`. Note that the latter does not strictly preserve logical equivalence, only implication. So it is possible to replace a provable goal by an unprovable one. But since not much is provable about a select term except via the axiom `SELECT_AX`, this is not likely in practice.

## Failure
Never fails.

## Example
Suppose we set the goal:

```
# g `(@n. n < 3) < 3 /\ (@n. n < 3) < 5`;;
```

An application of `SELECT_ELIM_TAC` returns:

```
# e SELECT_ELIM_TAC;;
val it : goalstack = 1 subgoal (1 total)

`!_73133. (!x. x < 3 ==> _73133 < 3) ==> (?n. n < 3) /\ _73133 < 5`
```

## Uses
This is already applied as an initial normalization by `MESON` and other rules. Users may occasionally find it helpful.

## See also
`SELECT_CONV`.

# SELECT_RULE

```
SELECT_RULE : thm -> thm
```

## Synopsis
Introduces an epsilon term in place of an existential quantifier.

## Description
The inference rule `SELECT_RULE` expects a theorem asserting the existence of a value `x` such that `P` holds. The equivalent assertion that the epsilon term `@x.P` denotes a value of `x` for which `P` holds is returned as a theorem.

```
      A |- ?x. P
   ------------------  SELECT_RULE
    A |- P[(@x.P)/x]
```

## Failure
Fails if applied to a theorem the conclusion of which is not existentially quantified.

## Example
The axiom `INFINITY_AX` in the theory `ind` is of the form:

```
   |- ?f. ONE_ONE f /\ ~ONTO f
```

Applying `SELECT_RULE` to this theorem returns the following.

```
  # SELECT_RULE INFINITY_AX;;
  val it : thm =
    |- ONE_ONE (@f. ONE_ONE f /\ ~ONTO f) /\ ~ONTO (@f. ONE_ONE f /\ ~ONTO f)
```

## Uses
May be used to introduce an epsilon term to permit rewriting with a constant defined using the epsilon operator.

## See also
`CHOOSE`, `SELECT_AX`, `SELECT_CONV`.

# self_destruct

```
self_destruct : string
```

## Synopsis
Exits HOL Light but saves current state ready to restart.

## Description
This operation is only available in HOL images created using checkpointing (as in the default Linux build arising from `make all`), not when the HOL Light sources have simply been loaded

into the OCaml toplevel without checkpointing. A call `self_destruct s` will exit the current OCaml / HOL Light session, but save the current state to an image `hol.snapshot`. Users can then start this image; it will display the usual banner and also the string `s`, and the user will then be in the same state as before `self_destruct`.

## Failure

Never fails, except in the face of OS-level problems such as running out of disc space.

## Uses

Very useful to start HOL Light quickly with many background theories or tools loaded, rather than needing to rebuild them from sources.

## Comments

Unfortunately I do not know of any checkpointing tool that can give this behavior under Windows. However, it works very nicely in Linux.

## Example

Suppose that all the proofs you are doing at the moment need more theorems about prime numbers, and also a list of all prime numbers up to 1000. We reach a suitable state:

```
# needs "Examples/prime.ml";;
...
# let primes_1000 = rev(rev_itlist
    (fun q ps -> if exists (fun p -> q mod p = 0) ps then ps else q::ps)
    (2--1000) []);;
...
```

and now issue the checkpointing command:

```
self_destruct "Preloaded with prime number material";;
```

HOL Light will exit and a new file `hol.snapshot` will be created. You might want to rename it as `hol.prime` in the OS so it has a more intuitive name and doesn't get overwritten by later checkpoints

```
$ mv hol.snapshot hol.prime
```

You can then start the new image just by `hol.prime`:

```
$ hol.prime
        HOL Light 2.10, built 16 March 2006 on OCaml 3.08.3
        Preloaded with prime number material

val it : unit = ()
#
```

and continue where you left off, with all the prime-number material available instantly:

```
# PRIME_DIVPROD;;
val it : thm =
  |- !p a b. prime p /\ p divides a * b ==> p divides a \/ p divides b
# el 100 primes_1000;;
val it : int = 547
```

## See also

`startup_banner`.

# SEMIRING_NORMALIZERS_CONV

```
SEMIRING_NORMALIZERS_CONV : thm -> thm -> (term -> bool) * conv * conv * conv -> (term -
```

## Synopsis

Produces normalizer functions over a ring or even a semiring.

## Description

The function `SEMIRING_NORMALIZERS_CONV` should be given two theorems about some binary operators that we write as infix '`+`', '`*`' and '`^`' and ground terms '`ZERO`' and '`ONE`'. (The conventional symbols make the import of the theorem easier to grasp, but they are essentially arbitrary.) The first theorem is of the following form, essentially stating that the operators form a semiring structure with '^' as the "power" operator:

```
|- (!x y z. x + (y + z) = (x + y) + z) /\
   (!x y. x + y = y + x) /\
   (!x. ZERO + x = x) /\
   (!x y z. x * (y * z) = (x * y) * z) /\
   (!x y. x * y = y * x) /\
   (!x. ONE * x = x) /\
   (!x. ZERO * x = ZERO) /\
   (!x y z. x * (y + z) = x * y + x * z) /\
   (!x. x^0 = ONE) /\
   (!x n. x^(SUC n) = x * x^n)
```

The second theorem may just be `TRUTH = |- T`, in which case it will be assumed that the structure is just a semiring. Otherwise, it may be of the following form for "negation" (`neg`) and "subtraction" functions, plus a ground term `MINUS1` thought of as `-1`:

```
|- (!x. neg x = MINUS1 * x) /\
   (!x y. x - y = x + MINUS1 * y)
```

If the second theorem is provided, the eventual normalizer will also handle the negation and subtraction operations. Generally this is beneficial, but is impossible on structures like `:num` with no negative numbers.

The remaining arguments are a tuple. The first is an ordering on terms, used to determine the polynomial form. Normally, the default OCaml ordering is fine. The rest are intended to be functions for operating on 'constants' (e.g. numerals), which should handle at least '`ZERO`', '`ONE`' and, in the case of a ring, '`MINUS1`'. The functions are: (i) a test for membership in the set of 'constants', (ii) an addition conversion on constants, (iii) a multiplication conversion on constants, and (iv) a conversion to raise a constant to a numeral power. Note that no subtraction or negation operations are needed explicitly because this is subsumed in the presence of `-1` as a constant.

The function then returns conversions for putting terms of the structure into a canonical form, essentially multiplied-out polynomials with a particular ordering. The functions respectively negate, add, subtract, multiply, exponentiate terms already in the canonical form, putting the result back in canonical form. The final return value is an overall normalization function.

## Failure

Fails if the theorems are malformed.

## Example

There are already instantiations of the main normalizer for natural numbers (`NUM_NORMALIZE_CONV`) and real numbers (`REAL_POLY_CONV`). Here is how the latter is first constructed (it is later enhanced to handle some additional functions more effectively, so use the inbuilt definition, not this one):

```
# let REAL_POLY_NEG_CONV,REAL_POLY_ADD_CONV,REAL_POLY_SUB_CONV,
      REAL_POLY_MUL_CONV,REAL_POLY_POW_CONV,REAL_POLY_CONV =
    SEMIRING_NORMALIZERS_CONV REAL_POLY_CLAUSES REAL_POLY_NEG_CLAUSES
     (is_ratconst,
      REAL_RAT_ADD_CONV,REAL_RAT_MUL_CONV,REAL_RAT_POW_CONV)
     (<);;
val ( REAL_POLY_NEG_CONV ) : term -> thm = <fun>
val ( REAL_POLY_ADD_CONV ) : term -> thm = <fun>
val ( REAL_POLY_SUB_CONV ) : term -> thm = <fun>
val ( REAL_POLY_MUL_CONV ) : term -> thm = <fun>
val ( REAL_POLY_POW_CONV ) : term -> thm = <fun>
val ( REAL_POLY_CONV ) : term -> thm = <fun>
```

For examples of the resulting main function in action, see `REAL_POLY_CONV`.

## Uses

This is a highly generic function, intended only for occasional use by experts. Users reasoning in any sort of ring structure may find it a useful building-block for a decision procedure.

## Comments

This is a subcomponent of more powerful generic decision procedures such as `RING`. These can handle more sophisticated reasoning that direct equality through normalization.

## See also

`ideal_cofactors`, `NUM_NORMALIZE_CONV`, `REAL_POLY_CONV`, `RING_AND_IDEAL_CONV`.

# set_basic_congs

```
set_basic_congs : thm list -> unit
```

## Synopsis

Change the set of basic congruences used by the simplifier.

## Description

The HOL Light simplifier (as invoked by `SIMP_TAC` etc.) uses congruence rules to determine how it uses context when descending through a term. These are essentially theorems showing how to decompose one equality to a series of other inequalities in context. A call to `set_basic_congs thl` sets the congruence rules to the list of theorems `thl`.

## Failure

Never fails.

## Comments
Normally, users only need to extend the congruences; for an example of how to do that see `extend_basic_congs`.

## See also
`basic_congs`, `extend_basic_congs`, `SIMP_CONV`, `SIMP_RULE`, `SIMP_TAC`.

# set_basic_convs

```
set_basic_convs : (string * (term * conv)) list -> unit
```

## Synopsis
Assign the set of default conversions.

## Description
The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_` variants. The latter are normally term transformations that cannot be expressed as single (conditional or unconditional) rewrite rules. A call to `set_basic_convs l` where `l` is a list of items ("name",(‘pat‘,conv)) will make the default conversions just that set, using the name `name` to refer to each one and restricting it to subterms encountered that match `pat`.

## Failure
Never fails.

## Comments
Normally, users will only want to extend the existing set of conversions using `extend_basic_convs`.

## See also
`basic_convs`, `extend_basic_convs`, `set_basic_rewrites`, `REWRITE_TAC`, `SIMP_TAC`.

# set_basic_rewrites

```
set_basic_rewrites : thm list -> unit
```

## Synopsis
Assign the set of default rewrites used by rewriting and simplification.

## Description
The HOL Light rewriter (`REWRITE_TAC` etc.) and simplifier (`SIMP_TAC` etc.) have default sets of (conditional) equations and other conversions that are applied by default, except in the `PURE_`

variants. A call to `extend_basic_rewrites thl` sets this to be the list of theorems `thl` (after processing into rewrite rules by `mk_rewrites`).

### Failure
Never fails.

### Comments
Users will most likely want to extend the existing set by `extend_basic_rewrites` rather than completely change it like this.

### See also
`basic_rewrites, extend_basic_convs, set_basic_convs.`

# set_eq

```
set_eq : 'a list -> 'a list -> bool
```

### Synopsis
Tests two 'sets' for equality.

### Description
`set_eq l1 l2` returns `true` if every element of `l1` appears in `l2` and every element of `l2` appears in `l1`. Otherwise it returns `false`. In other words, it tests if the lists are the same considered as sets, i.e. ignoring duplicates.

### Failure
Never fails.

### Example

```
# set_eq [1;2] [2;1;2];;
val it : bool = true
# set_eq [1;2] [1;3];;
val it : bool = false
```

### See also
`setify, union, intersect, subtract.`

# set_goal

```
set_goal : term list * term -> goalstack
```

## Synopsis
Initializes the subgoal package with a new goal.

## Description
The function `set_goal` initializes the subgoal management package. A proof state of the package consists of either a goal stack and a justification stack if a proof is in progress, or a theorem if a proof has just been completed. `set_goal` sets a new proof state consisting of an empty justification stack and a goal stack with the given goal as its sole goal. The goal is printed.

## Failure
Fails unless all terms in the goal are of type `bool`.

## Example

```
# set_goal([], `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`);;
val it : goalstack = 1 subgoal (1 total)

`HD [1; 2; 3] = 1 /\ TL [1; 2; 3] = [2; 3]`
```

## Uses
Starting an interactive proof session with the subgoal package.

The subgoal package implements a simple framework for interactive goal-directed proof. When conducting a proof that involves many subgoals and tactics, the user must keep track of all the justifications and compose them in the correct order. While this is feasible even in large proofs, it is tedious. The subgoal package provides a way of building and traversing the tree of subgoals top-down, stacking the justifications and applying them properly.

The package maintains a proof state consisting of either a goal stack of outstanding goals and a justification stack, or a theorem. Tactics are used to expand the current goal (the one on the top of the goal stack) into subgoals and justifications. These are pushed onto the goal stack and justification stack, respectively, to form a new proof state. All preceding proof states are saved and can be returned to if a mistake is made in the proof. The goal stack is divided into levels, a new level being created each time a tactic is successfully applied to give new subgoals. The subgoals of the current level may be considered in any order.

## See also
`b, e, g, p, r, top_goal, top_thm.`

# setify

```
setify : 'a list -> 'a list
```

## Synopsis
Removes repeated elements from a list. Makes a list into a ‘set’.

## Description

`setify l` removes repeated elements from `l`, leaving the last occurrence of each duplicate in the list.

## Failure

Never fails.

## Example

```
# setify [1;2;3;1;4;3];;
val it : int list = [1; 2; 3; 4]
```

## Comments

The current implementation will in fact return a sorted list according to the basic OCaml polymorphic ordering.

## See also

`uniq`.

# SET_RULE

```
SET_RULE : term -> thm
```

## Synopsis

Attempt to prove elementary set-theoretic lemma.

## Description

The function `SET_RULE` is a crude automated prover for set-theoretic facts. When applied to a term, it expands various set-theoretic definitions explicitly and then attempts to solve the result using `MESON`.

## Failure

Fails if the simple translation does not suffice, or the resulting goal is too deep for `MESON`.

## Example

```
# SET_RULE `{x | ~(x IN s <=> x IN t)} = (s DIFF t) UNION (t DIFF s)`;;
...
val it : thm = |- {x | ~(x IN s <=> x IN t)} = s DIFF t UNION t DIFF s

# SET_RULE
   `UNIONS {t y | y IN x INSERT s} = t x UNION UNIONS {t y | y IN s}`;;
val it : thm =
  |- UNIONS {t y | y IN x INSERT s} = t x UNION UNIONS {t y | y IN s}
```

## See also

`MESON`, `MESON_TAC[]`, `SET_TAC`.

# SET_TAC

`SET_TAC : thm list -> tactic`

## Synopsis
Attempt to prove goal using basic set-theoretic reasoning.

## Description
When applied to a goal and a list of lemmas to use, the tactic `SET_TAC` puts the lemmas into the goal as antecedents, expands various set-theoretic definitions explicitly and then attempts to solve the result using `MESON`.

## Failure
Fails if the simple translation does not suffice, or the resulting goal is too deep for `MESON`.

## Example
Given the following goal:

```
# g `!s. (UNIONS s = {}) <=> !t. t IN s ==> t = {}`;;
```

the following solves it:

```
# e(SET_TAC[]);;
...
val it : goalstack = No subgoals
```

## See also
`MESON`, `MESON_TAC`, `SET_RULE`.

# shareout

`shareout : 'a list list -> 'b list -> 'b list list`

## Synopsis
Shares out the elements of the second list according to pattern in first.

## Description
The call `shareout pat l` shares out the elements of `l` into the same groups as the pattern list `pat`, while keeping them in the same order. If there are more elements in `l` than needed, they will be discarded, but if there are fewer, failure will occur.

## Failure
Fails if there are too few elements in the second list.

## Example

```
# shareout [[1;2;3]; [4;5]; [6]; [7;8;9]] (explode "abcdefghijklmnopq");;
val it : string list list =
  [["a"; "b"; "c"]; ["d"; "e"]; ["f"]; ["g"; "h"; "i"]]
```

## See also
`chop_list`.

# SIMP_CONV

```
SIMP_CONV : thm list -> conv
```

## Synopsis
Simplify a term repeatedly by conditional contextual rewriting.

## Description
A call `SIMP_CONV thl tm` will return `|- tm = tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`).

The theorems are first split up into individual rewrite rules, either conditional (`|- c ==> l = r`) or unconditional (`|- l = r`), as described in the documentation for `mk_rewrites`. These are then applied repeatedly to replace subterms in the goal that are instances `l'` of the left-hand side with a corresponding `r'`. Rewrite rules that are permutative, with each side an instance of the other, have an ordering imposed on them so that they tend to force terms into canonical form rather than looping (see `ORDERED_REWR_CONV`). In the case of applying a conditional rewrite, the condition `c` needs to be eliminated before the rewrite can be applied. This is attempted by recursively applying the same simplifications to `c` in the hope of reducing it to `T`. If this can be done, the conditional rewrite is applied, otherwise not. To add additional provers to dispose of side-conditions beyond application of the basic rewrites, see `mk_prover` and `ss_of_provers`.

## Failure
Never fails, but may return a reflexive theorem `|- tm = tm` if no simplifications can be made.

## Example
Here we use the conditional and contextual facilities:

```
# SIMP_CONV[SUB_ADD; ARITH_RULE `0 < n ==> 1 <= n`]
      `0 < n ==> (n - 1) + 1 = n + f(k + 1)`;;
val it : thm =
|- 0 < n ==> n - 1 + 1 = n + f (k + 1) <=> 0 < n ==> n = n + f (k + 1)
```

and here we show how a permutative rewrite achieves normalization (the same would work with plain REWRITE_CONV:

```
# REWRITE_CONV[ADD_AC] `(a + c + e) + ((b + a + d) + e):num`;;
val it : thm = |- (a + c + e) + (b + a + d) + e = a + a + b + c + d + e + e
```

## Comments

For simply rewriting with unconditional equations, REWRITE_CONV and relatives are simpler and more efficient.

## See also

ASM_SIMP_TAC, ONCE_SIMP_CONV, SIMP_RULE, SIMP_TAC.

# SIMPLE_CHOOSE

SIMPLE_CHOOSE : term -> thm -> thm

## Synopsis

Existentially quantifies a hypothesis of a theorem.

## Description

A call SIMPLE_CHOOSE `v` th existentially quantifies a hypothesis of the theorem over the variable v. It is intended for use when there is only one hypothesis so that the choice of assumption is unambiguous. In general, it picks the one that happens to be first in the list.

## Failure

Fails if v is not a variable or if it is free in the conclusion of the theorem th.

## Example

```
# let th = SYM(ASSUME `x:num = y`);;
val th : thm = x = y |- y = x
# SIMPLE_EXISTS `x:num` th;;
val it : thm = x = y |- ?x. y = x

# SIMPLE_CHOOSE `x:num` it;;
val it : thm = ?x. x = y |- ?x. y = x
```

## Comments

The more general function is CHOOSE, but this is simpler in the special case.

## See also

CHOOSE, EXISTS, SIMPLE_EXISTS.

# SIMPLE_DISJ_CASES

```
SIMPLE_DISJ_CASES : thm -> thm -> thm
```

## Synopsis

Disjoins hypotheses of two theorems with same conclusion.

## Description

The rule `SIMPLE_DISJ_CASES` takes two 'case' theorems with alpha-equivalent conclusions and returns a theorem with the first hypotheses disjoined:

```
      A u {p} |- r    B u {q} |- r
   ----------------------------------- SIMPLE_DISJ_CASES
          A u B u {p \/ q} |- r
```

To avoid dependency on the order of the hypotheses, it is only recommended when each theorem has exactly one hypothesis:

```
       {p} |- r    {q} |- r
   ----------------------------------- SIMPLE_DISJ_CASES
              {p \/ q} |- r
```

For more sophisticated or-elimination, use `DISJ_CASES`.

## Failure

Fails if the conclusions of the theorems are not alpha-equivalent.

## Example

```
# let [th1; th2] = map (UNDISCH o TAUT)
    [`~p ==> p ==> q`; `q ==> p ==> q`];;
...
val th1 : thm = ~p |- p ==> q
val th2 : thm = q |- p ==> q

# SIMPLE_DISJ_CASES th1 th2;;
val it : thm = ~p \/ q |- p ==> q
```

## See also

`DISJ_CASES`, `DISJ_CASES_TAC`, `DISJ_CASES_THEN`, `DISJ_CASES_THEN2`, `DISJ1`, `DISJ2`.

# SIMPLE_EXISTS

```
SIMPLE_EXISTS : term -> thm -> thm
```

## Synopsis
Introduces an existential quantifier over a variable in a theorem.

## Description
When applied to a pair consisting of a variable `v` and a theorem `|- p`, `SIMPLE_EXISTS` returns the theorem `|- ?v. p`. It is not compulsory for `v` to appear free in `p`, but otherwise the quantification is vacuous.

## Failure
Fails only if `v` is not a variable.

## Example

```
# SIMPLE_EXISTS `x:num` (REFL `x:num`);;
val it : thm = |- ?x. x = x
```

## Comments
The `EXISTS` function is more general: it can introduce an existentially quantified variable to replace chosen instances of any term in the theorem. However, `SIMPLE_EXISTS` is easier to use when the simple case is needed.

## See also
`CHOOSE`, `EXISTS`.

# SIMPLIFY_CONV

```
SIMPLIFY_CONV : simpset -> thm list -> conv
```

## Synopsis
General simplification at depth with arbitrary simpset.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset'. Given a simpset `ss` and an additional list of theorems `thl` to be used as (conditional or unconditional) rewrite rules, `SIMPLIFY_CONV ss thl` gives a simplification conversion with a repeated top-down traversal strategy (`TOP_DEPTH_SQCONV`) and a nesting limit of 3 for the recursive solution of subconditions by further simplification.

## Failure
Never fails.

## Uses
Usually some other interface to the simplifier is more convenient, but you may want to use this to employ a customized simpset.

## See also
GEN_SIMPLIFY_CONV, ONCE_SIMPLIFY_CONV, SIMP_CONV, SIMP_RULE, SIMP_TAC, TOP_DEPTH_SQCONV.

# SIMP_RULE

```
SIMP_RULE : thm list -> thm -> thm
```

## Synopsis
Simplify conclusion of a theorem repeatedly by conditional contextual rewriting.

## Description
A call `SIMP_CONV thl (|- tm)` will return `|- tm'` where `tm'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details on this kind of conditional rewriting, see `SIMP_CONV`.

## Failure
Never fails, but may return the input theorem unchanged if no simplifications were applicable.

## See also
ONCE_SIMP_RULE, SIMP_CONV, SIMP_TAC.

# SIMP_TAC

```
SIMP_TAC : thm list -> tactic
```

## Synopsis
Simplify a goal repeatedly by conditional contextual rewriting.

## Description
When applied to a goal `A ?- g`, the tactic `SIMP_TAC thl` returns a new goal `A ?- g'` where `g'` results from applying the theorems in `thl` as (conditional) rewrite rules, as well as built-in simplifications (see `basic_rewrites` and `basic_convs`). For more details, see `SIMP_CONV`.

## Failure
Never fails, though may not change the goal if no simplifications are applicable.

## Comments
To add the assumptions of the goal to the rewrites, use `ASM_SIMP_TAC` (or just `ASM SIMP_TAC`).

## See also
`ASM`, `ASM_SIMP_TAC`, `mk_rewrites`, `ONCE_SIMP_CONV`, `REWRITE_TAC`, `SIMP_CONV`, `SIMP_RULE`.

# |=>

```
(|=>) : 'a -> 'b -> ('a, 'b) func
```

## Synopsis
Gives a one-point finite partial function.

## Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The call `x |=> y` gives a finite partial function that maps `x` to `y` and is undefined for all arguments other than `x`.

## Example

```
# let f = (1 |=> 2);;
val f : (int, int) func = <func>

# apply f 1;;
val it : int = 2

# apply f 2;;
Exception: Failure "apply".
```

## See also
`|->`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`, `undefined`.

# SKOLEM_CONV

```
SKOLEM_CONV : conv
```

## Synopsis

Completely Skolemize a term already in negation normal form.

## Description

Skolemization amounts to rewriting with the equivalence

```
# SKOLEM_THM;;
val it : thm = |- !P. (!x. ?y. P x y) <=> (?y. !x. P x (y x))
```

The conversion `SKOLEM_CONV` will apply this transformation and pull out quantifiers to give a form with all existential quantifiers pulled to the outside. However, it assumes that the input is in negation normal form, i.e. built up by conjunction and disjunction from possibly negated atomic formulas.

## Failure

Never fails.

## Example

Here is a simple example:

```
# SKOLEM_CONV `(!x. ?y. P x y) \/ (?u. !v. ?z. P (f u v) z)`;;
Warning: inventing type variables
val it : thm =
  |- (!x. ?y. P x y) \/ (?u. !v. ?z. P (f u v) z) <=>
     (?y u z. (!x. P x (y x)) \/ (!v. P (f u v) (z v)))
```

However, note that it doesn't work properly when the input involves implication, and hence is not in NNF:

```
# SKOLEM_CONV `(!x. ?y. P x y) ==> (?u. !v. ?z. P (f u v) z)`;;
Warning: inventing type variables
 val it : thm =
  |- (!x. ?y. P x y) ==> (?u. !v. ?z. P (f u v) z) <=>
     (?y. !x. P x (y x)) ==> (?u z. !v. P (f u v) (z v))
```

## Uses

A useful component in decision procedures, to simplify the class of formulas that need to be considered. Used internally in several such procedures like `MESON_TAC`.

## See also

`NNF_CONV`, `NNFC_CONV`, `PRENEX_CONV`.

# some

```
some : ('a -> bool) -> 'a list -> 'a * 'a list
```

## Synopsis
Parses any single item satisfying a predicate.

## Description
If `p` is a predicate on input tokens of some kind, `some p` is a parser that parses and returns any first token satisying the predicate `p`, and raises `Noparse` on a first token not satisfying `p`.

## Failure
The call `some p` never fails.

## Comments
This is one of a suite of combinators for manipulating "parsers". A parser is simply a function whose OCaml type is some instance of `:('a)list -> 'b * ('a)list`. The function should take a list of objects of type `:'a` (e.g. characters or tokens), parse as much of it as possible from left to right, and return a pair consisting of the object derived from parsing (e.g. a term or a special syntax tree) and the list of elements that were not processed.

## See also
`++`, `||`, `>>`, `a`, `atleast`, `elistof`, `finished`, `fix`, `leftbin`, `listof`, `many`, `nothing`, `possibly`, `rightbin`.

# sort

```
sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

## Synopsis
Sorts a list using a given transitive 'ordering' relation.

## Description
The call

```
sort op list
```

where `op` is a transitive relation on the elements of `list`, will topologically sort the list, i.e. will permute it such that if `x op y` but not `y op x` then `x` will occur to the left of `y` in the sorted list. In particular if `op` is a total order, the list will be sorted in the usual sense of the word.

## Failure
Never fails.

## Example

A simple example is:

```
# sort (<) [3; 1; 4; 1; 5; 9; 2; 6; 5; 3; 5; 8; 9; 7; 9];;
val it : int list = [1; 1; 2; 3; 3; 4; 5; 5; 5; 6; 7; 8; 9; 9; 9]
```

The following example is a little more complicated, and shows how a topological sorting under the relation 'is free in' can be achieved. This is actually sometimes useful to consider subterms of a term in an appropriate order:

```
# sort free_in [`(x + 1) + 2`; `x + 2`; `x:num`; `x + 1`; `1`];;
val it : term list = [`1`; `x`; `x + 1`; `x + 2`; `(x + 1) + 2`]
```

## Comments

This function uses the Quicksort algorithm internally, which has good typical-case performance and will sort topologically. However, its worst-case performance is quadratic. By contrast `mergesort` gives a good worst-case performance but requires a total order. Note that any comparison-based topological sorting function must have quadratic behaviour in the worst case. For an $n$-element list, there are $n(n-1)/2$ pairs. For any topological sorting algorithm, we can make sure the first $n(n-1)/2 - 1$ pairs compared are unrelated in either direction, while still leaving the option of choosing for the last pair $(a, b)$ either $a < b$ or $b < a$, eventually giving a partial order. So at least $n(n-1)/2$ comparisons are needed to distinguish these two partial orders correctly.

## See also

`mergesort`.

# SPEC_ALL

```
SPEC_ALL : thm -> thm
```

## Synopsis

Specializes the conclusion of a theorem with its own quantified variables.

## Description

When applied to a theorem `A |- !x1...xn. t`, the inference rule `SPEC_ALL` returns the theorem `A |- t[x1'/x1]...[xn'/xn]` where the `xi'` are distinct variants of the corresponding `xi`, chosen to avoid clashes with any variables free in the assumption list. Normally `xi'` is just `xi`, in which case `SPEC_ALL` simply removes all universal quantifiers.

```
       A |- !x1...xn. t
   ---------------------------  SPEC_ALL
    A |- t[x1'/x1]...[xn'/xn]
```

## Failure

Never fails.

## Example

The following example shows how variables are also renamed to avoid clashing with those in assumptions.

```
# let th = ADD_ASSUM `m = 1` ADD_SYM;;
val th : thm = m = 1 |- !m n. m + n = n + m

# SPEC_ALL th;;
val it : thm = m = 1 |- m' + n = n + m'
```

## See also

GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

# SPEC

```
SPEC : term -> thm -> thm
```

## Synopsis

Specializes the conclusion of a theorem.

## Description

When applied to a term `u` and a theorem `A |- !x. t`, then `SPEC` returns the theorem `A |- t[u/x]`. If necessary, variables will be renamed prior to the specialization to ensure that `u` is free for `x` in `t`, that is, no variables free in `u` become bound after substitution.

```
    A |- !x. t
  --------------  SPEC `u`
   A |- t[u/x]
```

## Failure

Fails if the theorem's conclusion is not universally quantified, or if `x` and `u` have different types.

## Example

The following example shows how `SPEC` renames bound variables if necessary, prior to substitution: a straightforward substitution would result in the clearly invalid theorem `|- ~y ==> (!y. y ==> ~y)`.

```
# let xv = `x:bool` and yv = `y:bool` in
       (GEN xv o DISCH xv o GEN yv o DISCH yv) (ASSUME xv);;
val it : thm = |- !x. x ==> (!y. y ==> x)

# SPEC `~y` it;;
val it : thm = |- ~y ==> (!y'. y' ==> ~y)
```

## See also

GEN, GENL, GEN_ALL, ISPEC, ISPECL, SPECL, SPEC_ALL, SPEC_VAR.

# SPECL

```
SPECL : term list -> thm -> thm
```

## Synopsis

Specializes zero or more variables in the conclusion of a theorem.

## Description

When applied to a term list `[u1;...;un]` and a theorem `A |- !x1...xn. t`, the inference rule `SPECL` returns the theorem `A |- t[u1/x1]...[un/xn]`, where the substitutions are made sequentially left-to-right in the same way as for `SPEC`, with the same sort of alpha-conversions applied to `t` if necessary to ensure that no variables which are free in `ui` become bound after substitution.

```
       A |- !x1...xn. t
   --------------------------  SPECL [`u1`;...;`un`]
     A |- t[u1/x1]...[un/xn]
```

It is permissible for the term-list to be empty, in which case the application of `SPECL` has no effect.

## Failure

Fails unless each of the terms is of the same as that of the appropriate quantified variable in the original theorem.

## Example

The following is a specialization of a theorem from theory `arithmetic`.

```
  # let t = ARITH_RULE `!m n p q. m <= p /\ n <= q ==> (m + n) <= (p + q)`;;
  val t : thm = |- !m n p q. m <= p /\ n <= q ==> m + n <= p + q

  # SPECL [`1`; `2`; `3`; `4`] t;;;
  val it : thm = |- 1 <= 3 /\ 2 <= 4 ==> 1 + 2 <= 3 + 4
```

## See also

`GEN`, `GENL`, `GEN_ALL`, `GEN_TAC`, `SPEC`, `SPEC_ALL`, `SPEC_TAC`.

# SPEC_TAC

```
SPEC_TAC : term * term -> tactic
```

## Synopsis
Generalizes a goal.

## Description
When applied to a pair of terms (`u`,`x`), where x is just a variable, and a goal A ?- t, the tactic SPEC_TAC generalizes the goal to A ?- !x. t[x/u], that is, all (free) instances of u are turned into x.

```
     A ?- t
================  SPEC_TAC (`u`,`x`)
 A ?- !x. t[x/u]
```

## Failure
Fails unless x is a variable with the same type as u.

## Uses
Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

## See also
GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, STRIP_TAC.

# SPEC_VAR

```
SPEC_VAR : thm -> term * thm
```

## Synopsis
Specializes the conclusion of a theorem, returning the chosen variant.

## Description
When applied to a theorem A |- !x. t, the inference rule SPEC_VAR returns the term x' and the theorem A |- t[x'/x], where x' is a variant of x chosen to avoid clashing with free variables in assumptions.

```
  A |- !x. t
--------------  SPEC_VAR
 A |- t[x'/x]
```

## Failure
Fails unless the theorem's conclusion is universally quantified.

## Example

Note how the variable is renamed to avoid the free m in the assumptions:

```
#  let th = ADD_ASSUM `m = 1` ADD_SYM;;
val th : thm = m = 1 |- !m n. m + n = n + m

# SPEC_VAR th;;
val it : term * thm = (`m'`, m = 1 |- !n. m' + n = n + m')
```

## See also

GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL.

# splitlist

```
splitlist : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
```

## Synopsis

Applies a binary destructor repeatedly in left-associative mode.

## Description

If a destructor function d inverts a binary constructor f, for example dest_comb for mk_comb, and fails when applied to y, then:

```
splitlist d (f(x1,f(x2,f(...f(xn,y)))))
```

returns

```
([x1; ... ; xn],y)
```

## Failure

Never fails.

## Example

The function strip_forall is actually just defined as splitlist dest_forall, which acts as follows:

```
# splitlist dest_forall `!x y z. x + y = z`;;
val it : term list * term = ([`x`; `y`; `z`], `x + y = z`)
```

## See also

itlist, rev_splitlist, striplist.

# ss_of_congs

```
ss_of_congs : thm list -> simpset -> simpset
```

## Synopsis
Add congruence rules to a simpset.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. A call `ss_of_congs thl ss` adds `thl` as new congruence rules to the simpset `ss` to yield a new simpset. For an illustration of how congruence rules can be used, see `extend_basic_congs`.

## Failure
Never fails unless the congruence rules are malformed.

## See also
`mk_rewrites`, `SIMP_CONV`, `ss_of_conv`, `ss_of_maker`, `ss_of_prover`, `ss_of_provers`, `ss_of_thms`.

# ss_of_conv

```
ss_of_conv : term -> conv -> simpset -> simpset
```

## Synopsis
Add a new conversion to a simpset.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. A call `ss_of_conv pat cnv ss` adds the conversion `cnv` to the simpset `ss` to yield a new simpset, restricting the initial filtering of potential subterms to those matching `pat`.

## Failure
Never fails.

## Example

```
# ss_of_conv `x + y:num` NUM_ADD_CONV empty_ss;;
...
```

## See also

mk_rewrites, SIMP_CONV, ss_of_congs, ss_of_maker, ss_of_prover, ss_of_provers, ss_of_thms.

# ss_of_maker

`ss_of_maker : (thm -> thm list -> thm list) -> simpset -> simpset`

## Synopsis

Change the rewrite maker in a simpset.

## Description

In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. A call `ss_of_maker maker ss` changes the "rewrite maker" in `ss` to yield a new simpset; use of this simpset with additional theorems will process those theorems using the new rewrite maker. The default rewrite maker is `mk_rewrites` with an appropriate flag, and it is unusual to want to change it.

## Failure

Never fails.

## See also

mk_rewrites, SIMP_CONV, ss_of_congs, ss_of_conv, ss_of_prover, ss_of_provers, ss_of_thms.

# ss_of_prover

`ss_of_prover : (strategy -> strategy) -> simpset -> simpset`

## Synopsis

Change the method of prover application in a simpset.

## Description

In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. The default 'prover use' method is to first recursively apply all the simplification to conditions and then try the provers, if any, one by one until one succeeds. It is unusual to want to change this, but if desired you can do it with `ss_of_prover str ss`.

## Failure

Never fails.

## See also

`mk_rewrites`, `SIMP_CONV`, `ss_of_congs`, `ss_of_conv`, `ss_of_maker`, `ss_of_provers`, `ss_of_thms`.

# ss_of_provers

```
ss_of_provers : prover list -> simpset -> simpset
```

## Synopsis

Add new provers to a simpset.

## Description

In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. A call `ss_of_provers prs ss` adds the provers in `prs` to the simpset `ss` to yield a new simpset. See `mk_prover` for more explanation of how to create something of type `prover`.

## Failure

Never fails.

## See also

`mk_prover`, `mk_rewrites`, `SIMP_CONV`, `ss_of_congs`, `ss_of_conv`, `ss_of_maker`, `ss_of_prover`, `ss_of_thms`.

# ss_of_thms

```
ss_of_thms : thm list -> simpset -> simpset
```

## Synopsis
Add theorems to a simpset.

## Description
In their maximal generality, simplification operations in HOL Light (as invoked by `SIMP_TAC`) are controlled by a 'simpset', which may contain conditional and unconditional rewrite rules, conversions and provers for conditions, as well as a determination of how to use the prover on the conditions and how to process theorems into rewrites. A call `ss_of_thms thl ss` processes the theorems `thl` according to the rewrite maker in the simpset `ss` (normally `mk_rewrites`) and adds them to the theorems in `ss` to yield a new simpset.

## Failure
Never fails.

## Example

```
# ss_of_thms [ADD_CLAUSES] empty_ss;;
...
```

## See also
`mk_rewrites`, `SIMP_CONV`, `ss_of_congs`, `ss_of_conv`, `ss_of_maker`, `ss_of_prover`, `ss_of_provers`.

# startup_banner

```
startup_banner : string
```

## Synopsis
The one-line startup banner for HOL Light.

## Description
This string is the startup banner for HOL Light, and is displayed when standalone images (see `self_destruct`) are started up. It is only available in HOL images created using checkpointing (as in the default Linux build arising from `make all`), not when the HOL Light sources have simply been loaded into the OCaml toplevel without checkpointing.

## Failure
Not applicable.

## Example

On my home computer, the value is currently:

```
# startup_banner;;
val it : string =
  "        HOL Light 2.10, built 16 March 2006 on OCaml 3.08.3"
```

## See also

self_destruct.

# string_of_term

```
string_of_term : term -> string
```

## Synopsis

Converts a HOL term to a string representation.

## Description

The call `string_of_term tm` produces a textual representation of the term `tm` as a string, similar to what is printed automatically at the toplevel, though without the surrounding quotes.

## Failure

Never fails.

## Example

```
# string_of_term `x + 1 < 2 <=> x = 0`;;
val it : string = "x + 1 < 2 <=> x = 0"
```

## Comments

The string may contain newlines for large terms, broken in a similar fashion to automatic printing.

## See also

string_of_thm, string_of_type.

# string_of_thm

```
string_of_thm : thm -> string
```

## Synopsis
Converts a HOL theorem to a string representation.

## Description
The call `string_of_thm th` produces a textual representation of the theorem `th` as a string, similar to what is printed automatically at the toplevel.

## Failure
Never fails.

## Example

```
# string_of_thm ADD_CLAUSES;;
val it : string =
  "|- (!n. 0 + n = n) /\\\n   (!m. m + 0 = m) /\\\n   (!m n. SUC m + n = SUC (m
+ n)) /\\\n   (!m n. m + SUC n = SUC (m + n))"

# print_string it;;
|- (!n. 0 + n = n) /\
   (!m. m + 0 = m) /\
   (!m n. SUC m + n = SUC (m + n)) /\
   (!m n. m + SUC n = SUC (m + n))
val it : unit = ()
```

## Comments
The string may contain newlines for large terms, broken in a similar fashion to automatic printing.

## See also
`string_of_thm`, `string_of_type`.

# string_of_type

```
string_of_type : hol_type -> string
```

## Synopsis
Converts a HOL type to a string representation.

## Description
The call `string_of_type ty` produces a textual representation of the type `ty` as a string, similar to what is printed automatically at the toplevel, though without the surrounding quotes and colon.

## Failure
Never fails.

## Example

```
# string_of_type bool_ty;;
val it : string = "bool"
```

## See also
string_of_term, string_of_thm.

# strip_abs

```
strip_abs : term -> term list * term
```

## Synopsis
Iteratively breaks apart abstractions.

## Description
strip_abs `\x1 ... xn. t` returns ([`x1`;...;`xn`],`t`). Note that

```
strip_abs(list_mk_abs([`x1`;...;`xn`],`t`))
```

will not return ([`x1`;...;`xn`],`t`) if t is an abstraction.

## Failure
Never fails.

## Example

```
# strip_abs `\x y z. x /\ y /\ z`;;
val it : term list * term = ([`x`; `y`; `z`], `x /\ y /\ z`)
```

## See also
list_mk_abs, dest_abs.

# STRIP_ASSUME_TAC

```
STRIP_ASSUME_TAC : thm_tactic
```

## Synopsis

Splits a theorem into a list of theorems and then adds them to the assumptions.

## Description

Given a theorem `th` and a goal `(A,t)`, `STRIP_ASSUME_TAC th` splits `th` into a list of theorems. This is done by recursively breaking conjunctions into separate conjuncts, cases-splitting disjunctions, and eliminating existential quantifiers by choosing arbitrary variables. Schematically, the following rules are applied:

```
        A ?- t
======================  STRIP_ASSUME_TAC (A' |- v1 /\ ... /\ vn)
 A u {v1,...,vn} ?- t

             A ?- t
=================================  STRIP_ASSUME_TAC (A' |- v1 \/ ... \/ vn)
 A u {v1} ?- t ... A u {vn} ?- t

       A ?- t
====================  STRIP_ASSUME_TAC (A' |- ?x.v)
 A u {v[x'/x]} ?- t
```

where `x'` is a variant of `x`.

If the conclusion of `th` is not a conjunction, a disjunction or an existentially quantified term, the whole theorem `th` is added to the assumptions.

As assumptions are generated, they are examined to see if they solve the goal (either by being alpha-equivalent to the conclusion of the goal or by deriving a contradiction).

The assumptions of the theorem being split are not added to the assumptions of the goal(s), but they are recorded in the proof. This means that if `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `STRIP_ASSUME_TAC (A' |- v)` results in an invalid tactic.

## Failure

Never fails.

## Example

When solving the goal

```
# g `m = 0 + m`;;
```

assuming the clauses for addition with `STRIP_ASSUME_TAC ADD_CLAUSES` results in the goal

```
# e(STRIP_ASSUME_TAC ADD_CLAUSES);;
val it : goalstack = 1 subgoal (1 total)

 0 [`!n. 0 + n = n`]
 1 [`!m. m + 0 = m`]
 2 [`!m n. SUC m + n = SUC (m + n)`]
 3 [`!m n. m + SUC n = SUC (m + n)`]

`m = 0 + m`
```

while the same tactic directly solves the goal

```
 ?- !m. 0 + m = m
```

## Uses

`STRIP_ASSUME_TAC` is used when applying a previously proved theorem to solve a goal, or when enriching its assumptions so that rewriting with assumptions and other operations involving assumptions have more to work with.

## See also

`ASSUME_TAC`, `CHOOSE_TAC`, `CHOOSE_THEN`, `CONJUNCTS_THEN`, `DISJ_CASES_TAC`, `DISJ_CASES_THEN`.

# strip_comb

```
strip_comb : term -> term * term list
```

## Synopsis

Iteratively breaks apart combinations (function applications).

## Description

`strip_comb `t t1 ... tn`` returns `(`t`,[`t1`;...;`tn`])`. Note that

```
strip_comb(list_mk_comb(`t`,[`t1`;...;`tn`]))
```

will not return `(`t`,[`t1`;...;`tn`])` if `t` is a combination.

## Failure
Never fails.

## Example

```
# strip_comb `x /\ y`;;
val it : term * term list = (`(/\)`, [`x`; `y`])

# strip_comb `T`;;
val it : term * term list = (`T`, [])
```

## See also
dest_comb, list_mk_comb, splitlist, striplist.

# strip_exists

```
strip_exists : term -> term list * term
```

## Synopsis
Iteratively breaks apart existential quantifications.

## Description
`strip_exists `?x1 ... xn. t`` returns `([`x1`;...;`xn`],`t`)`. Note that

```
strip_exists(list_mk_exists([`x1`;...;`xn`],`t`))
```

will not return `([`x1`;...;`xn`],`t`)` if `t` is an existential quantification.

## Failure
Never fails.

## See also
dest_exists, list_mk_exists.

# strip_forall

```
strip_forall : term -> term list * term
```

## Synopsis
Iteratively breaks apart universal quantifications.

## Description

strip_forall `!x1 ... xn. t` returns ([`x1`;...;`xn`],`t`). Note that

```
strip_forall(list_mk_forall([`x1`;...;`xn`],`t`))
```

will not return ([`x1`;...;`xn`],`t`) if t is a universal quantification.

## Failure

Never fails.

## See also

dest_forall, list_mk_forall.

# strip_gabs

```
strip_gabs : term -> term list * term
```

## Synopsis

Breaks apart an iterated generalized or basic abstraction.

## Description

If the term t is iteratively constructed by basic or generalized abstractions, i.e. is of the form \vs1. \vs2. ... \vsn. t, then the call strip_gabs t returns a pair of the list of varstructs and the term [vs1; vs2; ...; vsn],t.

## Failure

Never fails, though the list of varstructs will be empty if the initial term is no sort of abstraction.

## Example

```
# strip_gabs `\(a,b) c ((d,e),f). (a - b) + c + (d - e) * f`;;
val it : term list * term =
  ([`a,b`; `c`; `(d,e),f`], `a - b + c + (d - e) * f`)
```

## See also

dest_gabs, is_gabs, mk_gabs.

# STRIP_GOAL_THEN

```
STRIP_GOAL_THEN : thm_tactic -> tactic
```

## Synopsis

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

## Description

Given a theorem-tactic `ttac` and a goal `(A,t)`, `STRIP_GOAL_THEN` removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`. If `t` is a universally quantified term, then `STRIP_GOAL_THEN` strips off the quantifier:

```
      A ?- !x.u
   ==============  STRIP_GOAL_THEN ttac
     A ?- u[x'/x]
```

where `x'` is a primed variant that does not appear free in the assumptions `A`. If `t` is a conjunction, then `STRIP_GOAL_THEN` simply splits the conjunction into two subgoals:

```
      A ?- v /\ w
   =================  STRIP_GOAL_THEN ttac
    A ?- v   A ?- w
```

If `t` is an implication `"u ==> v"` and if:

```
      A ?- v
  ===============  ttac (u |- u)
     A' ?- v'
```

then:

```
      A ?- u ==> v
  ====================  STRIP_GOAL_THEN ttac
        A' ?- v'
```

Finally, a negation `~t` is treated as the implication `t ==> F`.

## Failure

`STRIP_GOAL_THEN ttac (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction. Failure also occurs if the application of `ttac` fails, after stripping the goal.

## Example

When solving the goal

```
# g `n = 1 ==> n * n = n`;;
Warning: Free variables in goal: n
val it : goalstack = 1 subgoal (1 total)

`n = 1 ==> n * n = n`
```

a possible initial step is to apply

```
# e(STRIP_GOAL_THEN SUBST1_TAC);;
val it : goalstack = 1 subgoal (1 total)

`1 * 1 = 1`
```

which is immediate by `ARITH_TAC`, for example.

## Uses

`STRIP_GOAL_THEN` is used when manipulating intermediate results (obtained by stripping outer connectives from a goal) directly, rather than as assumptions.

## See also

`CONJ_TAC`, `DISCH_THEN`, `GEN_TAC`, `STRIP_ASSUME_TAC`, `STRIP_TAC`.

# striplist

```
striplist : ('a -> 'a * 'a) -> 'a -> 'a list
```

## Synopsis

Applies a binary destructor repeatedly, flattening the construction tree into a list.

## Description

If a destructor function `d` inverts a binary constructor `f`, for example `dest_comb` for `mk_comb`, and fails when applied to components `xi`, then when applied to any object built up repeatedly by `f` applied to base values `xi` returns the list `[x1;...;xn]`.

## Failure

Never fails.

## Example

```
# striplist dest_conj `(a /\ (b /\ ((c /\ d) /\ e)) /\ f) /\ g`;;
val it : term list = [`a`; `b`; `c`; `d`; `e`; `f`; `g`]
```

## See also

`splitlist`, `rev_splitlist`, `end_itlist`.

# strip_ncomb

```
strip_ncomb : int -> term -> term * term list
```

## Synopsis
Strip away a given number of arguments from a combination.

## Description
Given a number `n` and a combination term `` `f a1 ... an` ``, the function `strip_ncomb` returns the result of stripping away exactly `n` arguments: the pair `` `f`,[`a1`;...;`an`] ``. Note that exactly `n` arguments are stripped even if `f` is a combination.

## Failure
Fails if there are not `n` arguments to strip off.

## Example
Note how the behaviour is more limited compared with simple `strip_comb`:

```
# strip_ncomb 2 `f u v x y z`;;
Warning: inventing type variables
val it : term * term list = (`f u v x`, [`y`; `z`])

# strip_comb `f u v x y z`;;
Warning: inventing type variables
val it : term * term list = (`f`, [`u`; `v`; `x`; `y`; `z`])
```

## Uses
Delicate term decompositions.

## See also
`strip_comb`.

# STRIP_TAC

```
STRIP_TAC : tactic
```

## Synopsis
Splits a goal by eliminating one outermost connective.

## Description

Given a goal (A,t), STRIP_TAC removes one outermost occurrence of one of the connectives !, ==>, ~ or /\ from the conclusion of the goal t. If t is a universally quantified term, then STRIP_TAC strips off the quantifier:

```
      A ?- !x.u
   ==============  STRIP_TAC
     A ?- u[x'/x]
```

where x' is a primed variant that does not appear free in the assumptions A. If t is a conjunction, then STRIP_TAC simply splits the conjunction into two subgoals:

```
      A ?- v /\ w
   =================  STRIP_TAC
    A ?- v   A ?- w
```

If t is an implication, STRIP_TAC moves the antecedent into the assumptions, stripping conjunctions, disjunctions and existential quantifiers according to the following rules:

```
    A ?- v1 /\ ... /\ vn ==> v            A ?- v1 \/ ... \/ vn ==> v
   ============================        =================================
       A u {v1,...,vn} ?- v             A u {v1} ?- v ... A u {vn} ?- v

     A ?- ?x.w ==> v
   ====================
    A u {w[x'/x]} ?- v
```

where x' is a primed variant of x that does not appear free in A. Finally, a negation ~t is treated as the implication t ==> F.

## Failure

STRIP_TAC (A,t) fails if t is not a universally quantified term, an implication, a negation or a conjunction.

## Example

Starting with the goal:

```
  # g `!m n. m <= n /\ n <= m ==> m = n`;;
```

the repeated application of STRIP_TAC strips off the universal quantifiers, breaks apart the antecedent and adds the conjuncts to the hypotheses:

```
# e(REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`m <= n`]
 1 [`n <= m`]

`m = n`
```

## Uses

When trying to solve a goal, often the best thing to do first is `REPEAT STRIP_TAC` to split the goal up into manageable pieces.

## See also

`CONJ_TAC`, `DISCH_TAC`, `DISCH_THEN`, `GEN_TAC`, `STRIP_ASSUME_TAC`, `STRIP_GOAL_THEN`.

# STRIP_THM_THEN

```
STRIP_THM_THEN : thm_tactical
```

## Synopsis

`STRIP_THM_THEN` applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

## Description

Given a theorem-tactic `ttac`, a theorem `th` whose conclusion is a conjunction, a disjunction or an existentially quantified term, and a goal `(A,t)`, `STRIP_THM_THEN ttac th` first strips apart the conclusion of `th`, next applies `ttac` to the theorem(s) resulting from the stripping and then applies the resulting tactic to the goal.

In particular, when stripping a conjunctive theorem `A' |- u /\ v`, the tactic

```
ttac(u |- u) THEN ttac(v |- v)
```

resulting from applying `ttac` to the conjuncts, is applied to the goal. When stripping a disjunctive theorem `A' |- u \/ v`, the tactics resulting from applying `ttac` to the disjuncts, are applied to split the goal into two cases. That is if

```
 A ?- t                          A ?- t
=========  ttac (u |- u)    and   =========  ttac (v |- v)
 A ?- t1                         A ?- t2
```

then:

```
      A ?- t
==================  STRIP_THM_THEN ttac (A' |- u \/ v)
 A ?- t1  A ?- t2
```

When stripping an existentially quantified theorem `A' |- ?x.u`, the tactic `ttac(u |- u)`, resulting from applying `ttac` to the body of the existential quantification, is applied to the goal. That is, if:

```
 A ?- t
=========  ttac (u |- u)
 A ?- t1
```

then:

```
   A ?- t
=============  STRIP_THM_THEN ttac (A' |- ?x. u)
   A ?- t1
```

The assumptions of the theorem being split are not added to the assumptions of the goal(s) but are recorded in the proof. If `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `STRIP_THM_THEN ttac th` results in an invalid tactic.

## Failure

`STRIP_THM_THEN ttac th` fails if the conclusion of `th` is not a conjunction, a disjunction or an existentially quantified term. Failure also occurs if the application of `ttac` fails, after stripping the outer connective from the conclusion of `th`.

## Uses

`STRIP_THM_THEN` is used enrich the assumptions of a goal with a stripped version of a previously-proved theorem.

## See also

CHOOSE_THEN, CONJUNCTS_THEN, DISJ_CASES_THEN, STRIP_ASSUME_TAC.

# STRUCT_CASES_TAC

```
STRUCT_CASES_TAC : thm_tactic
```

## Synopsis

Performs very general structural case analysis.

## Description

When it is applied to a theorem of the form:

```
th = A' |- ?y11...y1m. x = t1 /\ (B11 /\ ... /\ B1k) \/ ... \/
             ?yn1...ynp. x = tn /\ (Bn1 /\ ... /\ Bnp)
```

in which there may be no existential quantifiers where a 'vector' of them is shown above, `STRUCT_CASES_TAC th` splits a goal `A ?- s` into `n` subgoals as follows:

```
                            A ?- s
================================================================
 A u {B11,...,B1k} ?- s[t1/x] ... A u {Bn1,...,Bnp} ?- s[tn/x]
```

that is, performs a case split over the possible constructions (the `ti`) of a term, providing as assumptions the given constraints, having split conjoined constraints into separate assumptions. Note that unless `A'` is a subset of `A`, this is an invalid tactic.

## Failure

Fails unless the theorem has the above form, namely a conjunction of (possibly multiply existentially quantified) terms which assert the equality of the same variable `x` and the given terms.

## Example

Suppose we have the goal:

```
# g `~(l:(A)list = []) ==> LENGTH l > 0`;;
```

then we can get rid of the universal quantifier from the inbuilt list theorem `list_CASES`:

```
list_CASES = !l. l = [] \/ (?h t. l = CONS h t)
```

and then use `STRUCT_CASES_TAC`. This amounts to applying the following tactic:

```
# e(STRUCT_CASES_TAC (SPEC_ALL list_CASES));;
val it : goalstack = 2 subgoals (2 total)

`~(CONS h t = []) ==> LENGTH (CONS h t) > 0`

`~([] = []) ==> LENGTH [] > 0`
```

and both of these are solvable by `REWRITE_TAC[GT; LENGTH; LT_0]`.

## Uses

Generating a case split from the axioms specifying a structure.

## See also

`ASM_CASES_TAC`, `BOOL_CASES_TAC`, `COND_CASES_TAC`, `DISJ_CASES_TAC`.

# SUB_CONV

SUB_CONV : conv -> conv

## Synopsis
Applies a conversion to the top-level subterms of a term.

## Description
For any conversion c, the function returned by SUB_CONV c is a conversion that applies c to all the top-level subterms of a term. If the conversion c maps t to |- t = t', then SUB_CONV c maps an abstraction `\x. t` to the theorem:

```
|- (\x. t) = (\x. t')
```

That is, SUB_CONV c `\x. t` applies c to the body of the abstraction `\x. t`. If c is a conversion that maps `t1` to the theorem |- t1 = t1' and `t2` to the theorem |- t2 = t2', then the conversion SUB_CONV c maps an application `t1 t2` to the theorem:

```
|- (t1 t2) = (t1' t2')
```

That is, SUB_CONV c `t1 t2` applies c to the both the operator t1 and the operand t2 of the application `t1 t2`. Finally, for any conversion c, the function returned by SUB_CONV c acts as the identity conversion on variables and constants. That is, if `t` is a variable or constant, then SUB_CONV c `t` returns |- t = t.

## Failure
SUB_CONV c tm fails if tm is an abstraction `\x. t` and the conversion c fails when applied to t, or if tm is an application `t1 t2` and the conversion c fails when applied to either t1 or t2. The function returned by SUB_CONV c may also fail if the ML function c is not, in fact, a conversion (i.e. a function that maps a term t to a theorem |- t = t').

## See also
ABS_CONV, COMB_CONV, RAND_CONV, RATOR_CONV.

# SUBGOAL_TAC

SUBGOAL_TAC : string -> term -> tactic list -> tactic

## Synopsis
Encloses the sub-proof of a named lemma.

## Description

The call `SUBGOAL_TAC "name" ‘t‘ [tac]` introduces a new subgoal `t` with the current assumptions, runs on that subgoal the tactic `tac`, and attaches the resuly as a new hypothesis called `name` in the current subgoal. The `[tac]` argument is always a one-element list, for stylistic reasons. If `tac` does not prove the goal, any resulting subgoals from it will appear first.

## Failure

Fails if `t` is not Boolean or if `tac` fails on it.

## Example

If we want to prove

```
# g ‘(n MOD 2) IN {0,1}‘;;
```

we might start by establishing a lemma:

```
# e(SUBGOAL_TAC "ml2" ‘n MOD 2 < 2‘ [SIMP_TAC[DIVISION; ARITH]]);;
val it : goalstack = 1 subgoal (1 total)

 0 [‘n MOD 2 < 2‘] (ml2)

‘n MOD 2 IN {0, 1}‘
```

after which, for example, we could finish things with

```
# e(REWRITE_TAC[IN_INSERT; NOT_IN_EMPTY] THEN
    POP_ASSUM MP_TAC THEN ARITH_TAC);;
val it : goalstack = No subgoals
```

## Uses

Structuring proofs via a sequence of simple lemmas.

## See also

`SUBGOAL_THEN`.

# SUBGOAL_THEN

```
SUBGOAL_THEN : term -> thm_tactic -> tactic
```

## Synopsis

Introduces a lemma as a new subgoal.

## Description

The user proposes a lemma and is then invited to prove it under the current assumptions. The lemma is then used with the `thm_tactic` to apply to the goal. That is, if

```
  A1 ?- t1
==========  ttac (t |- t)
  A2 ?- t2
```

then

```
      A1 ?- t1
====================  SUBGOAL_THEN `t` ttac
 A1 ?- t   A2 ?- t2
```

In the quite common special case where `ttac` is `ASSUME_TAC`, the net behaviour is simply to present the user with two subgoals, one in which the lemma is to be proved and one in which it may be assumed:

```
      A1 ?- t1
============================  SUBGOAL_THEN `t` ASSUME_TAC
 A1 ?- t   A1 u {t} ?- t2
```

## Failure

`SUBGOAL_THEN` will fail if an attempt is made to use a non-boolean term as a lemma.

## Uses

Introducing lemmas into the same basic proof script without separately binding them to names. This is often a good structuring technique for large tactic proofs, where separate lemmas might look artificial because of all the ad-hoc context in which they occur.

## Example

Consider the proof of the Knaster-Tarski fixpoint theorem, to be found in `Examples/card.ml`. This (in its set-lattice context) states that every monotonic function has a fixpoint. After setting the initial goal:

```
# g `!f. (!s t. s SUBSET t ==> f(s) SUBSET f(t)) ==> ?s:A->bool. f(s) = s`;;
```

we start off the proof, already proceeding via a series of lemmas with `SUBGOAL_THEN`, though we will focus our attention on a later one:

```
# e(REPEAT STRIP_TAC THEN MAP_EVERY ABBREV_TAC
     [`Y = {b:A->bool | f(b) SUBSET b}`; `a:A->bool = INTERS Y`] THEN
    SUBGOAL_THEN `!b:A->bool. b IN Y <=> f(b) SUBSET b` ASSUME_TAC THENL
     [EXPAND_TAC "Y" THEN REWRITE_TAC[IN_ELIM_THM]; ALL_TAC] THEN
    SUBGOAL_THEN `!b:A->bool. b IN Y ==> f(a:A->bool) SUBSET b`
    ASSUME_TAC THENL
     [ASM_MESON_TAC[SUBSET_TRANS; IN_INTERS; SUBSET]; ALL_TAC]);;
...
val it : goalstack = 1 subgoal (1 total)

 0 [`!s t. s SUBSET t ==> f s SUBSET f t`]
 1 [`{b | f b SUBSET b} = Y`]
 2 [`INTERS Y = a`]
 3 [`!b. b IN Y <=> f b SUBSET b`]
 4 [`!b. b IN Y ==> f a SUBSET b`]

`?s. f s = s`
```

Now we select a particularly interesting lemma:

```
# e(SUBGOAL_THEN `f(a:A->bool) SUBSET a` ASSUME_TAC);;
val it : goalstack = 2 subgoals (2 total)

 0 [`!s t. s SUBSET t ==> f s SUBSET f t`]
 1 [`{b | f b SUBSET b} = Y`]
 2 [`INTERS Y = a`]
 3 [`!b. b IN Y <=> f b SUBSET b`]
 4 [`!b. b IN Y ==> f a SUBSET b`]
 5 [`f a SUBSET a`]

`?s. f s = s`

 0 [`!s t. s SUBSET t ==> f s SUBSET f t`]
 1 [`{b | f b SUBSET b} = Y`]
 2 [`INTERS Y = a`]
 3 [`!b. b IN Y <=> f b SUBSET b`]
 4 [`!b. b IN Y ==> f a SUBSET b`]

`f a SUBSET a`
```

The lemma is relatively easy to prove by giving `MESON_TAC` the right lemmas:

```
# e(ASM_MESON_TAC[IN_INTERS; SUBSET]);;
...
val it : goalstack = 1 subgoal (1 total)

 0 [`!s t. s SUBSET t ==> f s SUBSET f t`]
 1 [`{b | f b SUBSET b} = Y`]
 2 [`INTERS Y = a`]
 3 [`!b. b IN Y <=> f b SUBSET b`]
 4 [`!b. b IN Y ==> f a SUBSET b`]
 5 [`f a SUBSET a`]
```

rather large. If you step back three steps with

```
# b(); b(); b();;
```

then although the following works, it takes half a minute:

```
# e(ASM_MESON_TAC[IN_INTERS; SUBSET; SUBSET_ANTISYM]);;
....
val it : goalstack = No subgoals
```

### See also
MATCH_MP_TAC, MP_TAC, SUBGOAL_TAC.

## SUBS_CONV

```
SUBS_CONV : thm list -> term -> thm
```

### Synopsis
Substitution conversion.

### Description
The call `SUBS_CONV [th1; ...; th2] t`, where the theorems in the list are all equations, will return the theorem `|- t = t'` where `t'` results from substituting any terms that are the same as the left-hand side of some `thi` with the corresponding right-hand side. Note that no matching or instantiation is done, in contrast to rewriting conversions.

### Failure
May fail if the theorems are not equational.

### Example
Here we substitute with a simplification theorem, but only instances that are the same as the LHS:

```
# SUBS_CONV[ARITH_RULE `x + 0 = x`] `(x + 0) + (y + 0) + (x + 0) + (0 + 0)`;;
val it : thm =
  |- (x + 0) + (y + 0) + (x + 0) + 0 + 0 = x + (y + 0) + x + 0 + 0
```

By contrast, the analogous rewriting conversion will treat the variable `x` as universally quantified and replace more subterms by matching the LHS against them:

```
# REWRITE_CONV[ARITH_RULE `x + 0 = x`]
    `(x + 0) + (y + 0) + (x + 0) + (0 + 0)`;;
val it : thm = |- (x + 0) + (y + 0) + (x + 0) + 0 + 0 = x + y + x
```

### See also
GEN_REWRITE_CONV, REWR_CONV, REWRITE_CONV, PURE_REWRITE_CONV.

# SUBS

```
SUBS : thm list -> thm -> thm
```

## Synopsis

Makes simple term substitutions in a theorem using a given list of theorems.

## Description

Term substitution in HOL is performed by replacing free subterms according to the transformations specified by a list of equational theorems. Given a list of theorems `A1|-t1=v1,...,An|-tn=vn` and a theorem `A|-t`, `SUBS` simultaneously replaces each free occurrence of `ti` in `t` with `vi`:

```
        A1|-t1=v1 ... An|-tn=vn    A|-t
   ---------------------------------------------  SUBS[A1|-t1=v1;...;An|-tn=vn]
    A1 u ... u An u A |- t[v1,...,vn/t1,...,tn]       (A|-t)
```

No matching is involved; the occurrence of each `ti` being substituted for must be a free in `t` (see `SUBST_MATCH`). An occurrence which is not free can be substituted by using rewriting rules such as `REWRITE_RULE`, `PURE_REWRITE_RULE` and `ONCE_REWRITE_RULE`.

## Failure

`SUBS [th1;...;thn] (A|-t)` fails if the conclusion of each theorem in the list is not an equation. No change is made to the theorem `A |- t` if no occurrence of any left-hand side of the supplied equations appears in `t`.

## Example

Substitutions are made with the theorems

```
 # let thm1 = SPEC_ALL ADD_SYM
   and thm2 = SPEC_ALL(CONJUNCT1 ADD_CLAUSES);;
  val thm1 : thm = |- m + n = n + m
  val thm2 : thm = |- 0 + n = n
```

depending on the occurrence of free subterms

```
  # SUBS [thm1; thm2] (ASSUME `(n + 0) + (0 + m) = m + n`);;
  val it : thm = (n + 0) + 0 + m = m + n |- (n + 0) + 0 + m = n + m

  # SUBS [thm1; thm2] (ASSUME `!n. (n + 0) + (0 + m) = m + n`);;
  val it : thm = !n. (n + 0) + 0 + m = m + n |- !n. (n + 0) + 0 + m = m + n
```

## Uses

`SUBS` can sometimes be used when rewriting (for example, with `REWRITE_RULE`) would diverge and term instantiation is not needed. Moreover, applying the substitution rules is often much faster than using the rewriting rules.

## See also
ONCE_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE, SUBS_CONV.

# subset

```
subset : 'a list -> 'a list -> bool
```

## Synopsis
Tests if one list is a subset of another.

## Description
The call `subset l1 l2` returns `true` if every element of `l1` also occurs in `l2`, regardless of whether an element appears once or more than once in each list. So when `l1` and `l2` are regarded as sets, this is a subset test.

## Failure
Never fails.

## Example

```
# subset [1;1;2;2] [1;2;3];;
val it : bool = true
```

## See also
insert, intersect, set_eq, setify, subtract, union.

# SUBST1_TAC

```
SUBST1_TAC : thm_tactic
```

## Synopsis
Makes a simple term substitution in a goal using a single equational theorem.

## Description
Given a theorem `A' |- u = v` and a goal `(A ?- t)`, the tactic `SUBST1_TAC (A' |- u = v)` rewrites the term `t` into `t[v/u]`, by substituting `v` for each free occurrence of `u` in `t`:

```
     A ?- t
=============  SUBST1_TAC (A' |- u = v)
  A ?- t[v/u]
```

The assumptions of the theorem used to substitute with are not added to the assumptions of the goal but are recorded in the proof. If `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), then `SUBST1_TAC (A' |- u = v)` results in an invalid tactic. `SUBST1_TAC` automatically renames bound variables to prevent free variables in `v` becoming bound after substitution.

## Failure

`SUBST1_TAC th (A ?- t)` fails if the conclusion of `th` is not an equation. No change is made to the goal if no free occurrence of the left-hand side of `th` appears in `t`.

## Example

Suppose we start with the goal:

```
# g `!p x y. 1 = x /\ p(1) ==> p(x)`;;
```

We could, of course, solve it immediately with `MESON_TAC[]`. However, for a more "manual" proof, we might do:

```
# e(REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`x = 1`]
 1 [`p 1`]

`p x`
```

and then use `SUBST1_TAC` to substitute:

```
# e(FIRST_X_ASSUM(SUBST1_TAC o SYM));;
val it : goalstack = 1 subgoal (1 total)

 0 [`p 1`]

`p 1`
```

after which just `ASM_REWRITE_TAC[]` will finish.

## Uses

`SUBST1_TAC` can be used when rewriting with a single theorem using tactics such as `REWRITE_TAC` is too expensive or would diverge.

## See also

`ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, SUBST_ALL_TAC.`

# SUBST_ALL_TAC

```
SUBST_ALL_TAC : thm -> tactic
```

## Synopsis

Substitutes using a single equation in both the assumptions and conclusion of a goal.

## Description

`SUBST_ALL_TAC` breaches the style of natural deduction, where the assumptions are kept fixed. Given a theorem `A |- u = v` and a goal `([A1;...;An] ?- t)`, `SUBST_ALL_TAC (A |- u = v)` transforms the assumptions `A1`,...,`An` and the term `t` into `A1[v/u]`,...,`An[v/u]` and `t[v/u]` respectively, by substituting `v` for each free occurrence of `u` in both the assumptions and the conclusion of the goal.

```
         {A1,...,An} ?- t
 =================================  SUBST_ALL_TAC (A |- u = v)
  {A1[v/u],...,An[v/u]} ?- t[v/u]
```

The assumptions of the theorem used to substitute with are not added to the assumptions of the goal, but they are recorded in the proof. If `A` is not a subset of the assumptions of the goal (up to alpha-conversion), then `SUBST_ALL_TAC (A |- u = v)` results in an invalid tactic.

`SUBST_ALL_TAC` automatically renames bound variables to prevent free variables in `v` becoming bound after substitution.

## Failure

`SUBST_ALL_TAC th (A ?- t)` fails if the conclusion of `th` is not an equation. No change is made to the goal if no occurrence of the left-hand side of `th` appears free in `(A ?- t)`.

## Example

Suppose we start with the goal:

```
# g `!p x y. 1 = x /\ p(x - 1) ==> p(x EXP 2 - x)`;;
```

and, as often, begin by breaking it down routinely:

```
# e(REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

 0 [`1 = x`]
 1 [`p (x - 1)`]

`p (x EXP 2 - x)`
```

Now we can use `SUBST_ALL_TAC` to substitute `1` for `x` in both assumptions and conclusion:

```
# e(FIRST_X_ASSUM(SUBST_ALL_TAC o SYM));;
val it : goalstack = 1 subgoal (1 total)

 0 [`p (1 - 1)`]

`p (1 EXP 2 - 1)`
```

One can finish things off in various ways, e.g.

```
# e(POP_ASSUM MP_TAC THEN CONV_TAC NUM_REDUCE_CONV THEN REWRITE_TAC[]);;
```

## See also

`ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST1_TAC`, `SUBST_TAC`.

# subst

```
subst : (term * term) list -> term -> term
```

## Synopsis

Substitute terms for other terms inside a term.

## Description

The call `subst [t1',t1; ...; tn',tn] t` systmatically replaces free instances of each term `ti` inside `t` with the corresponding `ti'` from the instantiation list. (A subterm is considered free if none of its free variables are bound by its context.) Bound variables will be renamed if necessary to avoid capture.

## Failure

Fails if any of the pairs `ti',ti` in the instantiation list has `ti` and `ti'` with different types. Multiple instances of the same `ti` in the list are not trapped, but only the first one will be used consistently.

## Example

Here is a relatively simple example

```
# subst [`x + 1`,`1 + 2`] `(1 + 2) + 1 + 2 + 3`;;
val it : term = `(x + 1) + 1 + 2 + 3`
```

and here is a more complex instance where renaming of bound variables is needed:

```
# subst [`x:num`,`1`] `!x. x > 0 <=> x >= 1`;;
val it : term = `!x'. x' > 0 <=> x' >= x`
```

## Comments

This is the most general term substitution function, but if all the `ti` are variables, the `vsubst` function is more efficient.

## See also

`inst`, `vsubst`.

# SUBST_VAR_TAC

```
SUBST_VAR_TAC : thm -> tactic
```

## Synopsis

Use an equation to substitute "safely" in goal.

## Description

When applied to a theorem with an equational hypothesis `A |- s = t`, `SUBST_ VAR_TAC` has no effect if `s` and `t` are alpha-equivalent. Otherwise, if either side of the equation is a variable not free in the other side, or a constant, and the conclusion contains no free variables not free in some assumption of the goal, then the theorem is used to replace that constant or variable throughout the goal, assumptions and conclusions. If none of these cases apply, or the conclusion is not even an equation, the application fails.

## Failure

Fails if applied to a non-equation for which none of the cases above hold.

## Uses

By some sequence like `REPEAT(FIRST_X_ASSUM SUBST_VAR_TAC)` one can use all possible assumptions to substitute "safely", in the sense that it will not change the provability status of the goal. This is sometimes a useful prelude to other automatic techniques.

## Comments

## See also
SUBST1_TAC, SUBST_ALL_TAC.

# subtract

```
subtract : 'a list -> 'a list -> 'a list
```

## Synopsis
Computes the set-theoretic difference of two 'sets'.

## Description
`subtract l1 l2` returns a list consisting of those elements of `l1` that do not appear in `l2`. If both lists are initially free of repetitions, this can be considered a set difference operation.

## Failure
Never fails.

## Example

```
# subtract [1;2;3] [3;5;4;1];;
val it : int list = [2]
# subtract [1;2;4;1] [4;5];;
val it : int list = [1; 2; 1]
```

## See also
setify, set_equal, union, intersect.

# subtract'

```
subtract' : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

## Synopsis
Subtraction of sets modulo an equivalence.

## Description
The call `subtract' r l1 l2` removes from the list `l1` all elements `x` such that there is an `x'` in `l2` with `r x x'`. If `l1` and `l2` were free of equivalents under `r`, the resulting list will be too, so

this is a set operation modulo an equivalence. The function `subtract` is the special case where the relation is just equality.

## Failure
Fails only if the function `r` fails.

## Example

```
# subtract' (fun x y -> abs(x) = abs(y)) [-1; 2; 1] [-2; -3; 4; -4];;
val it : int list = [-1; 1]
```

## Uses
Maintaining sets modulo an equivalence such as alpha-equivalence.

## See also
`insert'`, `mem'`, `union`, `union'`, `unions'`.

# SYM_CONV

```
SYM_CONV : term -> thm
```

## Synopsis
Interchanges the left and right-hand sides of an equation.

## Description
When applied to an equational term `t1 = t2`, the conversion `SYM_CONV` returns the theorem:

```
|- t1 = t2 <=> t2 = t1
```

## Failure
Fails if applied to a term that is not an equation.

## Example

```
# SYM_CONV `2 = x`;;
val it : thm = |- 2 = x <=> x = 2
```

## See also
`SYM`.

# SYM

```
SYM : thm -> thm
```

## Synopsis

Swaps left-hand and right-hand sides of an equation.

## Description

When applied to a theorem `A |- t1 = t2`, the inference rule `SYM` returns `A |- t2 = t1`.

```
    A |- t1 = t2
   --------------  SYM
    A |- t2 = t1
```

## Failure

Fails unless the theorem is equational.

## Example

```
  # NUM_REDUCE_CONV `12 * 12`;;
  val it : thm = |- 12 * 12 = 144

  # SYM it;;
  val it : thm = |- 144 = 12 * 12
```

## See also

`GSYM`, `REFL`, `TRANS`.

# TAC_PROOF

```
TAC_PROOF : goal * tactic -> thm
```

## Synopsis

Attempts to prove a goal using a given tactic.

## Description

When applied to a goal-tactic pair `(A ?- t,tac)`, the `TAC_PROOF` function attempts to prove the goal `A ?- t`, using the tactic `tac`. If it succeeds, it returns the theorem `A' |- t` corresponding to the goal, where the assumption list `A'` may be a proper superset of `A` unless the tactic is valid; there is no inbuilt validity checking.

## Failure
Fails unless the goal has hypotheses and conclusions all of type `bool`, and the tactic can solve the goal.

## See also
`prove`, `VALID`.

# TAUT

`TAUT : term -> thm`

## Synopsis
Proves a propositional tautology.

## Description
The call ``TAUT `t` `` where `t` is a propositional tautology, will prove it automatically and return `|- t`. A propositional tautology is a formula built up using the logical connectives '`~`', '`/\`', '`\/`', '`==>`' and '`<=>`' from terms that can be considered "atomic" that is logically valid whatever truth-values are assigned to the atomic formulas.

## Failure
Fails if `t` is not a propositional tautology.

## Example
Here is a simple and potentially useful tautology:

```
# TAUT `a \/ b ==> c <=> (a ==> c) /\ (b ==> c)`;;
val it : thm = |- a \/ b ==> c <=> (a ==> c) /\ (b ==> c)
```

and here is a more surprising one:

```
# TAUT `(p ==> q) \/ (q ==> p)`;;
val it : thm = |- (p ==> q) \/ (q ==> p)
```

Note that the "atomic" formulas need not just be variables:

```
# TAUT `(x > 2 ==> y > 3) \/ (y < 3 ==> x > 2)`;;
val it : thm = |- (x > 2 ==> y > 3) \/ (y < 3 ==> x > 2)
```

## Uses
Solving a tautologous goal completely by `CONV_TAC TAUT`, or generating a tautology to massage the goal into a more convenient equivalent form by ``REWRITE_TAC[TAUT `...`]`` or ``ONCE_REWRITE_TAC[TAUT `...`]``.

## Comments
The algorithm used is quite naive, and not efficient on large formulas. For more general first-order reasoning, with quantifier instantiation, use MESON-based methods.

## See also
BOOL_CASES_TAC, ITAUT, ITAUT_TAC, MESON, MESON_TAC.

# temp_path

```
temp_path : string ref
```

## Synopsis
Directory in which to create temporary files.

## Description
Some HOL Light derived rules in the libraries (none in the core system) need to create temporary files. This is the directory in which they do so.

## Failure
Not applicable.

## Example
On my laptop:

```
# !temp_path;;
val it : string = "/tmp"
```

## See also
hol_dir.

# term_match

```
term_match : term list -> term -> term -> instantiation
```

## Synopsis
Match one term against another.

## Description
The call `term_match lcs t t'` attempts to find an instantiation for free variables in `t`, not permitting assignment of 'local constant' variables in the list `lcs`, so that it is alpha-equivalent

to `t'`. If it succeeds, the appropriate instantiation is returned. Otherwise it fails. The matching is higher-order in a limited sense; see `PART_MATCH` for more illustrations.

## Failure
Fails if terms cannot be matched.

## Example

```
# term_match [] `x + y + 1` `(y + 1) + z + 1`;;
val it : instantiation = ([], [(`z`, `y`); (`y + 1`, `x`)], [])

# term_match [] `~(?x:A. P x)` `~(?n. 5 < n /\ n < 6)`;;
val it : instantiation =
  ([(1, `P`)], [(`\n. 5 < n /\ n < 6`, `P`)], [(`:num`, `:A`)])
```

## See also
`instantiate`, `PART_MATCH`.

# term_of_preterm

```
term_of_preterm : preterm -> term
```

## Synopsis
Converts a preterm into a term.

## Description
HOL Light uses "pretypes" and "preterms" as intermediate structures for parsing and typechecking, which are later converted to types and terms. A call `term_of_preterm ptm` attempts to convert preterm `ptm` into a HOL term.

## Failure
Fails if some constants used in the preterm have not been defined, or if there are other inconsistencies in the types so that a consistent typing cannot be arrived at.

## Comments
Only users seeking to change HOL's parser and typechecker quite radically need to use this function.

## See also
`preterm_of_term`, `retypecheck`, `type_of_pretype`.

# term_of_rat

```
term_of_rat : num -> term
```

## Synopsis
Converts OCaml number to canonical rational literal of type `:real`.

## Description
The call `term_of_rat n`, where `n` is an OCaml rational number (type `num`), returns the canonical rational literal of type `:real` that represents it. The canonical literals are integer literals `&n` for numeral `n`, `-- &n` for a nonzero numeral `n`, or ratios `&p / &q` or `-- &p / &q` where `p` is nonzero, `q > 1` and `p` and `q` share no common factor.

## Failure
Never fails.

## Example

```
# term_of_rat (Int 3 // Int 2);;
val it : term = `&3 / &2`
```

## See also
`is_ratconst`, `mk_realintconst`, `rat_of_term`, `REAL_RAT_REDUCE_CONV`.

# term_order

```
term_order : term -> term -> bool
```

## Synopsis
Term order for use in AC-rewriting.

## Description
This binary predicate implements a crude but fairly efficient ordering on terms that is appropriate for ensuring that ordered rewriting will perform normalization.

## Failure
Never fails.

## Example

This example shows how using ordered rewriting with this term ordering can give normalization under associative and commutative laws given the appropriate rewrites:

```
# ADD_AC;;
val it : thm =
  |- m + n = n + m /\ (m + n) + p = m + n + p /\ m + n + p = n + m + p


# TOP_DEPTH_CONV
   (FIRST_CONV(map (ORDERED_REWR_CONV term_order) (CONJUNCTS ADD_AC)))
   `d + (f + a) + b + (c + e):num`;;
val it : thm = |- d + (f + a) + b + c + e = a + b + c + d + e + f
```

## Uses

It is used automatically when applying permutative rewrite rules inside rewriting and simplification. Users will not normally want to use it explicitly, though the example above shows roughly what goes on there.

## See also

ORDERED_IMP_REWR_CONV, ORDERED_REWR_CONV.

# term_unify

```
term_unify : term list -> term -> term -> instantiation
```

## Synopsis

Unify two terms.

## Description

Given two terms `tm1` and `tm2`, a call `term_unify vars tm1 tm2` attempts to find instantiations of the variables `vars` in the two terms to make them alpha-equivalent. At present, no type instantiation is done. The unification is also purely first-order. In these respects it is less general than `term_match`, and this may be improved in the future.

## Failure

Fails if the two terms are not first-order unifiable by instantiating the given variables without type instantiation.

## See also

instantiate, term_match.

# term_union

```
term_union : term list -> term list -> term list
```

## Synopsis
Union of two sets of terms up to alpha-equivalence.

## Description
The call `term_union l1 l2` for two lists of terms `l1` and `l2` returns a list including all of `l2` and all terms of `l1` for which no alpha-equivalent term occurs in `l2` or earlier in `l1`. If both lists were sets modulo alpha-conversion, i.e. contained no alpha-equivalent pairs, then so will be the result.

## Failure
Never fails.

## Example

```
# term_union [`1`; `2`] [`2`; `3`];;
val it : term list = [`1`; `2`; `3`]

# term_union [`!x. x >= 0`; `?u. u > 0`] [`?w. w > 0`; `!u. u >= 0`];;
val it : term list = [`?w. w > 0`; `!u. u >= 0`]
```

## Uses
For combining assumption lists of theorems without duplication of alpha-equivalent ones.

## See also
`aconv`, `union`, `union'`.

# the_definitions

```
the_definitions : thm list ref
```

## Synopsis
List of all definitions introduced so far.

## Description
The reference variable `the_definitions` holds the list of definitions made so far. Various definitional rules such as `new_definition` automatically augment it. Note that in some cases

(e.g. `new_inductive_definition`) the stored form of the definition may look very different from what the user sees or enters at the top level.

## Failure

Not applicable.

## Example

If we examine the list in HOL Light's initial state, we see the most recent definition at the head (`superadmissible` is connected with HOL's automated definitional rule `define`) and the oldest, logical truth `T`, at the tail:

```
# !the_definitions;;
val it : thm list =
  [|- !(<<) p s t.
          superadmissible (<<) p s t <=>
          admissible (<<) (\f a. T) s p ==> tailadmissible (<<) p s t;
   ...
   ...
   |- (/\) = (\p q. (\f. f p q) = (\f. f T T)); |- T <=> (\p. p) = (\p. p)]
```

If we make a new definition of any sort, e.g.

```
# new_definition `false <=> F`;;
val it : thm = |- false <=> F
```

we will see a new entry at the head:

```
# !the_definitions;;
val it : thm list =
  [|- false <=> F;
   ...
   ...
   |- (/\) = (\p q. (\f. f p q) = (\f. f T T)); |- T <=> (\p. p) = (\p. p)]
```

## Uses

This list is not logically necessary and is not part of HOL Light's logical core, but it is used outside the core so that multiple instances of the same definition are quietly "ignored" rather than rejected. (By contrast, the list of new constants introduced by definitions is logically necessary to avoid inconsistent redefinition.) Users may also sometimes find it convenient.

## See also

`axioms`, `constants`, `define`, `new_definition`, `new_inductive_definition`, `new_recursive_definition`, `new_specification`, `the_specifications`.

# the_inductive_types

`the_inductive_types : (string * (thm * thm)) list ref`

## Synopsis
List of previously declared inductive types.

## Description
This reference variable contains a list of the inductive types, together with their induction and recursion theorems as returned by `define_type`. The list is automatically extended by a call of `define_type`.

## Failure
Not applicable.

## See also
`define_type`.

# the_interface

`the_interface : (string * (string * hol_type)) list ref`

## Synopsis
List of active interface mappings.

## Description
HOL Light allows the same identifier to map to one or more underlying constants using an overloading mechanism with resolution based on type. The reference variable `the_interface` stores the current list of all interface mappings.

## See also
`make_overloadable`, `overload_interface`, `override_interface`, `prioritize_overload`, `reduce_interface`, `remove_interface`, `the_interface`, `the_overload_skeletons`.

# thenc_

`thenc_ : conv -> conv -> conv`

## Synopsis
Non-infix version of THENC.

## See also
THENC.

# THENC

```
(THENC) : conv -> conv -> conv
```

## Synopsis
Applies two conversions in sequence.

## Description
If the conversion c1 returns |- t = t' when applied to a term `t`, and c2 returns |- t' = t'' when applied to `t'`, then the composite conversion (c1 THENC c2) `t` returns |- t = t''. That is, (c1 THENC c2) `t` has the effect of transforming the term `t` first with the conversion c1 and then with the conversion c2.

## Failure
(c1 THENC c2) `t` fails if either the conversion c1 fails when applied to `t`, or if c1 `t` succeeds and returns |- t = t' but c2 fails when applied to `t'`. (c1 THENC c2) `t` may also fail if either of c1 or c2 is not, in fact, a conversion (i.e. a function that maps a term t to a theorem |- t = t').

## Example

```
# BETA_CONV `(\x. x + 1) 3`;;
val it : thm = |- (\x. x + 1) 3 = 3 + 1
# (BETA_CONV THENC NUM_ADD_CONV) `(\x. x + 1) 3`;;
val it : thm = |- (\x. x + 1) 3 = 4
```

## See also
EVERY_CONV, ORELSEC, REPEATC.

# then_

```
then_ : tactic -> tactic -> tactic
```

## Synopsis

Non-infix version of THEN.

## See also

THEN.

# THEN

```
(THEN) : tactic -> tactic -> tactic
```

## Synopsis

Applies two tactics in sequence.

## Description

If t1 and t2 are tactics, t1 THEN t2 is a tactic which applies t1 to a goal, then applies the tactic t2 to all the subgoals generated. If t1 solves the goal then t2 is never applied.

## Failure

The application of THEN to a pair of tactics never fails. The resulting tactic fails if t1 fails when applied to the goal, or if t2 does when applied to any of the resulting subgoals.

## Example

Suppose we want to prove the inbuilt theorem DELETE_INSERT ourselves:

```
# g `!x y. (x INSERT s) DELETE y =
           if x = y then s DELETE y else x INSERT (s DELETE y)`;;
```

We may wish to perform a case-split using COND_CASES_TAC, but since variables in the if-then-else construct are bound, this is inapplicable. Thus we want to first strip off the universally quantified variables:

```
# e(REPEAT GEN_TAC);;
val it : goalstack = 1 subgoal (1 total)

`(x INSERT s) DELETE y =
 (if x = y then s DELETE y else x INSERT (s DELETE y))`
```

and then apply COND_CASES_TAC:

```
# e COND_CASES_TAC;;
...
```

A quicker way (starting again from the initial goal) would be to combine the tactics using

THEN:

```
# e(REPEAT GEN_TAC THEN COND_CASES_TAC);;
...
```

## Comments

Although normally used to sequence tactics which generate a single subgoal, it is worth remembering that it is sometimes useful to apply the same tactic to multiple subgoals; sequences like the following:

```
EQ_TAC THENL [ASM_REWRITE_TAC[]; ASM_REWRITE_TAC[]]
```

can be replaced by the briefer:

```
EQ_TAC THEN ASM_REWRITE_TAC[]
```

If using this several times in succession, remember that THEN is left-associative.

## See also

EVERY, ORELSE, THENL.

# thenl_

```
thenl_ : tactic -> tactic list -> tactic
```

## Synopsis

Non-infix version of THENL.

## See also

THENL.

# THENL

```
(THENL) : tactic -> tactic list -> tactic
```

## Synopsis

Applies a list of tactics to the corresponding subgoals generated by a tactic.

## Description

If `t,t1,...,tn` are tactics, `t THENL [t1;...;tn]` is a tactic which applies `t` to a goal, and if it does not fail, applies the tactics `t1,...,tn` to the corresponding subgoals, unless `t` completely solves the goal.

## Failure

The application of `THENL` to a tactic and tactic list never fails. The resulting tactic fails if `t` fails when applied to the goal, or if the goal list is not empty and its length is not the same as that of the tactic list, or finally if `ti` fails when applied to the `i`'th subgoal generated by `t`.

## Example

If we want to prove the inbuilt theorem `LE_LDIV` ourslves:

```
# g `!a b n. ~(a = 0) /\ b <= a * n ==> b DIV a <= n`;;
...
```

we may start by proving a lemma `n = (a * n) DIV a` from the given hypotheses. The following step generates two subgoals:

```
# e(REPEAT STRIP_TAC THEN SUBGOAL_THEN `n = (a * n) DIV a` SUBST1_TAC);;
val it : goalstack = 2 subgoals (2 total)

 0 [`~(a = 0)`]
 1 [`b <= a * n`]

`b DIV a <= (a * n) DIV a`

 0 [`~(a = 0)`]
 1 [`b <= a * n`]

`n = (a * n) DIV a`
```

Each subgoal has a relatively short proof, but these proofs are quite different. We can combine them with the initial tactic above using `THENL`, so the following would solve the initial goal:

```
# e(REPEAT STRIP_TAC THEN SUBGOAL_THEN `n = (a * n) DIV a` SUBST1_TAC THENL
    [ASM_SIMP_TAC[DIV_MULT]; MATCH_MP_TAC DIV_MONO THEN ASM_REWRITE_TAC[]]);;
```

Note that it is quite a common situation for the same tactic to be applied to all generated subgoals. In that case, you can just use `THEN`, e.g. in the proof of the pre-proved theorem `ADD_0`:

```
# g `!m. m + 0 = m`;;
...
# e(INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);;
val it : goalstack = No subgoals
```

## Uses

Applying different tactics to different subgoals.

## See also

EVERY, ORELSE, THEN.

# then_tcl_

```
then_tcl_ : thm_tactical -> thm_tactical -> thm_tactical
```

## Synopsis

Non-infix version of THEN_TCL.

## See also

THEN_TCL.

# THEN_TCL

```
(THEN_TCL) : thm_tactical -> thm_tactical -> thm_tactical
```

## Synopsis

Composes two theorem-tacticals.

## Description

If ttl1 and ttl2 are two theorem-tacticals, ttl1 THEN_TCL ttl2 is a theorem-tactical which composes their effect; that is, if:

```
ttl1 ttac th1 = ttac th2
```

and

```
ttl2 ttac th2 = ttac th3
```

then

```
(ttl1 THEN_TCL ttl2) ttac th1 = ttac th3
```

## Failure

The application of THEN_TCL to a pair of theorem-tacticals never fails.

## See also

EVERY_TCL, FIRST_TCL, ORELSE_TCL.

# the_overload_skeletons

```
the_overload_skeletons : (string * hol_type) list ref
```

## Synopsis

List of overload skeletons for all overloadable identifiers.

## Description

HOL Light allows the same identifier to denote several different underlying constants, with the choice being determined by types and/or an order of priority (see `prioritize_overload`). The reference variable `the_overload_skeletons` contains a list of all the overloadable symbols (you can add more using `make_overloadable`) and their type skeletons. All constants to which an identifier is overloaded must have a type that is an instance of this skeleton, although you can make it a type variable in which case any type would be allowed.

## Failure

Not applicable.

## Example

In the initial state of HOL Light:

```
# !the_overload_skeletons;;
val it : (string * hol_type) list =
  [("gcd", `:A#A->A`); ("coprime", `:A#A->bool`); ("mod", `:A->A->A->bool`);
   ("divides", `:A->A->bool`); ("&", `:num->A`); ("min", `:A->A->A`);
   ("max", `:A->A->A`); ("abs", `:A->A`); ("inv", `:A->A`);
   ("pow", `:A->num->A`); ("--", `:A->A`); (">=", `:A->A->bool`);
   (">", `:A->A->bool`); ("<=", `:A->A->bool`); ("<", `:A->A->bool`);
   ("/", `:A->A->A`); ("*", `:A->A->A`); ("-", `:A->A->A`);
   ("+", `:A->A->A`)]
```

## See also

make_overloadable, overload_interface, override_interface, prioritize_overload, reduce_interface, remove_interface, the_interface.

# the_specifications

```
the_specifications : thm list ref
```

## Synopsis

List of all constant specifications introduced so far.

## Description

The reference variable `the_specifications` holds the list of constant specifications made so far by `new_specification`. It is a list of triples, with the first two components being the list of variables and the existential theorem used as input, and the last being the returned theorem.

## Failure

Not applicable.

## Uses

This list is not logically necessary and is not part of HOL Light's logical core, but it is used outside the core so that multiple instances of the same specification are quietly "ignored" rather than rejected. (By contrast, the list of new constants introduced by definitions is logically necessary to avoid inconsistent redefinition.) Users may also sometimes find it convenient.

## See also

`axioms`, `constants`, `define`, `new_definition`, `new_inductive_definition`, `new_recursive_definition`, `new_specification`.

# the_type_definitions

```
the_type_definitions : ((string * string * string) * (thm * thm)) list ref
```

## Synopsis

List of type definitions made so far.

## Description

The reference variable `the_type_definitions` holds a list of entries, one for each type definition made so far with `new_type_definition`. It is not normally explicitly manipulated by the user, but is automatically augmented by each call of `new_type_definition`. Each entry contains three strings (the type name, type constructor name and destructor name) and two theorems (the input nonemptiness theorem and the returned type bijections). That is, for a call:

```
  bijth = new_type_definition tyname (absname,repname) nonempth;;
```

the entry created in this list is:

```
 (tyname,absname,repname),(nonempth,bijth)
```

Note that the entries made using other interfaces to `new_basic_type_definition`, such as `define_type`, are not included in this list.

## Failure

Not applicable.

## Uses
This is mainly intended for internal use in `new_type_definition`, so that repeated instances of the same definition are ignored rather than rejected. Some users may find the information useful too.

## See also
`axioms`, `constants`, `new_type_definition`, `the_definitions`.

# thm_frees

```
thm_frees : thm -> term list
```

## Synopsis
Returns a list of the variables free in a theorem's assumptions and conclusion.

## Description
When applied to a theorem, `A |- t`, the function `thm_frees` returns a list, without repetitions, of those variables which are free either in `t` or in some member of the assumption list `A`.

## Failure
Never fails.

## Example

```
# let th = CONJUNCT1 (ASSUME `p /\ q`);;
val th : thm = p /\ q |- p

# thm_frees th;;
val it : term list = [`q`; `p`]
```

## See also
`frees`, `freesl`, `free_in`.

# time

```
time : ('a -> 'b) -> 'a -> 'b
```

## Synopsis
Report CPU time taken by a function.

## Description

A call `time f x` will evaluate `f x` as usual, but will also (provided the `report_timing` flag is `true` as it is by default) print the CPU time taken by that function evaluation.

## Failure

Never fails in itself, though it propagates any exception generated by the call `f x` itself.

## Example

```
#  time NUM_REDUCE_CONV `123 EXP 14`;;
CPU time (user): 0.09
val it : thm = |- 123 EXP 14 = 18141431786723807536841319600 9
```

## Uses

Monitoring CPU time taken, e.g. to test different algorithms or implementation optimizations.

## See also

`report_timing`.

# tl

```
tl : 'a list -> 'a list
```

## Synopsis

Computes the tail of a list (the original list less the first element).

## Description

`tl [x1;...;xn]` returns `[x2;...;xn]`.

## Failure

Fails with `tl` if the list is empty.

## See also

`hd`, `el`.

# TOP_DEPTH_CONV

```
TOP_DEPTH_CONV : conv -> conv
```

## Synopsis

Applies a conversion top-down to all subterms, retraversing changed ones.

## Description

TOP_DEPTH_CONV c tm repeatedly applies the conversion c to all the subterms of the term tm, including the term tm itself. The supplied conversion c is applied to the subterms of tm in top-down order and is applied repeatedly (zero or more times, as is done by REPEATC) at each subterm until it fails. If a subterm t is changed (except for alpha-equivalence) by virtue of the application of c to its own subterms, then the term into which t is transformed is retraversed by applying TOP_DEPTH_CONV c to it.

## Failure

TOP_DEPTH_CONV c tm never fails but can diverge.

## Example

Both TOP_DEPTH_CONV and REDEPTH_CONV repeatedly apply a conversion until no more applications are possible anywhere in the term. For example, TOP_DEPTH_CONV BETA_CONV or REDEPTH_CONV BETA_CONV will eliminate all beta redexes:

```
# TOP_DEPTH_CONV BETA_CONV `(\x. (\y. (\z. z + y) (y + 1)) (x + 2)) 3`;;
val it : thm =
  |- (\x. (\y. (\z. z + y) (y + 1)) (x + 2)) 3 = ((3 + 2) + 1) + 3 + 2
```

The main difference is that TOP_DEPTH_CONV proceeds top-down, whereas REDEPTH_CONV proceeds bottom-up. Reasons for preferring TOP_DEPTH_CONV might be that a transformation near the top obviates the need for transformations lower down. For example, this is quick because everything is done by one top-level rewrite:

```
# let conv = GEN_REWRITE_CONV I [MULT_CLAUSES] ORELSEC NUM_RED_CONV;;
val conv : conv = <fun>

# time (TOP_DEPTH_CONV conv) `0 * 25 EXP 100`;;
CPU time (user): 0.
val it : thm = |- 0 * 25 EXP 100 = 0
```

whereas the following takes markedly longer:

```
# time (REDEPTH_CONV conv) `0 * 25 EXP 100`;;
CPU time (user): 2.573
val it : thm = |- 0 * 25 EXP 100 = 0
```

## See also

DEPTH_CONV, ONCE_DEPTH_CONV, REDEPTH_CONV, TOP_SWEEP_CONV.

# TOP_DEPTH_SQCONV

```
TOP_DEPTH_SQCONV : strategy
```

## Synopsis
Applies simplification top-down to all subterms, retraversing changed ones.

## Description
HOL Light's simplification functions (e.g. `SIMP_TAC`) have their traversal algorithm controlled by a "strategy". `TOP_DEPTH_SQCONV` is a strategy corresponding to `TOP_DEPTH_CONV` for ordinary conversions: simplification is applied top-down to all subterms, retraversing changed ones.

## Failure
Not applicable.

## See also
`DEPTH_SQCONV`, `ONCE_DEPTH_SQCONV`, `REDEPTH_SQCONV`, `TOP_DEPTH_CONV`, `TOP_SWEEP_SQCONV`.

# top_goal

```
top_goal : unit -> term list * term
```

## Synopsis
Returns the current goal of the subgoal package.

## Description
The function `top_goal` is part of the subgoal package. It returns the top goal of the goal stack in the current proof state. For a description of the subgoal package, see `set_goal`.

## Failure
A call to `top_goal` will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved.

## Uses
Examining the proof state after a proof fails.

## See also
`b`, `e`, `g`, `p`, `r`, `set_goal`, `top_thm`.

# top_realgoal

```
top_realgoal : unit -> (string * thm) list * term
```

## Synopsis

Returns the actual internal structure of the current goal.

## Description

Returns the actual internal representation of the current goal, including the labels and the theorems that are the assumptions.

## Uses

For users interested in the precise internal structure of the goal, e.g. to debug subtle free variable problems. Normally the simpler structure returned by `top_goal` is entirely adequate.

## See also

`top_goal`.

# TOP_SWEEP_CONV

```
TOP_SWEEP_CONV : conv -> conv
```

## Synopsis

Repeatedly applies a conversion top-down at all levels, but after descending to subterms, does not return to higher ones.

## Description

The call `TOP_SWEEP_CONV conv` applies `conv` repeatedly at the top level of a term, and then descends into subterms of the result, recursively doing the same thing. However, once the subterms are dealt with, it does not, unlike `TOP_DEPTH_CONV conv`, return to re-examine them.

## Failure

Never fails.

## Example

If we create an equation between large tuples:

```
# let tm =
    let pairup x i t = mk_pair(mk_var(x^string_of_int i,aty),t) in
    let mkpairs x = itlist (pairup x) (1--200) (mk_var(x,aty)) in
  mk_eq(mkpairs "x",mkpairs "y");;
...
```

we can observe that

```
# time (TOP_DEPTH_CONV(REWR_CONV PAIR_EQ)); ();;
```

is a little bit slower than

```
# time (TOP_SWEEP_CONV(REWR_CONV PAIR_EQ)); ();;
```

## See also

DEPTH_CONV, ONCE_DEPTH_CONV, REDEPTH_CONV, TOP_DEPTH_CONV.

# TOP_SWEEP_SQCONV

```
TOP_SWEEP_SQCONV : strategy
```

## Synopsis

Applies simplification top-down at all levels, but after descending to subterms, does not return to higher ones.

## Description

HOL Light's simplification functions (e.g. SIMP_TAC) have their traversal algorithm controlled by a "strategy". TOP_SWEEP_SQCONV is a strategy corresponding to TOP_SWEEP_CONV for ordinary conversions: simplification is applied top-down at all levels, but after descending to subterms, does not return to higher ones.

## Failure

Not applicable.

## See also

DEPTH_SQCONV, ONCE_DEPTH_SQCONV, REDEPTH_SQCONV, TOP_DEPTH_SQCONV, TOP_SWEEP_CONV.

# top_thm

`top_thm : unit -> thm`

## Synopsis
Returns the theorem just proved using the subgoal package.

## Description
The function `top_thm` is part of the subgoal package. A proof state of the package consists of either goal and justification stacks if a proof is in progress or a theorem if a proof has just been completed. If the proof state consists of a theorem, `top_thm` returns that theorem. For a description of the subgoal package, see `set_goal`.

## Failure
`top_thm` will fail if the proof state does not hold a theorem. This will be so either because no goal has been set or because a proof is in progress with unproven subgoals.

## Uses
Accessing the result of an interactive proof session with the subgoal package.

## See also
`b, e, g, p, r, set_goal, top_goal.`

# TRANS

`TRANS : thm -> thm -> thm`

## Synopsis
Uses transitivity of equality on two equational theorems.

## Description
When applied to a theorem `A1 |- t1 = t2` and a theorem `A2 |- t2 = t3`, the inference rule `TRANS` returns the theorem `A1 u A2 |- t1 = t3`.

```
    A1 |- t1 = t2   A2 |- t2 = t3
   -------------------------------  TRANS
         A1 u A2 |- t1 = t3
```

## Failure
Fails unless the theorems are equational, with the right side of the first being the same as the left side of the second.

## Example

The following shows identical uses of TRANS, one on Boolean equations (shown as <=>) and one on numerical equations.

```
# let t1 = ASSUME `a:bool = b` and t2 = ASSUME `b:bool = c`;;
val t1 : thm = a <=> b |- a <=> b
val t2 : thm = b <=> c |- b <=> c
# TRANS t1 t2;;
val it : thm = a <=> b, b <=> c |- a <=> c

# let t1 = ASSUME `x:num = 1` and t2 = num_CONV `1`;;
val t1 : thm = x = 1 |- x = 1
val t2 : thm = |- 1 = SUC 0
# TRANS t1 t2;;
val it : thm = x = 1 |- x = SUC 0
```

## Comments

This is one of HOL Light's 10 primitive inference rules.

## See also

EQ_MP, IMP_TRANS, REFL, SYM.

# tryapplyd

```
tryapplyd : ('a, 'b) func -> 'a -> 'b -> 'b
```

## Synopsis

Applies a finite partial function, with a default for undefined points.

## Description

This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` is a finite partial function, `x` an element of its domain type and `y` of its range type, the call `tryapplyd f x y` tries to apply `f` to the value `x`, as with `apply f x`, but if it is undefined, simply returns `y`

## Failure

Never fails.

## Example

```
# tryapplyd (1 |=> 2) 1 (-1);;
val it : int = 2

# tryapplyd undefined 1 (-1);;
val it : int = -1
```

## See also

|->, |=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, undefine, undefined.

# TRY_CONV

TRY_CONV : conv -> conv

## Synopsis

Attempts to apply a conversion; applies identity conversion in case of failure.

## Description

TRY_CONV c ‘t‘ attempts to apply the conversion c to the term ‘t‘; if this fails, then the identity conversion is applied instead giving the reflexive theorem |- t = t.

## Failure

Never fails.

## Example

```
# num_CONV ‘0‘;;
Exception: Failure "num_CONV".
# TRY_CONV num_CONV ‘0‘;;
val it : thm = |- 0 = 0
```

## See also

ALL_CONV.

# TRY

TRY : tactic -> tactic

## Synopsis
Makes a tactic have no effect rather than fail.

## Description
For any tactic `t`, the application `TRY t` gives a new tactic which has the same effect as `t` if that succeeds, and otherwise has no effect.

## Failure
The application of `TRY` to a tactic never fails. The resulting tactic never fails.

## Example
We might want to try a certain tactic "speculatively", even if we're not sure that it will work, for example, to handle the "easy" subgoals from breaking apart a large conjunction. On a small scale, we might want to prove:

```
# g `(x + 1) EXP 2 = x EXP 2 + 2 * x + 1 /\
     (x EXP 2 = y EXP 2 ==> x = y) /\
     (x < y ==> 2 * x + 1 < 2 * y)`;;
...
```

and just see which conjuncts we can get rid of automatically by `ARITH_TAC`. It turns out that it only leaves one subgoal with some nonlinear reasoning:

```
# e(REPEAT CONJ_TAC THEN TRY ARITH_TAC);;
val it : goalstack = 1 subgoal (1 total)

`x EXP 2 = y EXP 2 ==> x = y`
```

## See also
`CHANGED_TAC`, `VALID`.

# tryfind

```
tryfind : ('a -> 'b) -> 'a list -> 'b
```

## Synopsis
Returns the result of the first successful application of a function to the elements of a list.

## Description
`tryfind f [x1;...;xn]` returns `(f xi)` for the first `xi` in the list for which application of `f` succeeds.

## Failure

Fails with `tryfind` if the application of the function fails for all elements in the list. This will always be the case if the list is empty.

## See also

`find, mem, exists, forall, assoc, rev_assoc.`

# try_user_parser

```
try_user_parser : lexcode list -> preterm * lexcode list
```

## Synopsis

Try all user parsing functions.

## Description

HOL Light allows user parsing functions to be installed, and will try them on all terms during parsing before the usual parsers. The call `try_user_parser l` attempts to parse the list of tokens `l` using all the user parsers, taking the results from whichever one succeeds first.

## Failure

Fails if all user parsers fail.

## See also

`delete_parser, install_parser, installed_parsers.`

# try_user_printer

```
try_user_printer : term -> unit
```

## Synopsis

Try user-defined printers on a term.

## Description

HOL Light allows arbitrary user printers to be inserted into the toplevel printer so that they are invoked on all applicable subterms (see `install_user_printer`). The call `try_user_printer tm` attempts all installed user printers on the term `tm` in an implementation-defined order. If one succeeds, the call returns `()`, and otherwise it fails.

## Failure

Fails if no user printer is applicable to the given term (e.g. if no user printers have been installed).

## Example

After installing the printer for variables with types in the example for `install_user_printer`, you can try:

```
# try_user_printer `x:num`;;
(x:num)val it : unit = ()

# try_user_printer `1`;;
Exception: Failure "tryfind".
```

## See also

`delete_user_printer`, `install_user_printer`.

# type_abbrevs

```
type_abbrevs : unit -> (string * hol_type) list
```

## Synopsis

Lists all current type abbreviations.

## Description

The call `type_abbrevs()` returns a list of all current type abbreviations, which are applied when parsing types but have no logical significance.

## Failure

Never fails.

## See also

`new_type_abbrev`, `remove_type_abbrev`.

# type_invention_warning

```
type_invention_warning : bool ref
```

## Synopsis

Determined if user is warned about invented type variables.

## Description

If HOL Light is unable to assign specific types to a term entered in quotation, it will invent its own type variables to use in the most general type. The flag `type_invention_warning` determines whether the user is warned in such situations. The default is `true`, since this can often indicate a user error (e.g. the user forgot to define a constant before using it in a term or overlooked more general types than expected). But to disable the warnings, set it to `false`.

## Failure

Not applicable.

## Example

When the following term is entered, HOL Light invents a type variable to use as the most general type:

```
# let tm = `x IN s`;;
Warning: inventing type variables
val tm : term = `x IN s`
```

which are not particularly intuitive, as you can see:

```
# map dest_var (frees tm);;
val it : (string * hol_type) list =
  [("x", `:?47676`); ("s", `:?47676->bool`)]
```

You can avoid this by explicitly giving appropriate types or type variables yourself:

```
# let tm = `(x:A) IN s`;;
val tm : term = `x IN s`
```

But if you often want to let HOL Light invent types for itself without warning you, set

```
# type_invention_warning := false;;
val it : unit = ()
```

One reason why you might find the warning more irritating than helpful is if you are rewriting with ad-hoc set theory lemmas generated like this:

```
# SET_RULE `x IN UNIONS (a INSERT t) <=> x IN UNIONS t \/ x IN a`;;
```

## See also

`retypecheck`, `term_of_preterm`.

# type_match

```
type_match : hol_type -> hol_type -> (hol_type * hol_type) list -> (hol_type * hol_type)
```

## Synopsis
Computes a type instantiation to match one type to another.

## Description
The call `type_match vty cty []` will if possible find an instantiation of the type variables in `vty` to make it the same as `cty`, and will fail if this is not possible. The instantiation is returned in a list of term-variable pairs as expected by type instantiation operations like `inst` and `INST_TYPE`. More generally, `type_match vty cty env` will attempt to find such a match assuming that the instantiations already in the list `env` are needed (this is helpful, for example, in matching multiple pairs of types in parallel).

## Failure
Fails if there is no match under the chosen constraints.

## Example
Here is a basic example with an empty last argument:

```
# type_match `:A->B->bool` `:num->num->bool` [];;
val it : (hol_type * hol_type) list = [(`:num`, `:A`); (`:num`, `:B`)]
```

and here is an illustration of how the extra argument can be used to perform parallel matches.

```
# itlist2 type_match
    [`:A->A->bool`; `:B->B->bool`] [`:num->num->bool`; `:bool->bool->bool`]
    [];;
val it : (hol_type * hol_type) list = [(`:num`, `:A`); (`:bool`, `:B`)]
```

## See also
`inst, INST_TYPE, mk_mconst, term_match.`

# type_of

```
type_of : term -> hol_type
```

## Synopsis
Returns the type of a term.

## Failure
Never fails.

## Example

```
# type_of `T`;;
val it : hol_type = `:bool`
```

# type_of_pretype

```
type_of_pretype : pretype -> hol_type
```

## Synopsis
Converts a pretype to a type.

## Description
HOL Light uses "pretypes" and "preterms" as intermediate structures for parsing and typechecking, which are later converted to types and terms. A call `type_of_pretype pty` attempts to convert pretype `pty` into a HOL type.

## Failure
Fails if some type constants used in the pretype have not been defined, or if the arities are wrong.

## Comments
Only users seeking to change HOL's parser and typechecker quite radically need to use this function.

## See also
`pretype_of_type`, `retypecheck`, `term_of_preterm`.

# types

```
types : unit -> (string * int) list
```

## Synopsis
Lists all the types presently declared.

## Description
The function `types` should be applied to `()` and returns a list of all the type constructors declared, in the form of arity-name pairs.

## Failure
Never fails.

### Example

In the initial state we have:

```
# types();;
val it : (string * int) list =
  [("finite_sum", 2); ("cart", 2); ("finite_image", 1); ("int", 0);
   ("real", 0); ("hreal", 0); ("nadd", 0); ("3", 0); ("2", 0); ("list", 1);
   ("option", 1); ("sum", 2); ("recspace", 1); ("num", 0); ("ind", 0);
   ("prod", 2); ("1", 0); ("bool", 0); ("fun", 2)]
```

### See also

`axioms`, `constants`, `new_type`, `new_type_definition`.

## type_subst

`type_subst : (hol_type * hol_type) list -> hol_type -> hol_type`

### Synopsis

Substitute chosen types for type variables in a type.

### Description

The call `type_subst [ty1,tv1; ... ; tyn,tvn] ty` where each `tyi` is a type and each `tvi` is a type variable, will systematically replace each instance of `tvi` in the type `ty` by the corresponding type `tyi`.

### Failure

Never fails. If some of the `tvi` are not type variables they will be ignored, and if several `tvi` are the same, the first one in the list will be used to determine the substitution.

### Example

```
# type_subst [`:num`,`:A`; `:bool`,`:B`] `:A->(B)list->A#B#C`;;
val it : hol_type = `:num->(bool)list->num#bool#C`
```

### See also

`inst`, `tysubst`.

## type_vars_in_term

`type_vars_in_term : term -> hol_type list`

## Synopsis
Returns the set of type variables used in a term.

## Description
The call `type_vars_in_term t` returns the set of all type variables occurring anywhere inside any subterm of `t`.

## Failure
Never fails.

## Example
Note that the list of types occurring somewhere in the term may be larger than the set of type variables in the term's toplevel type. For example:

```
# type_vars_in_term `!x:A. x = x`;;
val it : hol_type list = [`:A`]
```

whereas

```
# tyvars(type_of `!x:A. x = x`);;
val it : hol_type list = []
```

## See also
`frees, tyvars`.

# typify_universal_set

```
typify_universal_set : bool ref
```

## Synopsis
Determines whether the universe set on a type is printed just as the type.

## Description
The reference variable `typify_universal_set` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. When it is `true`, as it is by default, any universal set `UNIV:A->bool` (`UNIV` is a predefined set constant valid over all types) is printed just as `(:A)`. When `typify_universal_set` is `false`, it is printed as `UNIV`, just as for any other constant.

## Failure
Not applicable.

## Example

Note that having this setting is quite useful here:

```
# CART_EQ;;
val it : thm =
  |- !x y. x = y <=> (!i. 1 <= i /\ i <= dimindex (:B) ==> x $ i = y $ i)
```

## Uses

HOL Light's Cartesian power type (constructor '^') uses a type to index the power. When this flag is `true`, formulas often become easier to understand when printed, as in the above example.

## See also

`pp_print_term`, `prebroken_binops`, `print_all_thm`, `print_unambiguous_comprehensions`, `reverse_interface_mapping`, `unspaced_binops`.

# tysubst

`tysubst : (hol_type * hol_type) list -> hol_type -> hol_type`

## Description

The call `tysubst [ty1',ty1; ... ; tyn',tyn] ty` will systematically traverse the type `ty` and replace the topmost instances of any `tyi` encountered with the corresponding `tyi'`. In the (usual) case where all the `tyi` are type variables, this is the same as `type_subst`, but also works when they are not.

## Failure

Never fails. If several `tyi` are the same, the first one in the list will be used to determine the substitution.

## Example

```
# tysubst [`:num`,`:A`; `:bool`,`:B`] `:A->(B)list->A#B#C`;;
val it : hol_type = `:num->(bool)list->num#bool#C`
# tysubst [`:A`,`:(num)list`] `:num->(num)list->(num)list`;;
val it : hol_type = `:num->A->A`
```

## See also

`inst`, `tysubst`.

# tyvars

`tyvars : hol_type -> hol_type list`

## Synopsis
Returns a list of the type variables in a type.

## Description
When applied to a type, `tyvars` returns a list (possibly empty) of the type variables that it involves.

## Failure
Never fails.

## Example

```
# tyvars `:(A->bool)->A`;;
val it : hol_type list = [`:A`]
```

## See also
`type_vars_in_term`.

# uncurry

```
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

## Synopsis
Converts a function taking two arguments into a function taking a single paired argument.

## Description
The application `uncurry f` returns `fun (x,y) -> f x y`, so that

```
uncurry f (x,y) = f x y
```

## Failure
Never fails.

## See also
`curry`.

# undefined

```
undefined : ('a, 'b) func
```

## Synopsis
Completely undefined finite partial function.

## Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The value `undefined` is the 'empty' finite partial function that is nowhere defined.

## Failure
Not applicable.

## Example

```
# (undefined:(string,term)func);;
val it : (string, term) func = <func>
# apply it "anything";;
Exception: Failure "apply".
```

## Uses
Starting a function to be augmented pointwise.

## See also
`|->`, `|=>`, `apply`, `applyd`, `choose`, `combine`, `defined`, `dom`, `foldl`, `foldr`, `graph`, `is_undefined`, `mapf`, `ran`, `tryapplyd`, `undefine`.

# undefine

```
undefine : 'a -> ('a, 'b) func -> ('a, 'b) func
```

## Synopsis
Remove definition of a finite partial function on specific domain value.

## Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. The call `undefine x f` removes a definition for the domain value `x` in the finite partial function `f`; if there was none to begin with the function is unchanged.

## Failure
Never fails.

## Example

```
# let f = itlist I [1 |-> "1"; 2 |-> "2"; 3 |-> "3"] undefined;;
val f : (int, string) func = <func>
# dom f;;
val it : int list = [1; 2; 3]
# dom(undefine 2 f);;
val it : int list = [1; 3]
```

## See also

|->, |=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefined.

# UNDISCH_ALL

UNDISCH_ALL : thm -> thm

## Synopsis

Iteratively undischarges antecedents in a chain of implications.

## Description

```
 A |- t1 ==> ... ==> tn ==> t
------------------------------  UNDISCH_ALL
     A, t1, ..., tn |- t
```

## Failure

Unlike UNDISCH, UNDISCH_ALL will, when called on something other than an implication, return its argument unchanged rather than failing.

## Example

```
# UNDISCH_ALL(TAUT `p ==> q ==> r ==> p /\ q /\ r`);;
val it : thm = p, q, r |- p /\ q /\ r
```

## See also

DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_TAC.

# UNDISCH

UNDISCH : thm -> thm

## Synopsis
Undischarges the antecedent of an implicative theorem.

## Description

```
    A |- t1 ==> t2
   ----------------  UNDISCH
     A, t1 |- t2
```

## Failure
UNDISCH will fail on theorems which are not implications.

## Example

```
  # UNDISCH(TAUT `p /\ q ==> p`);;
  val it : thm = p /\ q |- p
```

## See also
DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, STRIP_TAC, UNDISCH_ALL, UNDISCH_TAC.

# UNDISCH_TAC

```
UNDISCH_TAC : term -> tactic
```

## Synopsis
Undischarges an assumption.

## Description

```
          A ?- t
   ====================  UNDISCH_TAC `v`
    A - {v} ?- v ==> t
```

## Failure
UNDISCH_TAC will fail if `v` is not an assumption.

## Comments
UNDISCHarging `v` will remove all assumptions that are alpha-equivalent to `v`.

## See also
DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_ALL,
UNDISCH_THEN.

# UNDISCH_THEN

```
UNDISCH_THEN : term -> thm_tactic -> tactic
```

## Synopsis
Undischarges an assumption and applies theorem-tactic to it.

## Description
The tactic `UNDISCH_THEN ‘a‘ ttac` applied to a goal `A |- t` takes `a` out of the assumptions to give a goal `A - {a} |- t`, and applies the theorem-tactic `ttac` to the assumption `.. |- a` and that new goal.

## Failure
Fails if `a` is not an assumption; when applied to the goal it fails exactly if the theorem-tactic fails on the modified goal.

## Comments
The tactic `UNDISCH_TAC ‘t‘` can be considered the special case of `UNDISCH_THEN ‘t‘ MP_TAC`.

## See also
`FIND_ASSUM`, `FIRST_X_ASSUM`, `UNDISCH_TAC`.

# unhide_constant

```
unhide_constant : string -> unit
```

## Synopsis
Restores recognition of a constant by the quotation parser.

## Description
A call `unhide_constant "c"`, where `c` is a hidden constant, will unhide the constant, that is, will make the quotation parser recognize it as such rather than parsing it as a variable. It reverses the effect of the call `hide_constant name`.

## Failure
Fails unless the given name is a hidden constant in the current theory.

## Comments
The hiding of a constant only affects the quotation parser; the constant is still there in a theory, and may not be redefined.

## See also

hide_constant, is_hidden.

# UNIFY_ACCEPT_TAC

```
UNIFY_ACCEPT_TAC : term list -> thm -> 'a * term -> ('b list * instantiation) * 'c list * (inst
```

## Synopsis

Unify free variables in theorem and metavariables in goal to accept theorem.

## Description

Given a list `l` of assignable metavariables, a theorem `th` of the form `A |- t` and a goal `A' ?- t'`, the tactic `UNIFY_ACCEPT_TAC` attempts to unify `t` and `t'` by instantiating free variables in `t` and metavariables in the list `l` in the goal `t'` so that they match, then accepts the theorem as the solution of the goal.

## Failure

Fails if no unification will work. In fact, type instantiation is not at present included in the unification.

## Example

An inherently uninteresting but instructive example is the goal:

```
# g `(?x:num. p(x) /\ q(x) /\ r(x)) ==> ?y. p(y) /\ (q(y) <=> r(y))`;;
```

which could of course be solved directly by `MESON_TAC[]` or `ITAUT_TAC`. In fact, the process we will outline is close to what `ITAUT_TAC` does automatically. Let's start with:

```
# e STRIP_TAC;;
val it : goalstack = 1 subgoal (1 total)

 0 [`p x`]
 1 [`q x`]
 2 [`r x`]

`?y. p y /\ (q y <=> r y)`
```

and defer the actual choice of existential witness by introducing a metavariable:

```
# e (X_META_EXISTS_TAC `n:num` THEN CONJ_TAC);;
val it : goalstack = 2 subgoals (2 total)

 0 [`p x`]
 1 [`q x`]
 2 [`r x`]

`q n <=> r n`

 0 [`p x`]
 1 [`q x`]
 2 [`r x`]

`p n`
```

Now we finally fix the metavariable to match our assumption:

```
# e(FIRST_X_ASSUM(UNIFY_ACCEPT_TAC [`n:num`]));;
val it : goalstack = 1 subgoal (1 total)

 0 [`p x`]
 1 [`q x`]
 2 [`r x`]


`q x <=> r x`
```

Note that the metavariable has also been correspondingly instantiated in the remaining goal, which we can solve easily:

```
# e(ASM_REWRITE_TAC[]);;
val it : goalstack = No subgoals
```

## Uses

Terminating proof search when using metavariables. Used in `ITAUT_TAC`

## See also

`ACCEPT_TAC`, `ITAUT`, `ITAUT_TAC`, `MATCH_ACCEPT_TAC`.

# union

```
union : 'a list -> 'a list -> 'a list
```

## Synopsis

Computes the union of two ‘sets’.

## Description

`union l1 l2` returns a list consisting of the elements of `l1` not already in `l2` concatenated with `l2`. If `l1` and `l2` are initially free from duplicates, this gives a set-theoretic union operation.

## Failure

Never fails.

## Example

```
# union [1;2;3] [1;5;4;3];;
val it : int list = [2; 1; 5; 4; 3]
# union [1;1;1] [1;2;3;2];;
val it : int list = [1; 2; 3; 2]
```

## See also

`setify`, `set_equal`, `intersect`, `subtract`.

# union'

```
union' : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

## Synopsis

Union of sets modulo an equivalence.

## Description

The call `union' r l1 l2` appends to the list `l2` all those elements `x` of `l1` for which there is not already an equivalent `x'` with `r x x'` in `l2` or earlier in `l1`. If `l1` and `l2` were free of equivalents under `r`, the resulting list will be too, so this is a set operation modulo an equivalence. The function `union` is the special case where the relation is just equality.

## Failure

Fails only if the function `r` fails.

## Example

```
# union' (fun x y -> abs(x) = abs(y)) [-1; 2; 1] [-2; -3; 4; -4];;
val it : int list = [1; -2; -3; 4; -4]
```

## Uses

Maintaining sets modulo an equivalence such as alpha-equivalence.

## See also

`insert'`, `mem'`, `subtract'`, `union`, `unions'`.

# unions

```
unions : 'a list list -> 'a list
```

## Synopsis

Performs the union of a set of sets.

## Description

Applied to a list of lists, `union` returns a list of all the elements of them, in some unspecified order, with no repetitions. It can be considered as the union of the family of 'sets'.

## Failure

Never fails.

## Example

```
# unions [[1;2]; [2;2;2;]; [2;3;4;5]];;
val it : int list = [1; 2; 3; 4; 5]
```

## See also

`intersect`, `subtract`.

# unions'

```
unions' : ('a -> 'a -> bool) -> 'a list list -> 'a list
```

## Synopsis
Compute union of a family of sets modulo an equivalence.

## Description
If `r` is an equivalence relation an `l` a list of lists, the call `unions' r l` returns a list with one representative of each `r`-equivalence class occurring in any of the members. It thus gives a union of a family of sets with no duplicates under the equivalence `r`.

## Failure
Fails only if the relation `r` fails.

## Example

```
# unions' (fun x y -> abs(x) = abs(y))
    [[-1; 2; 3]; [-2; -3; -4]; [4; 5; -6]];;
val it : int list = [-1; -2; -3; 4; 5; -6]
```

## See also
`insert', mem', subtract', union', unions.`

# uniq

```
uniq : 'a list -> 'a list
```

## Synopsis
Eliminate adjacent identical elements from a list.

## Description
When applied to a list, `uniq` gives a new list that results from coalescing adjacent (only) elements of the list into one.

## Failure
Never fails.

## Example

```
# uniq [1;2;3;1;2;3];;
val it : int list = [1; 2; 3; 1; 2; 3]

# uniq [1;1;1;2;3;3;3;3;4];;
val it : int list = [1; 2; 3; 4]
```

## See also
`setify, sort.`

# unparse_as_binder

`unparse_as_binder : string -> unit`

## Synopsis
Stops the quotation parser from treating a name as a binder.

## Description
Certain identifiers `c` have binder status, meaning that `‘c x. y‘` is parsed as a shirthand for `‘(c) (\x. y)’`. The call `unparse_as_binder "c"` will remove `c` from the list of binders if it is there.

## Failure
Never fails, even if the string was not a binder.

## Example

```
# ‘!x. x < 2‘;;
val it : term = ‘!x. x < 2‘

# unparse_as_binder "!";;
val it : unit = ()
# ‘!x. x < 2‘;;
Exception: Failure "Unexpected junk after term".
```

## Comments
Removing binder status for the pre-existing binders like the quantifiers should only be done with great care, since it can cause other parser invocations to break.

## See also
`binders`, `parses_as_binder`, `parse_as_binder`.

# unparse_as_infix

`unparse_as_infix : string -> unit`

## Synopsis
Removes string from the list of infix operators.

## Description
Certain identifiers are treated as infix operators with a given precedence and associativity (left or right). The call `unparse_as_infix "op"` removes `op` from the list of infix identifiers, if it was indeed there.

## Failure
Never fails, even if the given string did not originally have infix status.

## Comments
Take care with applying this to some of the built-in operators, or parsing may fail in existing libraries.

## See also
`get_infix_status`, `infixes`, `parse_as_infix`.

# unparse_as_prefix

```
unparse_as_prefix : string -> unit
```

## Synopsis
Removes prefix status for an identifier.

## Description
Certain identifiers `c` have prefix status, meaning that combinations of the form `c f x` will be parsed as `c (f x)` rather than the usual `(c f) x`. The call `unparse_as_prefix "c"` removes `c` from the list of such identifiers.

## Failure
Never fails, regardless of whether `c` originally did have prefix status.

## See also
`is_prefix`, `parse_as_prefix`, `prefixes`.

# unreserve_words

```
unreserve_words : string list -> unit
```

## Synopsis
Remove given strings from the set of reserved words.

## Description

Certain identifiers in HOL are reserved, e.g. '`if`', '`let`' and '`|`', meaning that they are special to the parser and cannot be used as ordinary identifiers. The call `unreserve_words l` removes all strings in `l` from the list of reserved identifiers.

## Failure

Never fails, regardless of whether the given strings were in fact reserved.

## Comments

The initial set of reserved words in HOL Light should be unreserved only with great care, since then various elementary constructs may fail to parse.

## See also

`is_reserved_word`, `reserved_words`, `reserve_words`.

# unspaced_binops

```
unspaced_binops : string list ref
```

## Synopsis

Determines which binary operators are printed with surrounding spaces.

## Description

The reference variable `unspaced_binops` is one of several settable parameters controlling printing of terms by `pp_print_term`, and hence the automatic printing of terms and theorems at the toplevel. It holds a list of the names of infix binary operators that are printed without surrounding spaces. By default, it contains just the pairing operation '`,`', the numeric range '`..`' and the cartesian power indexing '`$`'.

## Failure

Not applicable.

## Example

```
# `x + 1`;;
val it : term = `x + 1`

# unspaced_binops := "+"::(!unspaced_binops);;
val it : unit = ()
# `x + 1`;;
val it : term = `x+1`
```

## See also

`pp_print_term`, `prebroken_binops`, `print_all_thm`, `print_unambiguous_comprehensions`, `reverse_interface_mapping`, `typify_universal_set`.

# unzip

`unzip : ('a * 'b) list -> 'a list * 'b list`

## Synopsis
Converts a list of pairs into a pair of lists.

## Description
`unzip [(x1,y1);...;(xn,yn)]` returns `([x1;...;xn],[y1;...;yn])`.

## Failure
Never fails.

## See also
`zip`.

# --

`(--) : int -> int -> int list`

## Synopsis
Gives a finite list of integers between the given bounds.

## Description
The call `m--n` returns the list of consecutive numbers from `m` to `n`.

## Example

```
# 1--10;;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# 5--5;;
val it : int list = [5]
# (-1)--1;;
val it : int list = [-1; 0; 1]
# 2--1;;
val it : int list = []
```

# use_file

`use_file : string -> unit`

## Synopsis
Load a file, much like OCaml's #use directive.

## Description
Essentially the same as OCaml's #use directive, but a regular OCaml function and therefore easier to exploit programmatically.

## Failure
Only fails if the included file causes failure.

## See also
loads, loadt.

# USE_THEN

```
USE_THEN : string -> thm_tactic -> tactic
```

## Synopsis
Apply a theorem tactic to named assumption, removing the assumption.

## Description
The tactic USE_THEN "name" ttac applies the theorem-tactic ttac to the assumption labelled name (or the first in the list if there is more than one).

## Failure
Fails if there is no assumption of that name or if the theorem-tactic fails when applied to it.

## Example
See LABEL_TAC for an extended example.

## Uses
Using an assumption identified by name.

## See also
ASSUME, FIND_ASSUM, LABEL_TAC, REMOVE_THEN.

# VALID

```
VALID : tactic -> tactic
```

## Synopsis
Tries to ensure that a tactic is valid.

## Description
For any tactic `t`, the application `VALID t` gives a new tactic that does exactly the same as `t` except that it also checks validity of the tactic and will fail if it is violated. Validity means that the subgoals produced by `t` can, if proved, be used by the justification function given by `t` to construct a theorem corresponding to the original goal.

This check is performed by actually creating, using `mk_fthm`, theorems corresponding to the subgoals, and seeing if the result of applying the justification function to them gives a theorem corresponding to the original goal. If it does, then `VALID t` simply applies `t`, and if not it fails. In principle, the extra dummy hypothesis used by `mk_fthm` (necessary to ensure logical soundness) could interfere with the mechanism of the tactic, but this never seems to happen.

## Comments
You can always force validity checking whenever it is applied by using `VALID` on a tactic. But if the goal is initially proved by using the subgoal stack this is probably not necessary since `VALID` is already implicitly applied in the `e` (expand) function.

## See also
`CHANGED_TAC, e, mk_fthm, TRY.`

# |->

```
(|->) : 'a -> 'b -> ('a, 'b) func -> ('a, 'b) func
```

## Synopsis
Modify a finite partial function at one point.

## Description
This is one of a suite of operations on finite partial functions, type `('a,'b)func`. These may sometimes be preferable to ordinary functions since they permit more operations such as equality comparison, extraction of domain etc. If `f` is a finite partial function then `(x |-> y) f` gives a modified version that maps `x` to `y` (whether or not `f` was defined on `x` before and regardless of the old value) but is otherwise the same.

## Failure
Never fails.

## Example

```
# let f = (1 |-> 2) undefined;;
val f : (int, int) func = <func>
# let g = (1 |-> 3) f;;
val g : (int, int) func = <func>
# apply f 1;;
val it : int = 2
# apply g 1;;
val it : int = 3
```

## See also

|=>, apply, applyd, choose, combine, defined, dom, foldl, foldr, graph, is_undefined, mapf, ran, tryapplyd, undefine, undefined.

# variables

variables : term -> term list

## Synopsis

Determines the variables used, free or bound, in a given term.

## Description

Given a term argument, variables returns a list of variables that occur free or bound in that term.

## Example

```
# variables `\a:bool. a`;;
val it : term list = [`a`]
# variables `(a:num) + (b:num)`;;
val it : term list = [`b`; `a`]
```

## See also

frees, free_in.

# variant

variant : term list -> term -> term

## Synopsis
Modifies a variable name to avoid clashes.

## Description
When applied to a list of variables to avoid clashing with, and a variable to modify, `variant` returns a variant of the variable to modify, that is, it changes the name as intuitively as possible to make it distinct from any variables in the list. This is normally done by adding primes to the name.

The exact form of the variable name should not be relied on, except that the original variable will be returned unmodified unless it is itself in the list to avoid clashing with.

## Failure
`variant l t` fails if any term in the list `l` is not a variable or if `t` is neither a variable nor a constant.

## Example
The following shows a few typical cases:

```
# variant [`y:bool`; `z:bool`] `x:bool`;;
val it : term = `x`

# variant [`x:bool`; `x':num`; `x'':num`] `x:bool`;;
val it : term = `x'`

# variant [`x:bool`; `x':bool`; `x'':bool`] `x:bool`;;
val it : term = `x'''`
```

## Uses
The function `variant` is extremely useful for complicated derived rules which need to rename variables to avoid free variable capture while still making the role of the variable obvious to the user.

## See also
`genvar, hide_constant.`

# variants

```
variants : term list -> term list -> term list
```

## Synopsis
Pick a list of variants of variables, avoiding a list of variables and each other.

## Description

The call `variants av vs`,s where `av` and `vs` are both lists of variables, will return a list `vs'` of variants of the variables in the list `vs`, renamed as necessary by adding primes to avoid clashing with any in the list `av` or with each other.

## Failure

Fails if any of the terms in the list is not a variable.

## Example

```
# variants [`x':num`; `x'':num`; `y:bool`] [`x:num`; `x':num`];;
val it : term list = [`x`; `x'''`]
```

## See also

`genvar, mk_primed_var, variant.`

# verbose

```
verbose : bool ref
```

## Synopsis

Flag to control verbosity of informative output.

## Description

When the value of `verbose` is set to `true`, the function `remark` will output its string argument whenever called. This is used for most informative output in automated rules.

## Failure

Not applicable.

## Example

Consider this call MESON to prove a first-order formula:

```
# MESON[] `!f g:num->num. (?!x. x = g(f(x))) <=> (?!y. y = f(g(y)))`;;
0..0..1..solved at 4
CPU time (user): 0.01
0..0..1..2..6..11..19..28..37..46..94..151..247..366..584..849..solved at 969
CPU time (user): 0.12
0..0..1..solved at 4
CPU time (user): 0.
0..0..1..2..6..11..19..28..37..46..94..151..247..366..584..849..solved at 969
CPU time (user): 0.06
val it : thm = |- !f g. (?!x. x = g (f x)) <=> (?!y. y = f (g y))
```

By changing the verbosity level, most of the output disappears:

```
# verbose := false;;
val it : unit = ()
# MESON[] `!f g:num->num. (?!x. x = g(f(x))) <=> (?!y. y = f(g(y)))`;;
CPU time (user): 0.01
CPU time (user): 0.13
CPU time (user): 0.
CPU time (user): 0.081
val it : thm = |- !f g. (?!x. x = g (f x)) <=> (?!y. y = f (g y))
```

and if we also disable timing reporting the action is silent:

```
# report_timing := false;;
val it : unit = ()
# MESON[] `!f g:num->num. (?!x. x = g(f(x))) <=> (?!y. y = f(g(y)))`;;
val it : thm = |- !f g. (?!x. x = g (f x)) <=> (?!y. y = f (g y))
```

## See also

`remark`, `report_timing`.

# vfree_in

`vfree_in : term -> term -> bool`

## Synopsis

Teste whether a variable (or constant) occurs free in a term.

## Description

The call `vfree_in v t`, where `v` is a variable (or constant, though this is not usually exploited) and `t` any term, tests whether `v` occurs free in `t`, and returns `true` if so, `false` if not. This is functionally equivalent to `mem v (frees t)` but may be more efficient because it never constructs the list of free variables explicitly.

## Failure

Never fails.

## Example

Here's a simple example:

```
# vfree_in `x:num` `x + y + 1`;;
val it : bool = true

# vfree_in `x:num` `x /\ y /\ z`;;
val it : bool = false
```

To see how using `vfree_in` can be more efficient than examining the free variable list explicitly, consider a huge term with one free and one bound variable:

```
# let tm = mk_abs(`p:bool`,funpow 17 (fun s -> mk_conj(s,s)) `p /\ q`);;
....
```

It takes an appreciable time to get the list of free variables:

```
# time frees tm;;
CPU time (user): 0.31
val it : term list = [`q`]
```

yet we can test if `p` or `q` is free almost instantaneously. Only a little of the term needs to be traversed to find the answer (just one level in the case of `p`, since it is bound at the outer term constructor).

```
#  time (vfree_in `q:bool`) tm;;
CPU time (user): 0.
val it : bool = true
```

## See also

`free_in, frees, freesin.`

# vsubst

```
vsubst : (term * term) list -> term -> term
```

## Synopsis
Substitute terms for variables inside a term.

## Description
The call `vsubst [t1,x1; ...; tn,xn] t` systmatically replaces free instances of each variable `xi` inside `t` with the corresponding `ti` from the instantiation list. Bound variables will be renamed if necessary to avoid capture.

## Failure
Fails if any of the pairs `ti,xi` in the instantiation list has `xi` and `ti` with different types, or `xi` a non-variable. Multiple instances of the same `xi` in the list are not trapped, but only the first one will be used consistently.

## Example
Here is a relatively simple example

```
# vsubst [`1`,`x:num`; `2`,`y:num`] `x + y + 3`;;
val it : term = `1 + 2 + 3`
```

and here is a more complex instance where renaming of bound variables is needed:

```
# vsubst [`y:num`,`x:num`] `!y. x + y < x + y + 1`;;
val it : term = `!y'. y + y' < y + y' + 1`
```

## Comments
An analogous function `subst` is more general, and will substitute for free occurrences of any term, not just variables. However, `vsubst` is generally much more efficient if you do just need substitution for variables.

## See also
`inst`, `subst`.

# warn

```
warn : bool -> string -> unit
```

## Synopsis
Prints out a warning string

## Description
When applied to a boolean value `b` and a string `s`, the call `warn b s` prints out "`Warning: s`" and a following newline to the terminal if `b` is true and otherwise does nothing.

## Failure
Never fails.

## Example

```
# let n = 7;;
val n : int = 7
# warn (n <> 0) "Nonzero value";;
Warning: Nonzero value
val it : unit = ()
```

## See also

`remark`, `report`.

# W

`W : ('a -> 'a -> 'b) -> 'a -> 'b`

## Synopsis

Duplicates function argument : `W f x` = `f x x`.

## Failure

Never fails.

## See also

`C`, `F_F`, `I`, `K`, `o`.

# WEAK_CNF_CONV

`WEAK_CNF_CONV : conv`

## Synopsis

Converts a term already in negation normal form into conjunctive normal form.

## Description

When applied to a term already in negation normal form (see `NNF_CONV`), meaning that all other propositional connectives have been eliminated in favour of conjunction, disjunction and negation, and negation is only applied to atomic formulas, `WEAK_CNF_CONV` puts the term into an equivalent conjunctive normal form, which is a conjunction of disjunctions.

## Failure

Never fails; non-Boolean terms will just yield a reflexive theorem.

## Example

```
# WEAK_CNF_CONV `(a /\ b) \/ (a /\ b /\ c) \/ d`;;
val it : thm =
  |- a /\ b \/ a /\ b /\ c \/ d <=>
     ((a \/ a \/ d) /\ (b \/ a \/ d)) /\
     ((a \/ b \/ d) /\ (b \/ b \/ d)) /\
     (a \/ c \/ d) /\
     (b \/ c \/ d)
```

## Comments

The ordering and associativity of the resulting form are not guaranteed, and it may contain duplicates. See CNF_CONV for a stronger (but somewhat slower) variant where this is important.

## See also

CNF_CONV, DNF_CONV, NNF_CONV, WEAK_DNF_CONV.

# WEAK_DNF_CONV

```
WEAK_DNF_CONV : conv
```

## Synopsis

Converts a term already in negation normal form into disjunctive normal form.

## Description

When applied to a term already in negation normal form (see NNF_CONV), meaning that all other propositional connectives have been eliminated in favour of disjunction, disjunction and negation, and negation is only applied to atomic formulas, WEAK_DNF_CONV puts the term into an equivalent disjunctive normal form, which is a disjunction of conjunctions.

## Failure

Never fails; non-Boolean terms will just yield a reflexive theorem.

## Example

```
# WEAK_DNF_CONV `(a \/ b) /\ (a \/ c /\ e)`;;
val it : thm =
  |- (a \/ b) /\ (a \/ c /\ e) <=>
     (a /\ a \/ b /\ a) \/ a /\ c /\ e \/ b /\ c /\ e
```

## Comments

The ordering and associativity of the resulting form are not guaranteed, and it may contain duplicates. See DNF_CONV for a stronger (but somewhat slower) variant where this is important.

## See also

CNF_CONV, DNF_CONV, NNF_CONV, WEAK_CNF_CONV.

# WF_INDUCT_TAC

```
WF_INDUCT_TAC : term -> (string * thm) list * term -> goalstate
```

## Synopsis

Performs wellfounded induction with respect to a given 'measure'.

## Description

The tactic `WF_INDUCT_TAC` is applied to two arguments. The second is a goal to prove, and the first is an expression to use as a "measure". The result is a new subgoal where the same goal is to be proved but as an assumption it holds for all smaller values of the measure, universally quantified over the free variables in the measure term (which should also be free in the goal).

## Failure

Never fails.

## Example

Suppose we define a Euclidean GCD algorithm:

```
# let egcd = define
   `egcd(m,n) = if m = 0 then n
                else if n = 0 then m
                else if m <= n then egcd(m,n - m)
                else egcd(m - n,n)`;;
```

and after picking up from the library an infix '`divides`' relation for divisibility:

```
# needs "Examples/prime.ml";;
```

we want to prove something about the result, e.g.

```
# g `!m n d. d divides egcd(m,n) <=> d divides m /\ d divides n`;;
```

A natural way to proceed is by induction on the sum of the arguments:

```
# e(GEN_TAC THEN GEN_TAC THEN WF_INDUCT_TAC `m + n`);;
val it : goalstack = 1 subgoal (1 total)

 0 [`!m'' n'.
          m'' + n' < m + n
          ==> (!d. d divides egcd (m'',n') <=> d divides m'' /\ d divides n')`]

`!d. d divides egcd (m,n) <=> d divides m /\ d divides n`
```

Note that we have the same goal, but an assumption that it holds for smaller values of the measure term.

## Comments

Wellfounded induction can always be performed on any relation by using WF_IND together with an assumption of wellfoundedness such as num_WF or WF_MEASURE. This tactic is just a slightly more convenient packaging.

## See also

INDUCT_TAC, LIST_INDUCT_TAC.

# X_CHOOSE_TAC

```
X_CHOOSE_TAC : term -> thm_tactic
```

## Synopsis

Assumes a theorem, with existentially quantified variable replaced by a given witness.

## Description

X_CHOOSE_TAC expects a variable y and theorem with an existentially quantified conclusion. When applied to a goal, it adds a new assumption obtained by introducing the variable y as a witness for the object x whose existence is asserted in the theorem.

```
          A ?- t
   ===================  X_CHOOSE_TAC `y` (A1 |- ?x. w)
    A u {w[y/x]} ?- t         (`y` not free anywhere)
```

## Failure

Fails if the theorem's conclusion is not existentially quantified, or if the first argument is not a variable. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in w or t, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

## Example

Given a goal:

```
# g `(?y. x = y + 2) ==> x < x * x`;;
```

the following may be applied:

```
# e(DISCH_THEN(X_CHOOSE_TAC `d:num`));;
val it : goalstack = 1 subgoal (1 total)

 0 [`x = d + 2`]

`x < x * x`
```

after which the following will finish things:

```
# e(ASM_REWRITE_TAC[] THEN ARITH_TAC);;
val it : goalstack = No subgoals
```

## See also

CHOOSE, CHOOSE_THEN, X_CHOOSE_THEN.

# X_CHOOSE_THEN

```
X_CHOOSE_THEN : term -> thm_tactical
```

## Synopsis

Replaces existentially quantified variable with given witness, and passes it to a theorem-tactic.

## Description

X_CHOOSE_THEN expects a variable y, a tactic-generating function ttac, and a theorem of the form (A1 |- ?x. w) as arguments. A new theorem is created by introducing the given variable y as a witness for the object x whose existence is asserted in the original theorem, (w[y/x] |- w[y/x]). If the tactic-generating function ttac applied to this theorem produces results as follows when applied to a goal (A ?- t):

```
  A ?- t
=========  ttac ({w[y/x]} |- w[y/x])
  A ?- t1
```

then applying (X_CHOOSE_THEN `y` ttac (A1 |- ?x. w)) to the goal (A ?- t) produces the subgoal:

```
    A ?- t
   =========  X_CHOOSE_THEN `y` ttac (A1 |- ?x. w)
    A ?- t1         (`y` not free anywhere)
```

## Failure

Fails if the theorem's conclusion is not existentially quantified, or if the first argument is not a variable. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in `w` or `t`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

## Example

Suppose we have the following goal:

```
  # g `!m n. m < n ==> m EXP 2 + 2 * m <= n EXP 2`;;
```

and rewrite with a theorem to get an existential antecedent:

```
  # e(REPEAT GEN_TAC THEN REWRITE_TAC[LT_EXISTS]);;
  val it : goalstack = 1 subgoal (1 total)

`(?d. n = m + SUC d) ==> m EXP 2 + 2 * m <= n EXP 2`
```

we may then use `X_CHOOSE_THEN` to introduce the name `e` for the existential variable and immediately substitute it in the goal:

```
  # e(DISCH_THEN(X_CHOOSE_THEN `e:num` SUBST1_TAC));;
  val it : goalstack = 1 subgoal (1 total)

  `m EXP 2 + 2 * m <= (m + SUC e) EXP 2`
```

at which point `ARITH_TAC` will finish it.

## See also

CHOOSE, CHOOSE_THEN, CONJUNCTS_THEN, CONJUNCTS_THEN2, DISJ_CASES_THEN, DISJ_CASES_THEN2, STRIP_THM_THEN, X_CHOOSE_TAC.

# X_GEN_TAC

```
X_GEN_TAC : term -> tactic
```

## Synopsis

Specializes a goal with the given variable.

## Description

When applied to a term `x'`, which should be a variable, and a goal `A ?- !x. t`, the tactic `X_GEN_TAC` returns the goal `A ?- t[x'/x]`.

```
     A ?- !x. t
   ==============  X_GEN_TAC `x'`
    A ?- t[x'/x]
```

## Failure

Fails unless the goal's conclusion is universally quantified and the term a variable of the appropriate type. It also fails if the variable given is free in either the assumptions or (initial) conclusion of the goal.

## Uses

It is perhaps good practice to use this rather than `GEN_TAC`, to ensure that there is no dependency on the bound variable name in the goal, which can sometimes arise somewhat arbitrarily, e.g. in higher-order matching.

## See also

`GEN, GENL, GEN_ALL, SPEC, SPECL, SPEC_ALL, SPEC_TAC, STRIP_TAC.`

# X_META_EXISTS_TAC

```
X_META_EXISTS_TAC : term -> tactic
```

## Synopsis

Replaces existentially quantified variable with given metavariables.

## Description

Given a variable `v` and a goal of the form `A ?- ?x. t[x]`, the tactic `X_META_EXISTS_TAC` gives the new goal `A ?- t[v]` where `v` is a new metavariable. In the resulting proof, it is as if the variable has been assigned here to the later choice for this metavariable, which can be made through `UNIFY_ACCEPT_TAC`.

## Failure

Fails if the metavariable is not a variable.

## Example

See `UNIFY_ACCEPT_TAC` for an example of using metavariables.

## Uses

Delaying instantiations until the correct term becomes clearer.

## Comments

Users should probably steer clear of using metavariables if possible. Note that the metavariable instantiations apply across the whole fringe of goals, not just the current goal, and can lead to confusion.

## See also

`EXISTS_TAC`, `META_EXISTS_TAC`, `META_SPEC_TAC`, `UNIFY_ACCEPT_TAC`.

# zip

```
zip : 'a list -> 'b list -> ('a * 'b) list
```

## Synopsis

Converts a pair of lists into a list of pairs.

## Description

`zip [x1;...;xn] [y1;...;yn]` returns `[(x1,y1);...;(xn,yn)]`.

## Failure

Fails if the two lists are of different lengths.

## See also

`unzip`.

Chapter 2

# Pre-proved Theorems

The sections that follow list the most useful theorems built into the HOL Light system, which are proved and bound to ML identifiers when the system is built. Some theorems that were felt (subjectively) unlikely to be useful for most HOL Light users are omitted. Within the broad groupings, theorems are listed in alphabetical order, by the name of the OCaml identifier to which they are bound.

## 2.1 Theorems about basic logical notions

```
ABS_SIMP
    |- !t1 t2. (\x. t1) t2 = t1

AND_CLAUSES
    |- !t. (T /\ t <=> t) /\
           (t /\ T <=> t) /\
           (F /\ t <=> F) /\
           (t /\ F <=> F) /\
           (t /\ t <=> t)

AND_DEF
    |- (/\) = (\p q. (\f. f p q) = (\f. f T T))

AND_FORALL_THM
    |- !P Q. (!x. P x) /\ (!x. Q x) <=> (!x. P x /\ Q x)

BETA_THM
    |- !f y. (\x. f x) y = f y

BOOL_CASES_AX
    |- !t. (t <=> T) \/ (t <=> F)

COND_ABS
    |- !b f g. (\x. if b then f x else g x) = (if b then f else g)

COND_CLAUSES
    |- !t1 t2. (if T then t1 else t2) = t1 /\ (if F then t1 else t2) = t2

COND_DEF
    |- COND = (\t t1 t2. @x. ((t <=> T) ==> x = t1) /\ ((t <=> F) ==> x = t2))

COND_ELIM_THM
    |- P (if c then x else y) <=> (c ==> P x) /\ (~c ==> P y)

COND_EXPAND
    |- !b t1 t2. (if b then t1 else t2) <=> (~b \/ t1) /\ (b \/ t2)

COND_ID
    |- !b t. (if b then t else t) = t

COND_RAND
    |- !b f x y. f (if b then x else y) = (if b then f x else f y)

COND_RATOR
    |- !b f g x. (if b then f else g) x = (if b then f x else g x)
```

```
CONJ_ACI
    |- (p /\ q <=> q /\ p) /\
       ((p /\ q) /\ r <=> p /\ q /\ r) /\
       (p /\ q /\ r <=> q /\ p /\ r) /\
       (p /\ p <=> p) /\
       (p /\ p /\ q <=> p /\ q)

CONJ_ASSOC
    |- !t1 t2 t3. t1 /\ t2 /\ t3 <=> (t1 /\ t2) /\ t3

CONJ_SYM
    |- !t1 t2. t1 /\ t2 <=> t2 /\ t1

CONTRAPOS_THM
    |- !t1 t2. ~t1 ==> ~t2 <=> t2 ==> t1

DE_MORGAN_THM
    |- !t1 t2. (~(t1 /\ t2) <=> ~t1 \/ ~t2) /\ (~(t1 \/ t2) <=> ~t1 /\ ~t2)

DISJ_ACI
    |- (p \/ q <=> q \/ p) /\
       ((p \/ q) \/ r <=> p \/ q \/ r) /\
       (p \/ q \/ r <=> q \/ p \/ r) /\
       (p \/ p <=> p) /\
       (p \/ p \/ q <=> p \/ q)

DISJ_ASSOC
    |- !t1 t2 t3. t1 \/ t2 \/ t3 <=> (t1 \/ t2) \/ t3

DISJ_SYM
    |- !t1 t2. t1 \/ t2 <=> t2 \/ t1

EQ_CLAUSES
    |- !t. ((T <=> t) <=> t) /\
           ((t <=> T) <=> t) /\
           ((F <=> t) <=> ~t) /\
           ((t <=> F) <=> ~t)

EQ_EXT
    |- !f g. (!x. f x = g x) ==> f = g

EQ_IMP
    |- (a <=> b) ==> a ==> b

EQ_REFL
    |- !x. x = x

EQ_REFL_T
    |- !x. x = x <=> T
```

```
EQ_SYM
    |- !x y. x = y ==> y = x

EQ_SYM_EQ
    |- !x y. x = y <=> y = x

EQ_TRANS
    |- !x y z. x = y /\ y = z ==> x = z

ETA_AX
    |- !t. (\x. t x) = t

EXCLUDED_MIDDLE
    |- !t. t \/ ~t

EXISTS_BOOL_THM
    |- (?b. P b) <=> P T \/ P F

EXISTS_DEF
    |- (?) = (\P. !q. (!x. P x ==> q) ==> q)

EXISTS_NOT_THM
    |- !P. (?x. ~P x) <=> ~(!x. P x)

EXISTS_OR_THM
    |- !P Q. (?x. P x \/ Q x) <=> (?x. P x) \/ (?x. Q x)

EXISTS_REFL
    |- !a. ?x. x = a

EXISTS_SIMP
    |- !t. (?x. t) <=> t

EXISTS_THM
    |- (?) = (\P. P ((@) P))

EXISTS_UNIQUE
    |- !P. (?!x. P x) <=> (?x. P x /\ (!y. P y ==> y = x))

EXISTS_UNIQUE_ALT
    |- !P. (?!x. P x) <=> (?x. !y. P y <=> x = y)

EXISTS_UNIQUE_DEF
    |- (?!) = (\P. (?) P /\ (!x y. P x /\ P y ==> x = y))

EXISTS_UNIQUE_REFL
    |- !a. ?!x. x = a

EXISTS_UNIQUE_THM
    |- !P. (?!x. P x) <=> (?x. P x) /\ (!x x'. P x /\ P x' ==> x = x')
```

```
FORALL_AND_THM
    |- !P Q. (!x. P x /\ Q x) <=> (!x. P x) /\ (!x. Q x)

FORALL_BOOL_THM
    |- (!b. P b) <=> P T /\ P F

FORALL_DEF
    |- (!) = (\P. P = (\x. T))

FORALL_NOT_THM
    |- !P. (!x. ~P x) <=> ~(?x. P x)

FORALL_SIMP
    |- !t. (!x. t) <=> t

FUN_EQ_THM
    |- !f g. f = g <=> (!x. f x = g x)

F_DEF
    |- F <=> (!p. p)

IMP_CLAUSES
    |- !t. (T ==> t <=> t) /\
           (t ==> T <=> T) /\
           (F ==> t <=> T) /\
           (t ==> t <=> T) /\
           (t ==> F <=> ~t)

IMP_CONJ
    |- p /\ q ==> r <=> p ==> q ==> r

IMP_DEF
    |- (==>) = (\p q. p /\ q <=> p)

IMP_IMP
    |- p ==> q ==> r <=> p /\ q ==> r

LEFT_AND_EXISTS_THM
    |- !P Q. (?x. P x) /\ Q <=> (?x. P x /\ Q)

LEFT_AND_FORALL_THM
    |- !P Q. (!x. P x) /\ Q <=> (!x. P x /\ Q)

LEFT_EXISTS_AND_THM
    |- !P Q. (?x. P x /\ Q) <=> (?x. P x) /\ Q

LEFT_EXISTS_IMP_THM
    |- !P Q. (?x. P x ==> Q) <=> (!x. P x) ==> Q
```

```
LEFT_FORALL_IMP_THM
    |- !P Q. (!x. P x ==> Q) <=> (?x. P x) ==> Q

LEFT_FORALL_OR_THM
    |- !P Q. (!x. P x \/ Q) <=> (!x. P x) \/ Q

LEFT_IMP_EXISTS_THM
    |- !P Q. (?x. P x) ==> Q <=> (!x. P x ==> Q)

LEFT_IMP_FORALL_THM
    |- !P Q. (!x. P x) ==> Q <=> (?x. P x ==> Q)

LEFT_OR_DISTRIB
    |- !p q r. p /\ (q \/ r) <=> p /\ q \/ p /\ r

LEFT_OR_EXISTS_THM
    |- !P Q. (?x. P x) \/ Q <=> (?x. P x \/ Q)

LEFT_OR_FORALL_THM
    |- !P Q. (!x. P x) \/ Q <=> (!x. P x \/ Q)

MONO_AND
    |- (A ==> B) /\ (C ==> D) ==> A /\ C ==> B /\ D

MONO_COND
    |- (A ==> B) /\ (C ==> D) ==> (if b then A else C) ==> (if b then B else D)

MONO_EXISTS
    |- (!x. P x ==> Q x) ==> (?x. P x) ==> (?x. Q x)

MONO_FORALL
    |- (!x. P x ==> Q x) ==> (!x. P x) ==> (!x. Q x)

MONO_IMP
    |- (B ==> A) /\ (C ==> D) ==> (A ==> C) ==> B ==> D

MONO_NOT
    |- (B ==> A) ==> ~A ==> ~B

MONO_OR
    |- (A ==> B) /\ (C ==> D) ==> A \/ C ==> B \/ D

NOT_CLAUSES
    |- (!t. ~ ~t <=> t) /\ (~T <=> F) /\ (~F <=> T)

NOT_CLAUSES_WEAK
    |- (~T <=> F) /\ (~F <=> T)

NOT_DEF
    |- (~) = (\p. p ==> F)
```

```
NOT_EXISTS_THM
    |- !P. ~(?x. P x) <=> (!x. ~P x)

NOT_FORALL_THM
    |- !P. ~(!x. P x) <=> (?x. ~P x)

NOT_IMP
    |- !t1 t2. ~(t1 ==> t2) <=> t1 /\ ~t2

OR_CLAUSES
    |- !t. (T \/ t <=> T) /\
           (t \/ T <=> T) /\
           (F \/ t <=> t) /\
           (t \/ F <=> t) /\
           (t \/ t <=> t)

OR_DEF
    |- (\/) = (\p q. !r. (p ==> r) ==> (q ==> r) ==> r)

OR_EXISTS_THM
    |- !P Q. (?x. P x) \/ (?x. Q x) <=> (?x. P x \/ Q x)

REFL_CLAUSE
    |- !x. x = x <=> T

RIGHT_AND_EXISTS_THM
    |- !P Q. P /\ (?x. Q x) <=> (?x. P /\ Q x)

RIGHT_AND_FORALL_THM
    |- !P Q. P /\ (!x. Q x) <=> (!x. P /\ Q x)

RIGHT_EXISTS_AND_THM
    |- !P Q. (?x. P /\ Q x) <=> P /\ (?x. Q x)

RIGHT_EXISTS_IMP_THM
    |- !P Q. (?x. P ==> Q x) <=> P ==> (?x. Q x)

RIGHT_FORALL_IMP_THM
    |- !P Q. (!x. P ==> Q x) <=> P ==> (!x. Q x)

RIGHT_FORALL_OR_THM
    |- !P Q. (!x. P \/ Q x) <=> P \/ (!x. Q x)

RIGHT_IMP_EXISTS_THM
    |- !P Q. P ==> (?x. Q x) <=> (?x. P ==> Q x)

RIGHT_IMP_FORALL_THM
    |- !P Q. P ==> (!x. Q x) <=> (!x. P ==> Q x)
```

```
RIGHT_OR_DISTRIB
    |- !p q r. (p \/ q) /\ r <=> p /\ r \/ q /\ r

RIGHT_OR_EXISTS_THM
    |- !P Q. P \/ (?x. Q x) <=> (?x. P \/ Q x)

RIGHT_OR_FORALL_THM
    |- !P Q. P \/ (!x. Q x) <=> (!x. P \/ Q x)

SELECT_AX
    |- !P x. P x ==> P ((@) P)

SELECT_REFL
    |- !x. (@y. y = x) = x

SELECT_UNIQUE
    |- !P x. (!y. P y <=> y = x) ==> (@) P = x

SKOLEM_THM
    |- !P. (!x. ?y. P x y) <=> (?y. !x. P x (y x))

SWAP_EXISTS_THM
    |- !P. (?x y. P x y) <=> (?y x. P x y)

SWAP_FORALL_THM
    |- !P. (!x y. P x y) <=> (!y x. P x y)

TRIV_AND_EXISTS_THM
    |- !P Q. (?x. P) /\ (?x. Q) <=> (?x. P /\ Q)

TRIV_EXISTS_AND_THM
    |- !P Q. (?x. P /\ Q) <=> (?x. P) /\ (?x. Q)

TRIV_EXISTS_IMP_THM
    |- !P Q. (?x. P ==> Q) <=> (!x. P) ==> (?x. Q)

TRIV_FORALL_IMP_THM
    |- !P Q. (!x. P ==> Q) <=> (?x. P) ==> (!x. Q)

TRIV_FORALL_OR_THM
    |- !P Q. (!x. P \/ Q) <=> (!x. P) \/ (!x. Q)

TRIV_OR_FORALL_THM
    |- !P Q. (!x. P) \/ (!x. Q) <=> (!x. P \/ Q)

TRUTH
    |- T

T_DEF
    |- T <=> (\p. p) = (\p. p)
```

```
UNIQUE_SKOLEM_ALT
    |- !P. (!x. ?!y. P x y) <=> (?f. !x y. P x y <=> f x = y)

UNIQUE_SKOLEM_THM
    |- !P. (!x. ?!y. P x y) <=> (?!f. !x. P x (f x))

UNWIND_THM1
    |- !P a. (?x. a = x /\ P x) <=> P a

UNWIND_THM2
    |- !P a. (?x. x = a /\ P x) <=> P a

bool_INDUCT
    |- !P. P F /\ P T ==> (!x. P x)

bool_RECURSION
    |- !a b. ?f. f F = a /\ f T = b
```

## 2.2 Theorems about elementary constructs

```
EXISTS_ONE_REP
    |- ?b. b

I_DEF
    |- I = (\x. x)

I_O_ID
    |- !f. I o f = f /\ f o I = f

I_THM
    |- !x. I x = x

OUTL
    |- OUTL (INL x) = x

OUTR
    |- OUTR (INR y) = y

o_ASSOC
    |- !f g h. f o g o h = (f o g) o h

o_DEF
    |- !f g. f o g = (\x. f (g x))

o_THM
    |- !f g x. (f o g) x = f (g x)
```

```
one
    |- !v. v = one

one_Axiom
    |- !e. ?!fn. fn one = e

one_DEF
    |- one = (@x. T)

one_INDUCT
    |- !P. P one ==> (!x. P x)

one_RECURSION
    |- !e. ?fn. fn one = e

one_axiom
    |- !f g. f = g

one_tydef
    |- (!a. one_ABS (one_REP a) = a) /\ (!r. r <=> one_REP (one_ABS r) <=> r)

option_INDUCT
    |- !P. P NONE /\ (!a. P (SOME a)) ==> (!x. P x)

option_RECURSION
    |- !NONE' SOME'. ?fn. fn NONE = NONE' /\ (!a. fn (SOME a) = SOME' a)

sum_INDUCT
    |- !P. (!a. P (INL a)) /\ (!a. P (INR a)) ==> (!x. P x)

sum_RECURSION
    |- !INL' INR'. ?fn. (!a. fn (INL a) = INL' a) /\ (!a. fn (INR a) = INR' a)
```

## 2.3 Theorems about pairs

```
COMMA_DEF
    |- !x y. x,y = ABS_prod (mk_pair x y)

CURRY_DEF
    |- !f x y. CURRY f x y = f (x,y)

EXISTS_PAIR_THM
    |- (?p. P p) <=> (?p1 p2. P (p1,p2))

FORALL_PAIR_THM
    |- (!p. P p) <=> (!p1 p2. P (p1,p2))
```

```
FST
    |- !x y. FST (x,y) = x

FST_DEF
    |- !p. FST p = (@x. ?y. p = x,y)

PAIR
    |- !x. FST x,SND x = x

PAIR_EQ
    |- !x y a b. x,y = a,b <=> x = a /\ y = b

PAIR_EXISTS_THM
    |- ?x a b. x = mk_pair a b

PAIR_SURJECTIVE
    |- !p. ?x y. p = x,y

PASSOC_DEF
    |- !f x y z. PASSOC f (x,y,z) = f ((x,y),z)

REP_ABS_PAIR
    |- !x y. REP_prod (ABS_prod (mk_pair x y)) = mk_pair x y

SND
    |- !x y. SND (x,y) = y

SND_DEF
    |- !p. SND p = (@y. ?x. p = x,y)

UNCURRY_DEF
    |- !f x y. UNCURRY f (x,y) = f x y

mk_pair_def
    |- !x y. mk_pair x y = (\a b. a = x /\ b = y)

pair_INDUCT
    |- (!x y. P (x,y)) ==> (!p. P p)

pair_RECURSION
    |- !PAIR'. ?fn. !a0 a1. fn (a0,a1) = PAIR' a0 a1

prod_tybij
    |- (!a. ABS_prod (REP_prod a) = a) /\
       (!r. (?a b. r = mk_pair a b) <=> REP_prod (ABS_prod r) = r)
```

## 2.4 Theorems about wellfoundedness

```
MEASURE_LE
    |- (!y. measure m y a ==> measure m y b) <=> m a <= m b

WF
    |- !(<<). WF (<<) <=>
              (!P. (?x. P x) ==> (?x. P x /\ (!y. y << x ==> ~P y)))

WF_DCHAIN
    |- WF (<<) <=> ~(?s. !n. s (SUC n) << s n)

WF_EQ
    |- WF (<<) <=> (!P. (?x. P x) <=> (?x. P x /\ (!y. y << x ==> ~P y)))

WF_EREC
    |- WF (<<)
       ==> (!H. (!f g x. (!z. z << x ==> f z = g z) ==> H f x = H g x)
                ==> (?!f. !x. f x = H f x))

WF_FALSE
    |- WF (\x y. F)

WF_IND
    |- WF (<<) <=> (!P. (!x. (!y. y << x ==> P y) ==> P x) ==> (!x. P x))

WF_LEX
    |- !R S.
           WF R /\ WF S
           ==> WF (\(r1,s1). \(r2,s2). R r1 r2 \/ r1 = r2 /\ S s1 s2)

WF_LEX_DEPENDENT
    |- !R S.
           WF R /\ (!a. WF (S a))
           ==> WF (\(r1,s1). \(r2,s2). R r1 r2 \/ r1 = r2 /\ S r1 s1 s2)

WF_MEASURE
    |- !m. WF (measure m)

WF_MEASURE_GEN
    |- !m. WF (<<) ==> WF (\x x'. m x << m x')

WF_POINTWISE
    |- WF (<<) /\ WF (<<<) ==> WF (\(x1,y1). \(x2,y2). x1 << x2 /\ y1 <<< y2)
```

```
WF_REC
    |- WF (<<)
       ==> (!H. (!f g x. (!z. z << x ==> f z = g z) ==> H f x = H g x)
                ==> (?f. !x. f x = H f x))

WF_REC_INVARIANT
    |- WF (<<)
       ==> (!H S.
                (!f g x.
                     (!z. z << x ==> f z = g z /\ S z (f z))
                     ==> H f x = H g x /\ S x (H f x))
                ==> (?f. !x. f x = H f x))

WF_REC_TAIL
    |- !P g h. ?f. !x. f x = (if P x then f (g x) else h x)

WF_REC_TAIL_GENERAL
    |- !P G H.
           WF (<<) /\
           (!f g x.
                (!z. z << x ==> f z = g z)
                ==> (P f x <=> P g x) /\ G f x = G g x /\ H f x = H g x) /\
           (!f g x. (!z. z << x ==> f z = g z) ==> H f x = H g x) /\
           (!f x y. P f x /\ y << G f x ==> y << x)
           ==> (?f. !x. f x = (if P f x then f (G f x) else H f x))

WF_REC_WF
    |- (!H. (!f g x. (!z. z << x ==> f z = g z) ==> H f x = H g x)
            ==> (?f. !x. f x = H f x))
       ==> WF (<<)

WF_REC_num
    |- !H. (!f g n. (!m. m < n ==> f m = g m) ==> H f n = H g n)
           ==> (?f. !n. f n = H f n)

WF_REFL
    |- !x. WF (<<) ==> ~(x << x)

WF_SUBSET
    |- (!x y. x << y ==> x <<< y) /\ WF (<<<) ==> WF (<<)

WF_UREC
    |- WF (<<)
       ==> (!H. (!f g x. (!z. z << x ==> f z = g z) ==> H f x = H g x)
                ==> (!f g. (!x. f x = H f x) /\ (!x. g x = H g x) ==> f = g))
```

```
WF_UREC_WF
    |- (!H. (!f g x. (!z. z << x ==> (f z <=> g z)) ==> (H f x <=> H g x))
              ==> (!f g. (!x. f x <=> H f x) /\ (!x. g x <=> H g x) ==> f = g))
       ==> WF (<<)

WF_num
    |- WF (<)

measure
    |- !m. measure m = (\x y. m x < m y)
```

## 2.5 Theorems about natural number arithmetic

```
ADD
    |- (!n. 0 + n = n) /\ (!m n. SUC m + n = SUC (m + n))

ADD1
    |- !m. SUC m = m + 1

ADD_0
    |- !m. m + 0 = m

ADD_AC
    |- m + n = n + m /\ (m + n) + p = m + n + p /\ m + n + p = n + m + p

ADD_ASSOC
    |- !m n p. m + n + p = (m + n) + p

ADD_CLAUSES
    |- (!n. 0 + n = n) /\
       (!m. m + 0 = m) /\
       (!m n. SUC m + n = SUC (m + n)) /\
       (!m n. m + SUC n = SUC (m + n))

ADD_EQ_0
    |- !m n. m + n = 0 <=> m = 0 /\ n = 0

ADD_SUB
    |- !m n. (m + n) - n = m

ADD_SUB2
    |- !m n. (m + n) - m = n

ADD_SUBR
    |- !m n. n - (m + n) = 0

ADD_SUBR2
    |- !m n. m - (m + n) = 0
```

```
ADD_SUC
    |- !m n. m + SUC n = SUC (m + n)

ADD_SYM
    |- !m n. m + n = n + m

BIT0
    |- !n. BIT0 n = n + n

BIT0_THM
    |- !n. NUMERAL (BIT0 n) = NUMERAL n + NUMERAL n

BIT1
    |- !n. BIT1 n = SUC (n + n)

BIT1_THM
    |- !n. NUMERAL (BIT1 n) = SUC (NUMERAL n + NUMERAL n)

DIST_ADD2
    |- !m n p q. dist (m + n,p + q) <= dist (m,p) + dist (n,q)

DIST_ADD2_REV
    |- !m n p q. dist (m,p) <= dist (m + n,p + q) + dist (n,q)

DIST_ADDBOUND
    |- !m n. dist (m,n) <= m + n

DIST_ELIM_THM
    |- P (dist (x,y)) <=> (!d. (x = y + d ==> P d) /\ (y = x + d ==> P d))

DIST_EQ_0
    |- !m n. dist (m,n) = 0 <=> m = n

DIST_LADD
    |- !m p n. dist (m + n,m + p) = dist (n,p)

DIST_LADD_0
    |- !m n. dist (m + n,m) = n

DIST_LE_CASES
    |- !m n p. dist (m,n) <= p <=> m <= n + p /\ n <= m + p

DIST_LMUL
    |- !m n p. m * dist (n,p) = dist (m * n,m * p)

DIST_LZERO
    |- !n. dist (0,n) = n

DIST_RADD
    |- !m p n. dist (m + p,n + p) = dist (m,n)
```

```
DIST_RADD_0
    |- !m n. dist (m,m + n) = n

DIST_REFL
    |- !n. dist (n,n) = 0

DIST_RMUL
    |- !m n p. dist (m,n) * p = dist (m * p,n * p)

DIST_RZERO
    |- !n. dist (n,0) = n

DIST_SYM
    |- !m n. dist (m,n) = dist (n,m)

DIST_TRIANGLE
    |- !m n p. dist (m,p) <= dist (m,n) + dist (n,p)

DIST_TRIANGLES_LE
    |- !m n p q r s.
           dist (m,n) <= r /\ dist (p,q) <= s
           ==> dist (m,p) <= dist (n,q) + r + s

DIST_TRIANGLE_LE
    |- !m n p q. dist (m,n) + dist (n,p) <= q ==> dist (m,p) <= q

DIVISION
    |- !m n. ~(n = 0) ==> m = m DIV n * n + m MOD n /\ m MOD n < n

DIVISION_0
    |- !m n.
           if n = 0
           then m DIV n = 0 /\ m MOD n = 0
           else m = m DIV n * n + m MOD n /\ m MOD n < n

DIVMOD_ELIM_THM
    |- P (m DIV n) (m MOD n) <=>
       (!q r. n = 0 /\ q = 0 /\ r = 0 \/ m = q * n + r /\ r < n ==> P q r)

DIVMOD_ELIM_THM'
    |- P (m DIV n) (m MOD n) <=>
       (?q r. (n = 0 /\ q = 0 /\ r = 0 \/ m = q * n + r /\ r < n) /\ P q r)

DIVMOD_EXIST
    |- !m n. ~(n = 0) ==> (?q r. m = q * n + r /\ r < n)

DIVMOD_EXIST_0
    |- !m n. ?q r. if n = 0 then q = 0 /\ r = 0 else m = q * n + r /\ r < n
```

```
DIVMOD_UNIQ
    |- !m n q r. m = q * n + r /\ r < n ==> m DIV n = q /\ m MOD n = r

DIVMOD_UNIQ_LEMMA
    |- !m n q1 r1 q2 r2.
           (m = q1 * n + r1 /\ r1 < n) /\ m = q2 * n + r2 /\ r2 < n
           ==> q1 = q2 /\ r1 = r2

DIV_0
    |- !n. ~(n = 0) ==> 0 DIV n = 0

DIV_1
    |- !n. n DIV 1 = n

DIV_ADD_MOD
    |- !a b n.
           ~(n = 0)
           ==> ((a + b) MOD n = a MOD n + b MOD n <=>
                (a + b) DIV n = a DIV n + b DIV n)

DIV_DIV
    |- !m n p. ~(n * p = 0) ==> m DIV n DIV p = m DIV (n * p)

DIV_EQ_0
    |- !m n. ~(n = 0) ==> (m DIV n = 0 <=> m < n)

DIV_EQ_EXCLUSION
    |- b * c < (a + 1) * d /\ a * d < (c + 1) * b ==> a DIV b = c DIV d

DIV_LE
    |- !m n. ~(n = 0) ==> m DIV n <= m

DIV_LE_EXCLUSION
    |- !a b c d. ~(b = 0) /\ b * c < (a + 1) * d ==> c DIV d <= a DIV b

DIV_LT
    |- !m n. m < n ==> m DIV n = 0

DIV_MOD
    |- !m n p. ~(n * p = 0) ==> (m DIV n) MOD p = (m MOD (n * p)) DIV n

DIV_MONO
    |- !m n p. ~(p = 0) /\ m <= n ==> m DIV p <= n DIV p

DIV_MONO2
    |- !m n p. ~(p = 0) /\ p <= m ==> n DIV m <= n DIV p

DIV_MONO_LT
    |- !m n p. ~(p = 0) /\ m + p <= n ==> m DIV p < n DIV p
```

```
DIV_MULT
    |- !m n. ~(m = 0) ==> (m * n) DIV m = n

DIV_MULT2
    |- !m n p. ~(m * p = 0) ==> (m * n) DIV (m * p) = n DIV p

DIV_MUL_LE
    |- !m n. n * m DIV n <= m

DIV_REFL
    |- !n. ~(n = 0) ==> n DIV n = 1

DIV_UNIQ
    |- !m n q r. m = q * n + r /\ r < n ==> m DIV n = q

EQ_ADD_LCANCEL
    |- !m n p. m + n = m + p <=> n = p

EQ_ADD_LCANCEL_0
    |- !m n. m + n = m <=> n = 0

EQ_ADD_RCANCEL
    |- !m n p. m + p = n + p <=> m = n

EQ_ADD_RCANCEL_0
    |- !m n. m + n = n <=> m = 0

EQ_IMP_LE
    |- !m n. m = n ==> m <= n

EQ_MULT_LCANCEL
    |- !m n p. m * n = m * p <=> m = 0 \/ n = p

EQ_MULT_RCANCEL
    |- !m n p. m * p = n * p <=> m = n \/ p = 0

EQ_SUC
    |- !m n. SUC m = SUC n <=> m = n

EVEN
    |- (EVEN 0 <=> T) /\ (!n. EVEN (SUC n) <=> ~EVEN n)

EVEN_ADD
    |- !m n. EVEN (m + n) <=> EVEN m <=> EVEN n

EVEN_AND_ODD
    |- !n. ~(EVEN n /\ ODD n)

EVEN_DOUBLE
    |- !n. EVEN (2 * n)
```

```
EVEN_EXISTS
     |- !n. EVEN n <=> (?m. n = 2 * m)

EVEN_EXISTS_LEMMA
     |- !n. (EVEN n ==> (?m. n = 2 * m)) /\ (~EVEN n ==> (?m. n = SUC (2 * m)))

EVEN_EXP
     |- !m n. EVEN (m EXP n) <=> EVEN m /\ ~(n = 0)

EVEN_MOD
     |- !n. EVEN n <=> n MOD 2 = 0

EVEN_MULT
     |- !m n. EVEN (m * n) <=> EVEN m \/ EVEN n

EVEN_ODD_DECOMPOSITION
     |- !n. (?k m. ODD m /\ n = 2 EXP k * m) <=> ~(n = 0)

EVEN_OR_ODD
     |- !n. EVEN n \/ ODD n

EVEN_SUB
     |- !m n. EVEN (m - n) <=> m <= n \/ (EVEN m <=> EVEN n)

EXP
     |- (!m. m EXP 0 = 1) /\ (!m n. m EXP SUC n = m * m EXP n)

EXP_1
     |- !n. n EXP 1 = n

EXP_2
     |- !n. n EXP 2 = n * n

EXP_ADD
     |- !m n p. m EXP (n + p) = m EXP n * m EXP p

EXP_EQ_0
     |- !m n. m EXP n = 0 <=> m = 0 /\ ~(n = 0)

EXP_LT_0
     |- !n x. 0 < x EXP n <=> ~(x = 0) \/ n = 0

EXP_MULT
     |- !m n p. m EXP (n * p) = m EXP n EXP p

EXP_ONE
     |- !n. 1 EXP n = 1

FACT
     |- FACT 0 = 1 /\ (!n. FACT (SUC n) = SUC n * FACT n)
```

```
FACT_LE
    |- !n. 1 <= FACT n

FACT_LT
    |- !n. 0 < FACT n

FACT_MONO
    |- !m n. m <= n ==> FACT m <= FACT n

GE
    |- !n m. m >= n <=> n <= m

GT
    |- !n m. m > n <=> n < m

LE
    |- (!m. m <= 0 <=> m = 0) /\ (!m n. m <= SUC n <=> m = SUC n \/ m <= n)

LEFT_ADD_DISTRIB
    |- !m n p. m * (n + p) = m * n + m * p

LEFT_SUB_DISTRIB
    |- !m n p. m * (n - p) = m * n - m * p

LET_ADD2
    |- !m n p q. m <= p /\ n < q ==> m + n < p + q

LET_ANTISYM
    |- !m n. ~(m <= n /\ n < m)

LET_CASES
    |- !m n. m <= n \/ n < m

LET_TRANS
    |- !m n p. m <= n /\ n < p ==> m < p

LE_0
    |- !n. 0 <= n

LE_ADD
    |- !m n. m <= m + n

LE_ADD2
    |- !m n p q. m <= p /\ n <= q ==> m + n <= p + q

LE_ADDR
    |- !m n. n <= m + n

LE_ADD_LCANCEL
    |- !m n p. m + n <= m + p <=> n <= p
```

```
LE_ADD_RCANCEL
    |- !m n p. m + p <= n + p <=> m <= n

LE_ANTISYM
    |- !m n. m <= n /\ n <= m <=> m = n

LE_CASES
    |- !m n. m <= n \/ n <= m

LE_EXISTS
    |- !m n. m <= n <=> (?d. n = m + d)

LE_EXP
    |- !x m n.
           x EXP m <= x EXP n <=>
           (if x = 0 then m = 0 ==> n = 0 else x = 1 \/ m <= n)

LE_LDIV
    |- !a b n. ~(a = 0) /\ b <= a * n ==> b DIV a <= n

LE_LDIV_EQ
    |- !a b n. ~(a = 0) ==> (b DIV a <= n <=> b < a * (n + 1))

LE_LT
    |- !m n. m <= n <=> m < n \/ m = n

LE_MULT2
    |- !m n p q. m <= n /\ p <= q ==> m * p <= n * q

LE_MULT_LCANCEL
    |- !m n p. m * n <= m * p <=> m = 0 \/ n <= p

LE_MULT_RCANCEL
    |- !m n p. m * p <= n * p <=> m <= n \/ p = 0

LE_RDIV_EQ
    |- !a b n. ~(a = 0) ==> (n <= b DIV a <=> a * n <= b)

LE_REFL
    |- !n. n <= n

LE_SQUARE_REFL
    |- !n. n <= n * n

LE_SUC
    |- !m n. SUC m <= SUC n <=> m <= n

LE_SUC_LT
    |- !m n. SUC m <= n <=> m < n
```

LE_TRANS
```
|- !m n p. m <= n /\ n <= p ==> m <= p
```

LT
```
|- (!m. m < 0 <=> F) /\ (!m n. m < SUC n <=> m = n \/ m < n)
```

LTE_ADD2
```
|- !m n p q. m < p /\ n <= q ==> m + n < p + q
```

LTE_ANTISYM
```
|- !m n. ~(m < n /\ n <= m)
```

LTE_CASES
```
|- !m n. m < n \/ n <= m
```

LTE_TRANS
```
|- !m n p. m < n /\ n <= p ==> m < p
```

LT_0
```
|- !n. 0 < SUC n
```

LT_ADD
```
|- !m n. m < m + n <=> 0 < n
```

LT_ADD2
```
|- !m n p q. m < p /\ n < q ==> m + n < p + q
```

LT_ADDR
```
|- !m n. n < m + n <=> 0 < m
```

LT_ADD_LCANCEL
```
|- !m n p. m + n < m + p <=> n < p
```

LT_ADD_RCANCEL
```
|- !m n p. m + p < n + p <=> m < n
```

LT_ANTISYM
```
|- !m n. ~(m < n /\ n < m)
```

LT_CASES
```
|- !m n. m < n \/ n < m \/ m = n
```

LT_EXISTS
```
|- !m n. m < n <=> (?d. n = m + SUC d)
```

LT_EXP
```
|- !x m n.
       x EXP m < x EXP n <=> 2 <= x /\ m < n \/ x = 0 /\ ~(m = 0) /\ n = 0
```

```
LT_IMP_LE
    |- !m n. m < n ==> m <= n

LT_LE
    |- !m n. m < n <=> m <= n /\ ~(m = n)

LT_LMULT
    |- !m n p. ~(m = 0) /\ n < p ==> m * n < m * p

LT_MULT
    |- !m n. 0 < m * n <=> 0 < m /\ 0 < n

LT_MULT2
    |- !m n p q. m < n /\ p < q ==> m * p < n * q

LT_MULT_LCANCEL
    |- !m n p. m * n < m * p <=> ~(m = 0) /\ n < p

LT_MULT_RCANCEL
    |- !m n p. m * p < n * p <=> m < n /\ ~(p = 0)

LT_NZ
    |- !n. 0 < n <=> ~(n = 0)

LT_REFL
    |- !n. ~(n < n)

LT_SUC
    |- !m n. SUC m < SUC n <=> m < n

LT_SUC_LE
    |- !m n. m < SUC n <=> m <= n

LT_TRANS
    |- !m n p. m < n /\ n < p ==> m < p

MINIMAL
    |- !P. (?n. P n) <=> P ((minimal) P) /\ (!m. m < (minimal) P ==> ~P m)

MOD_0
    |- !n. ~(n = 0) ==> 0 MOD n = 0

MOD_1
    |- !n. n MOD 1 = 0

MOD_ADD_MOD
    |- !a b n. ~(n = 0) ==> (a MOD n + b MOD n) MOD n = (a + b) MOD n

MOD_EQ
    |- !m n p q. m = n + q * p ==> m MOD p = n MOD p
```

```
MOD_EQ_0
    |- !m n. ~(n = 0) ==> (m MOD n = 0 <=> (?q. m = q * n))

MOD_EXISTS
    |- !m n. (?q. m = n * q) <=> (if n = 0 then m = 0 else m MOD n = 0)

MOD_EXP_MOD
    |- !m n p. ~(n = 0) ==> (m MOD n) EXP p MOD n = m EXP p MOD n

MOD_LE
    |- !m n. ~(n = 0) ==> m MOD n <= m

MOD_LT
    |- !m n. m < n ==> m MOD n = m

MOD_MOD
    |- !m n p. ~(n * p = 0) ==> m MOD (n * p) MOD n = m MOD n

MOD_MOD_REFL
    |- !m n. ~(n = 0) ==> m MOD n MOD n = m MOD n

MOD_MULT
    |- !m n. ~(m = 0) ==> (m * n) MOD m = 0

MOD_MULT2
    |- !m n p. ~(m * p = 0) ==> (m * n) MOD (m * p) = m * n MOD p

MOD_MULT_ADD
    |- !m n p. (m * n + p) MOD n = p MOD n

MOD_MULT_LMOD
    |- !m n p. ~(n = 0) ==> (m MOD n * p) MOD n = (m * p) MOD n

MOD_MULT_MOD2
    |- !m n p. ~(n = 0) ==> (m MOD n * p MOD n) MOD n = (m * p) MOD n

MOD_MULT_RMOD
    |- !m n p. ~(n = 0) ==> (m * p MOD n) MOD n = (m * p) MOD n

MOD_UNIQ
    |- !m n q r. m = q * n + r /\ r < n ==> m MOD n = r

MULT
    |- (!n. 0 * n = 0) /\ (!m n. SUC m * n = m * n + n)

MULT_0
    |- !m. m * 0 = 0

MULT_2
    |- !n. 2 * n = n + n
```

```
MULT_AC
    |- m * n = n * m /\ (m * n) * p = m * n * p /\ m * n * p = n * m * p

MULT_ASSOC
    |- !m n p. m * n * p = (m * n) * p

MULT_CLAUSES
    |- (!n. 0 * n = 0) /\
       (!m. m * 0 = 0) /\
       (!n. 1 * n = n) /\
       (!m. m * 1 = m) /\
       (!m n. SUC m * n = m * n + n) /\
       (!m n. m * SUC n = m + m * n)

MULT_EQ_0
    |- !m n. m * n = 0 <=> m = 0 \/ n = 0

MULT_EQ_1
    |- !m n. m * n = 1 <=> m = 1 /\ n = 1

MULT_EXP
    |- !p m n. (m * n) EXP p = m EXP p * n EXP p

MULT_SUC
    |- !m n. m * SUC n = m + m * n

MULT_SYM
    |- !m n. m * n = n * m

NOT_EVEN
    |- !n. ~EVEN n <=> ODD n

NOT_LE
    |- !m n. ~(m <= n) <=> n < m

NOT_LT
    |- !m n. ~(m < n) <=> n <= m

NOT_ODD
    |- !n. ~ODD n <=> EVEN n

NOT_SUC
    |- !n. ~(SUC n = 0)

NUMERAL
    |- !n. NUMERAL n = n

ODD
    |- (ODD 0 <=> F) /\ (!n. ODD (SUC n) <=> ~ODD n)
```

```
ODD_ADD
    |- !m n. ODD (m + n) <=> ~(ODD m <=> ODD n)

ODD_DOUBLE
    |- !n. ODD (SUC (2 * n))

ODD_EXISTS
    |- !n. ODD n <=> (?m. n = SUC (2 * m))

ODD_EXP
    |- !m n. ODD (m EXP n) <=> ODD m \/ n = 0

ODD_MOD
    |- !n. ODD n <=> n MOD 2 = 1

ODD_MULT
    |- !m n. ODD (m * n) <=> ODD m /\ ODD n

ODD_SUB
    |- !m n. ODD (m - n) <=> n < m /\ ~(ODD m <=> ODD n)

ONE
    |- 1 = SUC 0

PRE
    |- PRE 0 = 0 /\ (!n. PRE (SUC n) = n)

PRE_ELIM_THM
    |- P (PRE n) <=> (!m. n = SUC m \/ m = 0 /\ n = 0 ==> P m)

PRE_ELIM_THM'
    |- P (PRE n) <=> (?m. (n = SUC m \/ m = 0 /\ n = 0) /\ P m)

RIGHT_ADD_DISTRIB
    |- !m n p. (m + n) * p = m * p + n * p

RIGHT_SUB_DISTRIB
    |- !m n p. (m - n) * p = m * p - n * p

SUB
    |- (!m. m - 0 = m) /\ (!m n. m - SUC n = PRE (m - n))

SUB_0
    |- !m. 0 - m = 0 /\ m - 0 = m

SUB_ADD
    |- !m n. n <= m ==> m - n + n = m

SUB_ADD_LCANCEL
    |- !m n p. (m + n) - (m + p) = n - p
```

```
SUB_ADD_RCANCEL
    |- !m n p. (m + p) - (n + p) = m - n

SUB_ELIM_THM
    |- P (a - b) <=> (!d. a = b + d \/ a < b /\ d = 0 ==> P d)

SUB_ELIM_THM'
    |- P (a - b) <=> (?d. (a = b + d \/ a < b /\ d = 0) /\ P d)

SUB_EQ_0
    |- !m n. m - n = 0 <=> m <= n

SUB_PRESUC
    |- !m n. PRE (SUC m - n) = m - n

SUB_REFL
    |- !n. n - n = 0

SUB_SUC
    |- !m n. SUC m - SUC n = m - n

SUC_INJ
    |- !m n. SUC m = SUC n <=> m = n

SUC_SUB1
    |- !n. SUC n - 1 = n

TWO
    |- 2 = SUC 1

WLOG_LE
    |- (!m n. P m n <=> P n m) /\ (!m n. m <= n ==> P m n) ==> (!m n. P m n)

WLOG_LT
    |- (!m. P m m) /\ (!m n. P m n <=> P n m) /\ (!m n. m < n ==> P m n)
       ==> (!m y. P m y)

dist
    |- !n m. dist (m,n) = m - n + n - m

minimal
    |- !P. (minimal) P = (@n. P n /\ (!m. m < n ==> ~P m))

num_Axiom
    |- !e f. ?!fn. fn 0 = e /\ (!n. fn (SUC n) = f (fn n) n)

num_Axiom
    |- !e f. ?!fn. fn _0 = e /\ (!n. fn (SUC n) = f (fn n) n)
```

```
num_CASES
    |- !m. m = 0 \/ (?n. m = SUC n)

num_INDUCTION
    |- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)

num_INDUCTION
    |- !P. P _0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)

num_MAX
    |- !P. (?x. P x) /\ (?M. !x. P x ==> x <= M) <=>
           (?m. P m /\ (!x. P x ==> x <= m))

num_RECURSION
    |- !e f. ?fn. fn 0 = e /\ (!n. fn (SUC n) = f (fn n) n)

num_WF
    |- !P. (!n. (!m. m < n ==> P m) ==> P n) ==> (!n. P n)

num_WOP
    |- !P. (?n. P n) <=> (?n. P n /\ (!m. m < n ==> ~P m))
```

## 2.6 Theorems about lists

```
ALL
    |- (ALL P [] <=> T) /\ (ALL P (CONS h t) <=> P h /\ ALL P t)

ALL2
    |- (ALL2 P [] [] <=> T) /\
       (ALL2 P (CONS h1 t1) [] <=> F) /\
       (ALL2 P [] (CONS h2 t2) <=> F) /\
       (ALL2 P (CONS h1 t1) (CONS h2 t2) <=> P h1 h2 /\ ALL2 P t1 t2)

ALL2_ALL
    |- !P l. ALL2 P l l <=> ALL (\x. P x x) l

ALL2_AND_RIGHT
    |- !l m P Q. ALL2 (\x y. P x /\ Q x y) l m <=> ALL P l /\ ALL2 Q l m

ALL2_DEF
    |- (ALL2 P [] l2 <=> l2 = []) /\
       (ALL2 P (CONS h1 t1) l2 <=>
        (if l2 = [] then F else P h1 (HD l2) /\ ALL2 P t1 (TL l2)))

ALL2_MAP
    |- !P f l. ALL2 P (MAP f l) l <=> ALL (\a. P (f a) a) l
```

```
ALL2_MAP2
    |- !l m. ALL2 P (MAP f l) (MAP g m) <=> ALL2 (\x y. P (f x) (g y)) l m

ALL_APPEND
    |- !P l1 l2. ALL P (APPEND l1 l2) <=> ALL P l1 /\ ALL P l2

ALL_IMP
    |- !P Q l. (!x. MEM x l /\ P x ==> Q x) /\ ALL P l ==> ALL Q l

ALL_MAP
    |- !P f l. ALL P (MAP f l) <=> ALL (P o f) l

ALL_MEM
    |- !P l. (!x. MEM x l ==> P x) <=> ALL P l

ALL_MP
    |- !P Q l. ALL (\x. P x ==> Q x) l /\ ALL P l ==> ALL Q l

ALL_T
    |- !l. ALL (\x. T) l

AND_ALL
    |- !l. ALL P l /\ ALL Q l <=> ALL (\x. P x /\ Q x) l

AND_ALL2
    |- !P Q l m. ALL2 P l m /\ ALL2 Q l m <=> ALL2 (\x y. P x y /\ Q x y) l m

APPEND
    |- (!l. APPEND [] l = l) /\
       (!h t l. APPEND (CONS h t) l = CONS h (APPEND t l))

APPEND_ASSOC
    |- !l m n. APPEND l (APPEND m n) = APPEND (APPEND l m) n

APPEND_EQ_NIL
    |- !l m. APPEND l m = [] <=> l = [] /\ m = []

APPEND_NIL
    |- !l. APPEND l [] = l

ASSOC
    |- ASSOC a (CONS h t) = (if FST h = a then SND h else ASSOC a t)

CONS_11
    |- !h1 h2 t1 t2. CONS h1 t1 = CONS h2 t2 <=> h1 = h2 /\ t1 = t2

EL
    |- EL 0 l = HD l /\ EL (SUC n) l = EL n (TL l)
```

```
EX
    |- (EX P [] <=> F) /\ (EX P (CONS h t) <=> P h \/ EX P t)

EXISTS_EX
    |- !P l. (?x. EX (P x) l) <=> EX (\s. ?x. P x s) l

EX_IMP
    |- !P Q l. (!x. MEM x l /\ P x ==> Q x) /\ EX P l ==> EX Q l

EX_MAP
    |- !P f l. EX P (MAP f l) <=> EX (P o f) l

EX_MEM
    |- !P l. (?x. P x /\ MEM x l) <=> EX P l

FILTER
    |- FILTER P [] = [] /\
       FILTER P (CONS h t) = (if P h then CONS h (FILTER P t) else FILTER P t)

FILTER_APPEND
    |- !P l1 l2. FILTER P (APPEND l1 l2) = APPEND (FILTER P l1) (FILTER P l2)

FILTER_MAP
    |- !P f l. FILTER P (MAP f l) = MAP f (FILTER (P o f) l)

FORALL_ALL
    |- !P l. (!x. ALL (P x) l) <=> ALL (\s. !x. P x s) l

HD
    |- HD (CONS h t) = h

ITLIST
    |- ITLIST f [] b = b /\ ITLIST f (CONS h t) b = f h (ITLIST f t b)

ITLIST2
    |- ITLIST2 f [] [] b = b /\
       ITLIST2 f (CONS h1 t1) (CONS h2 t2) b = f h1 h2 (ITLIST2 f t1 t2 b)

ITLIST2_DEF
    |- ITLIST2 f [] l2 b = b /\
       ITLIST2 f (CONS h1 t1) l2 b = f h1 (HD l2) (ITLIST2 f t1 (TL l2) b)

ITLIST_APPEND
    |- !f a l1 l2. ITLIST f (APPEND l1 l2) a = ITLIST f l1 (ITLIST f l2 a)

ITLIST_EXTRA
    |- !l. ITLIST f (APPEND l [a]) b = ITLIST f l (f a b)

LAST
    |- LAST (CONS h t) = (if t = [] then h else LAST t)
```

```
LAST_CLAUSES
    |- LAST [h] = h /\ LAST (CONS h (CONS k t)) = LAST (CONS k t)

LENGTH
    |- LENGTH [] = 0 /\ (!h t. LENGTH (CONS h t) = SUC (LENGTH t))

LENGTH_APPEND
    |- !l m. LENGTH (APPEND l m) = LENGTH l + LENGTH m

LENGTH_EQ_CONS
    |- !l n. LENGTH l = SUC n <=> (?h t. l = CONS h t /\ LENGTH t = n)

LENGTH_EQ_NIL
    |- !l. LENGTH l = 0 <=> l = []

LENGTH_MAP
    |- !l f. LENGTH (MAP f l) = LENGTH l

LENGTH_MAP2
    |- !f l m. LENGTH l = LENGTH m ==> LENGTH (MAP2 f l m) = LENGTH m

LENGTH_REPLICATE
    |- !n x. LENGTH (REPLICATE n x) = n

MAP
    |- (!f. MAP f [] = []) /\ (!f h t. MAP f (CONS h t) = CONS (f h) (MAP f t))

MAP2
    |- MAP2 f [] [] = [] /\
       MAP2 f (CONS h1 t1) (CONS h2 t2) = CONS (f h1 h2) (MAP2 f t1 t2)

MAP2_DEF
    |- MAP2 f [] l = [] /\
       MAP2 f (CONS h1 t1) l = CONS (f h1 (HD l)) (MAP2 f t1 (TL l))

MAP_APPEND
    |- !f l1 l2. MAP f (APPEND l1 l2) = APPEND (MAP f l1) (MAP f l2)

MAP_EQ
    |- !f g l. ALL (\x. f x = g x) l ==> MAP f l = MAP g l

MAP_EQ_ALL2
    |- !l m. ALL2 (\x y. f x = f y) l m ==> MAP f l = MAP f m

MAP_EQ_DEGEN
    |- !l f. ALL (\x. f x = x) l ==> MAP f l = l

MAP_FST_ZIP
    |- !l1 l2. LENGTH l1 = LENGTH l2 ==> MAP FST (ZIP l1 l2) = l1
```

```
MAP_SND_ZIP
    |- !l1 l2. LENGTH l1 = LENGTH l2 ==> MAP SND (ZIP l1 l2) = l2

MAP_o
    |- !f g l. MAP (g o f) l = MAP g (MAP f l)

MEM
    |- (MEM x [] <=> F) /\ (MEM x (CONS h t) <=> x = h \/ MEM x t)

MEM_APPEND
    |- !x l1 l2. MEM x (APPEND l1 l2) <=> MEM x l1 \/ MEM x l2

MEM_ASSOC
    |- !l x. MEM (x,ASSOC x l) l <=> MEM x (MAP FST l)

MEM_EL
    |- !l n. n < LENGTH l ==> MEM (EL n l) l

MEM_FILTER
    |- !P l x. MEM x (FILTER P l) <=> P x /\ MEM x l

MEM_MAP
    |- !f y l. MEM y (MAP f l) <=> (?x. MEM x l /\ y = f x)

MONO_ALL
    |- (!x. P x ==> Q x) ==> ALL P l ==> ALL Q l

MONO_ALL2
    |- (!x y. P x y ==> Q x y) ==> ALL2 P l l' ==> ALL2 Q l l'

NOT_ALL
    |- !P l. ~ALL P l <=> EX (\x. ~P x) l

NOT_CONS_NIL
    |- !h t. ~(CONS h t = [])

NOT_EX
    |- !P l. ~EX P l <=> ALL (\x. ~P x) l

NULL
    |- (NULL [] <=> T) /\ (NULL (CONS h t) <=> F)

REPLICATE
    |- REPLICATE 0 x = [] /\ REPLICATE (SUC n) x = CONS x (REPLICATE n x)

REVERSE
    |- REVERSE [] = [] /\ REVERSE (CONS x l) = APPEND (REVERSE l) [x]

REVERSE_APPEND
    |- !l m. REVERSE (APPEND l m) = APPEND (REVERSE m) (REVERSE l)
```

```
REVERSE_REVERSE
    |- !l. REVERSE (REVERSE l) = l

TL
    |- TL (CONS h t) = t

ZIP
    |- ZIP [] [] = [] /\
       ZIP (CONS h1 t1) (CONS h2 t2) = CONS (h1,h2) (ZIP t1 t2)

ZIP_DEF
    |- ZIP [] l2 = [] /\ ZIP (CONS h1 t1) l2 = CONS (h1,HD l2) (ZIP t1 (TL l2))

list_CASES
    |- !l. l = [] \/ (?h t. l = CONS h t)

list_INDUCT
    |- !P. P [] /\ (!a0 a1. P a1 ==> P (CONS a0 a1)) ==> (!x. P x)

list_RECURSION
    |- !NIL' CONS'.
             ?fn. fn [] = NIL' /\ (!a0 a1. fn (CONS a0 a1) = CONS' a0 a1 (fn a1))
```

## 2.7 Theorems about real numbers

```
REAL_ADD_ASSOC
    |- !x y z. x + y + z = (x + y) + z

REAL_ADD_LDISTRIB
    |- !x y z. x * (y + z) = x * y + x * z

REAL_ADD_LID
    |- !x. &0 + x = x

REAL_ADD_LINV
    |- !x. --x + x = &0

REAL_ADD_SYM
    |- !x y. x + y = y + x

REAL_INV_0
    |- inv (&0) = &0

REAL_LE_ANTISYM
    |- !x y. x <= y /\ y <= x <=> x = y

REAL_LE_LADD_IMP
    |- !x y z. y <= z ==> x + y <= x + z
```

```
REAL_LE_MUL
     |- !x y. &0 <= x /\ &0 <= y ==> &0 <= x * y

REAL_LE_REFL
     |- !x. x <= x

REAL_LE_TOTAL
     |- !x y. x <= y \/ y <= x

REAL_LE_TRANS
     |- !x y z. x <= y /\ y <= z ==> x <= z

REAL_MUL_ASSOC
     |- !x y z. x * y * z = (x * y) * z

REAL_MUL_LID
     |- !x. &1 * x = x

REAL_MUL_LINV
     |- !x. ~(x = &0) ==> inv x * x = &1

REAL_MUL_SYM
     |- !x y. x * y = y * x

REAL_OF_NUM_ADD
     |- !m n. &m + &n = &(m + n)

REAL_OF_NUM_EQ
     |- !m n. &m = &n <=> m = n

REAL_OF_NUM_LE
     |- !m n. &m <= &n <=> m <= n

REAL_OF_NUM_MUL
     |- !m n. &m * &n = &(m * n)

REAL_ABS_0
     |- abs (&0) = &0

REAL_ABS_1
     |- abs (&1) = &1

REAL_ABS_ABS
     |- !x. abs (abs x) = abs x

REAL_ABS_BETWEEN
     |- !x y d. &0 < d /\ x - d < y /\ y < x + d <=> abs (y - x) < d

REAL_ABS_BETWEEN1
     |- !x y z. x < z /\ abs (y - x) < z - x ==> y < z
```

```
REAL_ABS_BETWEEN2
     |- !x0 x y0 y.
            x0 < y0 /\
            &2 * abs (x - x0) < y0 - x0 /\
            &2 * abs (y - y0) < y0 - x0
            ==> x < y

REAL_ABS_BOUND
     |- !x y d. abs (x - y) < d ==> y < x + d

REAL_ABS_BOUNDS
     |- !x k. abs x <= k <=> --k <= x /\ x <= k

REAL_ABS_CASES
     |- !x. x = &0 \/ &0 < abs x

REAL_ABS_CIRCLE
     |- !x y h. abs h < abs y - abs x ==> abs (x + h) < abs y

REAL_ABS_DIV
     |- !x y. abs (x / y) = abs x / abs y

REAL_ABS_INV
     |- !x. abs (inv x) = inv (abs x)

REAL_ABS_LE
     |- !x. x <= abs x

REAL_ABS_MUL
     |- !x y. abs (x * y) = abs x * abs y

REAL_ABS_NEG
     |- !x. abs (--x) = abs x

REAL_ABS_NUM
     |- !n. abs (&n) = &n

REAL_ABS_NZ
     |- !x. ~(x = &0) <=> &0 < abs x

REAL_ABS_POS
     |- !x. &0 <= abs x

REAL_ABS_POW
     |- !x n. abs (x pow n) = abs x pow n

REAL_ABS_REFL
     |- !x. abs x = x <=> &0 <= x
```

```
REAL_ABS_SIGN
     |- !x y. abs (x - y) < y ==> &0 < x

REAL_ABS_SIGN2
     |- !x y. abs (x - y) < --y ==> x < &0

REAL_ABS_STILLNZ
     |- !x y. abs (x - y) < abs y ==> ~(x = &0)

REAL_ABS_SUB
     |- !x y. abs (x - y) = abs (y - x)

REAL_ABS_SUB_ABS
     |- !x y. abs (abs x - abs y) <= abs (x - y)

REAL_ABS_TRIANGLE
     |- !x y. abs (x + y) <= abs x + abs y

REAL_ABS_TRIANGLE_LE
     |- !x y z. abs x + abs (y - x) <= z ==> abs y <= z

REAL_ABS_TRIANGLE_LT
     |- !x y z. abs x + abs (y - x) < z ==> abs y < z

REAL_ABS_ZERO
     |- !x. abs x = &0 <=> x = &0

REAL_ADD2_SUB2
     |- !a b c d. (a + b) - (c + d) = a - c + b - d

REAL_ADD_AC
     |- m + n = n + m /\ (m + n) + p = m + n + p /\ m + n + p = n + m + p

REAL_ADD_RDISTRIB
     |- !x y z. (x + y) * z = x * z + y * z

REAL_ADD_RID
     |- !x. x + &0 = x

REAL_ADD_RINV
     |- !x. x + --x = &0

REAL_ADD_SUB
     |- !x y. (x + y) - x = y

REAL_ADD_SUB2
     |- !x y. x - (x + y) = --y
```

```
REAL_COMPLETE
    |- !P. (?x. P x) /\ (?M. !x. P x ==> x <= M)
           ==> (?M. (!x. P x ==> x <= M) /\
                    (!M'. (!x. P x ==> x <= M') ==> M <= M'))

REAL_DIFFSQ
    |- !x y. (x + y) * (x - y) = x * x - y * y

REAL_DIV_1
    |- !x. x / &1 = x

REAL_DIV_LMUL
    |- !x y. ~(y = &0) ==> y * x / y = x

REAL_DIV_POW2
    |- !x m n.
           ~(x = &0)
           ==> x pow m / x pow n =
               (if n <= m then x pow (m - n) else inv (x pow (n - m)))

REAL_DIV_POW2_ALT
    |- !x m n.
           ~(x = &0)
           ==> x pow m / x pow n =
               (if n < m then x pow (m - n) else inv (x pow (n - m)))

REAL_DIV_REFL
    |- !x. ~(x = &0) ==> x / x = &1

REAL_DIV_RMUL
    |- !x y. ~(y = &0) ==> x / y * y = x

REAL_DOWN
    |- !d. &0 < d ==> (?e. &0 < e /\ e < d)

REAL_DOWN2
    |- !d1 d2. &0 < d1 /\ &0 < d2 ==> (?e. &0 < e /\ e < d1 /\ e < d2)

REAL_ENTIRE
    |- !x y. x * y = &0 <=> x = &0 \/ y = &0

REAL_EQ_ADD_LCANCEL
    |- !x y z. x + y = x + z <=> y = z

REAL_EQ_ADD_LCANCEL_0
    |- !x y. x + y = x <=> y = &0

REAL_EQ_ADD_RCANCEL
    |- !x y z. x + z = y + z <=> x = y
```

```
REAL_EQ_ADD_RCANCEL_0
     |- !x y. x + y = y <=> x = &0

REAL_EQ_IMP_LE
     |- !x y. x = y ==> x <= y

REAL_EQ_LCANCEL_IMP
     |- !x y z. ~(z = &0) /\ z * x = z * y ==> x = y

REAL_EQ_LDIV_EQ
     |- !x y z. &0 < z ==> (x / z = y <=> x = y * z)

REAL_EQ_MUL_LCANCEL
     |- !x y z. x * y = x * z <=> x = &0 \/ y = z

REAL_EQ_MUL_RCANCEL
     |- !x y z. x * z = y * z <=> x = y \/ z = &0

REAL_EQ_NEG2
     |- !x y. --x = --y <=> x = y

REAL_EQ_RCANCEL_IMP
     |- !x y z. ~(z = &0) /\ x * z = y * z ==> x = y

REAL_EQ_RDIV_EQ
     |- !x y z. &0 < z ==> (x = y / z <=> x * z = y)

REAL_EQ_SUB_LADD
     |- !x y z. x = y - z <=> x + z = y

REAL_EQ_SUB_RADD
     |- !x y z. x - y = z <=> x = z + y

REAL_INV_1
     |- inv (&1) = &1

REAL_INV_1_LE
     |- !x. &0 < x /\ x <= &1 ==> &1 <= inv x

REAL_INV_DIV
     |- !x y. inv (x / y) = y / x

REAL_INV_EQ_0
     |- !x. inv x = &0 <=> x = &0

REAL_INV_INV
     |- !x. inv (inv x) = x

REAL_INV_LE_1
     |- !x. &1 <= x ==> inv x <= &1
```

```
REAL_INV_MUL
    |- !x y. inv (x * y) = inv x * inv y

REAL_INV_NEG
    |- !x. inv (--x) = --inv x

REAL_LET_ADD
    |- !x y. &0 <= x /\ &0 < y ==> &0 < x + y

REAL_LET_ADD2
    |- !w x y z. w <= x /\ y < z ==> w + y < x + z

REAL_LET_ANTISYM
    |- !x y. ~(x <= y /\ y < x)

REAL_LET_TOTAL
    |- !x y. x <= y \/ y < x

REAL_LET_TRANS
    |- !x y z. x <= y /\ y < z ==> x < z

REAL_LE_01
    |- &0 <= &1

REAL_LE_ADD
    |- !x y. &0 <= x /\ &0 <= y ==> &0 <= x + y

REAL_LE_ADD2
    |- !w x y z. w <= x /\ y <= z ==> w + y <= x + z

REAL_LE_ADDL
    |- !x y. y <= x + y <=> &0 <= x

REAL_LE_ADDR
    |- !x y. x <= x + y <=> &0 <= y

REAL_LE_DIV
    |- !x y. &0 <= x /\ &0 <= y ==> &0 <= x / y

REAL_LE_DIV2_EQ
    |- !x y z. &0 < z ==> (x / z <= y / z <=> x <= y)

REAL_LE_DOUBLE
    |- !x. &0 <= x + x <=> &0 <= x

REAL_LE_INV
    |- !x. &0 <= x ==> &0 <= inv x

REAL_LE_INV2
    |- !x y. &0 < x /\ x <= y ==> inv y <= inv x
```

```
REAL_LE_INV_EQ
    |- !x. &0 <= inv x <=> &0 <= x

REAL_LE_LADD
    |- !x y z. x + y <= x + z <=> y <= z

REAL_LE_LCANCEL_IMP
    |- !x y z. &0 < x /\ x * y <= x * z ==> y <= z

REAL_LE_LDIV_EQ
    |- !x y z. &0 < z ==> (x / z <= y <=> x <= y * z)

REAL_LE_LMUL
    |- !x y z. &0 <= x /\ y <= z ==> x * y <= x * z

REAL_LE_LMUL_EQ
    |- !x y z. &0 < z ==> (z * x <= z * y <=> x <= y)

REAL_LE_LNEG
    |- !x y. --x <= y <=> &0 <= x + y

REAL_LE_LT
    |- !x y. x <= y <=> x < y \/ x = y

REAL_LE_MAX
    |- !x y z. z <= max x y <=> z <= x \/ z <= y

REAL_LE_MIN
    |- !x y z. z <= min x y <=> z <= x /\ z <= y

REAL_LE_MUL2
    |- !w x y z. &0 <= w /\ w <= x /\ &0 <= y /\ y <= z ==> w * y <= x * z

REAL_LE_NEG
    |- !x y. --x <= --y <=> y <= x

REAL_LE_NEG2
    |- !x y. --x <= --y <=> y <= x

REAL_LE_NEGL
    |- !x. --x <= x <=> &0 <= x

REAL_LE_NEGR
    |- !x. x <= --x <=> x <= &0

REAL_LE_NEGTOTAL
    |- !x. &0 <= x \/ &0 <= --x

REAL_LE_POW2
    |- !n. &1 <= &2 pow n
```

```
REAL_LE_RADD
    |- !x y z. x + z <= y + z <=> x <= y

REAL_LE_RCANCEL_IMP
    |- !x y z. &0 < z /\ x * z <= y * z ==> x <= y

REAL_LE_RDIV_EQ
    |- !x y z. &0 < z ==> (x <= y / z <=> x * z <= y)

REAL_LE_RMUL
    |- !x y z. x <= y /\ &0 <= z ==> x * z <= y * z

REAL_LE_RMUL_EQ
    |- !x y z. &0 < z ==> (x * z <= y * z <=> x <= y)

REAL_LE_RNEG
    |- !x y. x <= --y <=> x + y <= &0

REAL_LE_SQUARE
    |- !x. &0 <= x * x

REAL_LE_SQUARE_ABS
    |- !x y. abs x <= abs y <=> x pow 2 <= y pow 2

REAL_LE_SUB_LADD
    |- !x y z. x <= y - z <=> x + z <= y

REAL_LE_SUB_RADD
    |- !x y z. x - y <= z <=> x <= z + y

REAL_LNEG_UNIQ
    |- !x y. x + y = &0 <=> x = --y

REAL_LTE_ADD
    |- !x y. &0 < x /\ &0 <= y ==> &0 < x + y

REAL_LTE_ADD2
    |- !w x y z. w < x /\ y <= z ==> w + y < x + z

REAL_LTE_ANTISYM
    |- !x y. ~(x < y /\ y <= x)

REAL_LTE_TOTAL
    |- !x y. x < y \/ y <= x

REAL_LTE_TRANS
    |- !x y z. x < y /\ y <= z ==> x < z

REAL_LT_01
    |- &0 < &1
```

```
REAL_LT_ADD
    |- !x y. &0 < x /\ &0 < y ==> &0 < x + y

REAL_LT_ADD1
    |- !x y. x <= y ==> x < y + &1

REAL_LT_ADD2
    |- !w x y z. w < x /\ y < z ==> w + y < x + z

REAL_LT_ADDL
    |- !x y. y < x + y <=> &0 < x

REAL_LT_ADDNEG
    |- !x y z. y < x + --z <=> y + z < x

REAL_LT_ADDNEG2
    |- !x y z. x + --y < z <=> x < z + y

REAL_LT_ADDR
    |- !x y. x < x + y <=> &0 < y

REAL_LT_ADD_SUB
    |- !x y z. x + y < z <=> x < z - y

REAL_LT_ANTISYM
    |- !x y. ~(x < y /\ y < x)

REAL_LT_DIV
    |- !x y. &0 < x /\ &0 < y ==> &0 < x / y

REAL_LT_DIV2_EQ
    |- !x y z. &0 < z ==> (x / z < y / z <=> x < y)

REAL_LT_GT
    |- !x y. x < y ==> ~(y < x)

REAL_LT_IMP_LE
    |- !x y. x < y ==> x <= y

REAL_LT_IMP_NE
    |- !x y. x < y ==> ~(x = y)

REAL_LT_IMP_NZ
    |- !x. &0 < x ==> ~(x = &0)

REAL_LT_INV
    |- !x. &0 < x ==> &0 < inv x

REAL_LT_INV2
    |- !x y. &0 < x /\ x < y ==> inv y < inv x
```

```
REAL_LT_INV_EQ
    |- !x. &0 < inv x <=> &0 < x

REAL_LT_LADD
    |- !x y z. x + y < x + z <=> y < z

REAL_LT_LADD_IMP
    |- !x y z. y < z ==> x + y < x + z

REAL_LT_LCANCEL_IMP
    |- !x y z. &0 < x /\ x * y < x * z ==> y < z

REAL_LT_LDIV_EQ
    |- !x y z. &0 < z ==> (x / z < y <=> x < y * z)

REAL_LT_LE
    |- !x y. x < y <=> x <= y /\ ~(x = y)

REAL_LT_LMUL
    |- !x y z. &0 < x /\ y < z ==> x * y < x * z

REAL_LT_LMUL_EQ
    |- !x y z. &0 < z ==> (z * x < z * y <=> x < y)

REAL_LT_LNEG
    |- !x y. --x < y <=> &0 < x + y

REAL_LT_MAX
    |- !x y z. z < max x y <=> z < x \/ z < y

REAL_LT_MIN
    |- !x y z. z < min x y <=> z < x /\ z < y

REAL_LT_MUL
    |- !x y. &0 < x /\ &0 < y ==> &0 < x * y

REAL_LT_MUL2
    |- !w x y z. &0 <= w /\ w < x /\ &0 <= y /\ y < z ==> w * y < x * z

REAL_LT_NEG
    |- !x y. --x < --y <=> y < x

REAL_LT_NEG2
    |- !x y. --x < --y <=> y < x

REAL_LT_NEGTOTAL
    |- !x. x = &0 \/ &0 < x \/ &0 < --x

REAL_LT_POW2
    |- !n. &0 < &2 pow n
```

```
REAL_LT_RADD
    |- !x y z. x + z < y + z <=> x < y

REAL_LT_RCANCEL_IMP
    |- !x y z. &0 < z /\ x * z < y * z ==> x < y

REAL_LT_RDIV_EQ
    |- !x y z. &0 < z ==> (x < y / z <=> x * z < y)

REAL_LT_REFL
    |- !x. ~(x < x)

REAL_LT_RMUL
    |- !x y z. x < y /\ &0 < z ==> x * z < y * z

REAL_LT_RMUL_EQ
    |- !x y z. &0 < z ==> (x * z < y * z <=> x < y)

REAL_LT_RNEG
    |- !x y. x < --y <=> x + y < &0

REAL_LT_SQUARE
    |- !x. &0 < x * x <=> ~(x = &0)

REAL_LT_SUB_LADD
    |- !x y z. x < y - z <=> x + z < y

REAL_LT_SUB_RADD
    |- !x y z. x - y < z <=> x < z + y

REAL_LT_TOTAL
    |- !x y. x = y \/ x < y \/ y < x

REAL_LT_TRANS
    |- !x y z. x < y /\ y < z ==> x < z

REAL_MAX_ACI
    |- max x y = max y x /\
       max (max x y) z = max x (max y z) /\
       max x (max y z) = max y (max x z) /\
       max x x = x /\
       max x (max x y) = max x y

REAL_MAX_ASSOC
    |- !x y z. max x (max y z) = max (max x y) z

REAL_MAX_LE
    |- !x y z. max x y <= z <=> x <= z /\ y <= z
```

```
REAL_MAX_LT
    |- !x y z. max x y < z <=> x < z /\ y < z

REAL_MAX_MAX
    |- !x y. x <= max x y /\ y <= max x y

REAL_MAX_MIN
    |- !x y. max x y = --min (--x) (--y)

REAL_MAX_SYM
    |- !x y. max x y = max y x

REAL_MIN_ACI
    |- min x y = min y x /\
       min (min x y) z = min x (min y z) /\
       min x (min y z) = min y (min x z) /\
       min x x = x /\
       min x (min x y) = min x y

REAL_MIN_ASSOC
    |- !x y z. min x (min y z) = min (min x y) z

REAL_MIN_LE
    |- !x y z. min x y <= z <=> x <= z \/ y <= z

REAL_MIN_LT
    |- !x y z. min x y < z <=> x < z \/ y < z

REAL_MIN_MAX
    |- !x y. min x y = --max (--x) (--y)

REAL_MIN_MIN
    |- !x y. min x y <= x /\ min x y <= y

REAL_MIN_SYM
    |- !x y. min x y = min y x

REAL_MUL_2
    |- !x. &2 * x = x + x

REAL_MUL_AC
    |- m * n = n * m /\ (m * n) * p = m * n * p /\ m * n * p = n * m * p

REAL_MUL_LINV_UNIQ
    |- !x y. x * y = &1 ==> inv y = x

REAL_MUL_LNEG
    |- !x y. --x * y = --(x * y)
```

```
REAL_MUL_LZERO
     |- !x. &0 * x = &0

REAL_MUL_RID
     |- !x. x * &1 = x

REAL_MUL_RINV
     |- !x. ~(x = &0) ==> x * inv x = &1

REAL_MUL_RINV_UNIQ
     |- !x y. x * y = &1 ==> inv x = y

REAL_MUL_RNEG
     |- !x y. x * --y = --(x * y)

REAL_MUL_RZERO
     |- !x. x * &0 = &0

REAL_NEGNEG
     |- !x. -- --x = x

REAL_NEG_0
     |- -- &0 = &0

REAL_NEG_ADD
     |- !x y. --(x + y) = --x + --y

REAL_NEG_EQ
     |- !x y. --x = y <=> x = --y

REAL_NEG_EQ_0
     |- !x. --x = &0 <=> x = &0

REAL_NEG_GE0
     |- !x. &0 <= --x <=> x <= &0

REAL_NEG_GT0
     |- !x. &0 < --x <=> x < &0

REAL_NEG_LE0
     |- !x. --x <= &0 <=> &0 <= x

REAL_NEG_LMUL
     |- !x y. --(x * y) = --x * y

REAL_NEG_LT0
     |- !x. --x < &0 <=> &0 < x

REAL_NEG_MINUS1
     |- !x. --x = -- &1 * x
```

```
REAL_NEG_MUL2
     |- !x y. --x * --y = x * y

REAL_NEG_NEG
     |- !x. -- --x = x

REAL_NEG_RMUL
     |- !x y. --(x * y) = x * --y

REAL_NEG_SUB
     |- !x y. --(x - y) = y - x

REAL_NOT_EQ
     |- !x y. ~(x = y) <=> x < y \/ y < x

REAL_NOT_LE
     |- !x y. ~(x <= y) <=> y < x

REAL_NOT_LT
     |- !x y. ~(x < y) <=> y <= x

REAL_OF_NUM_GE
     |- !m n. &m >= &n <=> m >= n

REAL_OF_NUM_GT
     |- !m n. &m > &n <=> m > n

REAL_OF_NUM_LT
     |- !m n. &m < &n <=> m < n

REAL_OF_NUM_POW
     |- !x n. &x pow n = &(x EXP n)

REAL_OF_NUM_SUB
     |- !m n. m <= n ==> &n - &m = &(n - m)

REAL_OF_NUM_SUC
     |- !n. &n + &1 = &(SUC n)

REAL_POS
     |- !n. &0 <= &n

REAL_POS_NZ
     |- !x. &0 < x ==> ~(x = &0)

REAL_POW2_ABS
     |- !x. abs x pow 2 = x pow 2

REAL_POW_1
     |- !x. x pow 1 = x
```

```
REAL_POW_1_LE
    |- !n x. &0 <= x /\ x <= &1 ==> x pow n <= &1

REAL_POW_2
    |- !x. x pow 2 = x * x

REAL_POW_ADD
    |- !x m n. x pow (m + n) = x pow m * x pow n

REAL_POW_DIV
    |- !x y n. (x / y) pow n = x pow n / y pow n

REAL_POW_EQ_0
    |- !x n. x pow n = &0 <=> x = &0 /\ ~(n = 0)

REAL_POW_INV
    |- !x n. inv x pow n = inv (x pow n)

REAL_POW_LE
    |- !x n. &0 <= x ==> &0 <= x pow n

REAL_POW_LE2
    |- !n x y. &0 <= x /\ x <= y ==> x pow n <= y pow n

REAL_POW_LE_1
    |- !n x. &1 <= x ==> &1 <= x pow n

REAL_POW_LT
    |- !x n. &0 < x ==> &0 < x pow n

REAL_POW_LT2
    |- !n x y. ~(n = 0) /\ &0 <= x /\ x < y ==> x pow n < y pow n

REAL_POW_MONO
    |- !m n x. &1 <= x /\ m <= n ==> x pow m <= x pow n

REAL_POW_MONO_LT
    |- !m n x. &1 < x /\ m < n ==> x pow m < x pow n

REAL_POW_MUL
    |- !x y n. (x * y) pow n = x pow n * y pow n

REAL_POW_NEG
    |- !x n. --x pow n = (if EVEN n then x pow n else --(x pow n))

REAL_POW_NZ
    |- !x n. ~(x = &0) ==> ~(x pow n = &0)

REAL_POW_ONE
    |- !n. &1 pow n = &1
```

```
REAL_POW_POW
     |- !x m n. x pow m pow n = x pow (m * n)

REAL_POW_SUB
     |- !x m n. ~(x = &0) /\ m <= n ==> x pow (n - m) = x pow n / x pow m

REAL_RNEG_UNIQ
     |- !x y. x + y = &0 <=> y = --x

REAL_SOS_EQ_0
     |- !x y. x pow 2 + y pow 2 = &0 <=> x = &0 /\ y = &0

REAL_SUB_0
     |- !x y. x - y = &0 <=> x = y

REAL_SUB_ABS
     |- !x y. abs x - abs y <= abs (x - y)

REAL_SUB_ADD
     |- !x y. x - y + y = x

REAL_SUB_ADD2
     |- !x y. y + x - y = x

REAL_SUB_INV
     |- !x y. ~(x = &0) /\ ~(y = &0) ==> inv x - inv y = (y - x) / (x * y)

REAL_SUB_LDISTRIB
     |- !x y z. x * (y - z) = x * y - x * z

REAL_SUB_LE
     |- !x y. &0 <= x - y <=> y <= x

REAL_SUB_LNEG
     |- !x y. --x - y = --(x + y)

REAL_SUB_LT
     |- !x y. &0 < x - y <=> y < x

REAL_SUB_LZERO
     |- !x. &0 - x = --x

REAL_SUB_NEG2
     |- !x y. --x - --y = y - x

REAL_SUB_RDISTRIB
     |- !x y z. (x - y) * z = x * z - y * z

REAL_SUB_REFL
     |- !x. x - x = &0
```

```
REAL_SUB_RNEG
     |- !x y. x - --y = x + y

REAL_SUB_RZERO
     |- !x. x - &0 = x

REAL_SUB_SUB
     |- !x y. x - y - x = --y

REAL_SUB_SUB2
     |- !x y. x - (x - y) = y

REAL_SUB_TRIANGLE
     |- !a b c. a - b + b - c = a - c

REAL_WLOG_LE
     |- (!x y. P x y <=> P y x) /\ (!x y. x <= y ==> P x y) ==> (!x y. P x y)

REAL_WLOG_LT
     |- (!x. P x x) /\ (!x y. P x y <=> P y x) /\ (!x y. x < y ==> P x y)
        ==> (!x y. P x y)

real_abs
     |- !x. abs x = (if &0 <= x then x else --x)

real_div
     |- !x y. x / y = x * inv y

real_ge
     |- !y x. x >= y <=> y <= x

real_gt
     |- !y x. x > y <=> y < x

real_lt
     |- !y x. x < y <=> ~(y <= x)

real_max
     |- !n m. max m n = (if m <= n then n else m)

real_min
     |- !m n. min m n = (if m <= n then m else n)

real_pow
     |- x pow 0 = &1 /\ (!n. x pow SUC n = x * x pow n)

real_sub
     |- !x y. x - y = x + --y
```

## 2.8 Theorems about integers

```
INT_ABS
    |- !x. abs x = (if &0 <= x then x else --x)

INT_ABS_0
    |- abs (&0) = &0

INT_ABS_1
    |- abs (&1) = &1

INT_ABS_ABS
    |- !x. abs (abs x) = abs x

INT_ABS_BETWEEN
    |- !x y d. &0 < d /\ x - d < y /\ y < x + d <=> abs (y - x) < d

INT_ABS_BETWEEN1
    |- !x y z. x < z /\ abs (y - x) < z - x ==> y < z

INT_ABS_BETWEEN2
    |- !x0 x y0 y.
           x0 < y0 /\
           &2 * abs (x - x0) < y0 - x0 /\
           &2 * abs (y - y0) < y0 - x0
           ==> x < y

INT_ABS_BOUND
    |- !x y d. abs (x - y) < d ==> y < x + d

INT_ABS_CASES
    |- !x. x = &0 \/ &0 < abs x

INT_ABS_CIRCLE
    |- !x y h. abs h < abs y - abs x ==> abs (x + h) < abs y

INT_ABS_LE
    |- !x. x <= abs x

INT_ABS_MUL
    |- !x y. abs (x * y) = abs x * abs y

INT_ABS_MUL_1
    |- !x y. abs (x * y) = &1 <=> abs x = &1 /\ abs y = &1

INT_ABS_NEG
    |- !x. abs (--x) = abs x
```

```
INT_ABS_NUM
    |- !n. abs (&n) = &n

INT_ABS_NZ
    |- !x. ~(x = &0) <=> &0 < abs x

INT_ABS_POS
    |- !x. &0 <= abs x

INT_ABS_POW
    |- !x n. abs (x pow n) = abs x pow n

INT_ABS_REFL
    |- !x. abs x = x <=> &0 <= x

INT_ABS_SIGN
    |- !x y. abs (x - y) < y ==> &0 < x

INT_ABS_SIGN2
    |- !x y. abs (x - y) < --y ==> x < &0

INT_ABS_STILLNZ
    |- !x y. abs (x - y) < abs y ==> ~(x = &0)

INT_ABS_SUB
    |- !x y. abs (x - y) = abs (y - x)

INT_ABS_SUB_ABS
    |- !x y. abs (abs x - abs y) <= abs (x - y)

INT_ABS_TRIANGLE
    |- !x y. abs (x + y) <= abs x + abs y

INT_ABS_ZERO
    |- !x. abs x = &0 <=> x = &0

INT_ADD2_SUB2
    |- !a b c d. (a + b) - (c + d) = a - c + b - d

INT_ADD_AC
    |- m + n = n + m /\ (m + n) + p = m + n + p /\ m + n + p = n + m + p

INT_ADD_ASSOC
    |- !x y z. x + y + z = (x + y) + z

INT_ADD_LDISTRIB
    |- !x y z. x * (y + z) = x * y + x * z

INT_ADD_LID
    |- !x. &0 + x = x
```

```
INT_ADD_LINV
    |- !x. --x + x = &0

INT_ADD_RDISTRIB
    |- !x y z. (x + y) * z = x * z + y * z

INT_ADD_RID
    |- !x. x + &0 = x

INT_ADD_RINV
    |- !x. x + --x = &0

INT_ADD_SUB
    |- !x y. (x + y) - x = y

INT_ADD_SUB2
    |- !x y. x - (x + y) = --y

INT_ADD_SYM
    |- !x y. x + y = y + x

INT_ARCH
    |- !x d. ~(d = &0) ==> (?c. x < c * d)

INT_DIFFSQ
    |- !x y. (x + y) * (x - y) = x * x - y * y

INT_ENTIRE
    |- !x y. x * y = &0 <=> x = &0 \/ y = &0

INT_EQ_ADD_LCANCEL
    |- !x y z. x + y = x + z <=> y = z

INT_EQ_ADD_LCANCEL_0
    |- !x y. x + y = x <=> y = &0

INT_EQ_ADD_RCANCEL
    |- !x y z. x + z = y + z <=> x = y

INT_EQ_ADD_RCANCEL_0
    |- !x y. x + y = y <=> x = &0

INT_EQ_IMP_LE
    |- !x y. x = y ==> x <= y

INT_EQ_MUL_LCANCEL
    |- !x y z. x * y = x * z <=> x = &0 \/ y = z

INT_EQ_MUL_RCANCEL
    |- !x y z. x * z = y * z <=> x = y \/ z = &0
```

```
INT_EQ_NEG2
    |- !x y. --x = --y <=> x = y

INT_EQ_SUB_LADD
    |- !x y z. x = y - z <=> x + z = y

INT_EQ_SUB_RADD
    |- !x y z. x - y = z <=> x = z + y

INT_FORALL_POS
    |- (!n. P (&n)) <=> (!i. &0 <= i ==> P i)

INT_GE
    |- !x y. x >= y <=> y <= x

INT_GT
    |- !x y. x > y <=> y < x

INT_GT_DISCRETE
    |- !x y. x > y <=> x >= y + &1

INT_IMAGE
    |- !x. (?n. x = &n) \/ (?n. x = -- &n)

INT_LET_ADD
    |- !x y. &0 <= x /\ &0 < y ==> &0 < x + y

INT_LET_ADD2
    |- !w x y z. w <= x /\ y < z ==> w + y < x + z

INT_LET_ANTISYM
    |- !x y. ~(x <= y /\ y < x)

INT_LET_TOTAL
    |- !x y. x <= y \/ y < x

INT_LET_TRANS
    |- !x y z. x <= y /\ y < z ==> x < z

INT_LE_01
    |- &0 <= &1

INT_LE_ADD
    |- !x y. &0 <= x /\ &0 <= y ==> &0 <= x + y

INT_LE_ADD2
    |- !w x y z. w <= x /\ y <= z ==> w + y <= x + z

INT_LE_ADDL
    |- !x y. y <= x + y <=> &0 <= x
```

```
INT_LE_ADDR
    |- !x y. x <= x + y <=> &0 <= y

INT_LE_ANTISYM
    |- !x y. x <= y /\ y <= x <=> x = y

INT_LE_DOUBLE
    |- !x. &0 <= x + x <=> &0 <= x

INT_LE_LADD
    |- !x y z. x + y <= x + z <=> y <= z

INT_LE_LADD_IMP
    |- !x y z. y <= z ==> x + y <= x + z

INT_LE_LMUL
    |- !x y z. &0 <= x /\ y <= z ==> x * y <= x * z

INT_LE_LNEG
    |- !x y. --x <= y <=> &0 <= x + y

INT_LE_LT
    |- !x y. x <= y <=> x < y \/ x = y

INT_LE_MAX
    |- !x y z. z <= max x y <=> z <= x \/ z <= y

INT_LE_MIN
    |- !x y z. z <= min x y <=> z <= x /\ z <= y

INT_LE_MUL
    |- !x y. &0 <= x /\ &0 <= y ==> &0 <= x * y

INT_LE_NEG
    |- !x y. --x <= --y <=> y <= x

INT_LE_NEG2
    |- !x y. --x <= --y <=> y <= x

INT_LE_NEGL
    |- !x. --x <= x <=> &0 <= x

INT_LE_NEGR
    |- !x. x <= --x <=> x <= &0

INT_LE_NEGTOTAL
    |- !x. &0 <= x \/ &0 <= --x

INT_LE_POW2
    |- !n. &1 <= &2 pow n
```

```
INT_LE_RADD
    |- !x y z. x + z <= y + z <=> x <= y

INT_LE_REFL
    |- !x. x <= x

INT_LE_RNEG
    |- !x y. x <= --y <=> x + y <= &0

INT_LE_SQUARE
    |- !x. &0 <= x * x

INT_LE_SUB_LADD
    |- !x y z. x <= y - z <=> x + z <= y

INT_LE_SUB_RADD
    |- !x y z. x - y <= z <=> x <= z + y

INT_LE_TOTAL
    |- !x y. x <= y \/ y <= x

INT_LE_TRANS
    |- !x y z. x <= y /\ y <= z ==> x <= z

INT_LNEG_UNIQ
    |- !x y. x + y = &0 <=> x = --y

INT_LT
    |- !x y. x < y <=> ~(y <= x)

INT_LTE_ADD
    |- !x y. &0 < x /\ &0 <= y ==> &0 < x + y

INT_LTE_ADD2
    |- !w x y z. w < x /\ y <= z ==> w + y < x + z

INT_LTE_ANTISYM
    |- !x y. ~(x < y /\ y <= x)

INT_LTE_TOTAL
    |- !x y. x < y \/ y <= x

INT_LTE_TRANS
    |- !x y z. x < y /\ y <= z ==> x < z

INT_LT_01
    |- &0 < &1

INT_LT_ADD
    |- !x y. &0 < x /\ &0 < y ==> &0 < x + y
```

```
INT_LT_ADD1
    |- !x y. x <= y ==> x < y + &1

INT_LT_ADD2
    |- !w x y z. w < x /\ y < z ==> w + y < x + z

INT_LT_ADDL
    |- !x y. y < x + y <=> &0 < x

INT_LT_ADDNEG
    |- !x y z. y < x + --z <=> y + z < x

INT_LT_ADDNEG2
    |- !x y z. x + --y < z <=> x < z + y

INT_LT_ADDR
    |- !x y. x < x + y <=> &0 < y

INT_LT_ADD_SUB
    |- !x y z. x + y < z <=> x < z - y

INT_LT_ANTISYM
    |- !x y. ~(x < y /\ y < x)

INT_LT_DISCRETE
    |- !x y. x < y <=> x + &1 <= y

INT_LT_GT
    |- !x y. x < y ==> ~(y < x)

INT_LT_IMP_LE
    |- !x y. x < y ==> x <= y

INT_LT_IMP_NE
    |- !x y. x < y ==> ~(x = y)

INT_LT_LADD
    |- !x y z. x + y < x + z <=> y < z

INT_LT_LE
    |- !x y. x < y <=> x <= y /\ ~(x = y)

INT_LT_LMUL_EQ
    |- !x y z. &0 < z ==> (z * x < z * y <=> x < y)

INT_LT_MAX
    |- !x y z. z < max x y <=> z < x \/ z < y

INT_LT_MIN
    |- !x y z. z < min x y <=> z < x /\ z < y
```

```
INT_LT_MUL
    |- !x y. &0 < x /\ &0 < y ==> &0 < x * y

INT_LT_NEG
    |- !x y. --x < --y <=> y < x

INT_LT_NEG2
    |- !x y. --x < --y <=> y < x

INT_LT_NEGTOTAL
    |- !x. x = &0 \/ &0 < x \/ &0 < --x

INT_LT_POW2
    |- !n. &0 < &2 pow n

INT_LT_RADD
    |- !x y z. x + z < y + z <=> x < y

INT_LT_REFL
    |- !x. ~(x < x)

INT_LT_RMUL_EQ
    |- !x y z. &0 < z ==> (x * z < y * z <=> x < y)

INT_LT_SUB_LADD
    |- !x y z. x < y - z <=> x + z < y

INT_LT_SUB_RADD
    |- !x y z. x - y < z <=> x < z + y

INT_LT_TOTAL
    |- !x y. x = y \/ x < y \/ y < x

INT_LT_TRANS
    |- !x y z. x < y /\ y < z ==> x < z

INT_MAX_ACI
    |- max x y = max y x /\
       max (max x y) z = max x (max y z) /\
       max x (max y z) = max y (max x z) /\
       max x x = x /\
       max x (max x y) = max x y

INT_MAX_ASSOC
    |- !x y z. max x (max y z) = max (max x y) z

INT_MAX_LE
    |- !x y z. max x y <= z <=> x <= z /\ y <= z
```

```
INT_MAX_LT
    |- !x y z. max x y < z <=> x < z /\ y < z

INT_MAX_MAX
    |- !x y. x <= max x y /\ y <= max x y

INT_MAX_MIN
    |- !x y. max x y = --min (--x) (--y)

INT_MAX_SYM
    |- !x y. max x y = max y x

INT_MIN_ACI
    |- min x y = min y x /\
       min (min x y) z = min x (min y z) /\
       min x (min y z) = min y (min x z) /\
       min x x = x /\
       min x (min x y) = min x y

INT_MIN_ASSOC
    |- !x y z. min x (min y z) = min (min x y) z

INT_MIN_LE
    |- !x y z. min x y <= z <=> x <= z \/ y <= z

INT_MIN_LT
    |- !x y z. min x y < z <=> x < z \/ y < z

INT_MIN_MAX
    |- !x y. min x y = --max (--x) (--y)

INT_MIN_MIN
    |- !x y. min x y <= x /\ min x y <= y

INT_MIN_SYM
    |- !x y. min x y = min y x

INT_MUL_AC
    |- m * n = n * m /\ (m * n) * p = m * n * p /\ m * n * p = n * m * p

INT_MUL_ASSOC
    |- !x y z. x * y * z = (x * y) * z

INT_MUL_LID
    |- !x. &1 * x = x

INT_MUL_LNEG
    |- !x y. --x * y = --(x * y)
```

```
INT_MUL_LZERO
     |- !x. &0 * x = &0

INT_MUL_RID
     |- !x. x * &1 = x

INT_MUL_RNEG
     |- !x y. x * --y = --(x * y)

INT_MUL_RZERO
     |- !x. x * &0 = &0

INT_MUL_SYM
     |- !x y. x * y = y * x

INT_NEGNEG
     |- !x. -- --x = x

INT_NEG_0
     |- -- &0 = &0

INT_NEG_ADD
     |- !x y. --(x + y) = --x + --y

INT_NEG_EQ
     |- !x y. --x = y <=> x = --y

INT_NEG_EQ_0
     |- !x. --x = &0 <=> x = &0

INT_NEG_GE0
     |- !x. &0 <= --x <=> x <= &0

INT_NEG_GT0
     |- !x. &0 < --x <=> x < &0

INT_NEG_LE0
     |- !x. --x <= &0 <=> &0 <= x

INT_NEG_LMUL
     |- !x y. --(x * y) = --x * y

INT_NEG_LT0
     |- !x. --x < &0 <=> &0 < x

INT_NEG_MINUS1
     |- !x. --x = -- &1 * x

INT_NEG_MUL2
     |- !x y. --x * --y = x * y
```

```
INT_NEG_NEG
    |- !x. -- --x = x

INT_NEG_RMUL
    |- !x y. --(x * y) = x * --y

INT_NEG_SUB
    |- !x y. --(x - y) = y - x

INT_NOT_EQ
    |- !x y. ~(x = y) <=> x < y \/ y < x

INT_NOT_LE
    |- !x y. ~(x <= y) <=> y < x

INT_NOT_LT
    |- !x y. ~(x < y) <=> y <= x

INT_OF_NUM_ADD
    |- !m n. &m + &n = &(m + n)

INT_OF_NUM_EQ
    |- !m n. &m = &n <=> m = n

INT_OF_NUM_GE
    |- !m n. &m >= &n <=> m >= n

INT_OF_NUM_GT
    |- !m n. &m > &n <=> m > n

INT_OF_NUM_LE
    |- !m n. &m <= &n <=> m <= n

INT_OF_NUM_LT
    |- !m n. &m < &n <=> m < n

INT_OF_NUM_MUL
    |- !m n. &m * &n = &(m * n)

INT_OF_NUM_OF_INT
    |- !x. &0 <= x ==> &(num_of_int x) = x

INT_OF_NUM_POW
    |- !x n. &x pow n = &(x EXP n)

INT_OF_NUM_SUB
    |- !m n. m <= n ==> &n - &m = &(n - m)

INT_OF_NUM_SUC
    |- !n. &n + &1 = &(SUC n)
```

```
INT_POS
    |- !n. &0 <= &n

INT_POS_NZ
    |- !x. &0 < x ==> ~(x = &0)

INT_POW
    |- x pow 0 = &1 /\ (!n. x pow SUC n = x * x pow n)

INT_POW2_ABS
    |- !x. abs x pow 2 = x pow 2

INT_POW_1
    |- !x. x pow 1 = x

INT_POW_1_LE
    |- !n x. &0 <= x /\ x <= &1 ==> x pow n <= &1

INT_POW_2
    |- !x. x pow 2 = x * x

INT_POW_ADD
    |- !x m n. x pow (m + n) = x pow m * x pow n

INT_POW_EQ_0
    |- !x n. x pow n = &0 <=> x = &0 /\ ~(n = 0)

INT_POW_LE
    |- !x n. &0 <= x ==> &0 <= x pow n

INT_POW_LE2
    |- !n x y. &0 <= x /\ x <= y ==> x pow n <= y pow n

INT_POW_LE_1
    |- !n x. &1 <= x ==> &1 <= x pow n

INT_POW_LT
    |- !x n. &0 < x ==> &0 < x pow n

INT_POW_LT2
    |- !n x y. ~(n = 0) /\ &0 <= x /\ x < y ==> x pow n < y pow n

INT_POW_MONO
    |- !m n x. &1 <= x /\ m <= n ==> x pow m <= x pow n

INT_POW_MONO_LT
    |- !m n x. &1 < x /\ m < n ==> x pow m < x pow n

INT_POW_MUL
    |- !x y n. (x * y) pow n = x pow n * y pow n
```

```
INT_POW_NEG
    |- !x n. --x pow n = (if EVEN n then x pow n else --(x pow n))

INT_POW_NZ
    |- !x n. ~(x = &0) ==> ~(x pow n = &0)

INT_POW_ONE
    |- !n. &1 pow n = &1

INT_POW_POW
    |- !x m n. x pow m pow n = x pow (m * n)

INT_RNEG_UNIQ
    |- !x y. x + y = &0 <=> y = --x

INT_SUB
    |- !x y. x - y = x + --y

INT_SUB_0
    |- !x y. x - y = &0 <=> x = y

INT_SUB_ABS
    |- !x y. abs x - abs y <= abs (x - y)

INT_SUB_ADD
    |- !x y. x - y + y = x

INT_SUB_ADD2
    |- !x y. y + x - y = x

INT_SUB_LDISTRIB
    |- !x y z. x * (y - z) = x * y - x * z

INT_SUB_LE
    |- !x y. &0 <= x - y <=> y <= x

INT_SUB_LNEG
    |- !x y. --x - y = --(x + y)

INT_SUB_LT
    |- !x y. &0 < x - y <=> y < x

INT_SUB_LZERO
    |- !x. &0 - x = --x

INT_SUB_NEG2
    |- !x y. --x - --y = y - x

INT_SUB_REFL
    |- !x. x - x = &0
```

```
INT_SUB_RNEG
    |- !x y. x - --y = x + y

INT_SUB_RZERO
    |- !x. x - &0 = x

INT_SUB_SUB
    |- !x y. x - y - x = --y

INT_SUB_SUB2
    |- !x y. x - (x - y) = y

INT_SUB_TRIANGLE
    |- !a b c. a - b + b - c = a - c

NUM_GCD
    |- !a b. &(gcd (a,b)) = gcd (&a,&b)

NUM_OF_INT
    |- !x. &0 <= x <=> &(num_of_int x) = x

NUM_OF_INT_OF_NUM
    |- !n. num_of_int (&n) = n

int_congruent
    |- !x y n. (x == y) (mod n) <=> (?d. x - y = n * d)

int_coprime
    |- !a b. coprime (a,b) <=> (?x y. a * x + b * y = &1)

int_divides
    |- !b a. a divides b <=> (?x. b = a * x)

int_gcd
    |- !a b.
           &0 <= gcd (a,b) /\
           gcd (a,b) divides a /\
           gcd (a,b) divides b /\
           (?x y. gcd (a,b) = a * x + b * y)

int_mod
    |- !n x y. mod n x y <=> n divides x - y

num_congruent
    |- !x y n. (x == y) (mod n) <=> (&x == &y) (mod &n)

num_coprime
    |- !a b. coprime (a,b) <=> coprime (&a,&b)
```

```
num_divides
    |- !a b. a divides b <=> &a divides &b

num_gcd
    |- !a b. gcd (a,b) = num_of_int (gcd (&a,&b))

num_mod
    |- !n x y. mod n x y <=> mod &n (&x) (&y)
```

## 2.9 Theorems about sets and functions

```
ABSORPTION
    |- !x s. x IN s <=> x INSERT s = s

BIJ
    |- !f s t. BIJ f s t <=> INJ f s t /\ SURJ f s t

BIJECTIONS_CARD_EQ
    |- !s t f g.
           (FINITE s \/ FINITE t) /\
           (!x. x IN s ==> f x IN t /\ g (f x) = x) /\
           (!y. y IN t ==> g y IN s /\ f (g y) = y)
           ==> CARD s = CARD t

BIJECTIONS_HAS_SIZE
    |- !s t f g.
           (!x. x IN s ==> f x IN t /\ g (f x) = x) /\
           (!y. y IN t ==> g y IN s /\ f (g y) = y) /\
           s HAS_SIZE n
           ==> t HAS_SIZE n

BIJECTIONS_HAS_SIZE_EQ
    |- !s t f g.
           (!x. x IN s ==> f x IN t /\ g (f x) = x) /\
           (!y. y IN t ==> g y IN s /\ f (g y) = y)
           ==> (!n. s HAS_SIZE n <=> t HAS_SIZE n)

CARD
    |- !s. CARD s = ITSET (\x n. SUC n) s 0

CARD_CLAUSES
    |- CARD {} = 0 /\
       (!x s.
            FINITE s
            ==> CARD (x INSERT s) = (if x IN s then CARD s else SUC (CARD s)))
```

```
CARD_CROSS
    |- !s t. FINITE s /\ FINITE t ==> CARD (s CROSS t) = CARD s * CARD t

CARD_DELETE
    |- !x s.
           FINITE s
           ==> CARD (s DELETE x) = (if x IN s then CARD s - 1 else CARD s)

CARD_EQ
    |- !t s. s CARD_EQ t <=> s CARD_LE t /\ t CARD_LE s

CARD_EQ_0
    |- !s. FINITE s ==> (CARD s = 0 <=> s = {})

CARD_EQ_BIJECTION
    |- !s t.
           FINITE s /\ FINITE t /\ CARD s = CARD t
           ==> (?f. (!x. x IN s ==> f x IN t) /\
                    (!y. y IN t ==> (?x. x IN s /\ f x = y)) /\
                    (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y))

CARD_EQ_BIJECTIONS
    |- !s t.
           FINITE s /\ FINITE t /\ CARD s = CARD t
           ==> (?f g.
                    (!x. x IN s ==> f x IN t /\ g (f x) = x) /\
                    (!y. y IN t ==> g y IN s /\ f (g y) = y))

CARD_FUNSPACE
    |- !s t.
           FINITE s /\ FINITE t
           ==> CARD
               {f | (!x. x IN s ==> f x IN t) /\ (!x. ~(x IN s) ==> f x = d)} =
               CARD t EXP CARD s

CARD_GE
    |- !t s. s CARD_GE t <=> (?f. !y. y IN t ==> (?x. x IN s /\ y = f x))

CARD_GE_REFL
    |- !s. s CARD_GE s

CARD_GE_TRANS
    |- !s t u. s CARD_GE t /\ t CARD_GE u ==> s CARD_GE u

CARD_GT
    |- !t s. s CARD_GT t <=> s CARD_GE t /\ ~(t CARD_GE s)
```

```
CARD_IMAGE_INJ
    |- !f s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y) /\ FINITE s
           ==> CARD (IMAGE f s) = CARD s

CARD_IMAGE_INJ_EQ
    |- !f s t.
           FINITE s /\
           (!x. x IN s ==> f x IN t) /\
           (!y. y IN t ==> (?!x. x IN s /\ f x = y))
           ==> CARD t = CARD s

CARD_IMAGE_LE
    |- !f s. FINITE s ==> CARD (IMAGE f s) <= CARD s

CARD_LE
    |- !t s. s CARD_LE t <=> t CARD_GE s

CARD_LE_INJ
    |- !s t.
           FINITE s /\ FINITE t /\ CARD s <= CARD t
           ==> (?f. IMAGE f s SUBSET t /\
                    (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y))

CARD_LT
    |- !t s. s CARD_LT t <=> s CARD_LE t /\ ~(t CARD_LE s)

CARD_NUMSEG
    |- !m n. CARD (m..n) = (n + 1) - m

CARD_NUMSEG_1
    |- !n. CARD (1..n) = n

CARD_NUMSEG_LE
    |- !n. CARD {m | m <= n} = n + 1

CARD_NUMSEG_LEMMA
    |- !m d. CARD (m..m + d) = d + 1

CARD_NUMSEG_LT
    |- !n. CARD {m | m < n} = n

CARD_POWERSET
    |- !s. FINITE s ==> CARD {t | t SUBSET s} = 2 EXP CARD s

CARD_PRODUCT
    |- !s t.
           FINITE s /\ FINITE t
           ==> CARD {x,y | x IN s /\ y IN t} = CARD s * CARD t
```

```
CARD_PSUBSET
    |- !a b. a PSUBSET b /\ FINITE b ==> CARD a < CARD b

CARD_SUBSET
    |- !a b. a SUBSET b /\ FINITE b ==> CARD a <= CARD b

CARD_SUBSET_EQ
    |- !a b. FINITE b /\ a SUBSET b /\ CARD a = CARD b ==> a = b

CARD_SUBSET_LE
    |- !a b. FINITE b /\ a SUBSET b /\ CARD b <= CARD a ==> a = b

CARD_UNION
    |- !s t.
           FINITE s /\ FINITE t /\ s INTER t = {}
           ==> CARD (s UNION t) = CARD s + CARD t

CARD_UNIONS_LE
    |- !s t m n.
           s HAS_SIZE m /\ (!x. x IN s ==> FINITE (t x) /\ CARD (t x) <= n)
           ==> CARD (UNIONS {t x | x IN s}) <= m * n

CARD_UNION_EQ
    |- !s t u.
           FINITE u /\ s INTER t = {} /\ s UNION t = u
           ==> CARD s + CARD t = CARD u

CARD_UNION_LE
    |- !s t. FINITE s /\ FINITE t ==> CARD (s UNION t) <= CARD s + CARD t

CHOICE
    |- !s. CHOICE s = (@x. x IN s)

CHOICE_DEF
    |- !s. ~(s = {}) ==> CHOICE s IN s

CHOOSE_SUBSET
    |- !s. FINITE s ==> (!n. n <= CARD s ==> (?t. t SUBSET s /\ t HAS_SIZE n))

COMPONENT
    |- !x s. x IN x INSERT s

COUNTABLE
    |- !t. COUNTABLE t <=> (:num) CARD_GE t

CROSS
    |- !s t. s CROSS t = {x,y | x IN s /\ y IN t}
```

```
DECOMPOSITION
    |- !s x. x IN s <=> (?t. s = x INSERT t /\ ~(x IN t))

DELETE
    |- !s x. s DELETE x = {y | y IN s /\ ~(y = x)}

DELETE_COMM
    |- !x y s. s DELETE x DELETE y = s DELETE y DELETE x

DELETE_DELETE
    |- !x s. s DELETE x DELETE x = s DELETE x

DELETE_INSERT
    |- !x y s.
           (x INSERT s) DELETE y =
           (if x = y then s DELETE y else x INSERT (s DELETE y))

DELETE_INTER
    |- !s t x. s DELETE x INTER t = (s INTER t) DELETE x

DELETE_NON_ELEMENT
    |- !x s. ~(x IN s) <=> s DELETE x = s

DELETE_SUBSET
    |- !x s. s DELETE x SUBSET s

DIFF
    |- !s t. s DIFF t = {x | x IN s /\ ~(x IN t)}

DIFF_DIFF
    |- !s t. s DIFF t DIFF t = s DIFF t

DIFF_EMPTY
    |- !s. s DIFF {} = s

DIFF_EQ_EMPTY
    |- !s. s DIFF s = {}

DIFF_INSERT
    |- !s t x. s DIFF x INSERT t = s DELETE x DIFF t

DIFF_UNIV
    |- !s. s DIFF UNIV = {}

DISJOINT
    |- !s t. DISJOINT s t <=> s INTER t = {}

DISJOINT_DELETE_SYM
    |- !s t x. DISJOINT (s DELETE x) t <=> DISJOINT (t DELETE x) s
```

```
DISJOINT_EMPTY
    |- !s. DISJOINT {} s /\ DISJOINT s {}

DISJOINT_EMPTY_REFL
    |- !s. s = {} <=> DISJOINT s s

DISJOINT_INSERT
    |- !x s t. DISJOINT (x INSERT s) t <=> DISJOINT s t /\ ~(x IN t)

DISJOINT_NUMSEG
    |- !m n p q. DISJOINT (m..n) (p..q) <=> n < p \/ q < m \/ n < m \/ q < p

DISJOINT_SYM
    |- !s t. DISJOINT s t <=> DISJOINT t s

DISJOINT_UNION
    |- !s t u. DISJOINT (s UNION t) u <=> DISJOINT s u /\ DISJOINT t u

EMPTY
    |- {} = (\x. F)

EMPTY_DELETE
    |- !x. {} DELETE x = {}

EMPTY_DIFF
    |- !s. {} DIFF s = {}

EMPTY_GSPEC
    |- {x | F} = {}

EMPTY_NOT_UNIV
    |- ~({} = UNIV)

EMPTY_SUBSET
    |- !s. {} SUBSET s

EMPTY_UNION
    |- !s t. s UNION t = {} <=> s = {} /\ t = {}

EMPTY_UNIONS
    |- !s. UNIONS s = {} <=> (!t. t IN s ==> t = {})

EQ_UNIV
    |- (!x. x IN s) <=> s = UNIV

EXISTS_IN_CLAUSES
    |- (!P. (?x. x IN {} /\ P x) <=> F) /\
       (!P a s. (?x. x IN a INSERT s /\ P x) <=> P a \/ (?x. x IN s /\ P x))
```

```
EXISTS_IN_IMAGE
    |- !f s. (?y. y IN IMAGE f s /\ P y) <=> (?x. x IN s /\ P (f x))

EXTENSION
    |- !s t. s = t <=> (!x. x IN s <=> x IN t)

FINITE_CASES
    |- !a. FINITE a <=> a = {} \/ (?x s. a = x INSERT s /\ FINITE s)

FINITE_CROSS
    |- !s t. FINITE s /\ FINITE t ==> FINITE (s CROSS t)

FINITE_DELETE
    |- !s x. FINITE (s DELETE x) <=> FINITE s

FINITE_DELETE_IMP
    |- !s x. FINITE s ==> FINITE (s DELETE x)

FINITE_DIFF
    |- !s t. FINITE s ==> FINITE (s DIFF t)

FINITE_FUNSPACE
    |- !s t.
           FINITE s /\ FINITE t
           ==> FINITE
               {f | (!x. x IN s ==> f x IN t) /\ (!x. ~(x IN s) ==> f x = d)}

FINITE_HAS_SIZE_LEMMA
    |- !s. FINITE s ==> (?n. {x | x < n} CARD_GE s)

FINITE_IMAGE
    |- !f s. FINITE s ==> FINITE (IMAGE f s)

FINITE_IMAGE_EXPAND
    |- !f s. FINITE s ==> FINITE {y | ?x. x IN s /\ y = f x}

FINITE_IMAGE_INJ
    |- !f A. (!x y. f x = f y ==> x = y) /\ FINITE A ==> FINITE {x | f x IN A}

FINITE_IMAGE_INJ_EQ
    |- !f s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)
           ==> (FINITE (IMAGE f s) <=> FINITE s)

FINITE_IMAGE_INJ_GENERAL
    |- !f A s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y) /\ FINITE A
           ==> FINITE {x | x IN s /\ f x IN A}
```

```
FINITE_INDEX_NUMBERS
    |- !s. FINITE s <=>
           (?k f.
                (!i j. i IN k /\ j IN k /\ f i = f j ==> i = j) /\
                FINITE k /\
                s = IMAGE f k)

FINITE_INDEX_NUMSEG
    |- !s. FINITE s <=>
           (?f. (!i j. i IN 1..CARD s /\ j IN 1..CARD s /\ f i = f j ==> i = j) /\
                s = IMAGE f (1..CARD s))

FINITE_INDUCT
    |- !FINITE'. FINITE' {} /\ (!x s. FINITE' s ==> FINITE' (x INSERT s))
                 ==> (!a. FINITE a ==> FINITE' a)

FINITE_INDUCT_DELETE
    |- !P. P {} /\
           (!s. FINITE s /\ ~(s = {})
                ==> (?x. x IN s /\ (P (s DELETE x) ==> P s)))
           ==> (!s. FINITE s ==> P s)

FINITE_INDUCT_STRONG
    |- !P. P {} /\ (!x s. P s /\ ~(x IN s) /\ FINITE s ==> P (x INSERT s))
           ==> (!s. FINITE s ==> P s)

FINITE_INSERT
    |- !s x. FINITE (x INSERT s) <=> FINITE s

FINITE_INTER
    |- !s t. FINITE s \/ FINITE t ==> FINITE (s INTER t)

FINITE_NUMSEG
    |- !m n. FINITE (m..n)

FINITE_NUMSEG_LE
    |- !n. FINITE {m | m <= n}

FINITE_NUMSEG_LT
    |- !n. FINITE {m | m < n}

FINITE_POWERSET
    |- !s. FINITE s ==> FINITE {t | t SUBSET s}

FINITE_PRODUCT
    |- !s t. FINITE s /\ FINITE t ==> FINITE {x,y | x IN s /\ y IN t}
```

```
FINITE_PRODUCT_DEPENDENT
    |- !s t.
           FINITE s /\ (!x. x IN s ==> FINITE (t x))
           ==> FINITE {x,y | x IN s /\ y IN t x}

FINITE_RECURSION
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> ITSET f {} b = b /\
               (!x s.
                    FINITE s
                    ==> ITSET f (x INSERT s) b =
                        (if x IN s then ITSET f s b else f x (ITSET f s b)))

FINITE_RECURSION_DELETE
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> ITSET f {} b = b /\
               (!x s.
                    FINITE s
                    ==> ITSET f s b =
                        (if x IN s
                         then f x (ITSET f (s DELETE x) b)
                         else ITSET f (s DELETE x) b))

FINITE_RESTRICT
    |- !s p. FINITE s ==> FINITE {x | x IN s /\ P x}

FINITE_RULES
    |- FINITE {} /\ (!x s. FINITE s ==> FINITE (x INSERT s))

FINITE_SET_OF_LIST
    |- !l. FINITE (set_of_list l)

FINITE_SUBSET
    |- !s t. FINITE t /\ s SUBSET t ==> FINITE s

FINITE_SUBSETS
    |- !s. FINITE s ==> FINITE {t | t SUBSET s}

FINITE_SUBSET_IMAGE
    |- !f s t.
           FINITE t /\ t SUBSET IMAGE f s <=>
           (?s'. FINITE s' /\ s' SUBSET s /\ t = IMAGE f s')
```

```
FINITE_SUBSET_IMAGE_IMP
    |- !f s t.
           FINITE t /\ t SUBSET IMAGE f s
           ==> (?s'. FINITE s' /\ s' SUBSET s /\ t SUBSET IMAGE f s')

FINITE_UNION
    |- !s t. FINITE (s UNION t) <=> FINITE s /\ FINITE t

FINITE_UNIONS
    |- !s. FINITE s ==> (FINITE (UNIONS s) <=> (!t. t IN s ==> FINITE t))

FINITE_UNION_IMP
    |- !s t. FINITE s /\ FINITE t ==> FINITE (s UNION t)

FINREC
    |- (FINREC f b s a 0 <=> s = {} /\ a = b) /\
       (FINREC f b s a (SUC n) <=>
        (?x c. x IN s /\ FINREC f b (s DELETE x) c n /\ a = f x c))

FINREC_1_LEMMA
    |- !f b s a. FINREC f b s a (SUC 0) <=> (?x. s = {x} /\ a = f x b)

FINREC_EXISTS_LEMMA
    |- !f b s. FINITE s ==> (?a n. FINREC f b s a n)

FINREC_FUN
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> (?g. g {} = b /\
                    (!s x. FINITE s /\ x IN s ==> g s = f x (g (s DELETE x))))

FINREC_FUN_LEMMA
    |- !P R.
           (!s. P s ==> (?a n. R s a n)) /\
           (!n1 n2 s a1 a2. R s a1 n1 /\ R s a2 n2 ==> a1 = a2 /\ n1 = n2)
           ==> (?f. !s a. P s ==> ((?n. R s a n) <=> f s = a))

FINREC_SUC_LEMMA
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> (!n s z.
                    FINREC f b s z (SUC n)
                    ==> (!x. x IN s
                             ==> (?w. FINREC f b (s DELETE x) w n /\ z = f x w)))
```

```
FINREC_UNIQUE_LEMMA
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> (!n1 n2 s a1 a2.
                    FINREC f b s a1 n1 /\ FINREC f b s a2 n2
                    ==> a1 = a2 /\ n1 = n2)

FORALL_IN_CLAUSES
    |- (!P. (!x. x IN {} ==> P x) <=> T) /\
       (!P a s. (!x. x IN a INSERT s ==> P x) <=> P a /\ (!x. x IN s ==> P x))

FORALL_IN_IMAGE
    |- !f s. (!y. y IN IMAGE f s ==> P y) <=> (!x. x IN s ==> P (f x))

FORALL_IN_UNIONS
    |- !P s. (!x. x IN UNIONS s ==> P x) <=> (!t x. t IN s /\ x IN t ==> P x)

FUNCTION_FACTORS_LEFT
    |- !f g. (!x y. g x = g y ==> f x = f y) <=> (?h. f = h o g)

FUNCTION_FACTORS_RIGHT
    |- !f g. (!x. ?y. g y = f x) <=> (?h. f = g o h)

GSPEC
    |- !p. GSPEC p = p

HAS_SIZE
    |- !s n. s HAS_SIZE n <=> FINITE s /\ CARD s = n

HAS_SIZE_0
    |- !s n. s HAS_SIZE 0 <=> s = {}

HAS_SIZE_CARD
    |- !s n. s HAS_SIZE n ==> CARD s = n

HAS_SIZE_CLAUSES
    |- (s HAS_SIZE 0 <=> s = {}) /\
       (s HAS_SIZE SUC n <=>
        (?a t. t HAS_SIZE n /\ ~(a IN t) /\ s = a INSERT t))

HAS_SIZE_CROSS
    |- !s t m n. s HAS_SIZE m /\ t HAS_SIZE n ==> s CROSS t HAS_SIZE m * n

HAS_SIZE_FUNSPACE
    |- !d n t m s.
           s HAS_SIZE m /\ t HAS_SIZE n
           ==> {f | (!x. x IN s ==> f x IN t) /\ (!x. ~(x IN s) ==> f x = d)} HAS_SIZE
               n EXP m
```

```
HAS_SIZE_IMAGE_INJ
    |- !f s n.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y) /\ s HAS_SIZE n
           ==> IMAGE f s HAS_SIZE n

HAS_SIZE_INDEX
    |- !s n.
           s HAS_SIZE n
           ==> (?f. (!m. m < n ==> f m IN s) /\
                    (!x. x IN s ==> (?!m. m < n /\ f m = x)))

HAS_SIZE_NUMSEG
    |- !m n. m..n HAS_SIZE (n + 1) - m

HAS_SIZE_NUMSEG_1
    |- !n. 1..n HAS_SIZE n

HAS_SIZE_NUMSEG_LE
    |- !n. {m | m <= n} HAS_SIZE n + 1

HAS_SIZE_NUMSEG_LT
    |- !n. {m | m < n} HAS_SIZE n

HAS_SIZE_POWERSET
    |- !s n. s HAS_SIZE n ==> {t | t SUBSET s} HAS_SIZE 2 EXP n

HAS_SIZE_PRODUCT
    |- !s m t n.
           s HAS_SIZE m /\ t HAS_SIZE n
           ==> {x,y | x IN s /\ y IN t} HAS_SIZE m * n

HAS_SIZE_PRODUCT_DEPENDENT
    |- !s m t n.
           s HAS_SIZE m /\ (!x. x IN s ==> t x HAS_SIZE n)
           ==> {x,y | x IN s /\ y IN t x} HAS_SIZE m * n

HAS_SIZE_SUC
    |- !s n.
           s HAS_SIZE SUC n <=>
           ~(s = {}) /\ (!a. a IN s ==> s DELETE a HAS_SIZE n)

HAS_SIZE_UNION
    |- !s t m n.
           s HAS_SIZE m /\ t HAS_SIZE n /\ DISJOINT s t
           ==> s UNION t HAS_SIZE m + n
```

```
HAS_SIZE_UNIONS
    |- !s t m n.
           s HAS_SIZE m /\
           (!x. x IN s ==> t x HAS_SIZE n) /\
           (!x y. x IN s /\ y IN s /\ ~(x = y) ==> DISJOINT (t x) (t y))
           ==> UNIONS {t x | x IN s} HAS_SIZE m * n

IMAGE
    |- !s f. IMAGE f s = {y | ?x. x IN s /\ y = f x}

IMAGE_CLAUSES
    |- IMAGE f {} = {} /\ IMAGE f (x INSERT s) = f x INSERT IMAGE f s

IMAGE_CONST
    |- !s c. IMAGE (\x. c) s = (if s = {} then {} else {c})

IMAGE_DELETE_INJ
    |- (!x. f x = f a ==> x = a)
       ==> IMAGE f (s DELETE a) = IMAGE f s DELETE f a

IMAGE_DIFF_INJ
    |- (!x y. f x = f y ==> x = y)
       ==> IMAGE f (s DIFF t) = IMAGE f s DIFF IMAGE f t

IMAGE_EQ_EMPTY
    |- !f s. IMAGE f s = {} <=> s = {}

IMAGE_IMP_INJECTIVE
    |- !s f.
           FINITE s /\ IMAGE f s = s
           ==> (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)

IMAGE_IMP_INJECTIVE_GEN
    |- !s t f.
           FINITE s /\ CARD s = CARD t /\ IMAGE f s = t
           ==> (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)

IMAGE_SUBSET
    |- !f s t. s SUBSET t ==> IMAGE f s SUBSET IMAGE f t

IMAGE_UNION
    |- !f s t. IMAGE f (s UNION t) = IMAGE f s UNION IMAGE f t

IMAGE_o
    |- !f g s. IMAGE (f o g) s = IMAGE f (IMAGE g s)

IN
    |- !P x. x IN P <=> P x
```

```
INFINITE
    |- !s. INFINITE s <=> ~FINITE s

INFINITE_DIFF_FINITE
    |- !s t. INFINITE s /\ FINITE t ==> INFINITE (s DIFF t)

INFINITE_IMAGE_INJ
    |- !f. (!x y. f x = f y ==> x = y)
           ==> (!s. INFINITE s ==> INFINITE (IMAGE f s))

INFINITE_NONEMPTY
    |- !s. INFINITE s ==> ~(s = {})

INJ
    |- !t s f.
           INJ f s t <=>
           (!x. x IN s ==> f x IN t) /\
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)

INJECTIVE_LEFT_INVERSE
    |- (!x y. f x = f y ==> x = y) <=> (?g. !x. g (f x) = x)

INJECTIVE_ON_LEFT_INVERSE
    |- !f s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y) <=>
           (?g. !x. x IN s ==> g (f x) = x)

INSERT
    |- x INSERT s = {y | y IN s \/ y = x}

INSERT_AC
    |- x INSERT y INSERT s = y INSERT x INSERT s /\
       x INSERT x INSERT s = x INSERT s

INSERT_COMM
    |- !x y s. x INSERT y INSERT s = y INSERT x INSERT s

INSERT_DEF
    |- !s x. x INSERT s = (\y. y IN s \/ y = x)

INSERT_DELETE
    |- !x s. x IN s ==> x INSERT (s DELETE x) = s

INSERT_DIFF
    |- !s t x.
           x INSERT s DIFF t =
           (if x IN t then s DIFF t else x INSERT (s DIFF t))
```

```
INSERT_INSERT
    |- !x s. x INSERT x INSERT s = x INSERT s

INSERT_INTER
    |- !x s t.
           x INSERT s INTER t =
           (if x IN t then x INSERT (s INTER t) else s INTER t)

INSERT_SUBSET
    |- !x s t. x INSERT s SUBSET t <=> x IN t /\ s SUBSET t

INSERT_UNION
    |- !x s t.
           x INSERT s UNION t =
           (if x IN t then s UNION t else x INSERT (s UNION t))

INSERT_UNION_EQ
    |- !x s t. x INSERT s UNION t = x INSERT (s UNION t)

INSERT_UNIV
    |- !x. x INSERT UNIV = UNIV

INTER
    |- !s t. s INTER t = {x | x IN s /\ x IN t}

INTERS
    |- !s. INTERS s = {x | !u. u IN s ==> x IN u}

INTERS_0
    |- INTERS {} = UNIV

INTERS_1
    |- INTERS {s} = s

INTERS_2
    |- INTERS {s, t} = s INTER t

INTERS_INSERT
    |- INTERS (s INSERT u) = s INTER INTERS u

INTER_ACI
    |- p INTER q = q INTER p /\
       (p INTER q) INTER r = p INTER q INTER r /\
       p INTER q INTER r = q INTER p INTER r /\
       p INTER p = p /\
       p INTER p INTER q = p INTER q

INTER_ASSOC
    |- !s t u. (s INTER t) INTER u = s INTER t INTER u
```

```
INTER_COMM
    |- !s t. s INTER t = t INTER s

INTER_EMPTY
    |- (!s. {} INTER s = {}) /\ (!s. s INTER {} = {})

INTER_IDEMPOT
    |- !s. s INTER s = s

INTER_OVER_UNION
    |- !s t u. s UNION t INTER u = (s UNION t) INTER (s UNION u)

INTER_SUBSET
    |- (!s t. s INTER t SUBSET s) /\ (!s t. t INTER s SUBSET s)

INTER_UNIV
    |- (!s. UNIV INTER s = s) /\ (!s. s INTER UNIV = s)

IN_CROSS
    |- !x y s t. x,y IN s CROSS t <=> x IN s /\ y IN t

IN_DELETE
    |- !s x y. x IN s DELETE y <=> x IN s /\ ~(x = y)

IN_DELETE_EQ
    |- !s x x'. (x IN s <=> x' IN s) <=> x IN s DELETE x' <=> x' IN s DELETE x

IN_DIFF
    |- !s t x. x IN s DIFF t <=> x IN s /\ ~(x IN t)

IN_DISJOINT
    |- !s t. DISJOINT s t <=> ~(?x. x IN s /\ x IN t)

IN_ELIM_PAIR_THM
    |- !P a b. a,b IN {x,y | P x y} <=> P a b

IN_ELIM_THM
    |- (!P x. x IN GSPEC (\v. P (SETSPEC v)) <=> P (\p t. p /\ x = t)) /\
       (!p x. x IN {y | p y} <=> p x) /\
       (!P x. GSPEC (\v. P (SETSPEC v)) x <=> P (\p t. p /\ x = t)) /\
       (!p x. {y | p y} x <=> p x) /\
       (!p x. x IN (\y. p y) <=> p x)

IN_IMAGE
    |- !y s f. y IN IMAGE f s <=> (?x. y = f x /\ x IN s)

IN_INSERT
    |- !x y s. x IN y INSERT s <=> x = y \/ x IN s
```

```
IN_INTER
    |- !s t x. x IN s INTER t <=> x IN s /\ x IN t

IN_INTERS
    |- !s x. x IN INTERS s <=> (!t. t IN s ==> x IN t)

IN_NUMSEG
    |- !m n p. p IN m..n <=> m <= p /\ p <= n

IN_REST
    |- !x s. x IN REST s <=> x IN s /\ ~(x = CHOICE s)

IN_SET_OF_LIST
    |- !x l. x IN set_of_list l <=> MEM x l

IN_SING
    |- !x y. x IN {y} <=> x = y

IN_UNION
    |- !s t x. x IN s UNION t <=> x IN s \/ x IN t

IN_UNIONS
    |- !s x. x IN UNIONS s <=> (?t. t IN s /\ x IN t)

IN_UNIV
    |- !x. x IN UNIV

ITSET
    |- !b f s.
           ITSET f s b =
           (@g. g {} = b /\
                (!x s.
                     FINITE s
                     ==> g (x INSERT s) = (if x IN s then g s else f x (g s))))
           s

ITSET_EQ
    |- !s f g b.
           FINITE s /\
           (!x. x IN s ==> f x = g x) /\
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s)) /\
           (!x y s. ~(x = y) ==> g x (g y s) = g y (g x s))
           ==> ITSET f s b = ITSET g s b

LENGTH_LIST_OF_SET
    |- !s. FINITE s ==> LENGTH (list_of_set s) = CARD s
```

```
LIST_OF_SET_PROPERTIES
    |- !s. FINITE s
           ==> set_of_list (list_of_set s) = s /\
               LENGTH (list_of_set s) = CARD s

MEMBER_NOT_EMPTY
    |- !s. (?x. x IN s) <=> ~(s = {})

MEM_LIST_OF_SET
    |- !s. FINITE s ==> (!x. MEM x (list_of_set s) <=> x IN s)

NOT_EMPTY_INSERT
    |- !x s. ~({} = x INSERT s)

NOT_EQUAL_SETS
    |- !s t. ~(s = t) <=> (?x. x IN t <=> ~(x IN s))

NOT_INSERT_EMPTY
    |- !x s. ~(x INSERT s = {})

NOT_IN_EMPTY
    |- !x. ~(x IN {})

NOT_PSUBSET_EMPTY
    |- !s. ~(s PSUBSET {})

NOT_UNIV_PSUBSET
    |- !s. ~(UNIV PSUBSET s)

NUMSEG_ADD_SPLIT
    |- !m n p. m <= n + 1 ==> m..n + p = (m..n) UNION (n + 1..n + p)

NUMSEG_CLAUSES
    |- (!m. m..0 = (if m = 0 then {0} else {})) /\
       (!m n. m..SUC n = (if m <= SUC n then SUC n INSERT (m..n) else m..n))

NUMSEG_COMBINE_L
    |- !m p m. m <= p /\ p <= n ==> (m..p - 1) UNION (p..n) = m..n

NUMSEG_COMBINE_R
    |- !m p m. m <= p /\ p <= n ==> (m..p) UNION (p + 1..n) = m..n

NUMSEG_EMPTY
    |- !m n. m..n = {} <=> n < m

NUMSEG_LREC
    |- !m n. m <= n ==> m INSERT (m + 1..n) = m..n
```

```
NUMSEG_OFFSET_IMAGE
    |- !m n p. m + p..n + p = IMAGE (\i. i + p) (m..n)

NUMSEG_REC
    |- !m n. m <= SUC n ==> m..SUC n = SUC n INSERT (m..n)

NUMSEG_RREC
    |- !m n. m <= n ==> n INSERT (m..n - 1) = m..n

NUMSEG_SING
    |- !n. n..n = {n}

PAIRWISE
    |- (PAIRWISE r [] <=> T) /\
       (PAIRWISE r (CONS h t) <=> ALL (r h) t /\ PAIRWISE r t)

PSUBSET
    |- !s t. s PSUBSET t <=> s SUBSET t /\ ~(s = t)

PSUBSET_INSERT_SUBSET
    |- !s t. s PSUBSET t <=> (?x. ~(x IN s) /\ x INSERT s SUBSET t)

PSUBSET_IRREFL
    |- !s. ~(s PSUBSET s)

PSUBSET_MEMBER
    |- !s t. s PSUBSET t <=> s SUBSET t /\ (?y. y IN t /\ ~(y IN s))

PSUBSET_SUBSET_TRANS
    |- !s t u. s PSUBSET t /\ t SUBSET u ==> s PSUBSET u

PSUBSET_TRANS
    |- !s t u. s PSUBSET t /\ t PSUBSET u ==> s PSUBSET u

PSUBSET_UNIV
    |- !s. s PSUBSET UNIV <=> (?x. ~(x IN s))

REST
    |- !s. REST s = s DELETE CHOICE s

SETSPEC
    |- !P v t. SETSPEC v P t <=> P /\ v = t

SET_CASES
    |- !s. s = {} \/ (?x t. s = x INSERT t /\ ~(x IN t))

SET_OF_LIST_APPEND
    |- !l1 l2. set_of_list (APPEND l1 l2) = set_of_list l1 UNION set_of_list l2
```

```
SET_OF_LIST_OF_SET
    |- !s. FINITE s ==> set_of_list (list_of_set s) = s

SET_RECURSION_LEMMA
    |- !f b.
           (!x y s. ~(x = y) ==> f x (f y s) = f y (f x s))
           ==> (?g. g {} = b /\
                    (!x s.
                         FINITE s
                         ==> g (x INSERT s) =
                             (if x IN s then g s else f x (g s))))

SIMPLE_IMAGE
    |- !f s. {f x | x IN s} = IMAGE f s

SING
    |- !s. SING s <=> (?x. s = {x})

SUBSET
    |- !s t. s SUBSET t <=> (!x. x IN s ==> x IN t)

SUBSET_ANTISYM
    |- !s t. s SUBSET t /\ t SUBSET s ==> s = t

SUBSET_DELETE
    |- !x s t. s SUBSET t DELETE x <=> ~(x IN s) /\ s SUBSET t

SUBSET_DIFF
    |- !s t. s DIFF t SUBSET s

SUBSET_EMPTY
    |- !s. s SUBSET {} <=> s = {}

SUBSET_IMAGE
    |- !f s t. s SUBSET IMAGE f t <=> (?u. u SUBSET t /\ s = IMAGE f u)

SUBSET_INSERT
    |- !x s. ~(x IN s) ==> (!t. s SUBSET x INSERT t <=> s SUBSET t)

SUBSET_INSERT_DELETE
    |- !x s t. s SUBSET x INSERT t <=> s DELETE x SUBSET t

SUBSET_INTER_ABSORPTION
    |- !s t. s SUBSET t <=> s INTER t = s

SUBSET_NUMSEG
    |- !m n p q. m..n SUBSET p..q <=> n < m \/ p <= m /\ n <= q
```

```
SUBSET_PSUBSET_TRANS
    |- !s t u. s SUBSET t /\ t PSUBSET u ==> s PSUBSET u

SUBSET_REFL
    |- !s. s SUBSET s

SUBSET_RESTRICT
    |- !s P. {x | x IN s /\ P x} SUBSET s

SUBSET_TRANS
    |- !s t u. s SUBSET t /\ t SUBSET u ==> s SUBSET u

SUBSET_UNION
    |- (!s t. s SUBSET s UNION t) /\ (!s t. s SUBSET t UNION s)

SUBSET_UNION_ABSORPTION
    |- !s t. s SUBSET t <=> s UNION t = t

SUBSET_UNIV
    |- !s. s SUBSET UNIV

SURJ
    |- !t s f.
           SURJ f s t <=>
           (!x. x IN s ==> f x IN t) /\
           (!x. x IN t ==> (?y. y IN s /\ f y = x))

SURJECTIVE_IFF_INJECTIVE
    |- !s f.
           FINITE s /\ IMAGE f s SUBSET s
           ==> ((!y. y IN s ==> (?x. x IN s /\ f x = y)) <=>
                (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y))

SURJECTIVE_IFF_INJECTIVE_GEN
    |- !s t f.
           FINITE s /\ FINITE t /\ CARD s = CARD t /\ IMAGE f s SUBSET t
           ==> ((!y. y IN t ==> (?x. x IN s /\ f x = y)) <=>
                (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y))

SURJECTIVE_ON_RIGHT_INVERSE
    |- !f t.
           (!y. y IN t ==> (?x. x IN s /\ f x = y)) <=>
           (?g. !y. y IN t ==> g y IN s /\ f (g y) = y)

SURJECTIVE_RIGHT_INVERSE
    |- (!y. ?x. f x = y) <=> (?g. !y. f (g y) = y)

UNION
    |- !s t. s UNION t = {x | x IN s \/ x IN t}
```

```
UNIONS
    |- !s. UNIONS s = {x | ?u. u IN s /\ x IN u}

UNIONS_0
    |- UNIONS {} = {}

UNIONS_1
    |- UNIONS {s} = s

UNIONS_2
    |- UNIONS {s, t} = s UNION t

UNIONS_INSERT
    |- UNIONS (s INSERT u) = s UNION UNIONS u

UNION_ACI
    |- p UNION q = q UNION p /\
       (p UNION q) UNION r = p UNION q UNION r /\
       p UNION q UNION r = q UNION p UNION r /\
       p UNION p = p /\
       p UNION p UNION q = p UNION q

UNION_ASSOC
    |- !s t u. (s UNION t) UNION u = s UNION t UNION u

UNION_COMM
    |- !s t. s UNION t = t UNION s

UNION_EMPTY
    |- (!s. {} UNION s = s) /\ (!s. s UNION {} = s)

UNION_IDEMPOT
    |- !s. s UNION s = s

UNION_OVER_INTER
    |- !s t u. s INTER (t UNION u) = s INTER t UNION s INTER u

UNION_SUBSET
    |- !s t u. s UNION t SUBSET u <=> s SUBSET u /\ t SUBSET u

UNION_UNIV
    |- (!s. UNIV UNION s = UNIV) /\ (!s. s UNION UNIV = UNIV)

UNIV
    |- UNIV = (\x. T)

UNIV_NOT_EMPTY
    |- ~(UNIV = {})
```

```
UNIV_SUBSET
    |- !s. UNIV SUBSET s <=> s = UNIV

list_of_set
    |- !s. list_of_set s = (@l. set_of_list l = s /\ LENGTH l = CARD s)

num_FINITE
    |- !s. FINITE s <=> (?a. !x. x IN s ==> x <= a)

num_FINITE_AVOID
    |- !s. FINITE s ==> (?a. ~(a IN s))

num_INFINITE
    |- INFINITE (:num)

numseg
    |- !m n. m..n = {x | m <= x /\ x <= n}

pairwise
    |- !s r. pairwise r s <=> (!x y. x IN s /\ y IN s /\ ~(x = y) ==> r x y)

set_of_list
    |- set_of_list [] = {} /\ set_of_list (CONS h t) = h INSERT set_of_list t
```

## 2.10 Theorems about iterated operations

```
CARD_EQ_NSUM
    |- !s. FINITE s ==> CARD s = nsum s (\x. 1)

CARD_EQ_SUM
    |- !s. FINITE s ==> &(CARD s) = sum s (\x. &1)

FINITE_SUPPORT
    |- !op f s. FINITE s ==> FINITE (support op f s)

FINITE_SUPPORT_DELTA
    |- !op f a. FINITE (support op (\x. if x = a then f x else neutral op) s)

IN_SUPPORT
    |- !op f x s. x IN support op f s <=> x IN s /\ ~(f x = neutral op)

ITERATE_BIJECTION
    |- !op. monoidal op
            ==> (!f p s.
                     (!x. x IN s ==> p x IN s) /\
                     (!y. y IN s ==> (?!x. x IN s /\ p x = y))
                     ==> iterate op s f = iterate op s (f o p))
```

```
ITERATE_CASES
    |- !op f s.
           iterate op s f =
           (if FINITE (support op f s)
            then iterate op (support op f s) f
            else neutral op)

ITERATE_CLAUSES
    |- !op. monoidal op
            ==> (!f. iterate op {} f = neutral op) /\
                (!f x s.
                     FINITE s
                     ==> iterate op (x INSERT s) f =
                         (if x IN s
                          then iterate op s f
                          else op (f x) (iterate op s f)))

ITERATE_CLAUSES_GEN
    |- !op. monoidal op
            ==> (!f. iterate op {} f = neutral op) /\
                (!f x s.
                     monoidal op /\ FINITE (support op f s)
                     ==> iterate op (x INSERT s) f =
                         (if x IN s
                          then iterate op s f
                          else op (f x) (iterate op s f)))

ITERATE_CLOSED
    |- !op. monoidal op
            ==> (!P. P (neutral op) /\ (!x y. P x /\ P y ==> P (op x y))
                     ==> (!f s.
                              FINITE s /\ (!x. x IN s ==> P (f x))
                              ==> P (iterate op s f)))

ITERATE_CLOSED_GEN
    |- !op. monoidal op
            ==> (!P. P (neutral op) /\ (!x y. P x /\ P y ==> P (op x y))
                     ==> (!f s.
                              FINITE (support op f s) /\
                              (!x. x IN s /\ ~(f x = neutral op) ==> P (f x))
                              ==> P (iterate op s f)))

ITERATE_DELETE
    |- !op. monoidal op
            ==> (!f s a.
                     FINITE s /\ a IN s
                     ==> op (f a) (iterate op (s DELETE a) f) = iterate op s f)
```

```
ITERATE_DELTA
    |- !op. monoidal op
            ==> (!f a s.
                     iterate op s (\x. if x = a then f x else neutral op) =
                     (if a IN s then f a else neutral op))

ITERATE_DIFF
    |- !op. monoidal op
            ==> (!f s t.
                     FINITE s /\ t SUBSET s
                     ==> op (iterate op (s DIFF t) f) (iterate op t f) =
                         iterate op s f)

ITERATE_DIFF_GEN
    |- !op. monoidal op
            ==> (!f s t.
                     FINITE (support op f s) /\
                     support op f t SUBSET support op f s
                     ==> op (iterate op (s DIFF t) f) (iterate op t f) =
                         iterate op s f)

ITERATE_EQ
    |- !op. monoidal op
            ==> (!f g s.
                     (!x. x IN s ==> f x = g x)
                     ==> iterate op s f = iterate op s g)

ITERATE_EQ_GENERAL
    |- !op. monoidal op
            ==> (!s t f g h.
                     (!y. y IN t ==> (?!x. x IN s /\ h x = y)) /\
                     (!x. x IN s ==> h x IN t /\ g (h x) = f x)
                     ==> iterate op s f = iterate op t g)

ITERATE_EQ_GENERAL_INVERSES
    |- !op. monoidal op
            ==> (!s t f g h k.
                     (!y. y IN t ==> k y IN s /\ h (k y) = y) /\
                     (!x. x IN s ==> h x IN t /\ k (h x) = x /\ g (h x) = f x)
                     ==> iterate op s f = iterate op t g)

ITERATE_EQ_NEUTRAL
    |- !op. monoidal op
            ==> (!f s.
                     (!x. x IN s ==> f x = neutral op)
                     ==> iterate op s f = neutral op)
```

```
ITERATE_IMAGE
    |- !op. monoidal op
           ==> (!f g s.
                    (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)
                    ==> iterate op (IMAGE f s) g = iterate op s (g o f))

ITERATE_INJECTION
    |- !op. monoidal op
           ==> (!f p s.
                    FINITE s /\
                    (!x. x IN s ==> p x IN s) /\
                    (!x y. x IN s /\ y IN s /\ p x = p y ==> x = y)
                    ==> iterate op s (f o p) = iterate op s f)

ITERATE_ITERATE_PRODUCT
    |- !op. monoidal op
           ==> (!s t x.
                    FINITE s /\ (!i. i IN s ==> FINITE (t i))
                    ==> iterate op s (\i. iterate op (t i) (x i)) =
                        iterate op {i,j | i IN s /\ j IN t i} (\(i,j). x i j))

ITERATE_RELATED
    |- !op. monoidal op
           ==> (!R. R (neutral op) (neutral op) /\
                    (!x1 y1 x2 y2.
                         R x1 x2 /\ R y1 y2 ==> R (op x1 y1) (op x2 y2))
                    ==> (!f g s.
                             FINITE s /\ (!x. x IN s ==> R (f x) (g x))
                             ==> R (iterate op s f) (iterate op s g)))

ITERATE_SING
    |- !op. monoidal op ==> (!f x. iterate op {x} f = f x)

ITERATE_SUPPORT
    |- !op f s. iterate op (support op f s) f = iterate op s f

ITERATE_UNION
    |- !op. monoidal op
           ==> (!f s t.
                    FINITE s /\ FINITE t /\ DISJOINT s t
                    ==> iterate op (s UNION t) f =
                        op (iterate op s f) (iterate op t f))
```

```
ITERATE_UNION_GEN
    |- !op. monoidal op
            ==> (!f s t.
                     FINITE (support op f s) /\
                     FINITE (support op f t) /\
                     DISJOINT (support op f s) (support op f t)
                     ==> iterate op (s UNION t) f =
                         op (iterate op s f) (iterate op t f))

MONOIDAL_ADD
    |- monoidal (+)

MONOIDAL_MUL
    |- monoidal (*)

MONOIDAL_REAL_ADD
    |- monoidal (+)

MONOIDAL_REAL_MUL
    |- monoidal (*)

NEUTRAL_ADD
    |- neutral (+) = 0

NEUTRAL_MUL
    |- neutral (*) = 1

NEUTRAL_REAL_ADD
    |- neutral (+) = &0

NEUTRAL_REAL_MUL
    |- neutral (*) = &1

NSUM_0
    |- !s. nsum s (\n. 0) = 0

NSUM_ADD
    |- !f g s. FINITE s ==> nsum s (\x. f x + g x) = nsum s f + nsum s g

NSUM_ADD_NUMSEG
    |- !f g m n. nsum (m..n) (\i. f i + g i) = nsum (m..n) f + nsum (m..n) g

NSUM_ADD_SPLIT
    |- !f m n p.
           m <= n + 1
           ==> nsum (m..n + p) f = nsum (m..n) f + nsum (n + 1..n + p) f
```

```
NSUM_BIJECTION
    |- !f p s.
           (!x. x IN s ==> p x IN s) /\
           (!y. y IN s ==> (?!x. x IN s /\ p x = y))
           ==> nsum s f = nsum s (f o p)

NSUM_BOUND
    |- !s f b. FINITE s /\ (!x. x IN s ==> f x <= b) ==> nsum s f <= CARD s * b

NSUM_BOUND_GEN
    |- !s t b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x <= b DIV CARD s)
           ==> nsum s f <= b

NSUM_BOUND_LT
    |- !s f b.
           FINITE s /\ (!x. x IN s ==> f x <= b) /\ (?x. x IN s /\ f x < b)
           ==> nsum s f < CARD s * b

NSUM_BOUND_LT_ALL
    |- !s f b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < b)
           ==> nsum s f < CARD s * b

NSUM_BOUND_LT_GEN
    |- !s t b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < b DIV CARD s)
           ==> nsum s f < b

NSUM_CLAUSES
    |- (!f. nsum {} f = 0) /\
       (!x f s.
            FINITE s
            ==> nsum (x INSERT s) f =
                (if x IN s then nsum s f else f x + nsum s f))

NSUM_CLAUSES_LEFT
    |- !f m n. m <= n ==> nsum (m..n) f = f m + nsum (m + 1..n) f

NSUM_CLAUSES_NUMSEG
    |- (!m. nsum (m..0) f = (if m = 0 then f 0 else 0)) /\
       (!m n.
            nsum (m..SUC n) f =
            (if m <= SUC n then nsum (m..n) f + f (SUC n) else nsum (m..n) f))

NSUM_CLAUSES_RIGHT
    |- !f m n. 0 < n /\ m <= n ==> nsum (m..n) f = nsum (m..n - 1) f + f n
```

```
NSUM_CONST
    |- !c s. FINITE s ==> nsum s (\n. c) = CARD s * c

NSUM_CONST_NUMSEG
    |- !c m n. nsum (m..n) (\n. c) = ((n + 1) - m) * c

NSUM_DELETE
    |- !f s a. FINITE s /\ a IN s ==> f a + nsum (s DELETE a) f = nsum s f

NSUM_DELTA
    |- !s a. nsum s (\x. if x = a then b else 0) = (if a IN s then b else 0)

NSUM_DIFF
    |- !f s t.
           FINITE s /\ t SUBSET s ==> nsum (s DIFF t) f = nsum s f - nsum t f

NSUM_EQ
    |- !f g s. (!x. x IN s ==> f x = g x) ==> nsum s f = nsum s g

NSUM_EQ_0
    |- !f s. (!x. x IN s ==> f x = 0) ==> nsum s f = 0

NSUM_EQ_0_IFF
    |- !s. FINITE s ==> (nsum s f = 0 <=> (!x. x IN s ==> f x = 0))

NSUM_EQ_0_IFF_NUMSEG
    |- !f m n. nsum (m..n) f = 0 <=> (!i. m <= i /\ i <= n ==> f i = 0)

NSUM_EQ_0_NUMSEG
    |- !f s. (!i. m <= i /\ i <= n ==> f i = 0) ==> nsum (m..n) f = 0

NSUM_EQ_GENERAL
    |- !s t f g h.
           (!y. y IN t ==> (?!x. x IN s /\ h x = y)) /\
           (!x. x IN s ==> h x IN t /\ g (h x) = f x)
           ==> nsum s f = nsum t g

NSUM_EQ_GENERAL_INVERSES
    |- !s t f g h k.
           (!y. y IN t ==> k y IN s /\ h (k y) = y) /\
           (!x. x IN s ==> h x IN t /\ k (h x) = x /\ g (h x) = f x)
           ==> nsum s f = nsum t g

NSUM_EQ_NUMSEG
    |- !f g m n.
           (!i. m <= i /\ i <= n ==> f i = g i)
           ==> nsum (m..n) f = nsum (m..n) g
```

```
NSUM_EQ_SUPERSET
    |- !f s t.
           FINITE t /\
           t SUBSET s /\
           (!x. x IN t ==> f x = g x) /\
           (!x. x IN s /\ ~(x IN t) ==> f x = 0)
           ==> nsum s f = nsum t g

NSUM_IMAGE
    |- !f g s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)
           ==> nsum (IMAGE f s) g = nsum s (g o f)

NSUM_IMAGE_GEN
    |- !f g s.
           FINITE s
           ==> nsum s g = nsum (IMAGE f s) (\y. nsum {x | x IN s /\ f x = y} g)

NSUM_INJECTION
    |- !f p s.
           FINITE s /\
           (!x. x IN s ==> p x IN s) /\
           (!x y. x IN s /\ y IN s /\ p x = p y ==> x = y)
           ==> nsum s (f o p) = nsum s f

NSUM_LE
    |- !f g s. FINITE s /\ (!x. x IN s ==> f x <= g x) ==> nsum s f <= nsum s g

NSUM_LE_NUMSEG
    |- !f g m n.
           (!i. m <= i /\ i <= n ==> f i <= g i)
           ==> nsum (m..n) f <= nsum (m..n) g

NSUM_LMUL
    |- !f c s. nsum s (\x. c * f x) = c * nsum s f

NSUM_LT
    |- !f g s.
           FINITE s /\ (!x. x IN s ==> f x <= g x) /\ (?x. x IN s /\ f x < g x)
           ==> nsum s f < nsum s g

NSUM_LT_ALL
    |- !f g s.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < g x)
           ==> nsum s f < nsum s g
```

```
NSUM_MULTICOUNT
    |- !R s t k.
           FINITE s /\
           FINITE t /\
           (!j. j IN t ==> CARD {i | i IN s /\ R i j} = k)
           ==> nsum s (\i. CARD {j | j IN t /\ R i j}) = k * CARD t

NSUM_MULTICOUNT_GEN
    |- !R s t k.
           FINITE s /\
           FINITE t /\
           (!j. j IN t ==> CARD {i | i IN s /\ R i j} = k j)
           ==> nsum s (\i. CARD {j | j IN t /\ R i j}) = nsum t (\i. k i)

NSUM_NSUM_PRODUCT
    |- !s t x.
           FINITE s /\ (!i. i IN s ==> FINITE (t i))
           ==> nsum s (\i. nsum (t i) (x i)) =
               nsum {i,j | i IN s /\ j IN t i} (\(i,j). x i j)

NSUM_NSUM_RESTRICT
    |- !R f s t.
           FINITE s /\ FINITE t
           ==> nsum s (\x. nsum {y | y IN t /\ R x y} (\y. f x y)) =
               nsum t (\y. nsum {x | x IN s /\ R x y} (\x. f x y))

NSUM_OFFSET
    |- !f m p. nsum (m + p..n + p) f = nsum (m..n) (\i. f (i + p))

NSUM_OFFSET_0
    |- !f m n. m <= n ==> nsum (m..n) f = nsum (0..n - m) (\i. f (i + m))

NSUM_POS_BOUND
    |- !f b s. FINITE s /\ nsum s f <= b ==> (!x. x IN s ==> f x <= b)

NSUM_RESTRICT
    |- !f s. FINITE s ==> nsum s (\x. if x IN s then f x else 0) = nsum s f

NSUM_RESTRICT_SET
    |- !s f r.
           FINITE s
           ==> nsum {x | x IN s /\ P x} f = nsum s (\x. if P x then f x else 0)

NSUM_RMUL
    |- !f c s. nsum s (\x. f x * c) = nsum s f * c

NSUM_SING
    |- !f x. nsum {x} f = f x
```

```
NSUM_SING_NUMSEG
    |- !f n. nsum (n..n) f = f n

NSUM_SUBSET
    |- !u v f.
           FINITE u /\ FINITE v /\ (!x. x IN u DIFF v ==> f x = 0)
           ==> nsum u f <= nsum v f

NSUM_SUBSET_SIMPLE
    |- !u v f. FINITE v /\ u SUBSET v ==> nsum u f <= nsum v f

NSUM_SUPERSET
    |- !f u v.
           FINITE u /\ u SUBSET v /\ (!x. x IN v /\ ~(x IN u) ==> f x = 0)
           ==> nsum v f = nsum u f

NSUM_SUPPORT
    |- !f s. nsum (support (+) f s) f = nsum s f

NSUM_SWAP
    |- !f s t.
           FINITE s /\ FINITE t
           ==> nsum s (\i. nsum t (f i)) = nsum t (\j. nsum s (\i. f i j))

NSUM_SWAP_NUMSEG
    |- !a b c d f.
           nsum (a..b) (\i. nsum (c..d) (f i)) =
           nsum (c..d) (\j. nsum (a..b) (\i. f i j))

NSUM_TRIV_NUMSEG
    |- !f m n. n < m ==> nsum (m..n) f = 0

NSUM_UNION
    |- !f s t.
           FINITE s /\ FINITE t /\ DISJOINT s t
           ==> nsum (s UNION t) f = nsum s f + nsum t f

NSUM_UNION_EQ
    |- !s t u.
           FINITE u /\ s INTER t = {} /\ s UNION t = u
           ==> nsum s f + nsum t f = nsum u f

NSUM_UNION_LZERO
    |- !f u v.
           FINITE v /\ (!x. x IN u /\ ~(x IN v) ==> f x = 0)
           ==> nsum (u UNION v) f = nsum v f
```

```
NSUM_UNION_RZERO
    |- !f u v.
           FINITE u /\ (!x. x IN v /\ ~(x IN u) ==> f x = 0)
           ==> nsum (u UNION v) f = nsum u f

REAL_OF_NUM_SUM
    |- !f s. FINITE s ==> &(nsum s f) = sum s (\x. &(f x))

REAL_OF_NUM_SUM_NUMSEG
    |- !f m n. &(nsum (m..n) f) = sum (m..n) (\i. &(f i))

SUM_0
    |- !s. sum s (\n. &0) = &0

SUM_ABS
    |- !f s. FINITE s ==> abs (sum s f) <= sum s (\x. abs (f x))

SUM_ABS_BOUND
    |- !s f b.
           FINITE s /\ (!x. x IN s ==> abs (f x) <= b)
           ==> abs (sum s f) <= &(CARD s) * b

SUM_ABS_NUMSEG
    |- !f m n. abs (sum (m..n) f) <= sum (m..n) (\i. abs (f i))

SUM_ADD
    |- !f g s. FINITE s ==> sum s (\x. f x + g x) = sum s f + sum s g

SUM_ADD_NUMSEG
    |- !f g m n. sum (m..n) (\i. f i + g i) = sum (m..n) f + sum (m..n) g

SUM_ADD_SPLIT
    |- !f m n p.
           m <= n + 1
           ==> sum (m..n + p) f = sum (m..n) f + sum (n + 1..n + p) f

SUM_BIJECTION
    |- !f p s.
           (!x. x IN s ==> p x IN s) /\
           (!y. y IN s ==> (?!x. x IN s /\ p x = y))
           ==> sum s f = sum s (f o p)

SUM_BOUND
    |- !s f b.
           FINITE s /\ (!x. x IN s ==> f x <= b) ==> sum s f <= &(CARD s) * b
```

```
SUM_BOUND_GEN
    |- !s t b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x <= b / &(CARD s))
           ==> sum s f <= b

SUM_BOUND_LT
    |- !s f b.
           FINITE s /\ (!x. x IN s ==> f x <= b) /\ (?x. x IN s /\ f x < b)
           ==> sum s f < &(CARD s) * b

SUM_BOUND_LT_ALL
    |- !s f b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < b)
           ==> sum s f < &(CARD s) * b

SUM_BOUND_LT_GEN
    |- !s t b.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < b / &(CARD s))
           ==> sum s f < b

SUM_CLAUSES
    |- (!f. sum {} f = &0) /\
       (!x f s.
            FINITE s
            ==> sum (x INSERT s) f =
                (if x IN s then sum s f else f x + sum s f))

SUM_CLAUSES_LEFT
    |- !f m n. m <= n ==> sum (m..n) f = f m + sum (m + 1..n) f

SUM_CLAUSES_NUMSEG
    |- (!m. sum (m..0) f = (if m = 0 then f 0 else &0)) /\
       (!m n.
            sum (m..SUC n) f =
            (if m <= SUC n then sum (m..n) f + f (SUC n) else sum (m..n) f))

SUM_CLAUSES_RIGHT
    |- !f m n. 0 < n /\ m <= n ==> sum (m..n) f = sum (m..n - 1) f + f n

SUM_CONST
    |- !c s. FINITE s ==> sum s (\n. c) = &(CARD s) * c

SUM_CONST_NUMSEG
    |- !c m n. sum (m..n) (\n. c) = &((n + 1) - m) * c

SUM_DELETE
    |- !f s a. FINITE s /\ a IN s ==> sum (s DELETE a) f = sum s f - f a
```

```
SUM_DELTA
    |- !s a. sum s (\x. if x = a then b else &0) = (if a IN s then b else &0)

SUM_DIFF
    |- !f s t. FINITE s /\ t SUBSET s ==> sum (s DIFF t) f = sum s f - sum t f

SUM_EQ
    |- !f g s. (!x. x IN s ==> f x = g x) ==> sum s f = sum s g

SUM_EQ_0
    |- !f s. (!x. x IN s ==> f x = &0) ==> sum s f = &0

SUM_EQ_0_NUMSEG
    |- !f s. (!i. m <= i /\ i <= n ==> f i = &0) ==> sum (m..n) f = &0

SUM_EQ_GENERAL
    |- !s t f g h.
           (!y. y IN t ==> (?!x. x IN s /\ h x = y)) /\
           (!x. x IN s ==> h x IN t /\ g (h x) = f x)
           ==> sum s f = sum t g

SUM_EQ_GENERAL_INVERSES
    |- !s t f g h k.
           (!y. y IN t ==> k y IN s /\ h (k y) = y) /\
           (!x. x IN s ==> h x IN t /\ k (h x) = x /\ g (h x) = f x)
           ==> sum s f = sum t g

SUM_EQ_NUMSEG
    |- !f g m n.
           (!i. m <= i /\ i <= n ==> f i = g i) ==> sum (m..n) f = sum (m..n) g

SUM_EQ_SUPERSET
    |- !f s t.
           FINITE t /\
           t SUBSET s /\
           (!x. x IN t ==> f x = g x) /\
           (!x. x IN s /\ ~(x IN t) ==> f x = &0)
           ==> sum s f = sum t g

SUM_IMAGE
    |- !f g s.
           (!x y. x IN s /\ y IN s /\ f x = f y ==> x = y)
           ==> sum (IMAGE f s) g = sum s (g o f)

SUM_IMAGE_GEN
    |- !f g s.
           FINITE s
           ==> sum s g = sum (IMAGE f s) (\y. sum {x | x IN s /\ f x = y} g)
```

```
SUM_INJECTION
    |- !f p s.
           FINITE s /\
           (!x. x IN s ==> p x IN s) /\
           (!x y. x IN s /\ y IN s /\ p x = p y ==> x = y)
           ==> sum s (f o p) = sum s f

SUM_LE
    |- !f g s. FINITE s /\ (!x. x IN s ==> f x <= g x) ==> sum s f <= sum s g

SUM_LE_NUMSEG
    |- !f g m n.
           (!i. m <= i /\ i <= n ==> f i <= g i)
           ==> sum (m..n) f <= sum (m..n) g

SUM_LMUL
    |- !f c s. sum s (\x. c * f x) = c * sum s f

SUM_LT
    |- !f g s.
           FINITE s /\ (!x. x IN s ==> f x <= g x) /\ (?x. x IN s /\ f x < g x)
           ==> sum s f < sum s g

SUM_LT_ALL
    |- !f g s.
           FINITE s /\ ~(s = {}) /\ (!x. x IN s ==> f x < g x)
           ==> sum s f < sum s g

SUM_MULTICOUNT
    |- !R s t k.
           FINITE s /\
           FINITE t /\
           (!j. j IN t ==> CARD {i | i IN s /\ R i j} = k)
           ==> sum s (\i. &(CARD {j | j IN t /\ R i j})) = &(k * CARD t)

SUM_MULTICOUNT_GEN
    |- !R s t k.
           FINITE s /\
           FINITE t /\
           (!j. j IN t ==> CARD {i | i IN s /\ R i j} = k j)
           ==> sum s (\i. &(CARD {j | j IN t /\ R i j})) = sum t (\i. &(k i))

SUM_NEG
    |- !f s. sum s (\x. --f x) = --sum s f

SUM_OFFSET
    |- !f m p. sum (m + p..n + p) f = sum (m..n) (\i. f (i + p))
```

```
SUM_OFFSET_0
    |- !f m n. m <= n ==> sum (m..n) f = sum (0..n - m) (\i. f (i + m))

SUM_POS_BOUND
    |- !f b s.
           FINITE s /\ (!x. x IN s ==> &0 <= f x) /\ sum s f <= b
           ==> (!x. x IN s ==> f x <= b)

SUM_POS_EQ_0
    |- !f s.
           FINITE s /\ (!x. x IN s ==> &0 <= f x) /\ sum s f = &0
           ==> (!x. x IN s ==> f x = &0)

SUM_POS_EQ_0_NUMSEG
    |- !f m n.
           (!p. m <= p /\ p <= n ==> &0 <= f p) /\ sum (m..n) f = &0
           ==> (!p. m <= p /\ p <= n ==> f p = &0)

SUM_POS_LE
    |- !f s. FINITE s /\ (!x. x IN s ==> &0 <= f x) ==> &0 <= sum s f

SUM_POS_LE_NUMSEG
    |- !m n f. (!p. m <= p /\ p <= n ==> &0 <= f p) ==> &0 <= sum (m..n) f

SUM_RESTRICT
    |- !f s. FINITE s ==> sum s (\x. if x IN s then f x else &0) = sum s f

SUM_RESTRICT_SET
    |- !s f r.
           FINITE s
           ==> sum {x | x IN s /\ P x} f = sum s (\x. if P x then f x else &0)

SUM_RMUL
    |- !f c s. sum s (\x. f x * c) = sum s f * c

SUM_SING
    |- !f x. sum {x} f = f x

SUM_SING_NUMSEG
    |- !f n. sum (n..n) f = f n

SUM_SUB
    |- !f g s. FINITE s ==> sum s (\x. f x - g x) = sum s f - sum s g
```

```
SUM_SUBSET
    |- !u v f.
          FINITE u /\
          FINITE v /\
          (!x. x IN u DIFF v ==> f x <= &0) /\
          (!x. x IN v DIFF u ==> &0 <= f x)
          ==> sum u f <= sum v f

SUM_SUBSET_SIMPLE
    |- !u v f.
          FINITE v /\ u SUBSET v /\ (!x. x IN v DIFF u ==> &0 <= f x)
          ==> sum u f <= sum v f

SUM_SUB_NUMSEG
    |- !f g m n. sum (m..n) (\i. f i - g i) = sum (m..n) f - sum (m..n) g

SUM_SUM_PRODUCT
    |- !s t x.
          FINITE s /\ (!i. i IN s ==> FINITE (t i))
          ==> sum s (\i. sum (t i) (x i)) =
              sum {i,j | i IN s /\ j IN t i} (\(i,j). x i j)

SUM_SUM_RESTRICT
    |- !R f s t.
          FINITE s /\ FINITE t
          ==> sum s (\x. sum {y | y IN t /\ R x y} (\y. f x y)) =
              sum t (\y. sum {x | x IN s /\ R x y} (\x. f x y))

SUM_SUPERSET
    |- !f u v.
          FINITE u /\ u SUBSET v /\ (!x. x IN v /\ ~(x IN u) ==> f x = &0)
          ==> sum v f = sum u f

SUM_SUPPORT
    |- !f s. sum (support (+) f s) f = sum s f

SUM_SWAP
    |- !f s t.
          FINITE s /\ FINITE t
          ==> sum s (\i. sum t (f i)) = sum t (\j. sum s (\i. f i j))

SUM_SWAP_NUMSEG
    |- !a b c d f.
          sum (a..b) (\i. sum (c..d) (f i)) =
          sum (c..d) (\j. sum (a..b) (\i. f i j))

SUM_TRIV_NUMSEG
    |- !f m n. n < m ==> sum (m..n) f = &0
```

```
SUM_UNION
    |- !f s t.
           FINITE s /\ FINITE t /\ DISJOINT s t
           ==> sum (s UNION t) f = sum s f + sum t f

SUM_UNION_EQ
    |- !s t u.
           FINITE u /\ s INTER t = {} /\ s UNION t = u
           ==> sum s f + sum t f = sum u f

SUM_UNION_LZERO
    |- !f u v.
           FINITE v /\ (!x. x IN u /\ ~(x IN v) ==> f x = &0)
           ==> sum (u UNION v) f = sum v f

SUM_UNION_RZERO
    |- !f u v.
           FINITE u /\ (!x. x IN v /\ ~(x IN u) ==> f x = &0)
           ==> sum (u UNION v) f = sum u f

SUPPORT_CLAUSES
    |- (!f. support op f {} = {}) /\
       (!f x s.
            support op f (x INSERT s) =
            (if f x = neutral op
             then support op f s
             else x INSERT support op f s)) /\
       (!f x s. support op f (s DELETE x) = support op f s DELETE x) /\
       (!f s t. support op f (s UNION t) = support op f s UNION support op f t) /\
       (!f s t. support op f (s INTER t) = support op f s INTER support op f t) /\
       (!f s t. support op f (s DIFF t) = support op f s DIFF support op f t) /\
       (!f g s. support op g (IMAGE f s) = IMAGE f (support op (g o f) s))

SUPPORT_DELTA
    |- !op s f a.
           support op (\x. if x = a then f x else neutral op) s =
           (if a IN s then support op f {a} else {})

SUPPORT_EMPTY
    |- !op f s. (!x. x IN s ==> f x = neutral op) <=> support op f s = {}

SUPPORT_SUBSET
    |- !op f s. support op f s SUBSET s

SUPPORT_SUPPORT
    |- !op f s. support op f (support op f s) = support op f s
```

```
iterate
    |- !f s op.
            iterate op s f =
            (if FINITE (support op f s)
             then ITSET (\x a. op (f x) a) (support op f s) (neutral op)
             else neutral op)

monoidal
    |- !op. monoidal op <=>
             (!x y. op x y = op y x) /\
             (!x y z. op x (op y z) = op (op x y) z) /\
             (!x. op (neutral op) x = x)

neutral
    |- !op. neutral op = (@x. !y. op x y = y /\ op y x = y)

nsum
    |- nsum = iterate (+)

sum
    |- sum = iterate (+)

support
    |- !s f op. support op f s = {x | x IN s /\ ~(f x = neutral op)}
```

## 2.11 Theorems about cartesian powers

```
CARD_FINITE_IMAGE
    |- !s. CARD (:(A)finite_image) = dimindex s

CART_EQ
    |- !x y. x = y <=> (!i. 1 <= i /\ i <= dimindex (:N) ==> x$i = y$i)

DIMINDEX_1
    |- dimindex (:1) = 1

DIMINDEX_2
    |- dimindex (:2) = 2

DIMINDEX_3
    |- dimindex (:3) = 3

DIMINDEX_FINITE_IMAGE
    |- !s t. dimindex s = dimindex t

DIMINDEX_FINITE_SUM
    |- dimindex (:(M,N)finite_sum) = dimindex (:M) + dimindex (:N)
```

```
DIMINDEX_GE_1
    |- !s. 1 <= dimindex s

DIMINDEX_HAS_SIZE_FINITE_SUM
    |- (:(M,N)finite_sum) HAS_SIZE dimindex (:M) + dimindex (:N)

DIMINDEX_NONZERO
    |- !s. ~(dimindex s = 0)

DIMINDEX_UNIQUE
    |- (:N) HAS_SIZE n ==> dimindex (:N) = n

DIMINDEX_UNIV
    |- !s. dimindex s = dimindex (:N)

EXISTS_PASTECART
    |- (?p. P p) <=> (?x y. P (pastecart x y))

FINITE_FINITE_IMAGE
    |- FINITE (:(A)finite_image)

FINITE_IMAGE_IMAGE
    |- (:(A)finite_image) = IMAGE finite_index (1..dimindex (:N))

FINITE_INDEX_INJ
    |- !i j.
           1 <= i /\ i <= dimindex (:N) /\ 1 <= j /\ j <= dimindex (:N)
           ==> (finite_index i = finite_index j <=> i = j)

FINITE_INDEX_WORKS
    |- !i. ?!n. 1 <= n /\ n <= dimindex (:N) /\ finite_index n = i

FINITE_SUM_IMAGE
    |- (:(M,N)finite_sum) =
       IMAGE mk_finite_sum (1..dimindex (:M) + dimindex (:N))

FORALL_FINITE_INDEX
    |- (!k. P k) <=> (!i. 1 <= i /\ i <= dimindex (:N) ==> P (finite_index i))

FORALL_PASTECART
    |- (!p. P p) <=> (!x y. P (pastecart x y))

FSTCART_PASTECART
    |- !x y. fstcart (pastecart x y) = x

HAS_SIZE_1
    |- (:1) HAS_SIZE 1

HAS_SIZE_2
    |- (:2) HAS_SIZE 2
```

```
HAS_SIZE_3
    |- (:3) HAS_SIZE 3

HAS_SIZE_FINITE_IMAGE
    |- !s. (:(A)finite_image) HAS_SIZE dimindex s

LAMBDA_BETA
    |- !i. 1 <= i /\ i <= dimindex (:N) ==> (lambda) g$i = g i

LAMBDA_ETA
    |- !g. (lambda i. g$i) = g

LAMBDA_UNIQUE
    |- !f g.
           (!i. 1 <= i /\ i <= dimindex (:N) ==> f$i = g i) <=> (lambda) g = f

PASTECART_EQ
    |- !x y. x = y <=> fstcart x = fstcart y /\ sndcart x = sndcart y

PASTECART_FST_SND
    |- !z. pastecart (fstcart z) (sndcart z) = z

SNDCART_PASTECART
    |- !x y. sndcart (pastecart x y) = y

cart_tybij
    |- (!a. mk_cart (dest_cart a) = a) /\ (!r. T <=> dest_cart (mk_cart r) = r)

dimindex
    |- !s. dimindex s = (if FINITE (:N) then CARD (:N) else 1)

finite_image_tybij
    |- (!a. finite_index (dest_finite_image a) = a) /\
       (!r. r IN 1..dimindex (:N) <=> dest_finite_image (finite_index r) = r)

finite_index
    |- !x i. x$i = dest_cart x (finite_index i)

finite_sum_tybij
    |- (!a. mk_finite_sum (dest_finite_sum a) = a) /\
       (!r. r IN 1..dimindex (:M) + dimindex (:N) <=>
            dest_finite_sum (mk_finite_sum r) = r)

fstcart
    |- !f. fstcart f = (lambda i. f$i)

lambda
    |- !g. (lambda) g = (@f. !i. 1 <= i /\ i <= dimindex (:N) ==> f$i = g i)
```

```
pastecart
     |- !f g.
            pastecart f g =
            (lambda i. if i <= dimindex (:M) then f$i else g$(i - dimindex (:M)))

sndcart
     |- !f. sndcart f = (lambda i. f$(i + dimindex (:M)))
```