Chapter 9

Examples

As we have said, ML was originally designed as a metalanguage for a computer theorem prover. However it is suitable for plenty of other applications, mostly from the same general field of 'symbolic computation'. In this chapter we will give some examples of characteristic uses of ML. It is not necessary that the reader should understand every detail of the particular examples, e.g. the analyses of real number approximations given below. However it is worth trying out these programs and similar examples for yourself, and doing some of the exercises. There is no better way of getting a feel for how ML is actually used.

9.1 Symbolic differentiation

The phrase 'symbolic computation' is a bit vague, but roughly speaking it covers applications where manipulation of mathematical *expressions*, in general containing variables, is emphasized at the expense of actual numerical calculation. There are several successful 'computer algebra systems' such as Axiom, Maple and Mathematica, which can do certain symbolic operations that are useful in mathematics, e.g. factorizing polynomials and differentiating and integrating expressions. (They are also capable of purely numerical work.) We will illustrate how this sort of application can be tackled using ML.

We use symbolic differentiation as our example, because there is an algorithm to do it which is fairly simple and well known. The reader is probably familiar with certain derivatives of basic operations, e.g $\frac{d}{dx} sin(x) = cos(x)$, as well as results such as the Chain Rule and Product Rule for calculating the derivatives of composite expressions in terms of the derivatives of the components. Just as one can use these systematically in order to find derivatives, it is possible to program a fairly straightforward algorithm to do it in ML.

9.1.1 First order terms

We will allow mathematical expressions in a fairly general form. They may be built up from variables and constants by the application of operators. These operators may be unary, binary, ternary, or in general n-ary. We represent this using the following recursive datatype:

```
#type term = Var of string
           | Const of string
           | Fn of string * (term list);;
Type term defined.
```

For example, the expression:

$$sin(x + y)/cos(x - exp(y)) - ln(1 + x)$$

is represented by:

```
Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                Fn("cos",[Fn("-",[Var "x"; Fn("exp",[Var "y"])])])]);
        Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
```

9.1.2 Printing

Reading and writing expressions in their raw form is rather unpleasant. This is a general problem in all symbolic computation systems, and typically the system's interface contains a *parser* and *prettyprinter* to allow respectively input and output in a readable form. A detailed discussion of parsing is postponed, since it is worth a section of its own, but we will now write a very simple printer for expressions, so that at least we can see what's going on. We want the printer to support ordinary human conventions, i.e.

- Variables and constants are just written as their names.
- Ordinary n-ary functions applied to arguments are written by juxtaposing the function and a bracketed list of arguments, e.g. $f(x_1, \ldots, x_n)$.
- Infix binary functions like + are written in between their arguments.
- Brackets are used where necessary for disambiguation.
- Infix operators have a notion of precedence to reduce the need for brackets.

First we declare a list of infixes, which is a list of pairs: each operator name together with its precedence.

```
#let infixes = ["+",10; "-",10; "*",20; "/",20];;
```

It is common to use lists in this way to represent finite partial functions, since it allows more flexibility. They are commonly called *association lists*, and are very common in functional programming.[1] In order to convert the list into a partial function we use `assoc`:

```
#let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
Toplevel input:
>let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
>        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

The compiler warns us that if the appropriate data is not found in the list, the function will fail. Now we can define a function to get the infix precedence of a function:

```
#let get_precedence s = assoc s infixes;;
get_precedence : string -> int = <fun>
#get_precedence "+";;
- : int = 10
#get_precedence "/";;
- : int = 20
#get_precedence "%";;
Uncaught exception: Match_failure ("", 6544, 6601)
```

Note that if we ever change this list of infixes, then any functions such as `get_precedence` that use it need to be redeclared. This is the main reason why many LISP programmers prefer dynamic binding. However we can make the set of infixes arbitrarily extensible by making it into a reference and dereferencing it on each application:

```
#let infixes = ref ["+",10; "-",10; "*",20; "/",20];;
infixes : (string * int) list ref =
  ref ["+", 10; "-", 10; "*", 20; "/", 20]
#let get_precedence s = assoc s (!infixes);;
get_precedence : string -> int = <fun>
#get_precedence "^";;
Uncaught exception: Match_failure ("", 6544, 6601)
#infixes := ("^",30)::(!infixes);;
- : unit = ()
#get_precedence "^";;
- : int = 30
```

[1] For more heavyweight applications, some alternative such as hash tables is much better, but for simple applications where the lists don't get too long, this kind of linear list is simple and adequate.

Note that by the ML evaluation rules, the dereferencing is only applied after the function `get_precedence` receives its argument, and hence results in the set of infixes at that time. We can also define a function to decide whether a function has any data associated with it. One way is simply to try the function `get_precedence` and see if it works:

```
#let is_infix s =
   try get_precedence s; true
   with _ -> false;;
```

An alternative is to use a general function `can` that finds out whether another function succeeds:

```
#let can f x = try f x; true with _ -> false;;
can : ('a -> 'b) -> 'a -> bool = <fun>
#let is_infix = can get_precedence;;
is_infix : string -> bool = <fun>
```

We will use the following functions that we have already considered:

```
#let hd(h::t) = h;;
#let tl(h::t) = t;;
#let rec length l = if l = [] then 0 else 1 + length(tl l);;
```

Without further ado, here is a function to convert a term into a string.

```
#let rec string_of_term prec =
   fun (Var s) -> s
     | (Const c) -> c
     | (Fn(f,args)) ->
         if length args = 2 & is_infix f then
           let prec' = get_precedence f in
           let s1 = string_of_term prec' (hd args)
           and s2 = string_of_term prec' (hd(tl args)) in
           let ss = s1^" "^f^" "^s2 in
           if prec' <= prec then "("^ss^")" else ss
         else
           f^"("^(string_of_terms args)^")"

 and string_of_terms t =
   match t with
       [] -> ""
     | [t] -> string_of_term 0 t
     | (h::t) -> (string_of_term 0 h)^","^(string_of_terms t);;
```

The first argument `prec` indicates the precedence level of the operator of which the currently considered expression is an immediate subterm. Now if the current expression has an infix binary operator at the top level, parentheses are needed if the precedence of this operator is lower, e.g. if we are considering the right hand subexpression of $x * (y + z)$. Actually we use parentheses even when they are equal, in order to make groupings clear. In the case of associative operators like + this doesn't matter, but we must distinguish $x - (y - z)$ and $(x - y) - z$. (A more sophisticated scheme would give each operator an associativity.) The second, mutually recursive, function `string_of_terms` is used to print a comma-separated list of terms, as they occur for non-infix function applications of the form $f(t_1, \ldots, t_n)$. Let us see this in action:

```
#let t =
  Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                  Fn("cos",[Fn("-",[Var "x";
                                    Fn("exp",[Var "y"])])])]);
          Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
t : term =
 Fn
  ("-",
   [Fn
     ("/",
      [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
       Fn ("cos", [Fn ("-", [Var "x"; Fn ("exp", [Var "y"])])])]);
    Fn ("ln", [Fn ("+", [Const "1"; Var "x"])])])
#string_of_term 0 t;;
- : string = "sin(x + y) / cos(x - exp(y)) - ln(1 + x)"
```

In fact, we don't need to convert a term to a string ourselves. CAML Light allows us to set up our own printer so that it is used to print anything of type `term` in the standard read-eval-print loop. This needs a few special commands to ensure that we interact properly with the toplevel loop:

```
##open "format";;
#let print_term s =
   open_hvbox 0;
   print_string("`"^(string_of_term 0 s)^"`");
   close_box();;
print_term : term -> unit = <fun>
#install_printer "print_term";;
- : unit = ()
```

Now compare the effect:

```
#let t =
    Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                    Fn("cos",[Fn("-",[Var "x";
                                      Fn("exp",[Var "y"])])])]);
            Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
t : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#let x = t
x : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
```

Once the new printer is installed, it is used whenever CAML wants to print something of type `term`, even if it is a composite datastructure such as a pair:

```
#(x,t);;
- : term * term =
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`,
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
```

or a list:

```
#[x; t; x];;
- : term list =
 [`sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`]
```

This printer is rather crude in that it will not break up large expressions intelligently across lines. The CAML library `format`, from which we used a few functions above, provides for a better approach. Instead of converting the term into a string then printing it in one piece, we can make separate printing calls for different parts of it, and intersperse these with some special function calls that help the printing mechanism to understand where to break the lines. Many of the principles used in this and similar 'prettyprinting engines' are due to Oppen (1980). We will not consider this in detail here — see the CAML manual for details.[2]

9.1.3 Derivatives

Now it is a fairly simple matter to define the derivative function. First let us recall the systematic method we are taught in school:

- If the expression is one of the standard functions applied to an argument that is the variable of differentiation, e.g. $sin(x)$, return the known derivative.
- If the expression is of the form $f(x) + g(x)$ then apply the rule for sums, returning $f'(x) + g'(x)$. Likewise for subtraction etc.
- If the expression is of the form $f(x) * g(x)$ then apply the product rule, i.e. return $f'(x) * g(x) + f(x) * g'(x)$.
- If the expression is one of the standard functions applied to a composite argument, say $f(g(x))$ then apply the Chain Rule and so give $g'(x) * f'(g(x))$

[2] There is also a tutorial, written by Pierre Weis, available online as part of the CAML page: '`http://pauillac.inria.fr/caml/FAQ/format-eng.html`'.

This is nothing but a recursive algorithm, though that terminology is seldom used in schools. We can program it in ML very directly:

```
#let rec differentiate x tm = match tm with
     Var y -> if y = x then Const "1" else Const "0"
   | Const c -> Const "0"
   | Fn("-",[t]) -> Fn("-",[differentiate x t])
   | Fn("+",[t1;t2]) -> Fn("+",[differentiate x t1;
                                differentiate x t2])
   | Fn("-",[t1;t2]) -> Fn("-",[differentiate x t1;
                                differentiate x t2])
   | Fn("*",[t1;t2]) ->
      Fn("+",[Fn("*",[differentiate x t1; t2]);
              Fn("*",[t1; differentiate x t2])])
   | Fn("inv",[t]) -> chain x t
      (Fn("-",[Fn("inv",[Fn("^",[t;Const "2"])])]))
   | Fn("^",[t;n]) -> chain x t
     (Fn("*",[n; Fn("^",[t; Fn("-",[n; Const "1"])])]))
   | Fn("exp",[t]) -> chain x t tm
   | Fn("ln",[t]) -> chain x t (Fn("inv",[t]))
   | Fn("sin",[t]) -> chain x t (Fn("cos",[t]))
   | Fn("cos",[t]) -> chain x t
      (Fn("-",[Fn("sin",[t])]))
   | Fn("/",[t1;t2]) -> differentiate x
      (Fn("*",[t1; Fn("inv",[t2])]))
   | Fn("tan",[t]) -> differentiate x
      (Fn("/",[Fn("sin",[t]); Fn("cos",[t])]))
 and chain x t u =  Fn("*",[differentiate x t; u]);;
```

The `chain` function just avoids repeating the chain rule construction for many operations. Apart from that, we just systematically apply rules for sums, products etc. and the known derivatives of standard functions. Of course, we could add other functions such as the hyperbolic functions and inverse trigonometric functions if desired. A couple of cases, namely division and tan, avoid a complicated definition by applying the differentiator to an alternative formulation.

9.1.4 Simplification

If we try the differentiator out on our running example, then it works quite well:

```
#t;;
- : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#differentiate "x" t;;
- : term =
 `(((1 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((1 - 0 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 1) * inv(1 + x)`
#differentiate "y" t;;
- : term =
 `(((0 + 1) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((0 - 1 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)`
#differentiate "z" t;;
- : term =
 `(((0 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((0 - 0 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)`
```

However, it fails to make various obvious simplifications such as $0 * x = 0$ and $x + 0 = x$. Some of these redundant expressions are created by the rather unsubtle differentiation strategy which, for instance, applies the chain rule to $f(t)$ even where t is just the variable of differentiation. However, some would be hard to avoid without making the differentiator much more complicated. Accordingly, let us define a separate simplification function that gets rid of some of these unnecessary expressions.

```
#let simp =
   fun (Fn("+",[Const "0"; t])) -> t
     | (Fn("+",[t; Const "0"])) -> t
     | (Fn("-",[t; Const "0"])) -> t
     | (Fn("-",[Const "0"; t])) -> Fn("-",[t])
     | (Fn("+",[t1; Fn("-",[t2])])) -> Fn("-",[t1; t2])
     | (Fn("*",[Const "0"; t])) -> Const "0"
     | (Fn("*",[t; Const "0"])) -> Const "0"
     | (Fn("*",[Const "1"; t])) -> t
     | (Fn("*",[t; Const "1"])) -> t
     | (Fn("*",[Fn("-",[t1]); Fn("-",[t2])])) -> Fn("*",[t1; t2])
     | (Fn("*",[Fn("-",[t1]); t2])) -> Fn("-",[Fn("*",[t1; t2])])
     | (Fn("*",[t1; Fn("-",[t2])])) -> Fn("-",[Fn("*",[t1; t2])])
     | (Fn("-",[Fn("-",[t])])) -> t
     | t -> t;;
```

This just applies the simplification at the top level of the term. We need to execute it in a bottom-up sweep. This will also bring negations upwards through products. In some cases it would be more efficient to simplify in a top-down sweep, e.g. $0 * t$ for a complicated expression t, but this would need repeating for terms like $(0 + 0) * 2$. The choice is rather like that between normal and applicative order reduction in lambda calculus.

```
#let rec dsimp =
   fun (Fn(fn,args)) -> simp(Fn(fn,map dsimp args))
     | t -> simp t;;
```

Now we get better results:

```
#dsimp(differentiate "x" t);;
- : term =
 `(cos(x + y) * inv(cos(x - exp(y))) +
  sin(x + y) * (sin(x - exp(y)) *
  inv(cos(x - exp(y)) ^ 2))) - inv(1 + x)`
#dsimp(differentiate "y" t);;
- : term =
 `cos(x + y) * inv(cos(x - exp(y))) -
  sin(x + y) * ((exp(y) * sin(x - exp(y))) *
  inv(cos(x - exp(y)) ^ 2))`
#dsimp(differentiate "z" t);;
- : term = `0`
```

In general, one can always add more and more sophisticated simplification. For example, consider:

```
#let t2 = Fn("tan",[Var "x"]);;
t2 : term = `tan(x)`
#differentiate "x" t2;;
- : term =
 `(1 * cos(x)) * inv(cos(x)) +
  sin(x) * ((1 * -(sin(x))) * -(inv(cos(x) ^ 2)))`
#dsimp(differentiate "x" t2);;
- : term = `cos(x) * inv(cos(x)) +
            sin(x) * (sin(x) * inv(cos(x) ^ 2))`
```

We would like to simplify `cos(x) * inv(cos(x))` simply to `1`, ignoring, as most commercial computer algebra systems do, the possibility that `cos(x)` might be zero. We can certainly add new clauses to the simplifier for that. Similarly, we might like to collect up terms in the second summand. However here we start to see that human psychology plays a role. Which of the following is preferable? It is hard for the machine to decide unaided.

```
sin(x) * (sin(x) * inv(cos(x) ^ 2))

sin(x) ^ 2 * inv(cos(x) ^ 2)

sin(x) ^ 2 / cos(x) ^ 2

(sin(x) / cos(x)) ^ 2

tan(x) ^ 2
```

And having chosen the last one, should it then convert

```
1 + tan(x) ^ 2
```

into

```
sec(x) ^ 2
```

Certainly, it is shorter and more conventional. However it relies on the fact that we know the definitions of these fairly obscure trig functions like `sec`. Whatever the machine decides to do, it is likely that some user will find it upsetting.

9.2 Parsing

A defect of our previous example was that the input had to be written explicitly in terms of type constructors. Here we will show how to write a parser for this situation, and make our code sufficiently general that it can easily be applied to similar formal languages. Generally speaking, the problem of parsing is as follows. A formal language is defined by a *grammar*, which is a set of *production rules* for each syntactic category. For a language of terms with two infix operators + and * and only alphanumeric variables and numeral (0, 1 etc.) constants, it might look like this:

$$
\begin{array}{rcl}
term & \longrightarrow & name\texttt{(}termlist\texttt{)} \\
 & | & name \\
 & | & \texttt{(}term\texttt{)} \\
 & | & numeral \\
 & | & \texttt{-}term \\
 & | & term\ \texttt{+}\ term \\
 & | & term\ \texttt{*}\ term \\
termlist & \longrightarrow & term\texttt{,}termlist \\
 & | & term
\end{array}
$$

We will assume for the moment that we know what names and numerals are, but we could have similar rules defining them in terms of single characters. Now this set of production rules gives us a way of generating the concrete, linear representation for all terms. Note that *name*, *numeral*, *term* and *termlist* are meant to be variables in the above, whereas elements like '+' and '(' written in bold type are actual characters. For example we have:

$$
\begin{array}{rcl}
term & \longrightarrow & term \texttt{ * } term \\
& \longrightarrow & term \texttt{ * } \texttt{(}term\texttt{)} \\
& \longrightarrow & term \texttt{ * } \texttt{(}term \texttt{ + } term\texttt{)} \\
& \longrightarrow & term \texttt{ * } \texttt{(}term \texttt{ + } term \texttt{ * } term\texttt{)} \\
& \longrightarrow & numeral \texttt{ * } \texttt{(}name \texttt{ + } name \texttt{ * } name\texttt{)}
\end{array}
$$

so, following similar rules for *name* and *numeral*, we can generate for example:

```
10 * (x + y * z)
```

The task of *parsing* is to reverse this process of applying production rules, i.e. to take a string and discover how it could have been generated by the production rules for a syntactic category. Typically, the output is a *parse tree*, i.e. a tree structure that shows clearly the sequence of applications of the rules.

One problem with parsing is that the grammar may be *ambiguous*, i.e. a given string can often be generated in several different ways. This is the case in our example above, for the string could also have been generated by:

$$
\begin{array}{rcl}
term & \longrightarrow & term \texttt{ * } term \\
& \longrightarrow & term \texttt{ * } \texttt{(}term\texttt{)} \\
& \longrightarrow & term \texttt{ * } \texttt{(}term \texttt{ * } term\texttt{)} \\
& \longrightarrow & term \texttt{ * } \texttt{(}term \texttt{ + } term \texttt{ * } term\texttt{)} \\
& \longrightarrow & numeral \texttt{ * } \texttt{(}name \texttt{ + } name \texttt{ * } name\texttt{)}
\end{array}
$$

The obvious way to fix this is to specify rules of precedence and associativity for infix operators. However, we can make the grammar unambiguous without introducing additional mechanisms at the cost of adding new syntactic categories:

$$
\begin{array}{rcl}
atom & \longrightarrow & name\texttt{(}termlist\texttt{)} \\
& | & name \\
& | & numeral \\
& | & \texttt{(}term\texttt{)} \\
& | & \texttt{-}atom \\
mulexp & \longrightarrow & atom \texttt{ * } mulexp \\
& | & atom \\
term & \longrightarrow & mulexp \texttt{ + } term \\
& | & mulexp \\
termlist & \longrightarrow & term\texttt{,}termlist \\
& | & term
\end{array}
$$

Note that this makes both infix operators right associative. For a more complicated example of this technique in action, look at the formal definition of expressions in the ANSI C Standard.

9.2.1 Recursive descent parsing

The production rules suggest a very simple way of going about parsing using a series of mutually recursive functions. The idea is to have a function for each syntactic category, and a recursive structure that reflects exactly the mutual recursion found in the grammar itself. For example, the procedure for parsing terms, say `term` will, on encountering a `-` symbol, make a recursive call to itself to parse the subterm, and on encountering a name followed by an opening parenthesis, will make a recursive call to `termlist`. This in itself will make at least one recursive call to `term`, and so on. Such a style of programming is particularly natural in a language like ML where recursion is the principal control mechanism.

In our ML implementation, we suppose that we have some input list of objects of arbitrary type α for the parser to consume. These might simply be characters, but in fact there is usually a separate level of *lexical analysis* which collects characters together into *tokens* such as '`x12`', '`:=`' and '`96`', so by the time we reach this level the input is a list of tokens. We avoid committing ourselves to any particular type. Similarly, we will allow the parser to construct objects of arbitrary type β from its input. These might for example be parse trees as members of a recursive datatype, or might simply be numbers if the parser is to take an expression and evaluate it. Now in general, a parser might not consume all its input, so we also make it output the remaining tokens. Therefore, a parser will have type

$$(\alpha)list \to \beta \times (\alpha)list$$

For example, when given the input characters `(x + y) * z` the function `atom` should process the characters `(x + y)` and leave the remaining characters `* z`. It might return a parse tree for the processed expression using our earlier recursive type, and hence we would have:

```
atom "(x + y) * z" = Fn("+",[Var "x"; Var "y"]),"* z"
```

Since any call of the function `atom` must be from within the function `mulexp`, this second function will use the result of `atom` and deal with the input that remains, making a second call to `atom` to process the expression '`z`'.

9.2.2 Parser combinators

Another reason why this technique works particularly well in ML is that we can define some useful combinators for plugging parsers together. In fact, by giving them infix status, we can make the ML parser program look quite similar in structure to the original grammar defining the language.

First we declare an exception to be used where parsing fails. Then we define an infix `++` that applies two parsers in sequence, pairing up their results, and and infix `||` which tries first one parser, then the other. We then define a many-fold version of `++` that repeats a parser as far as possible, putting the results into a list. Finally, the infix `>>` is used to modify the results of a parser according to a function.

The CAML rules for symbolic identifiers such as `++` make them infix automatically, so we suppress this during the definition using the `prefix` keyword. Their precedences are also automatic, following from the usual precedence of their first characters as arithmetic operations etc. That is, `++` is the strongest, `>>` falls in the middle, and `||` is the weakest; this is exactly what we want.

```
exception Noparse;;

let prefix || parser1 parser2 input =
  try parser1 input
  with Noparse -> parser2 input;;

let prefix ++ parser1 parser2 input =
  let result1,rest1 = parser1 input in
  let result2,rest2 = parser2 rest1 in
  (result1,result2),rest2;;

let rec many parser input =
  try let result,next = parser input in
      let results,rest = many parser next in
      (result::results),rest
  with Noparse -> [],input;;

let prefix >> parser treatment input =
  let result,rest = parser input in
  treatment(result),rest;;
```

We will, in what follows, use the following other functions. Most of these have been defined above, the main exception being `explode` which converts a string into a list of 1-elements strings. It uses the inbuilt functions `sub_string` and `string_length`; we do not describe them in detail but their function should be easy to grasp from the example.

```
let rec itlist f =
     fun [] b -> b
       | (h::t) b -> f h (itlist f t b);;

let uncurry f(x,y) = f x y;;

let K x y = x;;

let C f x y = f y x;;

let o f g x = f(g x);;
#infix "o";;

let explode s =
  let rec exap n l =
      if n < 0 then l else
      exap (n - 1) ((sub_string s n 1)::l) in
  exap (string_length s - 1) [];;
```

In order to get started, we define a few 'atomic' parsers. The function `some` accepts any item satisfying a predicate and returns it. The function `a` is similar, but demands a particular item, and finally `finished` just makes sure there are no items left to be parsed.

```
let some p =
  fun [] -> raise Noparse
    | (h::t) -> if p h then (h,t) else raise Noparse;;

let a tok = some (fun item -> item = tok);;

let finished input =
  if input = [] then 0,input else raise Noparse;;
```

9.2.3 Lexical analysis

Our parsing combinators can easily be used, in conjunction with a few simple character discrimination functions, to construct a lexical analyzer for our language of terms. We define a type of *tokens* (or lexemes), and the lexer translates the input string into a list of tokens. The 'other' cases cover symbolic identifiers; they are all straightforward as they consist of just a single character, rather than composite ones like '`:=`'.

```
type token = Name of string | Num of string | Other of string;;

let lex =
  let several p = many (some p) in
  let lowercase_letter s = "a" <= s & s <= "z" in
  let uppercase_letter s = "A" <= s & s <= "Z" in
  let letter s = lowercase_letter s or uppercase_letter s in
  let alpha s = letter s or s = "_" or s = "'" in
  let digit s = "0" <= s & s <= "9" in
  let alphanum s = alpha s or digit s in
  let space s = s = " " or s = "\n" or s = "\t" in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let rawname =
     some alpha ++ several alphanum >> (Name o collect) in
  let rawnumeral =
     some digit ++ several digit >> (Num o collect) in
  let rawother = some (K true) >> Other in
  let token =
    (rawname || rawnumeral || rawother) ++ several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;
```

For example:

```
#lex "sin(x + y) * cos(2 * x + y)";;
 - : token list =
 [Name "sin"; Other "("; Name "x"; Other "+"; Name "y"; Other ")";
  Other "*"; Name "cos"; Other "("; Num "2"; Other "*"; Name "x";
  Other "+"; Name "y"; Other ")"]
```

9.2.4 Parsing terms

Since we are now working at the level of tokens, we define trivial parsers to accept tokens of (only) a certain kind:

```
let name =
  fun (Name s::rest) -> s,rest
    | _ -> raise Noparse;;

let numeral =
  fun (Num s::rest) -> s,rest
    | _ -> raise Noparse;;

let other =
  fun (Other s::rest) -> s,rest
    | _ -> raise Noparse;;
```

Now we can define a parser for terms, in a form very similar to the original grammar. The main difference is that each production rule has associated with it some sort of special action to take as a result of parsing.

```
let rec atom input
      = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || name
             >> (fun s -> Var s)
      || numeral
             >> (fun s -> Const s)
      || a (Other "(") ++ term ++ a (Other ")")
             >> (snd o fst)
      || a (Other "-") ++ atom
             >> snd) input
  and mulexp input
      = (atom ++ a(Other "*") ++ mulexp
             >> (fun ((a,_),m) -> Fn("*",[a;m]))
      || atom) input
  and term input
      = (mulexp ++ a(Other "+") ++ term
             >> (fun ((a,_),m) -> Fn("+",[a;m]))
      || mulexp) input
  and termlist input
      = (term ++ a (Other ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input;;
```

Let us package everything up as a single parsing function:

```
let parser = fst o (term ++ finished >> fst) o lex;;
```

To see it in action, we try with and without the printer (see above) installed:

```
#parser "sin(x + y) * cos(2 * x + y)";;
- : term =
 Fn
  ("*",
   [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
    Fn ("cos", [Fn ("+", [Fn ("*", [Const "2"; Var "x"]);
                          Var "y"])])])
#install_printer "print_term";;
- : unit = ()
#parser "sin(x + y) * cos(2 * x + y)";;
- : term = `sin(x + y) * cos(2 * x + y)`
```

9.2.5 Automatic precedence parsing

In the above parser, we 'hardwired' a parser for the two infix operators. However it would be nicer, and consistent with our approach to printing, if changes to the set of infixes could propagate to the parser. Moreover, even with just two different binary operators, our production rules and consequent parser were already starting to look artificial; imagine if we had a dozen or so. What we would like is to automate the production of a layer of parsers, one for each operator, in order of precedence. This can easily be done. First, we define a general function that takes a parser for what, at this level, are regarded as atomic expressions, and produces a new parser for such expressions separated by a given operator. Note that this could also be defined using parsing combinators, but the function below is simple and more efficient.

```
let rec binop op parser input =
  let atom1,rest1 as result = parser input in
  if not rest1 = [] & hd rest1 = Other op then
    let atom2,rest2 = binop op parser (tl rest1) in
    Fn(op,[atom1; atom2]),rest2
  else result;;
```

Now we define functions that pick out the infix operator in our association list with (equal) lowest precedence, and delete it from the list. If we maintained the list already sorted in precedence order, we could simply take the head and tail of the list.

```
let findmin l = itlist
  (fun (_,pr1 as p1) (_,pr2 as p2) -> if pr1 <= pr2 then p1 else p2)
  (tl l) (hd l);;

let rec delete x (h::t) = if h = x then t else h::(delete x t);;
```

Now we can define the generic precedence parser:

```
let rec precedence ilist parser input =
  if ilist = [] then parser input else
  let opp = findmin ilist in
  let ilist' = delete opp ilist in
  binop (fst opp) (precedence ilist' parser) input;;
```

and hence can modify the main parser to be both simpler and more general:

```
let rec atom input
      = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || name
             >> (fun s -> Var s)
      || numeral
             >> (fun s -> Const s)
      || a (Other "(") ++ term ++ a (Other ")")
             >> (snd o fst)
      || a (Other "-") ++ atom
             >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
      = (term ++ a (Other ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input;;

let parser = fst o (term ++ finished >> fst) o lex;;
```

for example:

```
#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term =
 Fn
  ("+",
   [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "x"]);
                                   Const "2"])]);
    Fn
     ("-",
      [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "y"]);
                                      Const "2"])]);
       Const "2"])])
#install_printer "print_term";;
- : unit = ()
#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term = `2 * sin(x) ^ 2 + (2 * sin(y) ^ 2 - 2)`
```

9.2.6 Defects of our approach

Our approach to lexing and parsing is not particularly efficient. In fact, CAML and some other ML implementations include a generator for LR parsers similar to YACC (Yet Another Compiler Compiler) which is often used to generate parsers in C under Unix. Not only are these more general in the grammars that they can handle, but the resulting parsers are more efficient. However we believe the present approach is rather clear, adequate for most purposes, and a good introduction to programming with higher order functions.

The inefficiency of our approach can be reduced by avoiding a few particularly wasteful features. Note that if two different productions for the same syntactic category have a common prefix, then we should avoid parsing it more than once, unless it is guaranteed to be trivial. Our production rules for *term* have this property:

$$\begin{array}{rcl} term & \longrightarrow & name(termlist) \\ & | & name \\ & | & \cdots \end{array}$$

We carefully put the longer production first in our actual implementation, otherwise success in reading a name would cause the abandonment of attempts to read a parenthesized list of arguments. Nevertheless, because of the redundancy we have mentioned, this way of constructing the parser is wasteful. In this case it doesn't matter much, because the duplicated call only analyzes one token. However the case for *termlist* is more significant:

```
  let ...
  and termlist input
       = (term ++ a (Other ",") ++ termlist
              >> (fun ((h,_),t) -> h::t)
       || term
              >> (fun h -> [h])) input;;
```

Here we could reparse a whole term, which might be arbitrarily large. We can avoid this in several ways. One is to move the `||` alternation to after the initial term has been read. A convenient way to code it is to eschew explicit recursion in favour of `many`:

```
  let ...
  and termlist input
      = term ++ many (a (Other ",") ++ term >> snd)
              >> (fun (h,t) -> h::t) input;;
```

Therefore the final version of the parser is:

```
let rec atom input
       = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
              >> (fun (((name,_),args),_) -> Fn(name,args))
       || name
              >> (fun s -> Var s)
       || numeral
              >> (fun s -> Const s)
       || a (Other "(") ++ term ++ a (Other ")")
              >> (snd o fst)
       || a (Other "-") ++ atom
              >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
        = (term ++ many (a (Other ",") ++ term >> snd)
              >> (fun (h,t) -> h::t)) input;;
```

A general defect of our approach, and recursive descent parsing generally, is that so-called *left recursion* in productions causes problems. This is where a given category expands to a sequence including the category itself as the first item. For example, if we had wanted to make the addition operator left-associative in our earlier grammar, we could have used:

$$\begin{array}{rcl} \mathit{term} & \longrightarrow & \mathit{term}\ \texttt{+}\ \mathit{mulexp} \\ & | & \mathit{mulexp} \end{array}$$

The naive transcription into ML would loop indefinitely, calling `term` on the same subterm over and over. There are various slightly ad hoc ways of dealing with this problem. For example, one can convert this use of recursion, which in some ways is rather gratuitous, into the explicit use of repetition. We can do this using standard combinators like `many` to return a list of '*mulexp*'s and then run over the list building the tree left-associated.

A final problem is that we do not handle errors very gracefully. We always raise the exception `Noparse`, and this, when caught, initiates backtracking. However this is not always appropriate for typical grammars. At best, it can cause expensive reparsing, and can lead to baffling error indications at the top level. For example, on encountering `5 +`, we expect a term following the `+`, and if we don't find one, we should generate a comprehensible exception for the user, rather than raise `Noparse` and perhaps initiate several other pointless attempts to parse the expression in a different way.

9.3 Exact real arithmetic

Real arithmetic on computers is normally done via floating point approximations. In general, we can only manipulate a real number, either ourselves or inside a computer, via some sort of finite representation. Some question how numbers can be said to 'exist' if they have no finite representation. For example, Kronecker accepted integers and rationals because they can be written down explicitly, and even *algebraic* numbers[3] because they can be represented using the polynomials of which they are solutions. However he rejected transcendental numbers because apparently they could not be represented finitely. Allegedly he he greeted the famous proof by Lindemann (1882) that π is transcendental with the remark that this was 'interesting, except that π does not exist'.

However, given our modern perspective, we can say that after all many more numbers than Kronecker would have accepted *do* have a finite representation, namely the program, or in some more abstract sense the *rule*, used to calculate

[3]Algebraic numbers are those that are roots of polynomials with integer coefficients, e.g. $\sqrt{2}$, and transcendental numbers are ones that aren't.

them to greater and greater precision. For example, we can write a program that will produce for any argument n the first n digits of π. Alternatively it can produce a rational number r such that $|\pi - r| < 2^{-n}$. Whatever approach is taken to the successive approximation of a real number, the key point is that its representation, the program itself, is finite.

This works particularly nicely in a language like ML where higher order functions are available. What we have called a 'program' above will just be an ML function. We can actually represent the arithmetic operations on numbers as higher order functions that, given functions for approximating x and y, will produce new ones for approximating $x + y$, xy, $sin(x)$ and so on, for a wide range of functions. In an ordinary programming language, we would need to define a concrete representation for programs, e.g. 'Gödel numbering', and write an interpreter for this representation.[4]

9.3.1 Representation of real numbers

Our approach is to represent a real x by a function $f_x : \mathbb{N} \to \mathbb{Z}$ that for each $n \in \mathbb{N}$ returns an approximation of x to within 2^{-n}, appropriately scaled. In fact, we have:

$$|f_x(n) - 2^n x| < 1$$

This is of course equivalent to $|\frac{f_x(n)}{2^n} - x| < \frac{1}{2^n}$. We could return a rational approximation directly, but it would still be convenient for the rationals to have denominators that are all powers of some number, so that common denominators arising during addition don't get too big. Performing the scaling directly seems simpler, requiring only integer arithmetic. Our choice of the base 2 is largely arbitrary. A lower base is normally advantageous because it minimizes the *granularity* of the achievable accuracies. For example, if we used base 10, then even if we only need to increase the accuracy of an approximation slightly, we have to step up the requested accuracy by a factor of 10.

9.3.2 Arbitrary-precision integers

CAML's standard integers (type `int`) have a severely limited range, so first of all we need to set up a type of unlimited-precision integers. This is not so difficult to program, but is slightly tedious. Fortunately, the CAML release includes a library that gives a fast implementation of arbitrary precision integer (and in fact rational) arithmetic. A version of CAML Light with this library pre-loaded is installed on Thor. Assuming you have used the following path setting:

[4]Effectively, we are manipulating functions 'in extension' rather than 'in intension', i.e. not concerning ourselves with their representation.

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

then the augmented version of CAML Light can be fired up using:

```
$ camllight my_little_caml
```

When the system returns its prompt, use the `#open` directive to make the library functions available:

```
##open "num";;
```

This library sets up a new type `num` of arbitrary precision rational numbers; we will just use the integer subset. CAML does not provide overloading, so it is necessary to use different symbols for the usual arithmetic operations over `num`.

Note that small constants of type `num` must be written as `Int k` rather than simply `k`. In fact `Int` is a type constructor for `num` used in the case when the number is a machine-representable integer. Larger ones use `Big_int`.

The unary negation on type `num` is written `minus_num`. Another handy unary operator is `abs_num`, which finds absolute values, i.e. given x returns $|x|$.

The usual binary operations are also available. The (truncating) division and modulus function, called `quo_num` and `mod_num`, do not have infix status. However most of the other binary operators are infixes, and the names are derived from the usual ones by adding a slash '/'. It is important to realize that in general, one must use `=/` to compare numbers for equality. This is because the constructors of the type `num` are, as always, distinct by definition, yet in terms of the underlying meaning they may overlap. For example we get different result from using truncating division and true division on 2^{30} and 2, even though they are numerically the same. One uses type constructor `Int` and the other `Ratio`.

```
#(Int 2 **/ Int 30) // Int 2 = quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = false
#(Int 2 **/ Int 30) // Int 2 =/ quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = true
```

Here is a full list of the main infix binary operators:

Operator	Type	Meaning
`**/`	`num -> num -> num`	Exponentiation
`*/`	`num -> num -> num`	Multiplication
`+/`	`num -> num -> num`	Addition
`-/`	`num -> num -> num`	Subtraction
`=/`	`num -> num -> bool`	Equality
`<>/`	`num -> num -> bool`	Inequality
`</`	`num -> num -> bool`	Less than
`<=/`	`num -> num -> bool`	Less than or equal
`>/`	`num -> num -> bool`	Greater than
`>=/`	`num -> num -> bool`	Greater than or equal

Let us see some further examples:

```
#Int 5 */ Int 14;;
it : num = Int 70
#Int 2 **/ Int 30;;
it : num = Big_int <abstr>
#(Int 2 **/ Int 30) // Int 2;;
it : num = Ratio <abstr>
#quo_num (Int 2 **/ Int 30) (Int 2);;
it : num = Int 536870912
```

Note that large numbers are not printed. However we can always convert one to a string using `string_of_num`:

```
#string_of_num(Int 2 ** Int 150);;
- : string = "1427247692705959881058285969449495136382746624"
```

This also has an inverse, called naturally enough `num_of_string`.

9.3.3 Basic operations

Recall that our real numbers are supposed to be (represented by) functions $\mathbb{Z} \to \mathbb{Z}$. In ML we will actually use `int -> num`, since the inbuilt type of integers is more than adequate for indexing the level of accuracy. Now we can define some operations on reals. The most basic operation, which gets us started, is to produce the real number corresponding to an integer. This is easy:

```
#let real_of_int k n = (Int 2 **/ Int n) */ Int k;;
real_of_int : int -> int -> num = <fun>
#real_of_int 23;;
- : int -> num = <fun>
```

It is obvious that for any k this obeys the desired approximation criterion:

$$|f_k(n) - 2^n k| = |2^n k - 2^n k| = 0 < 1$$

Now we can define the first nontrivial operation, that of unary negation:

```
let real_neg f n = minus_num(f n);;
```

The compiler generalizes the type more than intended, but this will not trouble us. It is almost as easy to see that the approximation criterion is preserved. If we know that for each n:

$$|f_x(n) - 2^n x| < 1$$

then we have for any n:

$$\begin{aligned}
|f_{-x}(n) - 2^n(-x)| &= |-f_x(n) - 2^n(-x)| \\
&= |-(f_x(n) - 2^n x)| \\
&= |f_x(n) - 2^n x| \\
&< 1
\end{aligned}$$

Similarly, we can define an 'absolute value' function on real numbers, using the corresponding function `abs_num` on numbers:

```
let real_abs f n = abs_num (f n);;
```

The correctness of this is again straightforward, using the fact that $||x|-|y|| \leq |x-y|$. Now consider the problem of adding two real numbers. Suppose x and y are represented by f_x and f_y respectively. We could define:

$$f_{x+y}(n) = f_x(n) + f_y(n)$$

However this gives no guarantee that the approximation criterion is maintained; we would have:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |f_x(n) + f_y(n) - 2^n(x+y)| \\
&\leq |f_x(n) - 2^n x| + |f_y(n) - 2^n y|
\end{aligned}$$

We can guarantee that the sum on the right is less than 2, but not that it is less than 1 as required. Therefore, we need in this case to evaluate x and y to *greater* accuracy than required in the answer. Suppose we define:

$$f_{x+y}(n) = (f_x(n+1) + f_y(n+1))/2$$

Now we have:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |(f_x(n+1) + f_y(n+1))/2 - 2^n(x+y)| \\
&\leq |f_x(n+1)/2 - 2^n x| + |f_y(n+1)/2 - 2^n y| \\
&= \frac{1}{2}|f_x(n+1) - 2^{n+1}x| + \frac{1}{2}|f_y(n+1) - 2^{n+1}y| \\
&< \frac{1}{2}1 + \frac{1}{2}1 = 1
\end{aligned}$$

Apparently this just gives the accuracy required. However we have implicitly used real mathematical division above. Since the function is supposed to yield an integer, we are obliged to round the quotient to an integer. If we just use `quo_num`, the error from rounding this might be almost 1, after which we could never guarantee the bound we want, however accurately we evaluate the arguments. However with a bit more care we can define a division function that always returns the integer closest to the true result (or one of them in the case of two equally close ones), so that the rounding error never exceeds $\frac{1}{2}$. This could be done directly in integer arithmetic, but the most straightforward coding is to use rational division followed by rounding to the nearest integer, as there are built-in functions for both these operations:

```
#let ndiv x y = round_num(x // y);;
ndiv : num -> num -> num = <fun>
##infix "ndiv";;
#(Int 23) ndiv (Int 5);;
- : num = Int 5
#(Int 22) ndiv (Int 5);;
- : num = Int 4
#(Int(-11)) ndiv (Int 4);;
- : num = Int -3
#(Int(-9)) ndiv (Int 4);;
- : num = Int -2
```

Now if we define:

$$f_{x+y}(n) = (f_x(n+2) + f_y(n+2)) \text{ ndiv } 4$$

everything works:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |((f_x(n+2) + f_y(n+2)) \text{ ndiv } 4) - 2^n(x+y)| \\
&\leq \frac{1}{2} + |(f_x(n+2) + f_y(n+2))/4 - 2^n(x+y)| \\
&= \frac{1}{2} + \frac{1}{4}|(f_x(n+2) + f_y(n+2)) - 2^{n+2}(x+y)| \\
&\leq \frac{1}{2} + \frac{1}{4}|f_x(n+2) - 2^{n+2}x| + \frac{1}{4}|f_y(n+2) - 2^{n+2}y| \\
&< \frac{1}{2} + \frac{1}{4}1 + \frac{1}{4}1 \\
&= 1
\end{aligned}$$

Accordingly we make our definition:

```
let real_add f g n =
  (f(n + 2) +/ g(n + 2)) ndiv (Int 4);;
```

We can define subtraction similarly, but the simplest approach is to build it out of the functions that we already have:

```
#let real_sub f g = real_add f (real_neg g);;
real_sub : (num -> num) -> (num -> num) -> num -> num = <fun>
```

It is a bit trickier to define multiplication, inverses and division. However the cases where we multiply or divide by an integer are much easier and quite common. It is worthwhile implementing these separately as they can be made more efficient. We define:

$$f_{mx}(n) = (m f_x(n+p+1)) \text{ ndiv } 2^{p+1}$$

where p is chosen so that $2^p \geq |m|$. For correctness, we have:

$$\begin{aligned}
|f_{mx}(n) - 2^n(mx)| &\leq \frac{1}{2} + |\frac{m f_x(n+p+1)}{2^{p+1}} - 2^n(mx)| \\
&= \frac{1}{2} + \frac{|m|}{2^{p+1}}|f_x(n+p+1) - 2^{n+p+1}x| \\
&< \frac{1}{2} + \frac{|m|}{2^{p+1}} \\
&\leq \frac{1}{2} + \frac{1}{2}\frac{|m|}{2^p} \\
&\leq \frac{1}{2} + \frac{1}{2} = 1
\end{aligned}$$

In order to implement this, we need a function to find the appropriate p. The following is crude but adequate:

```
let log2 =
  let rec log2 x y =
    if x </ Int 1 then y
    else log2 (quo_num x (Int 2)) (y + 1) in
  fun x -> log2 (x -/ Int 1) 0;;
```

The implementation is simply:

```
let real_intmul m x n =
   let p = log2 (abs_num m) in
   let p1 = p + 1 in
   (m */ x(n + p1)) ndiv (Int 2 **/ Int p1);;
```

For division by an integer, we define:

$$f_{x/m}(n) = f_x(n) \text{ ndiv } m$$

For correctness, we can ignore the trivial cases when $m = 0$, which should never be used, and when $m = \pm 1$, since then the result is exact. Otherwise, we assume $|f_x(n) - 2^n x| < 1$, so $|f_x(n)/m - 2^n x/m| < \frac{1}{|m|} \leq \frac{1}{2}$, which, together with the fact that $|f_x(n) \text{ ndiv } m - f_x(n)/m| \leq \frac{1}{2}$, yields the result. So we have:

```
let real_intdiv m x n =
   x(n) ndiv (Int m);;
```

9.3.4 General multiplication

General multiplication is harder because the error in approximation for one number is multiplied by the magnitude of the second number. Therefore, before the final accuracies are calculated, a preliminary evaluation of the arguments is required to determine their approximate magnitude. We proceed as follows. Suppose that we want to evaluate $x + y$ to precision n. First we choose r and s so that $|r - s| \leq 1$ and $r + s = n + 2$. That is, both r and s are slightly more than half the required precision. We now evaluate $f_x(r)$ and $f_y(s)$, and select natural numbers p and q that are the corresponding 'binary logarithms', i.e. $|f_x(r)| \leq 2^p$ and $|f_y(s)| \leq 2^q$. If both p and q are zero, then it is easy to see that we may return just 0. Otherwise, remember that either $p > 0$ or $q > 0$ as we will need this later. Now set:

$$\begin{aligned} k &= n + q - s + 3 = q + r + 1 \\ l &= n + p - r + 3 = p + s + 1 \\ m &= (k + l) - n = p + q + 4 \end{aligned}$$

We claim that $f_{xy}(n) = (f_x(k)f_y(l))$ ndiv 2^m has the right error behaviour, i.e. $|f_{xy}(n) - 2^n(xy)| < 1$. If we write:

$$\begin{aligned} 2^k x &= f_x(k) + \delta \\ 2^l y &= f_y(l) + \epsilon \end{aligned}$$

with $|\delta| < 1$ and $|\epsilon| < 1$, we have:

$$\begin{aligned} |f_{xy}(n) - 2^n(xy)| &\leq \frac{1}{2} + |\frac{f_x(k)f_y(l)}{2^m} - 2^n(xy)| \\ &= \frac{1}{2} + 2^{-m}|f_x(k)f_y(l) - 2^{k+l}xy| \\ &= \frac{1}{2} + 2^{-m}|f_x(k)f_y(l) - (f_x(k) + \delta)(f_y(l) + \epsilon)| \\ &= \frac{1}{2} + 2^{-m}|\delta f_y(l) + \epsilon f_x(k) + \delta\epsilon| \\ &\leq \frac{1}{2} + 2^{-m}(|\delta f_y(l)| + |\epsilon f_x(k)| + |\delta\epsilon|) \\ &\leq \frac{1}{2} + 2^{-m}(|f_y(l)| + |f_x(k)| + |\delta\epsilon|) \\ &< \frac{1}{2} + 2^{-m}(|f_y(l)| + |f_x(k)| + 1) \end{aligned}$$

Now we have $|f_x(r)| \leq 2^p$, so $|2^r x| < 2^p + 1$. Thus $|2^k x| < 2^{q+1}(2^p + 1)$, so $|f_x(k)| < 2^{q+1}(2^p + 1) + 1$, i.e. $|f_x(k)| \leq 2^{q+1}(2^p + 1)$. Similarly $|f_y(l)| \leq 2^{p+1}(2^q + 1)$. Consequently:

$$\begin{aligned}|f_y(l)| + |f_x(k)| + 1 &\leq 2^{p+1}(2^q + 1) + 2^{q+1}(2^p + 1) + 1 \\ &= 2^{p+q+1} + 2^{p+1} + 2^{p+q+1} + 2^{q+1} + 1 \\ &= 2^{p+q+2} + 2^{p+1} + 2^{q+1} + 1\end{aligned}$$

Now for our error bound we require $|f_y(l)| + |f_x(k)| + 1 \leq 2^{m-1}$, or dividing by 2 and using the discreteness of the integers:

$$2^{p+q+1} + 2^p + 2^q < 2^{p+q+2}$$

We can write this as $(2^{p+q} + 2^p) + (2^{p+q} + 2^q) < 2^{p+q+1} + 2^{p+q+1}$, which is true because we have either $p > 0$ or $q > 0$. So at last we are justified in defining:

```
let real_mul x y n =
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m);;
```

9.3.5 Multiplicative inverse

Next we will define the multiplicative inverse function. In order to get any sort of upper bound on this, let alone a good approximation, we need to get a *lower* bound for the argument. In general, there is no better way than to keep evaluating the argument with greater and greater accuracy until we can bound it away from zero. We will use the following lemma to justify the correctness of our procedure:

Lemma 9.1 *If $2e \geq n + k + 1$, $|f_x(k)| \geq 2^e$ and $|f_x(k) - 2^k x| < 1$, where $f_x(k)$ is an integer and e, n and k are natural numbers, then if we define*

$$f_y(n) = 2^{n+k} \ ndiv \ f_x(k)$$

we have $|f_y(n) - 2^n x^{-1}| < 1$, i.e. the required bound.

Proof: *The proof is rather tedious and will not be given in full. We just sketch the necessary case splits. If $|f_x(k)| > 2^e$ then a straightforward analysis gives the result; the rounding in* `ndiv` *gives an error of at most $\frac{1}{2}$, and the remaining error is $< \frac{1}{2}$. If $|f_x(k)| = 2^e$ but $n + k \geq e$, then although the second component of the* *error may now be twice as much, i.e. < 1, there is no rounding error because $f_x(k) = \pm 2^e$ divides into 2^{n+k} exactly. (We use here the fact that $2^e - 1 \leq 2^{e-1}$, because since $2e \geq n + k + 1$, e cannot be zero.) Finally, if $|f_x(k)| = 2^e$ and $n + k < e$, we have $|f_y(n) - 2^n \frac{1}{x}| < 1$ because both $|f_y(n)| \leq 1$ and $0 < |2^n \frac{1}{x}| < 1$, and both these numbers have the same sign. Q.E.D.*

Now suppose we wish to find the inverse of x to accuracy n. First we evaluate $f_x(0)$. There are two cases to distinguish:

1. If $|f_x(0)| > 2^r$ for some natural number r, then choose the least natural number k (which may well be zero) such that $2r + k \geq n + 1$, and set $e = r + k$. We now return 2^{n+k} ndiv $f_x(k)$. It is easy to see that the conditions of the lemma are satisfied. Since $|f_x(0)| \geq 2^r + 1$ we have $|x| > 2^r$, and so $|2^k x| > 2^{r+k}$. This means $|f_x(k)| > 2^{r+k} - 1$, and as $f_x(k)$ is an integer, $|f_x(k)| \geq 2^{r+k} = 2^e$ as required. The condition that $2e \geq n = k + 1$ is easy to check. Note that if $r \geq n$ we can immediately deduce that $f_y(n) = 0$ is a valid approximation.

2. If $|f_x(0)| \leq 1$, then we call the function '`msd`' that returns the least p such that $|f_x(p)| > 1$. Note that this may in general fail to terminate if $x = 0$. Now we set $e = n + p + 1$ and $k = e + p$, and return 2^{n+k} ndiv $f_x(k)$. Once again the conditions for the lemma are satisfied. Since $|f_x(p)| \geq 2$, we have $|2^p x| > 1$, i.e. $|x| > \frac{1}{2^p}$. Hence $|2^k x| > 2^{k-p} = 2^e$, and so $|f_x(k)| > 2^e - 1$, i.e. $|f_x(k)| \geq 2^e$.

To implement this, we first define the `msd` function:

```
let msd =
  let rec msd n x =
    if abs_num(x(n)) >/ Int 1 then n else msd (n + 1) x in
  msd 0;;
```

and then translate the above mathematics into a simple program:

```
let real_inv x n =
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k));;
```

Now, of course, it is straightforward to define division:

```
let real_div x y = real_mul x (real_inv y);;
```

9.3.6 Ordering relations

The interesting things about all these functions is that they are in fact uncomputable in general. The essential point is that it is impossible to decide whether a given number is zero. We can keep approximating it to greater and greater accuracy, but if the approximations always return 0, we cannot tell whether it is going to do so indefinitely or whether the next step will give a nonzero answer.[5] If x is not zero, then the search will eventually terminate, but if x is zero, it will run forever.

Accepting this, it is not difficult to define the relational operators. To decide the ordering relation of x and y it suffices to find an n such that $|x_n - y_n| \geq 2$. For example, if $x_n \geq y_n + 2$ we have

$$2^n x > x_n - 1 \geq y_n + 1 > 2^n y$$

and so $x > y$. We first write a general procedure to perform this search, and then define all the orderings in terms of it. Note that the only way to arrive at the reflexive orderings is to justify the irreflexive version!

```
let separate =
  let rec separate n x y =
    let d = x(n) -/ y(n) in
    if abs_num(d) >/ Int 1 then d
    else separate (n + 1) x y in
separate 0;;

let real_gt x y = separate x y >/ Int 0;;
let real_ge x y = real_gt x y;;
let real_lt x y = separate x y </ Int 0;;
let real_le x y = real_lt x y;;
```

9.3.7 Caching

In order to try out all the above functions, we would like to be able to print out a decimal representation of some approximation to a real. This is not hard, using some more standard facilities in the CAML library. If d decimal places are desired, then we make n such that $2^n > 10^d$ and hence the accuracy corresponds at least to the number of digits printed.

```
let view x d =
  let n = 4 * d in
  let out = x(n) // (Int 2 **/ Int n) in
  approx_num_fix d out;;
```

[5]For a proof that it is uncomputable, simply wrap up the halting problem, by defining $f(n) = 1$ if a certain Turing machine has halted after no more than n iterations, and $f(n) = 0$ otherwise.

Now we can test out some simple examples, which seem to work:

```
#let x = real_of_int 3;;
x : int -> num = <fun>
#let xi = real_inv x;;
xi : int -> num = <fun>
#let wun = real_mul x xi;;
wun : int -> num = <fun>
#view x 20;;
it : string = "3.00000000000000000000"
#view xi 20;;
it : string = ".33333333333333333333"
#view wun 20;;
it : string = "1.00000000000000000000"
```

However there is a subtle and serious bug in our implementation, which shows itself if we try larger expressions. The problem is that subexpressions can be evaluated many times. Most obviously, this happens if they occur several times in the input expression. But even if they don't, the multiplication, and even more so, the inverse, require trial evaluations at differing accuracies. As the evaluation propagates down to the bottom level, there is often an exponential buildup of reevaluations. For example, the following is slow — it takes several seconds.

```
#let x1 = real_of_int 1 in
 let x2 = real_mul x1 x1 in
 let x3 = real_mul x2 x2 in
 let x4 = real_mul x3 x3 in
 let x5 = real_mul x4 x4 in
 let x6 = real_mul x5 x5 in
 let x7 = real_mul x6 x6 in
 view x7 10;;
 - : string = "+1.0000000000"
```

We can fix this problem using the idea of *caching* or *memo functions* (Michie 1968). We give each function a reference cell to remember the most accurate version already calculated. If there is a second request for the same accuracy, it can be returned immediately without further evaluation. What's more, we can always construct a lower-level approximation, say n, from a higher one, say $n+k$ with $k \geq 1$. If we know that $|f_x(n+k) - 2^{n+k}x| < 1$ then we have:

$$
\begin{array}{rcl}
|f_x(n+k) \text{ ndiv } 2^k - 2^n x| & \leq & \frac{1}{2} + |\frac{f_x(n+k)}{2^k} - 2^n x| \\
& = & \frac{1}{2} + \frac{1}{2^k}|f_x(n+k) - 2^{n+k}x| \\
& < & \frac{1}{2} + \frac{1}{2^k} \\
& \leq & 1
\end{array}
$$

Hence we are always safe in returning $f_x(n+k)$ ndiv 2^k. We can implement this memo technique via a generic function `memo` to be inserted in each of the previous functions:

```
let real_of_int k = memo (fun n -> (Int 2 **/ Int n) */ Int k);;

let real_neg f = memo (fun n -> minus_num(f n));;

let real_abs f = memo (fun n -> abs_num (f n));;

let real_add f g = memo (fun n ->
   (f(n + 2) +/ g(n + 2)) ndiv (Int 4));;

let real_sub f g = real_add f (real_neg g);;

let real_intmul m x = memo (fun n ->
   let p = log2 (abs_num m) in
   let p1 = p + 1 in
   (m */ x(n + p1)) ndiv (Int 2 **/ Int p1));;

let real_intdiv m x = memo (fun n ->
   x(n) ndiv (Int m));;

let real_mul x y = memo (fun n ->
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m));;

let real_inv x = memo (fun n ->
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k)));;

let real_div x y = real_mul x (real_inv y);;
```

where

```
let memo f =
  let mem = ref (-1,Int 0) in
  fun n -> let (m,res) = !mem in
           if n <= m then
             if m = n then res
             else res ndiv (Int 2 **/ Int(m - n))
           else
             let res = f n in
             mem := (n,res); res;;
```

Now the above sequence of multiplications is instantaneous. Here are a few more examples:

```
#let pi1 = real_div (real_of_int 22) (real_of_int 7);;
pi1 : int -> num = <fun>
#view pi1 10;;
it : string = "+3.1428571429"
#let pi2 = real_div (real_of_int 355) (real_of_int 113);;
pi2 : int -> num = <fun>
#view pi2 10;;
it : string = "+3.1415929204"
#let pidiff = real_sub pi1 pi2;;
pidiff : int -> num = <fun>
#view pidiff 20;;
it : string = "+0.00126422250316055626"
#let ipidiff = real_inv pidiff;;
ipidiff : int -> num = <fun>
#view ipidiff 20;;
it : string = "+791.00000000000000000000"
```

Of course, everything we have done to date could be done with rational arithmetic. Actually, it may happen that our approach is more efficient in certain situations since we avoid calculating numbers that might have immense numerators and denominators when we just need an approximation. Nevertheless, the present approach really comes into its own when we want to define transcendental functions like exp, sin etc. We will not cover this in detail for lack of space, but it makes an interesting exercise. One approach is to use truncated Taylor series. Note that finite summations can be evaluated directly rather than by iterating the addition function; this leads to much better error behaviour.

9.4 Prolog and theorem proving

The language Prolog is popular in Artificial Intelligence research, and is used in various practical applications such as expert systems and intelligent databases. Here we will show how the main mechanism of Prolog, namely depth-first search through a database of rules using unification and backtracking, can be implemented in ML. We do not pretend that this is a full-blown implementation of

Prolog, but it gives an accurate picture of the general flavour of the language, and we will be able to run some simple examples in our system.

9.4.1 Prolog terms

Prolog code and data is represented using a uniform system of first order terms. We have already defined a type of terms for our mathematical expressions, and associated parsers and printers. Here we will use something similar, but not quite the same. First of all, it simplifies some of the code if we treat constants as nullary functions, i.e. functions that take an empty list of arguments. Accordingly we define:

```
type term = Var of string
          | Fn of string * (term list);;
```

Where we would formerly have used `Const s`, we will now use `Fn(s,[])`. Note that we will treat functions of different arities (different numbers of arguments) as distinct, even if they have the same name. Thus there is no danger of our confusing constants with true functions.

9.4.2 Lexical analysis

In order to follow Prolog conventions, which include case sensitivity, we also modify lexical analysis. We will not attempt to conform exactly to the full details of Prolog, but one feature is very important: alphanumeric identifiers that begin with an *upper case* letter or an underscore are treated as variables, while other alphanumeric identifiers, along with numerals, are treated as constants. For example `X` and `Answer` are variables while `x` and `john` are constants. We will lump all symbolic identifiers together as constants too, but we will distinguish the punctuation symbols: left and right brackets, commas and semicolons. Non-punctuation symbols are collected together into the longest strings possible, so symbolic identifiers need not consist only of one character.

```
type token = Variable of string
           | Constant of string
           | Punct of string;;
```

The lexer therefore looks like this:

```
let lex =
  let several p = many (some p) in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let upper_alpha s = "A" <= s & s <= "Z" or s = "_"
  and lower_alpha s = "a" <= s & s <= "z" or "0" <= s & s <= "9"
  and punct s = s = "(" or s = ")" or s = "[" or s = "]"
                or s = "," or s = "."
  and space s = s = " " or s = "\n" or s = "\t" in
  let alphanumeric s = upper_alpha s or lower_alpha s in
  let symbolic s = not space s & not alphanumeric s & not punct s in
  let rawvariable =
    some upper_alpha ++ several alphanumeric >> (Variable o collect)
  and rawconstant =
    (some lower_alpha ++ several alphanumeric ||
     some symbolic ++ several symbolic) >> (Constant o collect)
  and rawpunct = some punct >>  Punct in
  let token =
    (rawvariable || rawconstant || rawpunct) ++
    several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;
```

For example:

```
#lex "add(X,Y,Z) :- X is Y+Z.";;
- : token list =
 [Constant "add"; Punct "("; Variable "X"; Punct ",";
  Variable "Y"; Punct ","; Variable "Z"; Punct ")";
  Constant ":-"; Variable "X"; Constant "is"; Variable "Y";
  Constant "+"; Variable "Z"; Punct "."]
```

9.4.3 Parsing

The basic parser is pretty much the same as before; the printer is exactly the same. The main modification to the printer is that we allow Prolog lists to be written in a more convenient notation. Prolog has '.' and 'nil' corresponding to ML's '`::`' and '`[]`', and we set up the parser so that lists can be written much as in ML, e.g. '`[1,2,3]`'. We also allow the Prolog notation '`[H|T]`' instead of '`cons(H,T)`', typically used for pattern-matching. After the basic functions:

```
let variable =
  fun (Variable s::rest) -> s,rest
    | _ -> raise Noparse;;

let constant =
  fun (Constant s::rest) -> s,rest
    | _ -> raise Noparse;;
```

we have a parser for terms and also for Prolog rules, of these forms:

$$\begin{array}{l} term. \\ term \quad \text{:-} \quad term, \ldots, term. \end{array}$$

The parsers are as follows:

```
let rec atom input
      = (constant ++ a (Punct "(") ++ termlist ++ a (Punct ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || constant
             >> (fun s -> Fn(s,[]))
      || variable
             >> (fun s -> Var s)
      || a (Punct "(") ++ term ++ a (Punct ")")
             >> (snd o fst)
      || a (Punct "[") ++ list
             >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
      = (term ++ a (Punct ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input
  and list input
      = (term ++ (a (Constant "|") ++ term ++ a (Punct "]")
                     >> (snd o fst)
                 || a (Punct ",") ++ list
                      >> snd
                 || a (Punct "]")
                     >> (K (Fn("[]",[]))))
             >> (fun (h,t) -> Fn(".",[h; t]))
      || a (Punct "]")
             >> (K (Fn("[]",[])))) input
  and rule input
     = (term ++ (a (Punct ".")
                     >> (K [])
                || a (Constant ":-") ++ term ++
                   many (a (Punct ",") ++ term >> snd) ++
                   a (Punct ".")
                     >> (fun (((_,h),t),_) -> h::t))) input;;

let parse_term = fst o (term ++ finished >> fst) o lex;;

let parse_rules = fst o (many rule ++ finished >> fst) o lex;;
```

9.4.4 Unification

Prolog uses a set of rules to solve a current *goal* by trying to match one of the rules against the goal. A rule consisting of a single term can solve a goal immediately.

In the case of a rule *term* :- $term_1, \ldots, term_n$., if the goal matches *term*, then Prolog needs to solve each $term_i$ as a subgoal in order to finish the original goal.

However, goals and rules do not have to be exactly the same. Instead, Prolog assigns variables in both the goal and the rule to make them match up, a process known as *unification*. This means that we can end up proving a special case of the original goal, e.g. $P(f(X))$ instead of $P(Y)$. For example:

- To unify $f(g(X), Y)$ and $f(g(a), X)$ we can set $X = a$ and $Y = a$. Then both terms are $f(g(a), a)$.

- To unify $f(a, X, Y)$ and $f(X, a, Z)$ we can set $X = a$ and $Y = Z$, and then both terms are $f(a, a, Z)$.

- It is impossible to unify $f(X)$ and X.

In general, unifiers are not unique. For example, in the second example we could choose to set $Y = f(b)$ and $Z = f(b)$. However one can always choose one that is *most general*, i.e. any other unification can be reached from it by further instantiation (compare most general types in ML). In order to find it, roughly speaking, one need only descend the two terms recursively in parallel, and on finding a variable on either side, assigns it to whatever the term on the other side is. One also needs to check that the variable hasn't already been assigned to something else, and that it doesn't occur in the term being assigned to it (as in the last example above). A simple implementation of this idea follows. We maintain an association list giving instantiations already made, and we look each variable up to see if it is already assigned before proceeding. We use the existing instantiations as an accumulator.

```
let rec unify tm1 tm2 insts =
  match tm1 with
    Var(x) ->
      (try let tm1' = assoc x insts in
           unify tm1' tm2 insts
       with Not_found ->
           augment (x,tm2) insts)
  | Fn(f1,args1) ->
      match tm2 with
        Var(y) ->
          (try let tm2' = assoc y insts in
               unify tm1 tm2' insts
           with Not_found ->
               augment (y,tm1) insts)
      | Fn(f2,args2) ->
          if f1 = f2
          then itlist2 unify args1 args2 insts
          else raise (error "functions do not match");;
```

where the instantiation lists need to be augmented with some care to avoid the so-called 'occurs check' problem. We must disallow instantiation of X to a non-trivial term involving X, as in the third example above. Most real Prologs ignore this, either for (claimed) efficiency reasons or in order to allow weird cyclic datastructures instead of simple first order terms.

```
let rec occurs_in x =
  fun (Var y) -> x = y
    | (Fn(_,args)) -> exists (occurs_in x) args;;

let rec subst insts = fun
   (Var y) -> (try assoc y insts with Not_found -> tm)
 | (Fn(f,args)) -> Fn(f,map (subst insts) args);;

let raw_augment =
  let augment1 theta (x,s) =
    let s' = subst theta s in
    if occurs_in x s & not(s = Var(x))
    then raise (error "Occurs check")
    else (x,s') in
  fun p insts -> p::(map (augment1 [p]) insts);;

let augment (v,t) insts =
  let t' = subst insts t in match t' with
    Var(w) -> if w <= v then
                if w = v then insts
                else raw_augment (v,t') insts
              else raw_augment (w,Var(v)) insts
  | _ -> if occurs_in v t'
         then raise (error "Occurs check")
         else raw_augment (v,t') insts;;
```

9.4.5 Backtracking

Prolog proceeds by depth-first search, but it may backtrack: even if a rule unifies successfully, if all the remaining goals cannot be solved under the resulting instantiations, then another initial rule is tried. Thus we consider the whole list of goals rather than one at a time, to give the right control strategy.

```
let rec first f =
  fun [] -> raise (error "No rules applicable")
    | (h::t) -> try f h with error _ -> first f t;;

let rec expand n rules insts goals =
  first (fun rule ->
    if goals = [] then insts else
    let conc,asms =
      rename_rule (string_of_int n) rule in
    let insts' = unify conc (hd goals) insts in
    let local,global = partition
      (fun (v,_) -> occurs_in v conc or
              exists (occurs_in v) asms) insts' in
    let goals' = (map (subst local) asms) @
                 (tl goals) in
    expand (n + 1) rules global goals') rules;;
```

Here we use a function 'rename' to generate fresh variables for the rules each time:

```
let rec rename s =
  fun (Var v) -> Var("~"^v^s)
    | (Fn(f,args)) -> Fn(f,map (rename s) args);;

let rename_rule s (conc,asms) =
  (rename s conc,map (rename s) asms);;
```

Finally, we can package everything together in a function `prolog` that tries to solve a given goal using the given rules:

```
type outcome = No | Yes of (string * term) list;;

let prolog rules goal =
  try let insts = expand 0 rules [] [goal] in
      Yes(filter (fun (v,_) -> occurs_in v goal)
                 insts)
  with error _ -> No;;
```

This says either that the goal cannot be solved, or that it can be solved with the given instantiations. Note that we only return one answer in this latter case, but this is easy to change if desired.

9.4.6 Examples

We can use the Prolog interpreter just written to try some simple examples from Prolog textbooks. For example:

```
#let rules = parse_rules
  "male(albert).
   male(edward).
   female(alice).
   female(victoria).
   parents(edward,victoria,albert).
   parents(alice,victoria,albert).
   sister_of(X,Y) :-
     female(X),
     parents(X,M,F),
     parents(Y,M,F).";;
rules : (term * term list) list =
 [`male(albert)`, []; `male(edward)`, [];
  `female(alice)`, []; `female(victoria)`, [];
  `parents(edward,victoria,albert)`, [];
  `parents(alice,victoria,albert)`, [];
  `sister_of(X,Y)`,
    [`female(X)`; `parents(X,M,F)`; `parents(Y,M,F)`]]
#prolog rules ("sister_of(alice,edward)");;
- : outcome = Yes []
#prolog rules (parse_term "sister_of(alice,X)");;
- : outcome = Yes ["X", `edward`]
#prolog rules (parse_term "sister_of(X,Y)");;
- : outcome = Yes ["Y", `edward`; "X", `alice`]
```

The following are similar to some elementary ML list operations. Since Prolog is relational rather than functional, it is possible to use Prolog queries in a more flexible way, e.g. ask what arguments would give a certain result, rather than vice versa:

```
#let r = parse_rules
  "append([],L,L).
   append([H|T],L,[H|A]) :- append(T,L,A).";;
r : (term * term list) list =
 [`append([],L,L)`, [];
  `append(H . T,L,H . A)`, [`append(T,L,A)`]]
#prolog r (parse_term "append([1,2],[3],[1,2,3])");;
- : outcome = Yes []
#prolog r (parse_term "append([1,2],[3,4],X)");;
- : outcome = Yes ["X", `1 . (2 . (3 . (4 . [])))`]
#prolog r (parse_term "append([3,4],X,X)");;
- : outcome = No
#prolog r (parse_term "append([1,2],X,Y)");;
- : outcome = Yes ["Y", `1 . (2 . X)`]
```

In such cases, Prolog seems to be showing an impressive degree of intelligence. However under the surface it is just using a simple search strategy, and this can easily be thwarted. For example, the following loops indefinitely:

```
#prolog r (parse_term "append(X,[3,4],X)");;
```

9.4.7 Theorem proving

Prolog is acting as a simple theorem prover, using a database of logical facts (the rules) in order to prove a goal. However it is rather limited in the facts it can prove, partly because its depth-first search strategy is incomplete, and partly because it can only make logical deductions in certain patterns. It is possible to make Prolog-like systems that are more powerful, e.g. see Stickel (1988). In what follows, we will just show how to build a more capable theorem prover using essentially similar technology, including the unification code and an identical backtracking strategy.

In particular, unification is an effective way of deciding how to specialize universally quantified variables. For example, given the facts that $\forall X.\, p(X) \Rightarrow q(X)$ and $p(f(a))$, we can unify the two expressions involving p and thus discover that we need to set X to $f(a)$. By contrast, the very earliest theorem provers tried all possible terms built up from the available constants and functions (the 'Herbrand base').

Usually, depth-first search would go into an infinite loop, so we need to modify the Prolog strategy slightly. We will use *depth first iterative deepening*. This means that the search depth has a hard limit, and attempts to exceed it cause backtracking. However, if no proof is found at a given depth, the bound is increased and another attempt is made. Thus, first one searches for proofs of depth 1, and if that fails, searches for one of depth 2, then depth 3 and so on. For 'depth' one can use various parameters, e.g. the height or overall size of the search tree; we will use the number of unifiable variables introduced.

Manipulating formulas

We will simply use our standard first order terms to denote formulas, introducing new constants for the logical operators. Many of these are written infix.

Operator	Meaning
~(p)	not p
p & q	p and q
p | q	p or q
p --> q	p implies q
p <-> q	p if and only if q
forall(X,p)	for all X, p
exists(X,p)	there exists an X such that p

An alternative would be to introduce a separate type of formulas, but this would require separate parsing and printing support. We will avoid this, for the sake of simplicity.

Preprocessing formulas

It's convenient if the main part of the prover need not cope with implications and 'if and only if's. Therefore we first define a function that eliminates these in favour of the other connectives.

```
let rec proc tm =
  match tm with
    Fn("~",[t]) -> Fn("~",[proc t])
  | Fn("&",[t1; t2]) -> Fn("&",[proc t1; proc t2])
  | Fn("|",[t1; t2]) -> Fn("|",[proc t1; proc t2])
  | Fn("-->",[t1; t2]) ->
        proc (Fn("|",[Fn("~",[t1]); t2]))
  | Fn("<->",[t1; t2]) ->
        proc (Fn("&",[Fn("-->",[t1; t2]);
                      Fn("-->",[t2; t1])]))
  | Fn("forall",[x; t]) -> Fn("forall",[x; proc t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; proc t])
  | t -> t;;
```

The next step is to push the negations down the formula, putting it into so-called 'negation normal form' (NNF). We define two mutually recursive functions that create NNF for a formula, and its negation.

```
let rec nnf_p tm =
  match tm with
    Fn("~",[t]) -> nnf_n t
  | Fn("&",[t1; t2]) -> Fn("&",[nnf_p t1; nnf_p t2])
  | Fn("|",[t1; t2]) -> Fn("|",[nnf_p t1; nnf_p t2])
  | Fn("forall",[x; t]) -> Fn("forall",[x; nnf_p t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; nnf_p t])
  | t -> t

and nnf_n tm =
  match tm with
    Fn("~",[t]) -> nnf_p t
  | Fn("&",[t1; t2]) -> Fn("|",[nnf_n t1; nnf_n t2])
  | Fn("|",[t1; t2]) -> Fn("&",[nnf_n t1; nnf_n t2])
  | Fn("forall",[x; t]) -> Fn("exists",[x; nnf_n t])
  | Fn("exists",[x; t]) -> Fn("forall",[x; nnf_n t])
  | t -> Fn("~",[t]);;
```

We will convert the negation of the input formula into negation normal form, and the main prover will then try to derive a contradiction from it. This will suffice to prove the original formula.

The main prover

At each stage, the prover has a current formula, a list of formulas yet to be considered, and a list of literals. It is trying to reach a contradiction. The following strategy is employed:

- If the current formula is ‘p & q’, then consider ‘p’ and ‘q’ separately, i.e. make ‘p’ the current formula and add ‘q’ to the formulas to be considered.
- If the current formula is ‘p | q’, then try to get a contradiction from ‘p’ and then one from ‘q’.
- If the current formula is ‘forall(X,p)’, invent a new variable to replace ‘X’: the right value can be discovered later by unification.
- If the current formula is ‘exists(X,p)’, invent a new constant to replace ‘X’.
- Otherwise, the formula must be a literal, so try to unify it with a contradictory literal.
- If this fails, add it to the list of literals, and proceed with the next formula.

We desire a similar backtracking strategy to Prolog: only if the current instantiation allow all remaining goals to be solved do we accept it. We could use lists again, but instead we use *continuations*. A continuation is a function that is passed to another function and can be called from within it to ‘perform the rest of the computation’. In our case, it takes a list of instantiations and tries to solve the remaining goals under these instantiations. Thus, rather than explicitly trying to solve all remaining goals, we simply try calling the continuation.

```
let rec prove fm unexp pl nl n cont i =
  if n < 0 then raise (error "No proof") else
  match fm with
    Fn("&",[p;q]) ->
        prove p (q::unexp) pl nl n cont i
  | Fn("|",[p;q]) ->
        prove p unexp pl nl n
        (prove q unexp pl nl n cont) i
  | Fn("forall",[Var x; p]) ->
        let v = mkvar() in
        prove (subst [x,Var v] p) (unexp@[fm]) pl nl (n - 1) cont i
  | Fn("exists",[Var x; p]) ->
        let v = mkvar() in
        prove (subst [x,Fn(v,[])] p) unexp pl nl (n - 1) cont i
  | Fn("~",[t]) ->
        (try first (fun t' -> let i' = unify t t' i in
                              cont i') pl
         with error _ ->
            prove (hd unexp) (tl unexp) pl (t::nl) n cont i)
  | t ->
        (try first (fun t' -> let i' = unify t t' i in
                              cont i') nl
         with error _ ->
            prove (hd unexp) (tl unexp) (t::pl) nl n cont i);;
```

We set up the final prover as follows:

```
let prover =
  let rec prove_iter n t =
    try let insts = prove t [] [] [] n I [] in
        let globinsts = filter
          (fun (v,_) -> occurs_in v t) insts in
        n,globinsts
    with error _ -> prove_iter (n + 1) t in
  fun t -> prove_iter 0 (nnf_n(proc(parse_term t)));;
```

This implements the iterative deepening strategy. It tries to find the proof with the fewest universal instantiations; if it succeeds, it returns the number required and any toplevel instantiations, throwing away instantiations for internally created variables.

Examples

Here are some simple examples from Pelletier (1986).

```
#let P1 = prover "p --> q <-> ~(q) --> ~(p)";;
P1 : int * (string * term) list = 0, []
#let P13 = prover "p | q & r <-> (p | q) & (p | r)";;
P13 : int * (string * term) list = 0, []
#let P16 = prover "(p --> q) | (q --> p)";;
P16 : int * (string * term) list = 0, []
#let P18 = prover "exists(Y,forall(X,p(Y)-->p(X)))";;
P18 : int * (string * term) list = 2, []
#let P19 = prover "exists(X,forall(Y,forall(Z,
                (p(Y)-->q(Z))-->p(X)-->q(X))))";;
P19 : int * (string * term) list = 6, []
```

A bigger example is the following:

```
#let P55 = prover
  "lives(agatha) & lives(butler) & lives(charles) &
   (killed(agatha,agatha) | killed(butler,agatha) |
    killed(charles,agatha)) &
   (forall(X,forall(Y,
     killed(X,Y) --> hates(X,Y) & ~(richer(X,Y))))) &
   (forall(X,hates(agatha,X)
             --> ~(hates(charles,X)))) &
   (hates(agatha,agatha) & hates(agatha,charles)) &
   (forall(X,lives(X) & ~(richer(X,agatha))
             --> hates(butler,X))) &
   (forall(X,hates(agatha,X) --> hates(butler,X))) &
   (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
             | ~(hates(X,charles))))
   --> killed(agatha,agatha)";;
P55 : int * (string * term) list = 6, []
```

In fact, the prover can work out 'whodunit':

```
#let P55' = prover
  "lives(agatha) & lives(butler) & lives(charles) &
   (killed(agatha,agatha) | killed(butler,agatha) |
    killed(charles,agatha)) &
   (forall(X,forall(Y,
     killed(X,Y) --> hates(X,Y) & ~(richer(X,Y))))) &
   (forall(X,hates(agatha,X)
            --> ~(hates(charles,X)))) &
   (hates(agatha,agatha) & hates(agatha,charles)) &
   (forall(X,lives(X) & ~(richer(X,agatha))
            --> hates(butler,X))) &
   (forall(X,hates(agatha,X) --> hates(butler,X))) &
   (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
            | ~(hates(X,charles))))
   --> killed(X,agatha)";;
P55' : int * (string * term) list = 6, ["X", `agatha`]
```

Further reading

Symbolic differentiation is a classic application of functional languages. Other symbolic operations are major research issues in their own right — Davenport, Siret, and Tournier (1988) give a readable overview of what computer algebra systems can do and how they work. Paulson (1983) discusses the kind of simplification strategy we use here in a much more general setting.

Parsing with higher order functions is another popular example. It seems to have existed in the functional programming folklore for a long time; an early treatment is given by Burge (1975). Our presentation has been influenced by the treatments in Paulson (1991) and Reade (1989).

The first definition of 'computable real number' in our sense was in the original paper of Turing (1936). His approach was based on decimal expansions, but he needed to modify it (Turing 1937) for reasons explored in the exercises below. The approach to real arithmetic given here follows the work of Boehm, Cartwright, O'Donnel, and Riggle (1986). More recently a high-performance implementation in CAML Light has been written by Ménissier-Morain (1994), whose thesis (in French) contains detailed proofs of correctness for all the algorithms for elementary transcendental functions. For an alternative approach using 'linear fractional transformations', see Potts (1996).

Our approach to Prolog search and backtracking, either using lists or continuations, is fairly standard. For more on implementing Prolog, see for example Boizumault (1993), while for more on the actual use of the language, the classic text by Clocksin and Mellish (1987) is recommended. A detailed discussion of continuations in their many guises is given by Reynolds (1993). The unification algorithm given here is simple and purely functional, but rather inefficient. For faster imperative algorithms, see Martelli and Montanari (1982). Our theorem prover is based on leanT^AP (Beckert and Posegga 1995). For another important theorem proving method conveniently based on Prolog-style search, look at the Prolog Technology Theorem Prover (Stickel 1988).

Exercises

1. Modify the printer so that each operator has an associativity, used when printing iterated cases of the operator to avoid excessive parentheses.

2. The differentiation rules for inverses $1/g(x)$ and quotients $f(x)/g(x)$ are not valid when $g(x) = 0$. Several others for functions like ln are also true only under certain conditions. Actually most commercial computer algebra systems ignore this fact. However, try to do better, by writing a new version of the differentiator that not only produces the derivative but also a list of conditions that must be assumed for the derivative to be valid.

3. Try programming a simple procedure for indefinite integration. In general, it will not be able to do every integral, but try to make it understand a few basic principles.[6]

4. Read the documentation for the `format` library, and try to implement a printer that behaves better when the output is longer than one line.

5. What happens if the parser functions like `term`, `atom` etc. are all eta-reduced by deleting the word `input`? What happens if `precedence` is consistently eta-reduced by deleting `input`?

6. How well do precedence parsers generated by `precedence` treat distinct operators with the same precedence? How can the behaviour be improved? Can you extend it to allow a mix of left and right associative operators? For example, one really wants subtraction to associate to the left.

7. Rewrite the parser so that it gives helpful error messages when parsing fails.

8. Try representing a real number by the function that generates the first n digits in a decimal expansion. Can you implement the addition function?

9. (*) How can you retain a positional representation while making the basic operations computable?

[6]In fact there are algorithms that can integrate all algebraic expressions (not involving sin, ln etc.) — see Davenport (1981).

10. Implement multiprecision integer arithmetic operations yourself. Make your multiplication algorithm $O(n^{log_2 3})$ where n is the number of digits in the arguments.

11. Using `memo` as we have defined it, is it the case that whatever series of arguments a function is applied to, it will always give the same answer for the same argument? That is, are the functions true mathematical functions despite the internal use of references?

12. Write a parser-evaluator to read real-valued expressions and create the functional representation for the answer.

13. (*) Extend the real arithmetic routines to some functions like exp and sin.

14. Our preprocessing and negation normal form process can take exponential time in the length of the input in the case of many 'if and only if's. Modify it so that it is linear in this number.

15. Extend the initial transformation into negation normal form so that it gives Skolem normal form.[7]

16. (*) Implement a more efficient unification algorithm, e.g. following Martelli and Montanari (1982).

17. (*) Implement a version of the Prolog Technology Theorem Prover (Stickel 1988)

[7]Skolem normal form is covered in most elementary logic books, e.g. Enderton (1972).