Introduction to Functional Programming

Anuj Dawar

Computer Laboratory
University of Cambridge
Lent Term 2003
Main Text:

Other Useful Reading:

Notes and exercises available from:

In an imperative programming language, the program provides a series of instructions (or commands) to the machine.

Examples of such languages include C, Pascal, Modula2, Java

In a declarative programming language, the program (in principle) describes the computational task.

Functional: ML, Scheme, Haskell,…
Logic: Prolog, Godel,…
Imperative languages present a level of abstraction above the machine, hiding some details (memory addresses, registers, etc.) Still, the view is *machine-centred*.

Declarative languages provide a still further level of abstraction.

A style of programming that is more *programmer-centred*.
Functional programming

In the functional programming style, the computational task to be programmed is taken to be a function (in the mathematical sense).

The job of the programmer is to describe this function.

Implicit in the description is a method for computing the function.

The function maps one domain (of inputs) to another (of outputs).

These may be: integers; real numbers; lists; strings; or even functions themselves

importance of types
Commands and Expressions

In a typical imperative language, commands are formed from assignments to variables:

\[ x := E \]

by application of various control structures.

Sequencing

\[ C_1; C_2 \]

Conditionals

\[ \text{if } B \text{ then } C_1 \text{ else } C_2 \]

Looping

\[ \text{while } B \text{ do } C \]
Expressions

A functional program is just an expression to be evaluated.

An expression is built up from simpler expressions by means of function applications.

\[ E_1 + E_2 \]

or

\[ \text{if } B \text{ then } E_1 \text{ else } E_2 \]

There are no explicit notions of variable assignment, sequencing or control.
The factorial function can be written imperatively in C as follows:

```c
int fact(int n)
{
    int x = 1;
    while (n > 0)
    {
        x = x * n;
        n = n - 1;
    }
    return x;
}
```

whereas it would be expressed in ML as a recursive function:

```ml
fun fact n =
    if n = 0 then 1
    else n * fact(n - 1);
```
Recursive definition of functions is crucial to functional programming.

There is no other mechanism for looping

Variables cannot be updated through assignment. They get their values from function calls.
ML provides type checking, which can help catch many programming errors.

Types in ML may be polymorphic.

```ml
fun length [] = 0
| length (x::l) = 1 + length l;
```
Introduction to Functional Programming

Advantages

“Attack complexity with simple abstractions”

- Clarity
- Expressiveness
- Shorter Programs
- Security through type system
- Ease of reasoning
- Better modularity
Disadvantages

- Input/Output
- Interactivity and continuously running programs
- Speed/Efficiency

There is no reasonable “pure” functional language
Brief History

- Lambda Calculus (Church 1936)
- LISP (McCarthy 1954)
- ISWIM (Landin 1966)
- ML (Milner et al., 1974), originally a Meta Language for the LCF Theorem Prover.
- Definition of Standard ML (Milner, Tofte and Harper 1990)
- Revised definition and standard library (1997)
Rest of the Course

11 more lectures covering

- Basic Types in Standard ML
- Lists and Recursion
- Sorting
- Datatypes
- Higher Order Functions
- Specification and Verification
- Types and Type Inference
- Substantial case study
Running ML

ML provides an interactive session.

Enter an expression. ML returns a value.

Moscow ML version 1.42 (July 1997)
Enter `quit();' to quit.
- (2*4) + 18;

> val it = 26 : int
- 2.0 * 2.0 * 3.14159;
> val it = 12.56636 : real
- val pi = 3.14159;
> val pi = 3.14159 : real
- val a = pi* 2.0 *2.0;
> val a = 12.56636 : real
- val a = 2 * pi;

...

! Type clash: expression of type
! real
! cannot have type
! int

- val area = fn r => pi*r*r;
> val area = fn : real -> real
- val sqr = fn r => r*r;
> val sqr = fn : int -> int
- val sqr = fn r:real => r*r;
> val sqr = fn : real -> real
- val sqr = fn r => r*r:real;
> val sqr = fn : real -> real
- fun area (r) = pi*r*r;
> val area = fn : real -> real
- val pi = "yabadabadoo";
> val pi = "yabadabadoo" : string
- area(2.0);
> val it = 12.56636 : real
- area;
> val it = fn : real -> real
- it(2.0);
> val it = 12.56636 : real
- area(2);

... ! Type clash: expression of type
!   int
! cannot have type
!   real
**Numeric Types**

**int:** the integers

- constants \(0 1 \sim 1 2 \sim 2 0032\ldots\)
- infix operators \(+ - \ast \text{div} \text{mod}\)

**real:** the floating point numbers

- constants \(0.0 \sim 1.414 2.0 3.94e^7\ldots\)
- infix operators \(+ - \ast /\)
Overloading

Functions defined for both `int` and `real`:

- operators  \( \sim + - * \)
- relations  \( < <= > >= \)

You must tell the type checker what type is intended, if there is any ambiguity.
Useful library of functions, collected together into structures.

```
Int   Real   Math
```

The basis library is automatically loaded when using SML/NJ.

May need to be explicitly loaded in Moscow ML.

```
> load "Math";
> val it = () : unit
- fun f u = Math.sin(u)/u;
> val f = fn : real -> real
```

To load your own file of definitions:

```
- use "myfile";
```
Strings

Type string

- constants "" "A" "yaba!!daba&doo\n"
- size: string -> int
determines the number of characters in a string.
- s1^s2
  the concatenation of strings s1 and s2
- relations  <  <=  >  >=

Structure String
**Characters**

**Type** char

- constants  
  ```
  "A"  "y"  " 
  ```
- `ord: char -> int` integer value of a character.
- `chr: int -> char`
- `relations  <  <=  >  >=`

**Structure** Char
Truth Values

Type bool

- constants true false
- not: bool -> bool
- if p then x else y

\[ p \text{ andalso } q \]

\[ \text{if } p \text{ then } q \text{ else false} \]

\[ p \text{ orelse } q \]

\[ \text{if } p \text{ then true else } q \]

Structure Bool
Pairs and Tuples

- (2,3);
> val it = (2, 3) : int * int

- (2.0,2,3,"aa");

> val it = (2.0, 2, 3, "aa") : real * int * int * string

Tuples are useful for representing vectors,
presenting functions with multiple arguments,
obtaining multiple results from a function, etc.

- fun addtwice (m,n) = m + 2*n;
> val addtwice = fn : int * int -> int
**Vectors**

- fun negvec(x,y):real*real = (~x,~y);

> val negvec = fn : real * real -> real * real

- negvec(1.0,1.0);
> val it = (~1.0, ~1.0) : real * real

- fun addvec((x1,y1),(x2,y2)):real*real = (x1+x2,y1+y2);

> val addvec = fn : (real * real) * (real * real) -> real * real

- fun subvec(v1,v2) = addvec(v1,negvec v2);

> val subvec = fn : (real * real) * (real * real) -> real * real
Evaluation Strategy

Strict (or eager) evaluation.
Also known as call-by-value

Given an expression, which is a function application

\[ f(E_1, \ldots, E_n) \]

evaluate \( E_1, \ldots, E_n \) and then apply \( f \) to the resulting values.

Call-by-name:
Substitute the expressions \( E_1, \ldots, E_n \) into the definition of \( f \) and then evaluate the resulting expression.
Lazy Evaluation

Also known as **call-by-need**.

Like call-by-name, but sub-expressions that appear more than once are not copied. Pointers are used instead.

Potentially more efficient, but difficult to implement.

Standard ML uses strict evaluation.
A list is an ordered collection (of any length) of elements of the same type

- \([1,2,4]\);
- \(\text{val it} = [1, 2, 4] : \text{int list}\)
- \(["a", "", "abc", "a"]\);
- \(\text{val it} = . . . : \text{string list}\)
- \(\text{val it} = . . . : \text{int list list}\)
- \(\text{val it} = [] : 'a \text{ list}\)
- \(\text{val it} = [] : \text{int list}\)
- \(\text{val it} = [1, 2, 3] : \text{int list}\)
Lists

There are two kinds of list:

\[ \text{nil or [] is the empty list} \]

\[ h::t \text{ is the list with head } h \text{ and tail } t \]

\[ :: \text{ is an infix operator of type} \]

\[ \text{fn : 'a * 'a list } \rightarrow \text{ 'a list} \]

\[ [x_1, \ldots, x_n] \text{ is shorthand for} \]

\[ x_1::(\cdots(x_n::\text{nil})\cdots) \]
null

\[ \text{fn} : \text{'a list} \rightarrow \text{bool} \]
determines if a list is empty

hd

\[ \text{fn} : \text{'a list} \rightarrow \text{'a} \]
gives the first element of the list

tl

\[ \text{fn} : \text{'a list} \rightarrow \text{'a list} \]
gives the tail of the list
length

\[ \text{fn} : \ 'a \text{ list} \to \text{ int} \]

gives the number of elements in a list

rev

\[ \text{fn} : \ 'a \text{ list} \to \ 'a \text{ list} \]

gives the list in reverse order

\(\@\)

appends two lists NB: infix!
fun null l =  
    if l = [] then true else false;

or, using pattern matching:

fun null [] = true
  | null (::_) = false;

fun hd (x::_) = x;

fun tl (_::l) = l;

NB: these functions are built-in and do not need to be defined
fun rlength []   = 0
     | rlength (h::t) = 1 + rlength(t);

fun append ([], l) = l
     | append (h::t, l) = h::append(t,l);

fun reverse []   = []
     | reverse (h::t) = reverse(t)@[h];

Purely recursive definitions can be very inefficient
Iterative Definitions

fun addlen ([],n) = n
  | addlen (h::t, n) = addlen (t, n+1);

  fn : 'a list * int -> int

fun ilength l = addlen(l,0);

fun revto ([],l) = l
  | revto (h::t, l) = revto (t, h::l);

  fn : 'a list * 'a list -> 'a list
load "List";

We can then use List.take, List.drop

fun take (k, []) = []
| take (k, h::t) =
  if k > 0 then h::take(k-1,t)
  else [];

fun drop (k, []) = []
| drop (k, h::t) =
  if k > 0 then drop(k-1,t)
  else h::t;

fn : int * 'a list -> 'a list
fun zip ([], []) = []
| zip (h1::t1, h2::t2) = (h1, h2)::zip(t1, t2);

! Warning: pattern matching is not exhaustive

> val zip = fn :
    'a list * 'b list -> ('a * 'b) list

Creates a list of pairs from a pair of lists.

What happens when the two lists are of different length?
fun unzip [] = ([], [])
| unzip ((x,y)::pairs) = 
    let val (t,u) = unzip pairs in 
    (x::t, y::u)
    end;

Note the local declaration

let $D$ in $E$ end

Compare this against applying functions first and second to extract the components of the pair.
Equality Types

We can test certain expressions for equality:

- \(2 = 1+1\);
  > val it = true : bool
- \(1.414*1.414 = 2.0\);
  > val it = false : bool
- \([] = [1]\);
  > val it = false : bool

Equality testing can be used with the basic types, and with tuples and lists, but not with functions.

- \((\text{fn } x \Rightarrow x+2) = (\text{fn } x \Rightarrow 2+x)\);
  ! Type clash: match rule of type
  ! 'a -> 'b
  ! cannot have equality type '"'c
fun member (x, []) = false
 | member (x, h::t) = (x=h) orelse member (x,t);

val member = fn : 'a * 'a list -> bool

'a is an equality type variable.

> val it = fn : 'a * 'a -> bool

fun inter ([], l) = []
 | inter (h::t,l) = if member (h,l) then h::inter(t,l) else inter(t,l);

fn : 'a list * 'a list -> 'a list
fun insert(x:real, []) = [x]
  | insert(x, h::t) = 
    if x <= h then x::(h::t)
    else h::insert(x,t);

fun insort [] = []
  | insort (h::t) = insert (h, insort t);

fn : real list -> real list

Insertion sort takes $O(n^2)$ comparisons on average and in the worst case.
fun merge ([], l) = l : real list
| merge (l, []) = l
| merge (h1::t1,h2::t2)=
    if h1 <= h2
        then h1::merge(t1, h2::t2)
        else h2::merge(h1::t1, t2);

fun mergesort [] = []
| mergesort [x] = [x]
| mergesort l =
    let val k = length l div 2 in
        merge(mergesort (List.take(l, k)),
              mergesort (List.drop(l, k)))
    end;

Merge sort takes $O(n \log n)$ comparisons on average and in the worst case.
fun quick [] = []
| quick [x] = [x] : real list
| quick (h::t) =

let fun part (left, right, []) =
    (quick left)@(h::quick right)
| part (left, right, x::l) =
    if x<=h
    then part (x::left, right, l)
    else part (left, x::right, l)
in
    part( [], [], t) end;

Quick sort takes $O(n \log n)$ comparisons on average and $O(n^2)$ in the worst case.
fun quik ([], sorted)  = sorted
 | quik ([x], sorted)  = (x:real)::sorted
 | quik (h::t, sorted) =

let

 fun part (left, right, []) =
  quik(left, h::quik(right, sorted))
 | part (left, right, x::l) =
  if x <= h
   then part (x::left, right, l)
   else part (left, x::right, l)

in

 part([], [], t) end;
Record Types

- \{ \text{name}="Jones", \text{salary}=20300, \text{age}=26\};

\text{val it =}
{\text{age} = 26, \text{name} = "Jones", \text{salary} = 20300}
: \{\text{age} : \text{int}, \text{name} : \text{string}, \text{salary} : \text{int}\}

- \{1="Jones", 2=20300,3=26\};

> \text{val it} = ("Jones", 20300, 26)
: \text{string * int * int}
- val emp1 =
{name="Jones", salary=20300, age=26};

> val emp1 =
{age = 26, name = "Jones", salary = 20300}
: {age : int, name : string, salary : int}

- val {name=n1,salary=s1,age=a1}= emp1;

   > val n1 = "Jones" : string
   val s1 = 20300 : int
   val a1 = 26 : int

- val {name=n1,salary=s1,...} = emp1;

   > val n1 = "Jones" : string
   val s1 = 20300 : int

- val {name,age,...} = emp1;

   > val name = "Jones" : string
   val age = 26 : int
Record Types

type employee = {name: string, 
salary: int, 
age: int}; 

> type employee = ...

fun tax (e: employee) = 
  real(#salary e)*0.22;

Or,

fun tax ({salary,...}: employee) = 
  real(salary)*0.22;
Enumerated Types

Consider the King and his court:

```plaintext
datatype degree = Duke
    | Marquis
    | Earl
    | Viscount
    | Baron;

datatype person =
    King
    | Peer of degree*string*int
    | Knight of string
    | Peasant of string;
```

All constructors are distinct.
Functions on Datatypes

[King, 
Peer(Duke, "Gloucester", 5),
Knight "Gawain",
Peasant "Jack Cade"];

val it = ... : person list

fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false;
Exceptions

Exceptions are raised when there is no matching pattern, when an overflow occurs, when a subscript is out of range, or some other run-time error occurs.

Exceptions can also be explicitly raised.

```haskell
exception Failure;
exception BadVal of Int;

raise Failure
raise (BadVal 5)
```

\[
E \text{ handle } P_1 \Rightarrow E_1 \mid \ldots \mid P_n \Rightarrow E_n
\]
Recursive Datatypes

The built-in type operator of lists might be defined as follows:

\[
\text{infix} :: \ ;
\]

\[
\text{datatype 'a list} = \text{nil} \\
| :: \text{of 'a * 'a list};
\]

Binary Trees:

\[
\text{datatype 'a tree} = \\
\text{Lf} \\
| \text{Br of 'a * 'a tree * 'a tree};
\]

\[
\text{Br}(1, \text{Br}(2, \text{Br}(4, \text{Lf}, \text{Lf}), \\
\text{Br}(5, \text{Lf}, \text{Lf})), \\
\text{Br}(3, \text{Lf}, \text{Lf}))
\]
Counting the number of branch nodes

fun count Lf = 0
  | count (Br(v,t1,t2)) = 1+count(t1)+count(t2);

val count = fn : 'a tree -> int

Depth of a tree

fun depth Lf = 0
  | depth (Br(v,t1,t2)) = 1+Int.max(depth t1, depth t2);

val depth = fn : 'a tree -> int
Listing a Tree

Three different ways to list the data elements of a tree

Pre-Order

fun preorder Lf = []
  | preorder (Br(v,t1,t2)) = [v] @ preorder t1 @ preorder t2;

In-Order

fun inorder Lf = []
  | inorder (Br(v,t1,t2)) = inorder t1 @ [v] @ inorder t2;

Post-Order

fun postorder Lf = []
  | postorder (Br(v,t1,t2)) = postorder t1 @ postorder t2 @ [v];
Multi-Branching Trees

To define a datatype of a tree where each node can have any number of children

datatype 'a mtree =
    Branch of 'a * ('a mtree) list;

To recursively define functions, we can use \texttt{map}.

\texttt{fun double (Branch(k,ts)) =}
    \texttt{Branch(2*k, map double ts);}
Arrays in an imperative language designate a contiguous block of memory locations.

An array can be updated in place.

\[ A[k] := x \]

There are no such arrays in a purely functional language.

A functional array can be thought of as a function from a finite set of integers.

\[ A : \{1, \ldots, k\} \rightarrow \alpha \]

We require an update operation:

\[ B = \text{update}(A, k, x) \]

This creates a new copy of the array, which is the same as \( A \) except for the value at \( k \).
Arrays as Binary Trees

A tree whose depth is equal to the maximum number of bits in a subscript.

For a subscript $k$, take its binary representation, and at each level, follow the left branch if the bit is 0 and the right branch if it is 1.
exception Array; (*out of range*)

fun asub (Lf, _) = raise Array |
  asub (Br(v,t1,t2), k) = |
    if k = 1 then v |
    else if k mod 2 = 0 then asub(t1, k div 2) |
    else asub(t2, k div 2);

val asub = fn : 'a tree * int -> 'a

For an array $A$ and integer $k$ \texttt{asub(A,k)} gives the value of $A[k]$. 
fun aupdate (Lf, k, w) = 
  if k = 1 then Br(w, Lf, Lf) 
  else raise Array 
| aupdate (Br(v,t1,t2), k, w) = 
  if k = 1 then 
    Br(w, t1, t2) 
  else if k mod 2 = 0 then 
    Br(v, aupdate(t1,k div 2,w),t2) 
  else 
    Br(v, t1, aupdate(t2,k div 2,w));

val aupdate = 
fn : 'a tree * int * 'a -> 'a tree
A *Directory* associates *values* with *keys*.

If a directory is implemented as a list, each update and each lookup takes $O(n)$ operations on the average.

When the directory is implemented as a binary search tree, the operations take $O(\log n)$ on the average.
exception Bsearch of string;

fun blookup (Lf, b) = raise Bsearch(b)

| blookup (Br((a,x), t1, t2), b) =
  if    b<a then blookup(t1, b)
else if a<b then blookup(t2, b)
else x;

exn Bsearch = fn : string -> exn

val blookup =
fn : (string * 'a) tree * string -> 'a
fun bupdate (Lf, k:string, v) = 
    Br((k,v), Lf, Lf) 
  | bupdate (Br((a,x),t1,t2), k, v) = 
    if k < a 
      then Br((a,x),bupdate(t1,k,v),t2) 
    else if a < k 
      then Br((a,x),t1,bupdate(t2,k,v)) 
    else Br((k,v),t1,t2); (*a=k*)

val bupdate = 
fn : (string * 'a) tree * string * 'a 
    -> (string * 'a) tree
A higher order function (also called a *functional*) is a function that either takes a function as an argument, or yields a function as a result.

Higher order functions are a key feature that distinguishes functional from imperative programming.

- partial evaluation
- general purpose functionals (such as `map`)
- sequences or infinite lists
- search strategies
The expression

\[ \text{fn } x \Rightarrow E \]

is a function-valued expression.

Pattern-matching can be used:

\[ \text{fn } P_1 \Rightarrow E_1 \mid \ldots \mid P_n \Rightarrow E_n \]

The declaration

\[ \text{val double } = \text{fn } n \Rightarrow n\cdot n; \]

is the same as:

\[ \text{fun double } n = n\cdot n; \]

\[ \text{if } E \text{ then } E_1 \text{ else } E_2 \]

is defined as:

\[ (\text{fn } \text{true } \Rightarrow E_1 \mid \text{false } \Rightarrow E_2) \ E \]
Curried Functions

Every function in ML takes only one argument. For functions that require two arguments, we can use pairs:

- op+ (5,4);
> val it = 9 : int
- op+;
> val it = fn : int * int -> int

Or we can define:

- fun plus m n = m+n;
> val plus = fn : int -> int -> int

This is the curried version of plus

Partial evaluation:

- map (plus 4) [2, 3, 4];
> val it = [6, 7, 8] : int list
More Curried Functions

- hd;

> val it = fn : 'a list -> 'a

- hd [op+,op-,op*,op div] (5,4);

> val it = 9 : int

Here the type of hd is:

(int*int -> int) list -> int*int -> int

An analogy can be made with nested arrays, as in Pascal:

A: array [1..10] of
   array [1..10] of real

fun insort lessequal = 
  let fun ins (x,[]) = [x]
    | ins (x,h::t)=
      if lessequal(x,h) then x::h::t
      else h::ins(x,t)
  fun sort [] = []
    | sort (x::l) = ins(x,sort l)
  in sort end;

> val insort = fn :
  ('a * 'a -> bool) ->
  ('a list -> 'a list)
- insort (op<=) [5,3,5,7,2,9];
> val it = [2, 3, 5, 5, 7, 9] : int list
- insort (op>=) [5,3,5,7,2,9];
> val it = [9, 7, 5, 5, 3, 2] : int list
fun sum f 0 = 0.0
  | sum f m = f(m-1) + sum f (m-1);

> val sum =
  fn : (int -> real) -> int -> real

\[ \text{sum } f \text{ m} = \sum_{i=0}^{m-1} f(i) \]

\[ \text{sum } (\text{sum } f) \text{ m} = \sum_{i=0}^{m-1} \sum_{j=0}^{i-1} f(j) \]
Matrix Transpose

The map functional applies a function to every element of a list

```haskell
fun map f [] = []
  | map f (h::t) = (f h)::(map f t);
```

Representing a matrix as a list of lists, the following defines the transpose function.

```haskell
fun transp ([]::_ ) = []
  | transp rows =
      (map hd rows)::
      (transp (map tl rows));
```

```haskell
fn : 'a list list -> 'a list list
```
Matrix Multiplication

The dot product of two vectors as a curried function:

fun dotprod [] [] = 0.0
    | dotprod (h1::t1) (h2::t2) =
        h1*h2 + dotprod t1 t2;

Matrix multiplication:

fun matmult (Arows, Brows) =
    let val cols = transp Brows
    in map (fn row => map (dotprod row) cols)
        Arows
    end;
foldl and foldr are built-in functionals which can be defined as:

fun foldl f e [] = e
  | foldl f e (h::t) =
      foldl f (f(h,e)) t;

fun foldr f e [] = e
  | foldr f e (h::t) =
      f(h, foldr f e t);

These can be used to give simple definitions of many list functions

foldl op+ 0 sum
foldl (fn (_,n) => n+1) 0 length
foldr op:: xs ys xs@ys
fun exists p [] = false
  | exists p (h::t) = (p h) orelse
    exists p t;

fn : ('a -> bool) -> 'a list -> bool

Determines whether there is any element in a list
that satisfies the predicate p.

fun filter p [] = []
  | filter p (h::t) = if p h then
    h::(filter p t)
  else filter p t;

fn : ('a -> bool) -> 'a list -> 'a list

Anuj Dawar
University of Cambridge Computer Laboratory, January 7, 2003
Lazy Lists

Lists where the next element is computed on demand.

Also known as *streams* in the literature.

(not to be confused with input/output streams)

- Avoids waste if the entire list is not needed.
- Infinite objects are a useful abstraction.
- Models simple I/O.

Disadvantage: Termination is lost
() — the unique tuple of length 0 has type unit.

A datatype for lazy lists:

```ml
datatype 'a seq = Nil
  | Cons of 'a * (unit -> 'a seq);
```

Cons(h, tf) is the sequence with head h and tail function tf.

Example: the infinite sequence k, k + 1, k + 2, ...

```ml
fun from k = 
  Cons(k, fn () => from(k+1));
val from = fn : int -> int seq
```
exception Empty;

fun hdq Nil = raise Empty
  | hdq (Cons(h,_)) = h;

fun tlq Nil = raise Empty
  | tlq (Cons(_,tf)) = tf();

from 1;
> val it = Cons(1, fn) : int seq
- tlq it;
> val it = Cons(2, fn) : int seq
- tlq it;
> val it = Cons(3, fn) : int seq
fun \texttt{takeq} \ (0,s) \ = \ []
| \texttt{takeq} \ (n,\texttt{Nil}) \ = \ []
| \texttt{takeq} \ (n,\texttt{Cons(h,tf)}) =
\hspace{1cm} h::\texttt{takeq}(n-1,tf());

\texttt{takeq} takes an integer \( n \) and a sequence \( s \) and
produces a list of the first \( n \) elements of \( s \)

fun \texttt{squares} \ \texttt{Nil} :\texttt{int seq} = \texttt{Nil}
| \texttt{squares} \ (\texttt{Cons(h,tf)}) =
\hspace{1cm} \texttt{Cons(h*h,fn () => squares (tf()))};

- \texttt{squares} (from 1);
> \texttt{val it} = \texttt{Cons(1, fn)} : \texttt{int seq}
- \texttt{takeq} (5,\texttt{it});
> \texttt{val it} = \texttt{[1, 4, 9, 16, 25]} : \texttt{int list}
Joining Two Sequences

fun appendq (Nil, yf) = yf
  | appendq (Cons(h,tf),yf) =
      Cons(h, fn() => appendq(tf(),yf));

appendq: 'a seq * 'a seq -> 'a seq

If the first sequence is infinite,
appendq(x,y) = x

fun interleave (Nil, yf) = yf
  | interleave (Cons(h,tf),yf) =
      Cons(h, fn() => interleave(yf,tf()));
fun mapq f Nil = Nil
    | mapq f (Cons(h,tf)) =
        Cons(f h, fn() => mapq f (tf()));

- mapq (fn x => x*x) (from 1);
> val it = Cons(1, fn) : int seq
- takeq(5,it);
> val it = [1, 4, 9, 16, 25] : int list

fun filterq p Nil = Nil
    | filterq p (Cons(h,tf)) =
        if p h then
            Cons(h,fn() => filterq p (tf()))
        else filterq p (tf());
Searching Infinite Trees

Many problem solving applications involve searching a very large search space, structured as a tree.

Usually the tree is not available in its entirety, but the nodes are generated as needed.

We represent such a tree as a function of type

\[
\text{next} : \alpha \rightarrow \alpha \text{ list}
\]

Different search strategies may be appropriate.

- *Depth First Search* searches the entire sub-tree rooted at \( a \) before proceeding to its sibling.

- *Breadth First Search* searches all nodes at a given level before proceeding to the next level.
Depth First Search

Fast, when it works

Requires space proportional to the *height* of the tree.

If the tree is infinite, it may not find a solution even if one exists.

```ml
fun depth next x = 
  let fun dfs [] = Nil
  | dfs (h::t)=
      Cons(h,fn() => dfs((next h)@t))
  in dfs [x] end;

fn : ('a -> 'a list) -> 'a -> 'a seq
```
Breadth First Search

Guaranteed to find a solution in finite time, if one exists.

Finds the nearest solution first.

Requires space proportional to the size of the tree.

fun breadth next x =
  let fun bfs [] = Nil
  | bfs (h::t) =
    Cons(h, fn() => bfs(t@(next h)))
  in bfs [x] end;

fn : (’a -> ’a list) -> ’a -> ’a seq
Testing and Verification

We wish to establish that a program is correct. That is to say, it meets its specifications.

Testing

Try a selection of inputs and check against results

There is no guarantee that all bugs will be found. Behaviour of a program is not continuous

Verification

Prove that the program is correct.

Proofs can be long, tedious and complicated. Functional programs are easier to do proofs with.
Formal vs. Rigorous proof

Formal Proof:

- what logicians study
- purely symbolic
- needs an axiom system
- needs machine support

Rigorous Proof:

- what mathematicians do
- in natural language (with some symbols)
- needs clear foundations
- understandable to people

A rigorous proof is a convincing mathematical argument.
Assumptions

In order to reason with ML programs, we make some assumptions about the expressions that will be used in proofs.

- Expressions are purely functional.
- Expressions are well-formed and well-typed.
- Types are interpreted as sets.
- Execution of programs always terminates.

```ml
fun undef(x) = undef(x + 1);
```

`factorial` terminates for \( n \geq 0 \).
We have a property $\phi(n)$ we wish to prove for all integers $n \geq 0$.

Prove:

- $\phi(0)$; and
- $\phi(k)$ implies $\phi(k + 1)$, for all $k$.

This proves $\phi(n)$ for all $n$.

For any particular $n$, $\phi(n)$ can be derived in $n$ steps.
Two Factorial Functions

fun fact n = 
    if n = 0 then 1 else n*(fact n-1);

Or, a more efficient, iterative, version

fun facti (n,p) = 
    if n=0 then p else
        facti(n-1, n*p);

Are the two functions equivalent?
Can we show that for all \(n\):

\[
\text{facti}(n,1) = \text{fact}(n)
\]
Inductive Proof

We make the stronger inductive hypothesis.

\[ \forall p \text{ facti}(n, p) = \text{fact}(n) \times p \]

Base case:

\[ \text{facti}(0, p) = p \quad \text{defn. of facti} \]
\[ = 1 \times p \]
\[ = \text{fact}(0) \times p \quad \text{defn. of fact} \]

Induction Step:

\[ \text{facti}(k + 1, p) = \text{facti}(k, (k + 1) \times p) \]
\[ = \text{fact}(k) \times (k + 1) \times p \]
\[ = \text{fact}(k + 1) \times p \]
Complete Induction

To show that $\phi(n)$ is true for all $n \geq 0$, it is also sufficient to show that:

$$(\forall i < k \phi(i)) \text{ implies } \phi(k)$$

- no separate base case is necessary
- $\phi(n)$ is still provable in $n$ steps
- a more general form of proof by induction
fun power(x,k) = 
    if k=1 then x 
    else if k mod 2 = 0 then 
        power(x*x, k div 2) 
    else x*power(x*x, k div 2); 

Prove by induction (on \( k \)) that for all \( k \geq 1 \):

\[ \forall x \ power(x, k) = x^k \]
if $k = 1$ then $\text{power}(x, 1) = x = x^1$

if $k = 2j$ then

$$\text{power}(x, k) = \text{power}(x, 2j)$$
$$= \text{power}(x^2, j)$$
$$= (x^2)^j$$
$$= x^{2j}$$

if $k = 2j + 1$ then

$$\text{power}(x, k) = \text{power}(x, 2j + 1)$$
$$= x \times \text{power}(x^2, j)$$
$$= x \times (x^2)^j$$
$$= x^{2j+1}$$
Structural Induction

To prove a property $\phi(l)$ for all lists $l$, it suffices to show:

- base case: $\phi([])$
- induction step: for any $h$ and $t$, if $\phi(t)$, then $\phi(h :: t)$

$\phi([x_1, \ldots, x_n])$ can be proved in $n$ steps.

Similarly for other recursive datatypes.
Some List Functions

fun app([], l) = l
  | app(h::t, l) = h::app(t, l);

fun nrev [] = []
  | nrev (h::t) = (nrev t)@[h];

fun revto([], l) = l
  | revto(h::t, l) = revto(t,h::l);

We can prove properties such as:

\[
\text{app(app}(l_1, l_2), l_3) = \text{app}(l_1, \text{app}(l_2, l_3))
\]

\[
\text{revto}(l_1, l_2) = \text{nrev}(l_1)@[l_2]
\]
Append is Associative

For all \( l1, l2 \) and \( l3 \):
\[
\operatorname{app}(\operatorname{app}(l1, l2), l3) = \operatorname{app}(l1, \operatorname{app}(l2, l3))
\]

By structural induction on \( l1 \).

Base case:
\[
\operatorname{app}(\operatorname{app}([], l2), l3) = \operatorname{app}(l2, l3) = \operatorname{app}([], \operatorname{app}(l2, l3))
\]

Induction Step:
\[
\operatorname{app}(\operatorname{app}(h :: t, l2), l3) = \operatorname{app}(h :: \operatorname{app}(t, l2), l3) = h :: \operatorname{app}(\operatorname{app}(t, l2), l3) = h :: \operatorname{app}(t, \operatorname{app}(l2, l3)) = \operatorname{app}(h :: t, \operatorname{app}(l2, l3))
\]
\forall l2 \ revto(l1, l2) = nrev(l1)@l2

By structural induction on \( l1 \).

Base case:

\[
\text{revto}([], l2) = l2 = []@l2 = nrev[]@l2
\]

Induction Step:

\[
\begin{align*}
\text{revto}(h :: t, l2) &= \text{revto}(t, h :: l2) \\
&= nrev(t)@(h :: l2) \\
&= nrev(t)@[h]@l2 \\
&= nrev(h :: t)@l2
\end{align*}
\]
Other Examples

\[
\begin{align*}
n\text{length}(l1@l2) &= (n\text{length }l1) + (n\text{length }l2) \\
n\text{rev}(l1@l2) &= (n\text{rev }l2)@n\text{rev }l1 \\
n\text{rev}(n\text{rev }l) &= l \\
l@[] &= l \\
(map f) \circ (map g) &= map(f \circ g)
\end{align*}
\]

Correctness preserving program transformations.
Structural Induction on Trees

datatype 'a tree =
    Lf
    | Br of 'a*'a tree*'a tree

To show $\phi(t)$ for all trees $t$, it suffices to show:

- $\phi(Lf)$ Base case
- For any $t_1$, $t_2$ and $x$:
  if $\phi(t_1)$ and $\phi(t_2)$ then $\phi(Br(x, t_1, t_2))$. Induction step

$\phi(t)$ can be proved in $\text{size}(t)$ steps.
fun preorder Lf = []
   | preorder (Br(x,t1,t2)) =
       x::preorder(t1)@preorder(t2);

fun preord (Lf, l) = l
   | preord (Br(x,t1,t2),l) =
       x::preord(t1, preord (t2,l));

We can show that the two are equivalent:

\[ \forall l \text{ preorder}(t, l) = \text{preorder}(t)@l \]

By induction on \( t \)
Introduction to Functional Programming

Preorder

Base Case:

\[ \text{preord}(\text{Lf}, l) = l = [] @ l = \text{preorder}(\text{Lf}) @ l \]

Induction Step:

\[
\text{preord}(\text{Br}(x, t_1, t_2), l) \\
= x :: \text{preord}(t_1, \text{preord}(t_2, l)) \\
= x :: \text{preorder}(t_1) @ \text{preorder}(t_2, l) \\
= x :: \text{preorder}(t_1) @ \text{preorder}(t_2) @ l \\
= \text{preorder}(\text{Br}(x, t_1, t_2)) @ l
\]
Induction Principles

Structural Induction works well on functions whose recursive definition follows the definition of the datatype.

Structural Recursion.

Some functions follow other recursive patterns.

mergesort works by splitting its list in two.

We can show that $\phi(l)$ holds for all lists $l$ by showing that:

$$\text{if } \forall l' \text{ length}(l') < \text{length}(l) \rightarrow \phi(l') \text{ then } \phi(l)$$

More generally, well-founded induction works on any well-ordered set
Types help us rule out certain programs that don’t seem to make sense.

Can we apply a function to itself $f \circ f$?

Sometimes, it might be sensible:

$$\text{fn } x \Rightarrow x \text{ or } \text{fn } x \Rightarrow y.$$  

But in general it looks very suspicious.

This sort of self-application can lead to inconsistencies in formal logics.

Russell’s paradox considers $\{x \mid x \not\in x\}$.

To avoid this, Russell introduced a system of types.

Type theory is an *alternative* to set theory as a foundation for mathematics.

There are interesting links between type theory and programming.
Programming reasons for types

- More efficient code, and more effective use of space.
- ‘Sanity check’ for programs, catching a lot of programming errors before execution.
- Documentation.
- Data hiding and modularization.

At the same time, some programmers find them an irksome restriction. How can we achieve the best balance?
We can distinguish between

- **Strong typing**, as in Modula-3, where types must match up exactly.
- **Weak typing**, as in C, where greater latitude is allowed.

and also between

- **Static typing**, as in FORTRAN, which happens at compilation
- **Dynamic typing**, as in LISP, which happens during execution.

ML is statically and strongly typed.

At the same time, *polymorphism* gives great flexibility.
A type expression is:

- **One of the basic types**: `unit`, `bool`, `int`, `real`, `string`.
- **A type variable** \( \alpha \).
- \( \sigma \rightarrow \tau \)
- \( \sigma \times \tau \)
- \( \sigma \text{ list} \)

Where \( \sigma \) and \( \tau \) are any type expressions.
Polymorphism

Some functions can have various different types — *polymorphism*. We distinguish between:

- True ('parametric') polymorphism, where all the possible types for an expression are instances of some schematic type.
  \[ \text{fn } x \Rightarrow x \]

- Ad hoc polymorphism, or *overloading*, where this isn’t so.
  \[ + \]

Overloading is limited to a few special cases.

Except for overloading, the ML type inference system can infer a type for every expression.
Type variables

If an expression has a type involving a variable \( \alpha \) then it can also be given any type that results from consistently replacing \( \alpha \) by another type expression.

A type \( \sigma \) is more general than \( \tau \), \( (\sigma \leq \tau) \), when we can substitute types for type variables in \( \sigma \) and get \( \tau \). For example:

\[
\begin{align*}
\alpha & \leq \text{bool} \\
\beta & \leq \alpha \\
(\alpha \to \alpha) & \leq (\text{int} \to \text{int}) \\
(\alpha \to \alpha) & \not\leq (\text{int} \to \text{bool}) \\
(\alpha \to \beta) & \leq (\beta \to \beta) \\
(\alpha \to \beta) & \not\leq \alpha
\end{align*}
\]
Most general types

Every expression in ML that has a type has a most general type.

(Hindley/Milner)

There is an algorithm for finding the most general type of any expression, even if it contains no type information at all.

ML implementations use this algorithm.

Thus, except for overloading, it is never necessary to write down a type.

This makes the ML type system much less onerous than some others.

For every pair of type expressions $\sigma$ and $\tau$, if they can be unified, there is a most general unifier, up to re-naming of type variables.
ML type inference (1)

Example:

\[
\begin{align*}
\text{fn } a &\Rightarrow (\text{fn } f \Rightarrow \text{fn } x \Rightarrow f \ x) \ (\text{fn } x \Rightarrow x) \\
\text{Attach distinct type variables to distinct variables in the expression.}
\end{align*}
\]

\[
\begin{align*}
\text{fn } (a:\alpha) &\Rightarrow (\text{fn } (f:\beta) \Rightarrow \text{fn } (x:\gamma) \Rightarrow (f:\beta) \ (x:\gamma)) \ (\text{fn } (x:\delta) \Rightarrow (x:\delta)) \\
\text{Note:}
\end{align*}
\]

- previously defined constants get their assigned type.
- distinct instances of polymorphic constants get assigned types with distinct type variables.
- different bindings of the same variable are really different variables.
ML type inference (2)

$f \; x$ can only be well-typed if $f : \sigma \rightarrow \tau$ and $x : \sigma$
for some $\sigma$ and $\tau$.

$(f \; x) : \tau$.

\texttt{fn (x:}\beta) => \texttt{E:}\gamma \text{ has type } \beta \rightarrow \gamma.

Using these facts, we can find relations among the type variables.

Essentially, we get a series of simultaneous equations, and use them to eliminate some unknowns.

The remaining unknowns, if any, parametrize the final polymorphic type.

If the types can’t be matched up, or some type variable has to be equal to some composite type containing itself, then typechecking fails.

It is a form of term \textit{unification}.
To solve a list of equations:

\[ t_1 = s_1, \ldots, t_n = s_n \]

- if \( t_1 \) is a variable, replace all occurrences of it in the rest of the list by \( s_1 \);
- otherwise, if \( s_1 \) is a variable, replace all occurrences of it in the rest of the list by \( t_1 \);
- otherwise, if they are distinct constants, then fail;
- otherwise, if they have distinct head constructors (\( \rightarrow, \ast, \text{list} \)) then fail;
- otherwise, add to the list the list of equations of their corresponding parts;

Repeat until all equations are processed.
For \((f: \beta) \ (x: \gamma)\) to be well-typed,

For some \(\epsilon\)
\[\beta = \gamma \rightarrow \epsilon.\]

\[(\text{fn } f \Rightarrow \text{fn } x \Rightarrow f \ x):(\gamma \rightarrow \epsilon) \rightarrow (\gamma \rightarrow \epsilon)\]

and this is applied to
\[(\text{fn } x \Rightarrow x):\delta \rightarrow \delta\]

So \((\gamma \rightarrow \epsilon) = (\delta \rightarrow \delta)\) and
\[\gamma = \delta \text{ and } \epsilon = \delta\]

So, the whole expression has type \(\alpha \rightarrow (\delta \rightarrow \delta).\)

It doesn’t matter how we name the type variables now, so we can call it \(\alpha \rightarrow (\beta \rightarrow \beta).\)
Let polymorphism

Let allows us to have local bindings
let val v = E in E’ end.

This is like (fn v => E’) E, but type-checking this way leads to a problem,

If v is bound to something polymorphic, multiple instances must be allowed to have different types.

let val I = fn x => x
  in if I true then I 1 else 0
  end;

One way to type-check this is by substitution of the bound variable.

There are more efficient ways.
Type preservation

Typechecking ML is static, completed before evaluation.

Evaluation of well-typed expressions cannot give rise to ill-typed expressions.  

The main step in evaluation is:

\[(\text{fn } x \Rightarrow t[x]) \ u \Rightarrow t[u]\]

\(x\) and \(u\) must have the same types at the outset, so this preserves typability.

The reverse is not true,

\[(\text{fn } a \Rightarrow \text{fn } b \Rightarrow b) (\text{fn } x \Rightarrow x \ x)\]
fn a => let fun pair x y = fn z => z x y
    val x1 = fn y => pair y y
    val x2 = fn y => x1(x1 y)
    val x3 = fn y => x2(x2 y)
    val x4 = fn y => x3(x3 y)
    val x5 = fn y => x4(x4 y)
    in x5 (fn z => z)
end;

The type of this expression takes about a minute to calculate, and when printed out takes 50,000 lines.
Case Study: Parsing

- Recursive Descent Parsing.
- Parsing functionals that closely correspond to the rules of a grammar.
- Functionals that correspond to concatenation, alternatives, repetition, etc.
- Easy to allow backtracking.
- Construct syntax trees to which it is easy to attach semantics.
- Similar functionals in other domains.

_caveat:_ The functions are easy to write - the parsing algorithms are the same.
A lexical analyser converts the string of symbols into a list of tokens.

```plaintext
datatype token = Key of string | Id of string;
```

Key for keywords, and Id for identifiers.

A parser is a function of type:

```plaintext
type 'a phrase =
    token list -> 'a * token list;
```

The parser removes tokens from the front of the list that match the rule of type 'a.
Basic Parsers

"string" : string phrase

id : string phrase

empty : 'a list phrase

exception Error of string;

fun id (Id a::toks) = (a, toks)
  | id toks =
      raise Error "Identifier expected";

fun empty toks = ([], toks);
Alternatives and Consecutives

infix 0 || ;

fun (ph1 || ph2) toks =
  ph1 toks handle Error _ => ph2 toks;

infix 5 -- ;

fun (ph1 -- ph2) toks =
  let val (x, toks2) = ph1 toks
  val (y, toks3) = ph2 toks2
  in ((x,y), toks3) end;
infix 3 >> ;

fun (ph >> f) toks =  
  let val (x, toks2) = ph toks  
  in (f x, toks2) end;

fun repeat ph toks =  
  (ph -- repeat ph >> (op::)  
    || empty) toks;
Propositional Logic 1

Grammar rules:

\[
\begin{align*}
Prop & = \ Term \{ "|" \ Term \} \\
Term & = \ Factor \{ "\&" \ Factor \} \\
Factor & = \ Id \mid "\~" \ Factor \mid "(" \ Prop\)"
\end{align*}
\]

fun orl(p, []) = p
   | orl(p,(_,q)::l) = orl(Disj(p,q), l);

fun andl(p, []) = p
   | andl(p,(_,q)::l) = andl(Conj(p,q), l);
fun prop toks = 
  (term -- repeat($"|" -- term)
     >> orl) toks;

fun term toks = 
  (factor -- repeat($"&" -- factor)
     >> andl) toks;

fun factor toks = 
  (id >> Atom
     || $"~" -- factor
        >>> (fn (_,p) => Neg p)
     || "$"(" -- prop -- ")"
        >>> (fn ((_,p),_) => p)
  ) toks;
The parser yields 1 outcome, or an error.

We can obtain a backtracking parser by replacing the exception by a lazy list of outcomes.

[]
[(x, toks)]
[(x1, toks1), (x2, toks2),...]

Depending on whether the string of tokens is a syntax error, a unique parse, or an ambiguous parse.

-- and || can be modified to combine lazy lists.

The lazy consumption of outcomes leads to backtracking.