Introduction to Functional Programming

Marcelo Fiore

Computer Laboratory
University of Cambridge
Lent 2008

References

Main:

Other:

Lecture I

Keywords:
functional programming; expressions and values; functions; recursion; types.

References:
♦ [MLWP, Chapter 1]

Programming

♦ Programming is an intellectual activity.

It is somehow close to proving theorems in mathematics (cf., analysis of algorithms, program verification).

♦ Programming is hard.

Software is notoriously unreliable. We need all the tools, principles, etc. that we can have to aid programming and thinking about it.
Why Functional Programming?

- Offers a novel way of thinking about programming.
  - Highlights expressiveness and clarity.
- Suitable for quick, easy, reliable, etc. prototyping.
  - Security via type discipline.
- Susceptible to program correctness and/or verification.
  - Ease of mathematical reasoning about programs.

Functional Programming

Input/Output-based computation (= Mathematical style):
A functional program is an expression, and executing a program amounts to evaluating the expression to a value.

Features:
- No state (⇒ no memory cells and no assignment).
- No side effects.
- Referential transparency: One may replace equals by equals.
- Higher-order: Functions are first-class values.
- Static, strong, polymorphic typing.

Imperative Programming

State-based computation (= von Neumann style):
Imperative programs rely on modifying a state by using commands.
Programs are instructions specifying how to modify the state.

Commands
- Assignment
- Control
  - Sequencing
  - Conditionals
  - Iteration

Imperative vs. Functional

Factorial

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

given

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

Advantages:

- Clearer semantics: programs correspond more directly to abstract mathematical objects.
- Conciseness and elegance: programs are shorter.
- Type system assists in the detection of errors and aids rapid prototyping.
- Better parametrisation and modularity of programs.
- Freedom in implementation; *e.g.*, parallelisation, lazy evaluation.

Disadvantages:

- Some programming needs are harder to fit into a purely functional model; *e.g.*, input/output modes, interactivity and continuously running programs (operating systems, process controllers).
- Historically functional languages have been less efficient than imperative ones; better compilers and runtime systems have largely closed the performance gap.

Imperative vs. Functional

<table>
<thead>
<tr>
<th>State-based computation</th>
<th>Input/Output-based computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing</td>
<td>Composition</td>
</tr>
<tr>
<td>Iteration</td>
<td>Recursion</td>
</tr>
<tr>
<td>Datatypes</td>
<td>Structured datatypes</td>
</tr>
<tr>
<td>—</td>
<td>Higher-order</td>
</tr>
</tbody>
</table>

Difficulties

Some standard responses:

- “It’s too hard.”
- “My employer doesn’t use it.”
- “Programs don’t run as fast as in C.”
- “I hate and/or don’t understand all those type errors.”
- “I want to do garbage collection/memory management myself.”

**NB:** You will most surely need to change your way of thinking about programming.
Expressions

Expressions have a recursive, tree-like, structure. They are built-up from operators and arguments, by means of applications.

Examples:
1. \( \text{fact}(1+(2\cdot3)) \)
2. \( \text{fact}(\text{fact}(4))+1 \)
3. \( 1 = 1+1 \)

In the context of pure expressions (i.e., in the absence state change or side-effects), an expression always evaluates to the same value, and can thus be replaced by that value without affecting the program. This is called referential transparency.

Functions

Expressions consist mainly of function applications.

Functions may take any type of argument and return any type of result; ‘any type’ includes functions themselves—which are treated like other data.

Example:

\[
\text{fun doubleORsquare } n \\
= \begin{cases} 
\text{if } n \geq 0 \text{ then } \text{op}+ \text{ else } \text{op}^* \end{cases}(n,n)
\]

Recursion

Recursive definition of functions is crucial to functional programming; there is no other mechanism for looping!

Examples:
1. \( \text{fun gcd } (m,n) \)
   \[
   = \begin{cases} 
   \text{if } m = 0 \text{ then } m \text{ else } \text{gcd}(n \mod m,m) 
   \end{cases}
   \]
2. \( \text{fun even} (n) \)
   \[
   = \begin{cases} 
   \text{if } n = 0 \text{ then } \text{true} \text{ else } \text{odd}(n-1) 
   \end{cases}
   \]
   and \( \text{odd} (n) \)
   \[
   = \begin{cases} 
   \text{if } n = 0 \text{ then } \text{false} 
   \end{cases}
   \]
   else if \( n = 1 \) then true
   else even(n-1)

Static, strong, polymorphic typing

Types classify data and let us ensure that they are used sensibly.

ML provides static (i.e., compile-time), strong, polymorphic type checking, which can help catch programming errors. Polymorphism abstracts the types of parametric components.

Types are inferred automatically by the interpreter or compiler. Typically, type declarations are not required.
This course

- Basic types and tuples.
- Functions and recursion.
- List manipulation.
- Higher-order functions.
- Sorting.
- Abstraction and modularisation.
- Recursive Datatypes.
- Searching.
- Exceptions.
- Trees.
- Lazy lists.
- Types and type inference.
- Reasoning about functional programs.
- Case studies.

Lecture II

Keywords:
mosml; sml; value declarations; static binding; basic types (integers, reals, truth values, characters, strings);
function declarations; overloading; tuples; recursion;
expression evaluation; call-by-value.

References:

- [MLWP, Chapter 2]
- SML/NJ (http://www.smlnj.org/)
- Moscow ML (http://www.dina.dk/~sestof/mosml.html)

Running ML

$ mosml
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
-
%

Standard ML of New Jersey v110.57
-

Most functional languages are interactive:

- val pi = 3.14159 ;
> val pi = 3.14159 : real
- val area = pi * 2.0 * 2.0 ;
> val area = 12.56636 : real
-

A declaration gives something a name, or binds something to a name. In ML many things can be named: values, types, ...
Static binding

If a name is redeclared then the new meaning is adopted afterwards, but does not affect existing uses of the name.

```ml
val pi = 3.14159;
val radius = 2.0;
val area = pi * radius * radius;
val pi = 0.0;
area;
~"p01"
- use"p01";
```

Read a file into ML

[opening file "p01"]
> val pi = 3.14159 : real
> val radius = 2.0 : real
> val area = 12.56636 : real
> val pi = 0.0 : real
> val it = 12.56636 : real
[closing file "p01"]

The name `it` always has the value of the last expression typed at top level.

Arithmetic

Integers and Reals

- The type of integers: `int`.
  Constants: `...`, `-2`, `-1`, `0`, `1`, `2`, ...
  Built-in operators and functions: Try the following in ML
    - load"Int"; (* needed in mosml, but not in sml *)
    - open Int;
- The type of reals: `real`.
  Constants:
    `...`, `-1E6`, `-1.41`, `-1E-10`, `1E-10`, `1.41`, `1E6`, ...
  Built-in operators and functions: Try the following in ML
    - load"Real";open Real;
    - load"Math";open Math;

Truth values

The type of booleans: `bool`.
Constants: `false` `true`

Built-in operators and functions:
Try the following in ML
- load"Bool";open Bool;
Characters and strings

- The type of characters: `char`.
- Constants: `#"A", #"a", ..., #"1", ..., #" ", ..., #"\n"
- Built-in operators and functions:
  Try the following in ML: `load"Char"`; `open Char`;

- The type of strings: `string`.
- Constants:
  "", " ", "A", "z", "0 a 1 A ... 0 z 1 Z"
  "Bye, bye ... \n"
- Built-in operators and functions:
  Try the following in ML: `load"String"`; `open String`;

Overloading

Certain built-in operators are overloaded, having more than one meaning. For instance, `+` and `*` are defined both for integers and reals.

The type of an overloaded function must be determined from the context; occasionally types must be stated explicitly.

- `fun int_square (x:int) = x * x`;
- `val int_square = fn : int -> int`

NB: SML'97 defines a notion of default type. The SML compiler will resolve the overloading in a predefined way; relying on this is bad programming style.

- `fun default_square x = x * x`;
- `val default_square = fn : int -> int`

Declaring functions

Declaring functions

- `val pi = 3.14159`;
- `fun square (x:real) = x * x`;
- `fun area (radius) = pi * square(radius)`;
- `area (0.5)`;
- `val pi = 0.0`;
- `area 0.5`;
- ~"p02"

Charlotte's note

- `fun int_square (x:int) = x * x`;
- `val int_square = fn : int -> int`

Declaring functions

Conditional expressions

To define a function by cases —where the result depends on the outcome of a test— we employ a conditional expression.

- `fun sign n = if n>0 then 1 else if n=0 then 0 else ~1`;
- `fun absval x = if x >= 0.0 then x else ~x`;

Charlotte's note

- `fun int_square (x:int) = x * x`;
- `val int_square = fn : int -> int`

Declaring functions

Conditional expressions

To define a function by cases —where the result depends on the outcome of a test— we employ a conditional expression.

- `fun sign n = if n>0 then 1 else if n=0 then 0 else ~1`;
- `fun absval x = if x >= 0.0 then x else ~x`;

Charlotte's note

- `fun int_square (x:int) = x * x`;
- `val int_square = fn : int -> int`
The boolean infix operators \andalso\ and \orelse\ are not functions, but stand for conditional expressions:
\begin{itemize}
  \item E1 \andalso E2 \equiv if E1 then E2 else false
  \item E1 \orelse E2 \equiv if E1 then true else E2
\end{itemize}

\* The \textbf{empty tuple} is given by () which is of \textbf{unit} type:
\begin{verbatim}
  - ();
  > val it = () : unit
\end{verbatim}

\* The components of a non-empty tuple can be \textbf{selected} (or \textbf{projected}).

\* With functions, tuples give the effect of multiple arguments and/or results.

\textbf{Tuples}

A \textit{tuple} is an ordered, possibly empty, collection of values.

The tuple whose components are \(v_1, \ldots, v_n\) \((n \geq 0)\) is written \(\langle v_1, \ldots, v_n \rangle\).

\* A tuple is constructed by an expression of the form \((E_1, \ldots, E_n)\).

\begin{center}
\begin{tabular}{|l|}
\hline
If \(E_1\) has type \(\tau_1\), and \ldots, \(E_n\) has type \(\tau_n\)
then \((E_1, \ldots, E_n)\) has type \(\tau_1 \times \ldots \times \tau_n\).  \\
\hline
\end{tabular}
\end{center}

In particular, the \textbf{unit} type is often used with procedural programming in ML.

A \textit{procedure} is typically a ‘function’ whose result type is \textbf{unit}. The procedure is called for its effect; not for its value, which is always (). For instance,
\begin{verbatim}
  - use;
  > val it = fn : string \rightarrow unit
  - load; (** in mosml ***)
  > val it = fn : string \rightarrow unit
\end{verbatim}
**Complex numbers**

load"Math"; (* needed in mosml, but not in sml *)
type complex = real * real;  
val origin = ( 0.0 , 0.0 ) : complex;
fun X( (x,y):complex ) = x;
fun Y( (x,y):complex ) = y;
fun norm v = Math.sqrt( X(v)*X(v) + Y(v)*Y(v) ) ;
fun scalevec( r , v ) = ( r*X(v) , r*Y(v) ) ;
fun normal v = scalevec( 1.0/(norm v) , v ) ;

"p04"

**Declaring functions**

**Infix operators**

An *infix operator* is a function that is written between its two arguments.

infix xor ;  (* exclusive or *)
fun ( p xor q )
   = ( p orelse q ) andalso not( p andalso q ) ;
true xor false xor true ;

"p05"

> infix 0 xor
> val xor = fn : bool * bool -> bool
> val it = false : bool

In ML the keyword *op* overrides infix status:

- op xor;
> val it = bool * bool -> bool
- op xor ( true , false ) ;
> val it = true : bool
Declaring functions
Recursion

Examples

♦ Factorial

fun fact n
    = if n = 0 then 1
    else n * fact(n - 1);
> val fact = fn : int -> int

♦ Greatest Common Divisor

fun gcd(m, n)
    = if m = 0 then n
    else gcd(n mod m, m);
> val gcd = fn : int * int -> int

♦ Power-of-two test

fun powoftwo n
    = (n=1) or else (n mod 2 = 0) andalso powoftwo(n div 2);
> val powoftwo = fn : int -> bool

♦ Fibonacci numbers

\[
F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-2} + F_{n-1} \quad (n \geq 2)
\]

fun nextfib(Fn,Fsuccn) : int * int
    = (Fsuccn,Fn+Fsuccn);
> val nextfib = fn : int * int -> int * int
> val fibpair = fn : int -> int * int

♦ Mutual recursion

Examples

♦ \pi

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{1}{4k+1} - \frac{1}{4k+3} + \cdots
\]

fun pos k
    = if k < 0 then 0.0
    else (if k = 0 then 0.0 else neg(k-1))
        + 1.0/real(4*k+1)
and neg k
    = if k < 0 then 0.0
    else pos(k) - 1.0/real(4*k+3);
> val pos = fn : int -> real
> val neg = fn : int -> real
Parity test

```haskell
fun even n
    = n = 0 orelse odd( n-1 )
and odd n
    = n<>0 andalso ( n = 1 orelse even( n-1 ) )
> val even = fn : int -> bool
val odd = fn : int -> bool
```

The evaluation rule in ML is **call-by-value** (or **strict** evaluation).

### Call-by-value evaluation

To compute the value of $F(E)$, first compute the value of the expression $F$ to obtain a function value, say $f$. Then compute the value of the expression $E$, say $v$, to obtain an actual argument for $f$. Finally compute the value of the expression obtained by substituting the value $v$ for the formal parameter of the function $f$ into its body.

**NB:** Most purely functional languages adopt **call-by-name** (or **lazy** evaluation).

The manual evaluation of expressions is helpful when understanding and/or debugging programs.

---

**Evaluation of expressions**

Execution is the **evaluation** (or **reduction**) of an expression to its value, replacing equals by equals.

### Evaluation of conditionals

To compute the value of the conditional expression `if E then E1 else E2`, first compute the value of the expression $E$. If the value so obtained is `true` then return the value of the computation of the expression $E1$; otherwise, return the value of the computation of the expression $E2$.

### Examples

1. fun minORmax b
   = (if b then Int.min else Int.max)( 1+3,2 )
   
   minORmax true
   ~ (if true then Int.min else Int.max)( 1+3,2 )
   ~ Int.min
   ~ (1+3,2)
   ~ 1+3 ~ 4
   ~ (4,2)
   ~ Int.min(4,2)
   ~ 2
In this vein, thus

\[
\text{fact(3)} \\
\leadsto \text{if 3 = 0 then 1 else 3 \ast fact(3-1)} \\
\leadsto 3 \ast \text{fact(3-1)} \\
\leadsto 3 \ast \text{fact(2)} \\
\leadsto 3 \ast ( \text{if 2 = 0 then 1 else 2 \ast fact(2-1)} ) \\
\leadsto 3 \ast ( 2 \ast \text{fact(2-1)} ) \\
\leadsto 3 \ast ( 2 \ast \text{fact(1)} ) \\
\leadsto 3 \ast ( 2 \ast ( \text{if 1 = 0 then 1 else 1 \ast fact(1-1)} ) ) \\
\leadsto 3 \ast ( 2 \ast ( 1 \ast \text{fact(1-1)} ) ) \\
\leadsto 3 \ast ( 2 \ast ( 1 \ast \text{fact(0)} ) ) \\
\leadsto 3 \ast ( 2 \ast ( 1 \ast ( \text{if 0 = 0 then 1 else 0 \ast fact(0-1)} ) ) ) \\
\leadsto 3 \ast ( 2 \ast ( 1 \ast 1 ) ) \\
\leadsto 3 \ast ( 2 \ast 1 ) \\
\leadsto 3 \ast 2 \\
\leadsto 6
\]

**NB:** Due to call-by-value, one cannot define an ML function \text{cond} such that \text{cond}(E,E1,E2) is evaluated like the conditional expression \text{if E then E1 else E2} for whatever expressions E, E1, E2.
**Types**

Every well-formed ML expression has a type. A type denotes a collection of values. All types are determined statically. Given little of no explicit type information, ML can infer all the types involved with a value or function declaration.

Via a mathematical theorem typically known as Subject Reduction, ML guarantees that the value obtained by evaluating an expression coincides with that of the evaluated expression. Thus, type-correct programs cannot suffer run-time type errors.

**Polymorphism**

In parametric polymorphism, a function (or datatype) is general enough to work with objects of different types.

A polymorphic type is a type scheme constructed from type variables and basic types (like int, real, char, string, bool, unit) using type constructors (like the product type constructor *, the function type constructor ->, etc.).

Polymorphic types represent families of types; viz. the family of instances obtained by substituting types for type variables.

**Examples:**

1. **Swapping**

   ```ml
   fun swap( x , y ) = ( y , x );
   fun int_swap( p:int*int ) = swap p ;
   fun real_swap( p:real*real ) = swap p ;
   fun unit_swap( p:unit*unit ) = swap p ;
   ``

   "poly"

   > val (‘a,’b) swap = fn : ‘a * ‘b -> ‘b * ‘a

2. **Associating**

   ```ml
   fun assocLR( ( x , y ) , z ) = ( x , ( y , z ) ) ;
   fun assocRL( x , ( y , z ) ) = ( ( x , y ) , z ) ;
   
   "poly"

   > val (‘a,’b,’c) assocLR = fn :
       (‘a * ’b) * ’c -> ’a * (’b * ’c)
   > val (‘a,’b,’c) assocRL = fn :
       ’a * (’b * ’c) -> (’a * ’b) * ’c
   ```
Declaring functions

Curried functions

Since an ML function can have only one argument, functions
taking more than one argument have so far corresponded to
ML functions taking tuples.

However, functions admitting multiple arguments can also be
realised by the process of *currying* them, to produce another
function that takes each of its arguments in turn returning a
function as result.

**Examples:**

1. Ternary multiplication

   ```
   fun termult(a, b, c) = a * b * c : int;
   fun curried_termult a b c = termult(a, b, c);
   ``

   "curry"

   ```
   > val termult = fn : int * int * int -> int
   > val curried_termult = fn : int -> int -> int -> int
   ```

   **NB:** Function application associates to the left, whilst the
   function-type constructor associates to the right.

2. Composition

   ```
   fun compose f g x = f(g x);
   fun uncurried_compose((f,g), x) = compose f g x;
   ``

   "curry"

   ```
   > val ('a,'b,'c) compose = fn : ('a->'b)->('c->'a)->'c->'b
   > val ('a,'b,'c) uncurried_compose = fn :
   ( ('a -> 'b) * ('c -> 'a) ) * 'c -> 'b
   ```

**NB:** Curried functions permit *partial evaluation:*

```
fun mult (x,y) = curried_termult 1 x y ;
fun double x = curried_termult 1 2 x ;
fun pow3 x = curried_termult x x x ;
```

"curry"

```
> val mult = fn : int * int -> int
> val double = fn : int -> int
> val pow3 = fn : int -> int
```
**Polymorphism**

*Warning*

In ML, a polymorphic type can be instantiated in multiple ways, but all type variables for a given instance must be instantiated as a group.

```ml
fun p x y = (x,y);
val p12 = p 1 2 ;
val ptrue = p 1 true ;
> val ('a,'b) p = fn : 'a -> 'b -> 'a * 'b
> val p12 = (1, 2) : int * int
> val ptrue = (1, true) : int * bool
```

```ml
fun qx = p1x;
val q2 = q2 ;
val qtrue = qtrue ;
> val 'a q = fn : 'a -> int * 'a
> val q2 = (1, 2) : int * int
> val qtrue = (1, true) : int * bool
```

```ml
- val q'2 = q' 2 ;

! Warning: the free type variable ’a has been
! instantiated to int
> val q'2 = (1, 2) : int * int
- val q’ttrue = q’ true ;

! Toplevel input:
! val q’ttrue = q’ true ;
! Type clash: expression of type
! bool
! cannot have type
! int
```

```
fun q x = p 1 x ;
val q2 = q 2 ;
val qtrue = q true ;
> val 'a q = fn : 'a -> int * 'a
> val q2 = (1, 2) : int * int
> val qtrue = (1, true) : int * bool
- val q’ = p 1 ;

! Warning: Value polymorphism:
! Free type variable(s) at top level in value
! identifier q’
> val q’ = fn : 'a -> int * 'a
```

---

**Function values**

Most functional languages give *function values* full rights, free of arbitrary restrictions. Like other values, functions may be arguments and results of other functions and may belong to other data structures (pairs, *etc.*).

**Nameless functions**

An ML function need not have a name. Indeed, the expression

```ml
fn x => E
```

is a *function value* with formal argument (or parameter) *x* and body *E*. In particular, the declarations

```ml
fun myfun x = E ; and val myfun = fn x => E ;
```

are equivalent.
Examples:

1. `fun Curry f`  
   = fn x => fn y => f(x, y);  
   `fun unCurry f`  
   = fn(x, y) => fxy;  
   "fn"  
   > val ('a,'b,'c) Curry = fn:  
      ('a*'b->'c)->'a->'b->'c  
   > val ('a,'b,'c) unCurry = fn:  
      ('a->'b->'c)->'a*'b->'c

2. `fun split f`  
   = ( fn x => ( fn(y,z)=>y ) (f x),  
       fn x => ( fn(y,z)=>z ) (f x) );  
   `fun pack(f, g)`  
   = fn x => (f x, g x);  
   "fn"  
   > val ('a,'b,'c) split = fn:  
      ('a->'b*'c)->('a->'b)*('a->'c)  
   > val ('a,'b,'c) pack = fn:  
      ('a->'b)*('a->'c)->'a->'b*'c

Lists

_A list_ is a finite sequence of elements of the same type.

For instance:

- _int list_ is the type of lists of integers,
- _string list_ is the type of lists of strings,
- _int list list_ is the type of lists whose elements are themselves lists of integers.

Examples

```plaintext
[ 1, 2, 4, 2, 1 ];  
[ "a", "b", "b", "a " ];  
[ [3,6,9], [5], [7] ];  
[] ;  
"p01"
```

> val it = [1, 2, 4, 2, 1] : int list  
> val it = ["a", "b", "b", " a "] : string list  
> val it = [[3, 6, 9], [5], [7]] : int list list  
> val 'a it = [] : 'a list
List are either of two kinds:

**Empty:**

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

**Compound:**

\[ h::t \]

with \( h \) the head of the list and \( t \) the tail

**NB:**

\( \text{::} \) is an infix operator:

\[
\begin{align*}
\text{op} & \text{:: } \\
> \text{val } \text{'a it = fn : } \text{'a } \text{'}a\text{ list } \to \text{'a list}
\end{align*}
\]

The notation \([e1, e2, \ldots, en]\) is a shorthand for \( e1 :: e2 :: \ldots :: en :: [] \).

---

**Built-in functions**

**Head:**

\[
\begin{align*}
\text{head } & \text{hd; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{'a list}
\end{align*}
\]

**Tail:**

\[
\begin{align*}
\text{tail } & \text{tl; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{'a list}
\end{align*}
\]

**Append:**

\[
\begin{align*}
\text{append } & \text{op@; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{'a list}
\end{align*}
\]

**Reverse:**

\[
\begin{align*}
\text{reverse } & \text{rev; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{'a list}
\end{align*}
\]

**Nil testing:**

\[
\begin{align*}
\text{null testing } & \text{null; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{bool}
\end{align*}
\]

**Length:**

\[
\begin{align*}
\text{length } & \text{length; } \\
> \text{val } \text{'a it = fn : } \text{'a list } \to \text{int}
\end{align*}
\]

---

**Pattern matching**

The use of data constructors as patterns allows to deconstruct data.

A pattern match takes place between a pattern\( (= \text{linear expressions built from variables, constants, and data constructors}) \) and data (built from constants and data constructors).

The variables appearing in a pattern are bound to (or unified with) the corresponding parts of the data object. (Constants require an exact match and \( _{} \) is used to match anything without producing any binding.)
Examples:
1. \texttt{val a :: b = [0,1];}
   \texttt{val [x,y] = [0,1];}
   \texttt{val [c,_] = [0,1];}
   \texttt{val [_,_] = [0,1];}

   "p02"

   \texttt{> val a = 0 : int}
   \texttt{val b = [1] : int list}
   \texttt{> val x = 0 : int}
   \texttt{val y = 1 : int}
   \texttt{> val c = 0 : int}

   \texttt{~}

   \texttt{"p02"}

2. \texttt{val [a,a] = [1,1];}

   "p05"

   \texttt{! val [a,a] = [1,1];}
   \texttt{^}

   \texttt{! Unbound value identifier: y}

Non-examples
1. \texttt{val [a,2] = [0,1];}

   "p03"

   ! Uncaught exception:
   ! Bind

2. \texttt{val [a,2] = [1,y];}

   "p04"

   ! val [a,2] = [1,y];
   ! ^

   ! Unbound value identifier: y

Deep patterns

Patterns can be as deep or as shallow as required.

They can match a value or the components that make up that value.

\texttt{val x = [ (1, ( [0,"F",false) , (1,"T",true) ] ) ] ;}
\texttt{val y :: z = x ;}
\texttt{val (a, (b,c,d) :: e) :: [] = x ;}

"p06"
Using patterns

Only one pattern can be used with `val`, but `fun`, `fn`, and `case` expressions can include multiple patterns. They are tried in order until one is successful.

Examples:

1. `fun null0 l = l = [] ;
   fun null1 [] = true
   | null1 _ = false ;
   val null2 = fn [] => true | _ => false ;
   > val `a null0 = fn `a list -> bool
   > val `a null1 = fn `a list -> bool
   > val `a null2 = fn `a list -> bool

   fun null13 [] = true ;
   null13 [] ;
   null13 [1,2,3] ;
   ~
   "p07"
   ! fun null13 [] = true ;
   ! ___________________________
   ! Warning: pattern matching is not exhaustive
   > val `a null13 = fn `a list -> bool
   > val it = true : bool
   ! Uncaught exception:
   ! Match

2. The conditional expression

   if E then E1 else E2

   abbreviates the function application

   (fn true => E1 | false => E2)(E)
Patterns in case expressions

fun null3 l
  = case l of
    [] => true
    | h::t => false;

fun null4 l
  = case l of
    [] => true;

"p08"

> val 'a null3 = fn : 'a list -> bool
!  [] => true ;
! ! -----------
! Warning: pattern matching is not exhaustive
> val 'a null4 = fn : 'a list -> bool

List manipulation

List manipulation

Examples

1. - fun length [] = 0
    | length (h::t) = 1 + length t ;
> val 'a length = fn : 'a list -> int

2. fun append [] l = l
    | append (h::t) l = h :: append t l ;
fun longreverse [] = []
    | longreverse (h::t) = append (longreverse t) [h] ;

"p09"

> val 'a append = fn : 'a list -> 'a list -> 'a list
> val 'a longreverse = fn : 'a list -> 'a list

Evaluate longreverse [0,1,2,3,4] !

List manipulation

Tail recursion

Example:

fun lastelem [ e ] = e
  | lastelem (h::t) = lastelem t ;
lastelem [ 0, 1, 2, 3, 4 ] ;
lastelem [ ] ;
"p10"
! ....lastelem [ e ] = e
!  | lastelem (h::t) = lastelem t ..
! Warning: pattern matching is not exhaustive
> val 'a lastelem = fn : 'a list -> 'a
> val it = 4 : int
! Uncaught exception:
! Match

List manipulation

Tail recursion

A tail call is the last thing that happens in a function.
A function is said to be tail recursive (or iterative) if the recursive calls are tail calls.

No computation is required after the recursive call returns; so the call could be replaced with a return.
There is no stack of pending operations.

Simple tail recursive functions are the functional equivalent of while loops.

Tail-call optimisation works with mutually recursive functions.
List manipulation
Accumulators

Some functions (like, for instance, `length`, `reverse`, etc.) can be made tail recursive by using an accumulator that gathers the running total of the computation.

1. ```
   fun addlength [] a = a
   | addlength (h::t) a = addlength t (a+1)
   fun length l = addlength l 0;
```

2. ```
   fun accadder [] a = a
   | accadder (h::t) a = accadder t (h+a)
   fun adder l = accadder l 0;
```

3. ```
   fun auxrev [] l = l
   | auxrev(h::t) l = auxrev t (h::l);
   fun reverse l = auxrev l[];
```

Examples

1. ```
   fun reverse l
       = let
           fun auxrev [] l = l
           | auxrev (h::t) l = auxrev t (h::l);
           in
           auxrev l[]
       end;
```

2. ```
   fun accadder [] a = a
   | accadder (h::t) a = accadder t (h+a)
   fun adder l = accadder l 0;
```

3. ```
   fun auxrev [] l = l
   | auxrev (h::t) l = auxrev t (h::l);
   fun reverse l = auxrev l[];
```

Local bindings

Local functions are defined and used inside other functions to present the desired function to the outside world without exporting its internal implementation.

Local functions and other values are defined by means of the `let...in...end` construct.

Remark: For patterns `P1, ..., Pn`, the expressions

```fn P1 => E1 | ... | Pn => En```

and

```let fun f(P1) = E1 | ... | f(Pn) => En in f end```

have the same meaning, provided that the name `f` is fresh.
2. fun fact n 
   = let 
     fun accfact n x 
       = if n = 0 then x 
       else accfact (n-1) (n*x) 
     in 
     accfact n 1 
   end ; 

"p15"

> val fact = fn : int -> int

! Compare with the imperative version!

3. fun unzip [] = ( [], [] )
   | unzip ( (h1,h2)::t )
     = let val (t1,t2) = unzip t 
       in ( h1::t1 , h2::t2 ) 
       end ;

fun unzip ( h1::t1 , h2::t2 ) = (h1,h2) :: zip( t1,t2 )

| zip_ = [] ;

"p16"

Recall that patterns are evaluated in the order that they are written.

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

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

List manipulation
Library functions

Try the following in ML - load"List"; open List;

Examples:

- List.concat ;
  > val 'a it = fn : 'a list list -> 'a list

- List.take ;
  > val 'a it = fn : 'a list * int -> 'a list

- List.drop ;
  > val 'a it = fn : 'a list * int -> 'a list

- List.find ;
  > val 'a it = fn :
     ( 'a -> bool) -> 'a list -> 'a option

- List.partition ;
  > val 'a it = fn :
     ( 'a -> bool) -> 'a list -> 'a list * 'a list

! Investigate what they do and provide your own implementations!
Higher-order functions

A higher-order function (or functional) is a function that operates on other functions; e.g., it 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. They naturally lead to:

- Partial evaluation.
- General-purpose functionals.
- Infinite lists.

Functionals with numbers

1. Summation.

\[ \text{sum } f_{ij} = \sum_{n=i}^{j} f(n) \]

fun sum f i j
  = if i > j then 0.0
  else f(i) + sum f (i+1) j
fi

"p01"
> val sum = fn : (int -> real) -> int -> int -> real

What do the functions

\[ fn f => fn i => fn k => fn l => \text{sum } (\text{sum } f i) k l \]

and

fn h => fn i => fn j => fn k => fn l =>
  \text{sum } (fn n => \text{sum } (h n) i j) k l
do?

2. Iterated composition.

\[ \text{iterate } f_{n x} = f^{n}(x) \]

fun iterate fn x
  = if n > 0 then iterate f (n-1) (f x)
  else x
fi

"p02"
> val 'a iterate
  = fn : ('a -> 'a) -> int -> 'a -> 'a
Map

It is often useful to systematically transform a list into another one by mapping each element via a function as follows

\[
\text{map } f \ [a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]
\]

where \( a_i \in A \) \((i = 1, \ldots, n)\) and \( f : A \rightarrow B \).

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

"p03"

> val (\'a, \'b) map
   = fn : (\'a -> \'b) -> \'a list -> \'b list

Examples

\[
\text{load"Real";}
\text{fun cast} \ l = \text{map} \ (\text{fn} \ x -> \text{real} \ x) \ l ;
\text{fun scaleby} \ n = \text{map} \ (\text{fn} \ x -> \text{Real}.*(n,x)) ;
\text{fun lift} \ l = \text{map} \ (\text{fn} \ x -> \text{SOME} \ x) \ l ;
\text{fun transp} \ ( [] :: _ ) = []
   | transp rows = (\text{map} \ hd \ rows) :: transp( \text{map} \ tl \ rows) ;
-
"p04"

> val scaleby = fn : int -> int list -> int list
> val \'a lift = fn : \'a list -> \'a option list
> val \'a matrixtransp = fn : \'a list list -> \'a list list

Filter

This functional applies a predicate (= boolean-valued function) to a list, returning the list of all the elements satisfying the predicate in the original order; thus filtering those that do not.

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

> val \'a filter
   = fn : (\'a -> bool) -> \'a list -> \'a list

Fold

When folding a list, we compute a single value by folding each new list element into the result so far, with an initial value provided by the caller.

For \( a_i \in A \) \((i = 1, \ldots, n)\), \( f : A \times B \rightarrow B \), and \( b \in B \) we have

\[
\begin{align*}
\text{fold left} & : \text{foldl} \ f \ b \ [a_1, \ldots, a_n] = f(a_n, \ldots f(a_1, b) \ldots) \\
\text{and} & : \text{foldr} \ f \ b \ [a_1, \ldots, a_n] = f(a_1, \ldots f(a_n, b) \ldots)
\end{align*}
\]

NB: \text{foldl} and \text{foldr} are built-in functionals; \text{foldl} is tail recursive but \text{foldr} is not.
fun foldl f b [] = b
  | foldl f b (h::t) = foldl f (f(h,b)) t ;
fun foldr f b [] = b
  | foldr f b (h::t) = f( h , foldr f b t ) ;

"p06"

> val ('d, 'e) foldl
  = fn : ('d * 'e -> 'e) -> 'e -> 'd list -> 'e
> val ('d, 'e) foldr
  = fn : ('d * 'e -> 'e) -> 'e -> 'd list -> 'e

Examples:
1. val addall = foldl op+ 0 ;
   val multall = foldl op* 1 ;
   ~
   "p07"
   > val addall = fn : int list -> int
   > val multall = fn : int list -> int

2. fun reverse l = foldl op:: [] l ;
   fun length l = foldl ( fn (_,n) => n+1 ) 0 l ;
   fun append l = foldr op:: l ;
   fun concat l = foldr op@ [] l ;
   fun map f = foldr ( fn (h,t) => (f h)::t ) [] ;
   ~
   "p07"

   > val 'a reverse = fn : 'a list -> 'a list
   > val 'a length = fn : 'a list -> int
   > val 'a append = fn : 'a list -> 'a list -> 'a list
   > val 'a concat = fn : 'a list list -> 'a list
   > val ('a, 'b) map
     = fn : ('a -> 'b) -> 'a list -> 'b list

Further list functionals

   ♦ fun takewhile P [] = []
     | takewhile P (h::t)
       = if P h then h :: takewhile P t else [] ;
   fun dropwhile P l
     = if null l then []
     else if P (hd l) then dropwhile P (tl l) else l ;
   ~
   "p08"

   > val 'a takewhile
     = fn : ('a -> bool) -> 'a list -> 'a list
   > val 'a dropwhile
     = fn : ('a -> bool) -> 'a list -> 'a list
Example

fun find P l
  = case dropwhile ( fn x => not(P x) ) l of
    [] => NONE
    | (h::_) => SOME h;

"p08"
> val 'a find
  = fn : ('a -> bool) -> 'a list -> 'a option

Examples:

infix isin ;
fun x isin l = exists (fn y => y = x) l ;
fun disjoint l1 l2
  = all (fn x => all (fn y => x<>y) l2) l1 ;

"p09"
> infix 0 isin
> val ''a isin = fn : ''a * ''a list -> bool
> val ''a disjoint
  = fn : ''a list -> ''a list -> bool

Matrix multiplication

fun dotprod l1 l2
  = foldl op+ 0.0 ( map op* (ListPair.zip( l1 , l2 )) ) ;
fun matmult Rows1 Rows2
  = let
    val Cols2 = transp Rows2
    in
    map (fn row => map (dotprod row) Cols2) Rows1
  end ;

"p10"
> val dotprod = fn : real list -> real list -> real
> val matmult = fn :
  real list list -> real list list -> real list list
Generating permutations

A combinatorial version of the factorial function:

```ml
fun permgen [] = []
| permgen l
  = let fun
      pickeach [] = []
      | pickeach (h::t)
      = (h,t) :: map (fn (x,l) => (x,h::l)) (pickeach t);
    in
    List.concat
    ( map (fn (h,l) => map (fn l => h::l) (permgen l))
      (pickeach l) )
  end;

"p11"
```

Simple substitution cipher

```ml
load"ListPair";
fun makedict s t
  = ListPair.zip( explode s , explode t);

> val makedict
  = fn : string -> string -> (char * char) list

fun lookupD x
  = case List.find( fn (s,t) => s=x) D of
    SOME(_,y) => y
  | NONE => x;

> val 'a lookup = fn:('a*'a)list->'a->'a
```

```ml
fun encode D
  = foldl op'^"" o map (str o (lookup D)) o explode;
val decode
  = encode o map (fn (s,t) => (t,s))

"p12"
```

Lecture V

Keywords:
- sorting: insertion sort, quick sort, merge sort; parametric sorting
- queues; signatures; structures; functors; generic sorting.

References:
- [MLWP] Chapter 2, § Introduction to modules
  Chapter 3, § Sorting: A case study
  Chapter 7, § Three representations of queues
- [PFDS] Section 5.2, § Queues
### Insertion sort

Insertion sort works by inserting the items, one at a time, into a sorted list.  

```plaintext
fun ins(x, []) = [x]
| ins(x, h::t) =  
  if x <= h then x::h::t  
  else h::ins(x, t);
```

val sort = fold1 ins [] ;

Insertion sort takes $O(n^2)$ comparisons in average and in the worst case.

### Quick sort

Quick sort works by picking up a pivot value and partitioning the list into two lists: values less than or equal to the pivot and values greater than the pivot. The sort is applied recursively to the two partitions and the resulting (sorted) lists are concatenated.  

```plaintext
fun sort [] = []
| sort(h::t) =  
  case List.partition ( fn x => x <= h ) t of  
    (left,right) => sort left @ h::sort right ;

> val sort = fn : int list -> int list
```

Quick sort takes $O(n \log n)$ comparisons on average and $O(n^2)$ in the worst case.

### Merge sort

Merge sort is another divide-and-conquer sorting algorithm. It works by dividing the input into two halves, sorting each of them, and then merging them in order.

The implementation below is top-down; it sorts one half of the list before starting with the other half. A bottom-up approach is also possible; starting with individual elements and merging them into larger lists until the sort is complete.

Merge sort takes $O(n \log n)$ comparisons on average and in the worst case.
fun merge( [], 12 ) = 12
  | merge( 11 , [] ) = 11
  | merge( 11 as h1::t1 , 12 as h2::t2 ) =
    if h1 <= h2 then h1::merge( t1 , 12 )
    else h2::merge( 11 , t2 )

fun sort [] = []
| sort [x] = [x]
| sort 1 = let
  val n = length 1 div 2
  in
  merge( sort( List.take(1,n) ) ,
         sort( List.drop(1,n) ) )
  end ;

Why do we need two base cases? What happens if we declare \texttt{val n = 1} in the \texttt{sort}ing function?

\textbf{Parametric sorting}

fun msort comp [] = []
  | msort comp [x] = [x]
  | msort comp 1 =
    let fun merge( [], 12 ) = 12
        | merge( 11 , [] ) = 11
        | merge( 11 as h1::t1 , 12 as h2::t2 ) =
          if comp(h1,h2)
            then h1 :: merge( t1 , 12 )
            else h2 :: merge( 11 , t2 )
        val n = length 1 div 2
        in
        merge( msort comp ( List.take(1,n) ) ,
               msort comp ( List.drop(1,n) ) )
        end ;

fun sort 1 =
  let fun msort( 0 , 1 ) = ( [], 1 )
        | msort( 1 , h::t ) = ( [h] , t )
        | msort( n , 1 ) =
          let val (sl1,l1) = msort( (n+1) div 2 , 1 )
              val (sl2,l2) = msort( n div 2 , l1 )
          in msort comp (merge(sl1,sl2) , l2 ) end
  in
case msort( length 1 , 1 ) of
  (sl,_) => sl
  end ;
**Queues**

*Queue* is an abstract data type allowing for the insertion and removal of elements in a first-in-first-out (FIFO) discipline. It provides the following operations:

```plaintext
signature QUEUE = sig
  type 'a t                  (* type of queues *)
  val empty: 'a t            (* the empty queue *)
  val null: 'a t -> bool     (* test for empty queue *)
  val enq: 'a -> 'a t -> 'a t (* add to end *)
  val deq: 'a t -> 'a t      (* remove from front *)
  val hd: 'a t -> 'a         (* return the front element *)
end;
```

An implementation

A fast and simple implementation of queues can be done with an ordered pair of lists. The first list contains the front elements of the queue in order and the second list contains the rear elements in reverse order.

For instance, the queue with integers \([-2..2]\) is represented by any one of the following:

```
( [], [2,1,0,~1,~2] )  ( [~2,~1,0] , [2,1] )
( [~2,~1] , [2,1,0] )   ( [~2,~1,0,1,2] , [] )
```

The head of the queue is the head of the first list, so `hd` returns this element and `deq` removes it. To add an element to the queue, we just add it to the beginning of the second list.

To ensure that `hd` always succeeds on a non-empty queue, we must maintain the *invariant* that if the first list is empty then so is the second one. When the first list is exhausted, we move the reverse of the second list to the front. This needs to happen in `enq` when the queue is empty, and in `deq` when the first list is a singleton.
structure Queue : QUEUE =
    struct
        type 'a t = 'a list * 'a list ;
        val empty = ([],[]);
        fun null(([],[])) = true
            | null _ = false;
        fun enq x (front,back) = (front, x::back);
        fun deq( _::[] ,back) = (rev back, [])
            | deq( _::rest,back) = (rest, back);
        fun hd( (head::rest,_)) = head;
    end;

Generic orders

signature ORDER =
    sig
        type t
        val leq: t * t -> bool
    end;

> signature ORDER
    = /	{type t = t, val leq : t * t -> bool}

structure LeqIntOrder =
    struct
        type t = int ;
        val leq = op<= ;
    end;

load"Real" ;
structure LeqRealOrder =
    struct
        type t = real ;
        val leq = Real.<= ;
    end;

> structure LeqIntOrder :
    {type t = int, val leq : int * int -> bool}
> structure LeqRealOrder :
    {type t = real, val leq : real * real -> bool}
functor Op( O: ORDER ) : ORDER =
    struct
        type t = O.t ;
        fun leq(x,y) = O.leq(y,x) ;
    end ;
> functor Op :
    !t.
{type t = t, val leq : t * t -> bool}
-> {type t = t, val leq : t * t -> bool}
structure GeqIntOrder = Op(LeqIntOrder) ;
structure GeqRealOrder = Op(LeqRealOrder) ;
> structure GeqIntOrder :
    {type t = int, val leq : int * int -> bool}
> structure GeqRealOrder :
    {type t = real, val leq : real * real -> bool}

struct (* MergeSort *)
    type t = O.t ;
    fun merge( [], 12 ) = 12
    | merge( 11 , [] ) = 11
    | merge( 11 as h1::t1 , 12 as h2::t2 ) =
        if O.leq(h1,h2) then h1::merge( t1 , 12 )
        else h2::merge( 11 , t2 ) ;
    fun sort [] = []
    | sort [x] = [x]
    | sort l = let val n = length l div 2 in
        merge( sort( List.take(l,n) ) ,
            sort( List.drop(l,n) ) )
    end ;
end (* MergeSort *);
Keywords:
- enumerated types; polymorphic datatypes:
- option type, disjoint-union type; abstract types;
- error handling; exceptions.

References:
- [MLWP] Section 2.9, Records
- Chapter 4, § The datatype declaration
- Section 7.6, § The abstype declaration
- Chapter 4, § Exceptions

The datatype declaration defines the set of constructors of the datatype:

```ml
> New type names: =S
datatype S
  = (S,{con bot : S, con ff : S,
      con top : S, con tt : S})
con bot = bot : S
con ff = ff : S
con top = top : S
con tt = tt : S
> val lt = fn : S * S -> bool
```

Example:
The lattice

\[ S = \begin{array}{c}
   \text{bot} \\
   \text{tt} \\
   \text{ff} \\
   \text{top}
\end{array} \]

may be implemented as

```ml
datatype S = bot | tt | ff | top ;
fun lt (bot,_) = true
    | lt (_,top) = true
    | lt _ = false ;
```

Enumerated types

Polymorphic datatypes

1. The standard library declares the datatype option:

```ml
datatype 'a option = NONE | SOME of 'a ;
```

```ml
> New type names: =option
datatype 'a option =
  ('a option,{con 'a NONE : 'a option,
    con 'a SOME : 'a -> 'a option})
con 'a NONE = NONE : 'a option
con 'a SOME = fn : 'a -> 'a option
```

Type τ option contains a copy of type τ, augmented with the extra value NONE.
The option type can be used to supply optional data to a function, but its most obvious use is to indicate errors.

Example:

```
fun find P l
  = case dropwhile ( fn x => not(P x) ) l of
    [] => NONE
    | (h::_) => SOME h;
> val 'a find
  = fn : ('a -> bool) -> 'a list -> 'a option
```

2. Disjoint-union type.

```
datatype ('a,'b)sum
  = In1 of 'a | In2 of 'b ;
~
"p05"
> New type names: =sum
datatype ('a, 'b) sum =
  (('a, 'b) sum,
   {con ('a, 'b) In1 : 'a -> ('a, 'b) sum,
     con ('a, 'b) In2 : 'b -> ('a, 'b) sum})
con ('a, 'b) In1 = fn : 'a -> ('a, 'b) sum
ccon ('a, 'b) In2 = fn : 'b -> ('a, 'b) sum
```

Examples:

```
type intORreal = (int,real)sum;
type 'a myoption = (unit,'a)sum;
```

Abstract types

Examples:

1. Complex numbers.

```
abstype complex = C of real * real
with
  fun ComplexRep( x , y ) = C(x,y) ;
  val origin = C( 0.0 , 0.0 ) ;
  fun X( C(x,y) ) = x ;
  fun Y( C(x,y) ) = y ;
  fun norm v = Math.sqrt( X(v)*X(v) + Y(v)*Y(v) ) ;
  fun scalevec( r , v ) = C( r*X(v) , r*Y(v) ) ;
  fun normal v = scalevec( 1.0/(norm v) , v ) ;
  fun CartRep v = ( X v , Y v ) ;
end ;
```

> New type names: complex
type complex = complex
val ComplexRep = fn : real * real -> complex
val origin = <complex>:complex
val X = fn : complex -> real
val Y = fn : complex -> real
val norm = fn : complex -> real
val scalevec = fn : real * complex -> complex
val normal = fn : complex -> complex
val CartRep = fn : complex -> real * real
val v1 = normal( ComplexRep(2.0,2.0) );
CartRep v1 ;
norm v1 ;
> val v1 = <complex> : complex
> val it
  = (0.707106781187, 0.707106781187) : real * real
> val it = 1.0 : real

2. Queues.

abstype 'a Queue = Q of 'a list * 'a list
with
val empty = Q([],[]);
fun null( Q([],[]) ) = true
  | null _ = false;
fun enq x ( Q([],_)) = Q([x],[])
  | enq x ( Q(front,back) ) = Q(front, x::back);
fun deq( Q(_,[]),back ) = Q( rev back, [] )
  | deq( Q(_,rest,back) ) = Q( rest, back )
  | deq(_,_) = empty;
fun hd( Q(head::rest,_) ) = SOME head
  | hd( Q([],_)) = NONE;
end;

enq 0 empty; enq 1 it; deq it; hd it;
val it = - : int Queue
val it = - : int Queue
val it = - : int Queue
val it = SOME 1 : int option
enq 0 empty; enq 1 it; deq it; deq it; hd it;
val it = - : int Queue
val it = - : int Queue
val it = - : int Queue
val it = NONE : int option
Exceptions

Exceptions are raised on various runtime failures including failed pattern match, overflow, etc. One can also define custom exceptions and raise them explicitly.

Examples:

1. exception Error of string;
   
   fun f b
   = ( if b
       then raise Error "It’s true\n"
       else raise Error "It’s false\n"
       ) handle Error m => print m;

   val f = fn : bool -> unit
   - f true;
   It’s true

2. Queues.
   abstype 'a Queue = Q of 'a list * 'a list
   
   with
   
   exception E;
   val empty = Q([], []);
   fun null( Q([],__)) = true
   | null _ = false;
   fun enq x ( Q([],_)) = Q([x], [])
   | enq x ( Q(front,back)) = Q(front, x::back);
   fun deq( Q(_,[])) = Q(rev back, [])
   | deq( Q(_::rest,back)) = Q(rest, back)
   | deq(_ ) = empty;
   fun hd( Q(head::rest,_) ) = head
   | hd( Q([],_)) = raise E;

   end;

datatype 'a instruction
   = create of 'a Queue -> 'a Queue
   | observe of 'a Queue -> 'a;

   fun process obs [] q = obs
   | process obs ((create f)::ins) q
     = process obs ins (f q)
   | process obs ((observe f)::ins) q
     = process (f q)::obs handle E => [] ins q;

   val process = fn :
   'a list -> 'a instruction list -> 'a Queue -> 'a list

--- Lecture VII ---

Keywords:

recursive datatypes: lists, trees, λ calculus; tree manipulation; tree listings: preorder, inorder, postorder; tree exploration: breadth-first and depth-first search; polymorphic exceptions; isomorphisms.

References:

♦ [MLWP, Chapter 4]
Recursive datatypes

Datatype definitions, including polymorphic ones, can be recursive.

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

\[
\text{infixr} \ 5 ::; \\
\text{datatype} \ \tau \text{ list} \\
= \text{nil} \mid :: \text{ of } \tau \ \times \ \tau \text{ list} ;
\]

In the same vein, the polymorphic datatype of (planar) binary trees with nodes where data items are stored is given by:

\[
\text{datatype} \ \tau \text{ tree} \\
= \text{empty} \mid \text{node of } \tau \ \times \ \tau \text{ tree} \times \tau \text{ tree} ;
\]

Further recursive datatypes

Examples:
1. Non-empty planar finitely-branching trees and forests.

   (a) Recursive version.
   \[
   \text{datatype} \\
   \tau \text{ FBtree} = \text{node of } \tau \ \times \ \tau \text{ FBtree list} ; \\
   \text{type} \\
   \tau \text{ FBforest} = \tau \text{ FBtree list} ;
   \]

   (b) Mutual-recursive version.
   \[
   \text{datatype} \\
   \tau \text{ FBtree} = \text{node of } \tau \ \times \ \tau \text{ FBforest} \\
   \text{and} \\
   \tau \text{ FBforest} = \text{forest of } \tau \text{ FBtree list} ;
   \]

   What are the set of values of \( \tau \text{ FBtree} \) and \( \tau \text{ FBforest} \)?

2. \( \lambda \) calculus.

   \[
   \text{datatype} \\
   D = f \ of \ D \rightarrow D ;
   \]

   \textbf{NB}: It is non-trivial to give semantics to \( D \). This was done by Dana Scott in the early 70’s, and gave rise to Domain Theory.

References:


Semantics

The set \( \text{Val}(\tau \text{ list}) \) of values of the type \( \tau \text{ list} \) is inductively given by the following rules:

\[
\begin{align*}
\text{nil} & \in \text{Val}(\tau \text{ list}) \\
\nu \in \text{Val}(\tau) & \quad \ell \in \text{Val}(\tau \text{ list}) \\
\nu :: \ell & \in \text{Val}(\tau \text{ list})
\end{align*}
\]

That is, \( \text{Val}(\tau \text{ list}) \) is the smallest set containing \( \text{nil} \) and closed under performing the operation \( \nu :: \_ \) for values \( \nu \) of type \( \tau \).

What is the set of values of \( \tau \text{ tree} \)?
Tree manipulation

Examples:

1. **fun** count empty = 0
   
   \[|\text{count}(\text{node}(_,l,r)) = 1 + \text{count} \ l + \text{count} \ r;\]

2. **fun** depth empty = 0
   
   \[|\text{depth}(\text{node}(_,l,r)) = 1 + \text{Int.} \max(\text{depth} \ l, \text{depth} \ r);\]

3. **fun** treemap f empty = empty
   
   \[|\text{treemap} \ f(\text{node}(n,l,r)) = \text{node}(f \ n, \text{treemap} \ f \ l, \text{treemap} \ f \ r);\]

4. **datatype**
   
   dir = L | R;

   exception E;

   **fun** subtree [] t = t
   
   \[|\text{subtree} (L::D)(\text{node}(_,l,_)) = \text{subtree} \ D \ l;\]

   \[|\text{subtree} (R::D)(\text{node}(_,_,r)) = \text{subtree} \ D \ r;\]

   \[|\text{subtree} _ _ = \text{raise} \ E;\]

Tree listings

1. **Preorder.**

   **fun** preorder empty = []
   
   \[|\text{preorder}(\text{node}(n,l,r)) = n :: (\text{preorder} \ l) @ (\text{preorder} \ r);\]

2. **Inorder.**

   **fun** inorder empty = []
   
   \[|\text{inorder}(\text{node}(n,l,r)) = (\text{inorder} \ l) @ n :: (\text{inorder} \ r);\]

3. **Postorder.**

   **fun** postorder empty = []
   
   \[|\text{postorder}(\text{node}(n,l,r)) = (\text{postorder} \ l) @ (\text{postorder} \ r) @ [n];\]

Inorder without append

**fun** inorder t

\[= \text{let}\]

\[\text{fun} \ \text{accinorder} \ \text{acc} \ \text{empty} = \text{acc}\]

\[|\text{accinorder} \ \text{acc} (\text{node}(n,l,r)) = \text{accinorder} (n :: \text{accinorder} \ \text{acc} \ r) \ l\]

\[\text{in}\]

\[\text{accinorder} [] \ t\]

\[\text{end};\]

\[= \text{inorder} (\text{node}(3,\text{node}(2,\text{node}(1,\text{empty},\text{empty}),\text{empty}),\text{node}(4,\text{empty},\text{node}(5,\text{empty},\text{empty}))))\];

\[\text{val} \ \text{it} = [1,2,3,4,5] : \text{int list}\]
### Tree exploration

#### Breadth-first search

datatype

`'a FBtree = node of `'a * `'a FBtree list ;`

fun bfs P t

= let fun auxbfs [] = NONE
   | auxbfs( node(n,F)::T )
   = if P n then SOME n
   else auxbfs( T @ F ) ;

in auxbfs [t] end ;

val bfs = fn : ( `'a -> bool ) -> `'a FBtree -> `'a option

---

#### Depth-first search

1. fun dfs P t

= let fun auxdfs [] = NONE
   | auxdfs( node(n,F)::T )
   = if P n then SOME n
   else auxdfs( F @ T ) ;

in auxdfs [t] end ;

val dfs = fn : ( `'a -> bool ) -> `'a FBtree -> `'a option

2. DFS without append

fun dfs' P t

= let

  fun auxdfs( node(n,F) )
  = if P n then SOME n
    else foldl (fn(t,r) => case r of
      NONE => auxdfs t | _ => r )

  ( fn(t,_) => auxdfst ) NONE F ;

in auxdfs t end ;

val dfs' = fn : ( `'a -> bool ) -> `'a FBtree -> `'a option

---

3. DFS without append; raising an exception when successful

fun dfs0 P (t: `'a FBtree)

= let

    exception Ok of `'a;

    fun auxdfs( node(n,F) )
    = if P n then raise Ok n
      else foldl (fn(t,_ => auxdfs t) NONE F ;

in auxdfs t handle Ok n => SOME n end ;

val dfs0 = fn : ( `'a -> bool ) -> `'a FBtree -> `'a option

---

Warning: When a polymorphic exception is declared, ML ensures that it is used with only one type. The type of a top level exception must be monomorphic and the type variables of a local exception are frozen.

Consider the following nonsense:

```python
exception Poly of 'a ; (*** ILLEGAL!!! ***)
(raise Poly true) handle Poly x => x+1 ;
```

Further topics

Quite surprisingly, there are very sophisticated non-recursive programs between recursive datatypes.

References:


Binary search trees

A binary search tree is a binary tree with nodes where data items are stored with the property that, for every node in the tree, the elements of the left subtree are smaller than or equal to that of the node which in turn is smaller than the elements of the right subtree.

Thus,

the inorder listing of a binary search tree is a sorted list.

Keywords:

- tree-based data structures; binary search trees; red/black trees; flexible functional arrays; heaps; priority queues.

References:

- [MLWP, Chapters 4 and 7]
- [PFDS, Chapters 2(§2), 3(§3), and 5(§2)]
Applications:

1. Binary search trees offer a simple way to represent sets; in which case, to eliminate repetitions, it is natural to impose the extra condition that elements of left subtrees are strictly smaller than their respective roots. When balanced, they admit $O(n \log n)$ runtime for all basic operations.

2. Binary search trees can also be easily extended to act as dictionaries, mapping keys to values.

We can test membership in a binary search tree using `lookup`:

```ml
fun lookup x empty = false
| lookup x ( node(v,l,r) )
  = ( x = v )
  orelse
  ( if x < v then (lookup x l) 
    else (lookup x r) ) ;

val lookup = fn : int -> int tree -> bool
```

To insert a new value, we just need to find its proper place:

```ml
fun insert x empty = node( x , empty , empty )
| insert x ( node(v,l,r) )
  = if x <= v then node( v , insert x l , r )
  else node( v , l , insert x r ) ;

val insert = fn : int -> int tree -> int tree
```

We could thus sort a list by the following procedure:

```ml
val sort
= inorder o ( foldl ( fn(x,t)=> insert x t ) empty ) ;

val sort = fn : int list -> int list
```

Red/Black trees

Binary search trees are simple and perform well on random or unordered data, but they perform poorly on ordered data, degrading to $O(n)$ performance for common operations. Red/black trees are a popular family of balanced binary search trees.

Every node in a red/black tree is colored either red or black.

```ml
datatype ‘a RBtree
  = 0
  | R of ’a * ’a RBtree * ’a RBtree
  | B of ’a * ’a RBtree * ’a RBtree ;
```

**NB:** Empty nodes are considered to be black.
We insist that every red/black tree satisfies the following two balance invariants:

1. No red node has a red child.
2. Every path from the root to an empty node contains the same number of black nodes.

Together, these invariants ensure that the longest possible path, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.

This function extends the insert function for unbalanced search trees in three significant ways.

1. When we create a new node, we initially color it red.
2. We force the final root to be black, regardless of the color returned by \( \text{ins} \).
3. We include calls to a BALANCE function, that massages its arguments as indicated in the figure on the following page to enforce the balance invariants.

\textbf{NB:} We allow a single red-red violation at a time, and percolate this violation up the search path towards the root during rebalancing.

\[
\begin{align*}
\text{fun RBinsert}\ x\ t \\
&= \text{let fun ins}\ (\ O) = R\ (x, O, O) \quad (*1*) \\
&\quad | \text{ins}\ (R\ (y, l, r)) \\
&\quad = \text{if } x \leq y \text{ then } R\ (y, \text{ins}\ l, r) \\
&\quad \quad \text{else } R\ (y, l, \text{ins}\ r) \\
&\quad | \text{ins}\ (B\ (y, l, r)) \\
&\quad = \text{if } x \leq y \\
&\quad \quad \text{then } \text{BALANCE}(B\ (y, \text{ins}\ l, r)) \quad (*3*) \\
&\quad \quad \text{else } \text{BALANCE}(B\ (y, l, \text{ins}\ r)) ; \quad (*3*) \\
&\text{in case}\ \text{ins}\ t\ \text{of} \\
&\quad R(x',l,r) \Rightarrow B(x',l,r) \quad (*2*) \\
&\quad | t' \Rightarrow t' \\
&\text{end} ;
\end{align*}
\]
We will provide an implementation based on the following tree-based data structure.

```ocaml
fun update( O , k , v )
  = if k = 1 then ( V(v,O,O) , 1 )
  else raise Subscript
| update( V(x,t1,tl), k , v )
  = if k = 1 then ( V(v,t1,tl), 0 )
  elseif k mod 2 = 0
    then let
       val (t,i) = update(tl,k div 2,v)
       in ( V(x,t1,tl), i ) end
  else let
       val (t,i) = update(tr,k div 2,v)
       in ( V(x,t1,tl), i ) end ;
```

A *functional* array provides a mapping from an initial segment of natural numbers to elements, together with *lookup* and *update* operations.

A *flexible* array augments the above operations to insert or delete elements from either end of the array.

```ocaml
signature UpperFlexARRAY =
  sig  type 'a t
       exception Subscript
    val empty: 'a t
    val null: 'a t -> bool
    val length: 'a t -> int
    val lookup: 'a t * int -> 'a
    val update: 'a t * int * 'a -> 'a t
    val shrink: 'a t -> 'a t
  end ;
```

```ocaml
fun update( O , k , v )
  = if k = 1 then ( V(v,O,O) , 1 )
  else raise Subscript
| update( V(x,t1,tl), k , v )
  = if k = 1 then ( V(v,t1,tl), 0 )
  elseif k mod 2 = 0
    then let
       val (t,i) = update(tl,k div 2,v)
       in ( V(x,t1,tl), i ) end
  else let
       val (t,i) = update(tr,k div 2,v)
       in ( V(x,t1,tl), i ) end ;
```

```ocaml
fun update( O , k , v )
  = if k = 1 then ( V(v,O,O) , 1 )
  else raise Subscript
| update( V(x,t1,tl), k , v )
  = if k = 1 then ( V(v,t1,tl), 0 )
  elseif k mod 2 = 0
    then let
       val (t,i) = update(tl,k div 2,v)
       in ( V(x,t1,tl), i ) end
  else let
       val (t,i) = update(tr,k div 2,v)
       in ( V(x,t1,tl), i ) end ;
```
fun delete( 0 , n )
  = raise Subscript
  | delete( V(v,tl,tr) , n )
  = if n = 1 then 0
  else if n mod 2 = 0
    then V( v , delete(tl,n div 2) , tr )
    else V( v , tl , delete(tr,n div 2) )
  end;
end;

/173

An implementation of \textit{upper flexible} functional arrays follows.

\textbf{NB:} The implementation can be extended to also provide \textit{downwards} flexibility in logarithmic time. Consider this as an exercise, or consult [MLWP, 4.15].

structure TreeArray : UpperFlexARRAY =
  struct
    abstype 'a t = A of int * 'a TreeArrayMod.t with
    exception Subscript;
    val empty = A( 0 , TreeArrayMod.empty ) ;
    fun null( A(l,_) ) = l = 0 ;
    fun length( A(l,_) ) = l ;
  end;
end;

fun lookup( A(l,t) , k )
  = if l = 0 orelse ( k < 1 andalso k > l )
  then raise Subscript
  else TreeArrayMod.lookup(t,k) ;

fun update( A(l,t) , k , v )
  = if 1 <= k andalso k <= l+1
  then let
    val (u,i) = TreeArrayMod.update( t , k , v )
    in
    A( l+i , u )
  end
  else raise Subscript ;

fun shrink( A(l,t) )
  = if l = 0 then empty
  else A( l-1 , TreeArrayMod.delete(t,l) ) ;
end ;
end ;
Priority queues using heaps

A priority queue is an ordered collection of items. Items may be inserted in any order, but only the highest priority item (typically taken to be that with lower numerical value) may be seen or deleted.

signature PRIORITY_QUEUE =
  sig exception Size
    type item
    val empty: t
    val null: t -> bool
    val insert: item -> t -> t
    val min: t -> item
    val delmin: t -> t
  end;

A heap is a binary tree in which the labels are arranged so that every label is less than or equal to all labels below it in the tree. This heap condition puts the labels in no strict order, but does put the least label at the root.

The following functional priority queues are based on flexible arrays that are heaps.

structure Heap: PRIORITY_QUEUE =
  struct
    exception Size;
    type item = real;
    abstype 'a tree
      = O | V of 'a * 'a tree * 'a tree with
    type t = item tree;

local
  exception Impossible;

  fun leftrem(V(v,O,O)) = (v,O)
    | leftrem(V(v,l,r)) = let
      val(w,t) = leftreml
    in
      (w,V(v,r,t))
  end
    | leftrem_ = raise Impossible;

  fun insert(w:item)O
    = V(w,O,O)
    | insert(w,V(v,l,r)) = if w <= v then V(w,insert v r,l)
      else V(v,insert w r,l);

  fun min(V(v,_,_)) = v
    | min_0 = raise Size;

/ 177
/ 178
/ 179
/ 180
fun siftdown(w:item, O, O) = V(w, O, O)
  | siftdown(w, t as V(v,0,0), O)
      = if w <= v then V(w, t, 0)
          else V(v, V(w,O,0), O)
  | siftdown(w, l as V(u,ll,lr), r as V(v,rl,rr))
      = if w <= u andalso w <= v then V(w,l,r)
          else if u <= v then V(u, siftdown(w,ll,lr),r)
              else V(v, l, siftdown(w,rl,rr))
  | siftdown_ = raise Impossible;

fun heapToList h
  = if Heap.null h then []
      else (Heap.min h)::heapToList(Heap.delmin h);

val sort
  = heapToList o foldl (fn(v,h) => Heap.insert v h) Heap.empty;

in
  fun delmin 0 = raise Size
  | delmin( V(v,0,_) ) = 0
  | delmin( V(v,l,r) )
      = let
          val (w,t) = leftrem l
          in
              siftdown( w, r, t )
          end
      end;

~ Lecture IX ~

Keywords:
call-by-value, call-by-name, and call-by-need evaluation;
lazy datatypes: sequences, streams, trees; lazy
evaluation; sieve of Eratosthenes; breadth-first and
depth-first traversals.

References:
[MLWP, Chapter 5]
Call-by-name evaluation

To compute the value of $F(E)$, first compute the value of the expression $F$ to obtain a function value, say $f$. Then compute the value of the expression obtained by substituting the expression $E$ for the formal parameter of the function $f$ into its body.

NB: Call-by-need is similar, but duplicated expressions are only evaluated once. (Haskell is the most widely used call-by-need purely-functional programming language.)

Example: Consider the following function definitions.

```haskell
fun pred n
    = if n = 0 then []
       else n :: pred(n-1) ;
fun lsum([], l) = 0
| lsum(h::t, l) = h + lsum(t, l) ;
```

```haskell
val pred = fn : int -> int list
val lsum = fn : int list * 'a -> int
```

1. Call-by-value evaluation.

\[ lsum( \text{pred}(2) , \text{pred}(10000) ) \]
\[ \begin{array}{l}
\text{pred}(2) \sim 2 :: \text{pred}(1) \sim 2 :: 1 :: \text{pred}(0) \sim 2 :: 1 :: [] \\
\text{pred}(10000) \sim 10000 :: \text{pred}(9999) \sim \ldots \\
\ldots \sim 10000 :: 9999 :: \ldots 1 :: [] \\
\sim lsum( 2 :: 1 :: [] , 10000 :: 9999 :: \ldots 1 :: [] ) \\
\sim 2 + lsum( 1 :: [] , 10000 :: 9999 :: \ldots 1 :: [] ) \\
\sim 2 + ( 1 + lsum( [] , 10000 :: 9999 :: \ldots 1 :: [] ) ) \\
\sim 2 + ( 1 + 0 ) \\
\sim 2 + 1 \\
\sim 3
\end{array} \]

2. Call-by-name evaluation.

\[ lsum( \text{pred}(2) , \text{pred}(10000) ) \]
\[ \begin{array}{l}
\text{pred}(2) \sim 2 :: \text{pred}(2-1) \\
\sim 2 + lsum( \text{pred}(2-1) , \text{pred}(10000) ) \\
\sim 2 + ( 1 + lsum( \text{pred}(1-1) , \text{pred}(10000) ) ) \\
\sim 2 + ( 1 + 0 ) \\
\sim 2 + 1 \\
\sim 3
\end{array} \]
**Lazy datatypes**

Lazy datatypes are one of the most celebrated features of functional programming. The elements of a lazy datatype are not evaluated until their values are required. Thus a lazy datatype may have infinite values, of which we may view any finite part but never the whole.

In a call-by-value functional language, like ML, we implement lazy datatypes by explicitly delaying evaluation. Indeed, to delay the evaluation of an expression $E$, we can use the nameless function $fn() \Rightarrow E$ instead, and we force the evaluation of this expression by the function application $(fn() \Rightarrow E)()$.

---

**Lazy evaluation in ML**

**Example:** Consider the following function definitions in ML.

```ml
fun seqpred n
    = if n = 0 then nil
      else cons( n, fn() => seqpred(n-1) ) ;
fun seqlsum( nil , l ) = 0
    | seqlsum( cons(h,t) , l )
      = h + seqlsum( t() , l ) ;
val seqpred = fn : int -> int seq
val seqlsum = fn : int seq * 'a -> int
```

Evaluate `seqlsum( seqpred(2) , seqpred(10000) )` and compare the process with the call-by-name evaluation of `lsum( pred(2) , pred(10000) )`!

---

**Examples:**

1. Sequences (= finite and infinite lists).
   ```ml
datatype 'a seq
    = nil | cons of 'a * ( unit -> 'a seq ) ;
```

2. Streams (= infinite lists).
   ```ml
datatype 'a stream
    = cons of unit -> 'a * 'a stream ;
```

3. Finite and infinite non-empty finitely-branching trees.
   ```ml
datatype 'a infFBtree
    = W of 'a * (unit -> 'a infFBtree list) ;
```

---

**Sequence manipulation**

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

1. Head, tail, and null testing.
   ```ml
   exception Empty ;
   fun seqhd nil = raise Empty
       | seqhd( cons(h,t) ) = h ;
   fun seqtl nil = raise Empty
       | seqtl( cons(h,t) ) = t() ;
   fun seqnull nil = true
       | seqnull _ = false ;
   val seqhd = fn : 'a seq -> 'a
   val seqtl = fn : 'a seq -> 'a seq
   val seqnull = fn : 'a seq -> bool
   ```
2. Constant sequences.

```ml
fun Kseq x = cons(x, fn() => Kseq x);
val Kseq = fn: 'a -> 'a seq
```

3. Traces.

```ml
fun trace f s
    = case s of
      NONE => nil
    | SOME x => cons(x, fn() => trace f (f x));
fun from n = trace (fn x => SOME(x+1)) (SOME n);
val trace = fn: ('a -> 'a option) -> 'a option -> 'a seq
val from = fn: int -> int seq
```

4. Sequence display.

```ml
exception Negative;
fun display n s
    = if n < 0 then raise Negative
    else if n = 0 then []
    else (seqhds :: display(n-1) (seqtl s));
val display = fn: int -> 'a seq -> 'a list
```

5. Append and shuffle.

```ml
fun seqappend( nil, s ) = s
    | seqappend( s, t )
      = cons(seqhds, fn() => seqappend(seqtls,t));
val seqappend = fn: 'a seq * 'a seq -> 'a seq
```

```ml
fun shuffle( nil, s ) = s
    | shuffle( s, t )
      = cons(seqhds, fn() => shuffle(t,seqtls));
val shuffle = fn: 'a seq * 'a seq -> 'a seq
```

6. Functionals: filter, map, fold.

```ml
fun seqfilter P nil = nil
    | seqfilter P s
      = let val h = seqhd s in
        if P h
        then cons(h, fn() => seqfilter P (seqtl s))
        else seqfilter P (seqtl s)
      end;
val seqfilter = fn: ('a -> bool) -> 'a seq -> 'a seq
```

```ml
fun seqmap f nil = nil
    | seqmap f s
      = cons(f(seqhd s), fn() => seqmap f (seqtl s));
val seqmap = fn: ('a -> 'b) -> 'a seq -> 'b seq
```

```ml
fun seqnfold n f x s
    = if n < 0 then raise Negative
    else if n = 0 then x
    else if seqnull s then raise Empty
    else seqnfold(n-1) f (f(seqhd s,x)) (seqtl s);
val seqnfold = fn: int -> ('a * 'b) -> 'b -> 'a seq -> 'b
```

```ml
/193
```

```ml
/194
```

```ml
/195
```

```ml
/196
```
**Generating the prime numbers**

**Streams**

```
datatype 'a stream = cons of unit -> 'a * 'a stream;

fun head (cons f) = let val (h,_) = f() in h end;

fun tail (cons f) = let val (_,t) = f() in t end;

val head = fn : 'a stream -> 'a
val tail = fn : 'a stream -> 'a stream
```

**Sieve of Eratosthenes (I)**

```
fun sieve s = let
  val h = head s
  val sift = filter (fn n => n mod h <> 0);
  in
  cons( fn() => ( h, sieve( sift (tail s) ) ) )
  end;

val sieve = fn : int stream -> int stream

fun from n = cons( fn() => ( n, from(n+1) ) );

val primes = sieve( from 2);

val from = fn : int -> int stream
val primes = cons fn : int stream
```

**Sieve of Eratosthenes (II)**

```
fun sieve s = case head s of
  NONE => cons( fn() => ( NONE, sieve( tail s ) ) )
| SOME h => let fun sweep s = itsweep s 1
  and itsweep s n = cons( fn() =>
    if n = h then ( NONE, sweep (tail s) )
    else ( head s, itsweep (tail s) (n+1) ) )
  in
  cons( fn() => ( SOME h, sieve( sweep (tail s) ) ) )
  end;

val sieve = fn : int option stream -> int option stream
```
### Infinite-tree manipulation

**datatype**

```plaintext
'null infFBtree
  = W of 'a * (unit -> 'a infFBtree list);
```

1. **Computation trees.**

```plaintext
fun CT fs
  = W( s, fn() => map(CT f)(f s) );
val CT = fn:'a->'alist->'a->'a infFBtree
```

2. **Breadth-first traversal.**

```plaintext
fun BFseq [] = nil
  | BFseq( W(x,F):: T )
    = cons( x, fn() => BFseq( T @ F() ) );
val BFseq = fn:'a infFBtree list -> 'a seq
```

3. **Depth-first traversal.**

```plaintext
fun DFseq [] = nil
  | DFseq( W(x,F):: T )
    = cons( x, fn() => DFseq( F() @ T ) );
val DFseq = fn:'a infFBtree list -> 'a seq
```

### Testing and verification

**Functional programs are easier to reason about**

- We wish to establish that a program is correct, in that it meets its specification.

- **Testing.**
  
  Try a selection of inputs and check against expected results.
  
  There is no guarantee that all bugs will be found.

- **Verification.**

  Prove that the program is correct within a mathematical model.
  
  Proofs can be long, tedious, complicated, hard, etc.

---

**Keywords:**

- testing and verification; rigorous and formal proofs;
- structural induction on lists; law of extensionality;
- multisets; structural induction on trees.

**References:**

- [MLWP, Chapter 6]
Rigorous vs. formal proof

A rigorous proof is a convincing mathematical argument

- Rigorous proof.
  - What mathematicians and some computer scientists do.
  - Done in the mathematical vernacular.
  - Needs clear foundations.

- Formal proof.
  - What logicians and some computer scientists study.
  - Done within a formal proof system.
  - Needs machine support.

Modelling assumptions

- Proofs treat programs as mathematical objects, subject to mathematical laws.
- Only purely functional programs will be allowed.
- Types will be interpreted as sets, which restricts the form of datatype declarations.
- We shall allow only well-defined expressions. They must be legally typed, and must denote terminating computations. By insisting upon termination, we can work within elementary set theory.

Structural induction on lists

Let $P$ be a property on lists that we would like to prove.

To establish $P(\ell)$ for all $\ell$ of type $\tau$ list

by structural induction, it suffices to prove.

1. The base case: $P([])$.
2. The inductive step: For all $h$ of type $\tau$ and $t$ of type $\tau$ list,
   $P(t)$ implies $P(h::t)$

Example: No list equals its own tail.

    For all $h$ of type $\tau$ and all $t$ of type $\tau$ list, $h::t \neq t$.

Applications

    fun nlen [] = 0
    | nlen (h::t) = 1 + nlen(t) ;

    fun len 1
    = let
      fun addlen( n , [] ) = n
      | addlen( n , h::t ) = addlen( n+1 , t )
      in
      addlen( 0 , 1 )
    end ;
Equality of functions

The law of extensionality states that functions $f, g : \alpha \to \beta$ are equal iff $f(x) = g(x)$ for all $x \in \alpha$.

Example:

For all lists $\ell, \ell_1, \ell_2$,
1. $\text{nlen}(\ell_1 @ \ell_2) = \text{nlen}(\ell_1) + \text{nlen}(\ell_2)$.
2. $\text{revApp}(\ell_1, \ell_2) = \text{nrev}(\ell_1) @ \ell_2$.
3. $\text{nrev}(\ell_1 @ \ell_2) = \text{nrev}(\ell_2) @ \text{nrev}(\ell_1)$.
4. $\ell @ [] = \ell$.
5. $\ell @ (\ell_1 @ \ell_2) = (\ell @ \ell_1) @ \ell_2$.
6. $\text{nrev}(\text{nrev}(\ell)) = \ell$.
7. $\text{nlen}(\ell) = \text{len}(\ell)$.

Applications

For all lists $\ell, \ell_1, \ell_2$,

1. **Functoriality** of $\text{id}$.
   \[
   \text{map id} = \text{id}
   \]
   For all $f : \alpha \to \beta$ and $g : \beta \to \gamma$,
   \[
   \text{map}(g \circ f) = \text{map}(g) \circ \text{map}(f) : \alpha \text{list} \to \gamma \text{list}
   \]
   2. For all $f : \alpha \to \beta$, and $\ell_1, \ell_2 : \alpha \text{ list}$,
      \[
      \text{map}(f \ell_1 @ \ell_2) = (\text{map}(f \ell_1)) @ (\text{map}(f \ell_2)) : \beta \text{ list}
      \]
   3. For all $f : \alpha \to \beta$,
      \[
      \text{map}(\text{nrev}(f)) = \text{nrev}(\text{map}(f)) : \beta \text{ list}
      \]

\*This is a technical term from Category Theory.
**Multisets**

Multisets are a useful abstraction to specify properties of functions operating on lists.

A *multiset*, also referred to as a *bag*, is a collection of elements that takes account of their number but not their order.

Formally, a multiset $m$ on a set $S$ is represented as a function $m : S \to \mathbb{N}$.

Some ways of forming multisets:

1. the *empty multiset* contains no elements and corresponds to the constantly $0$ function
   \[
   \emptyset : x \mapsto 0
   \]
2. the *singleton $s$ multiset* contains one occurrence of $s$, and corresponds to the function
   \[
   \langle s \rangle : x \mapsto \begin{cases} 
   1 & \text{if } x = s \\
   0 & \text{otherwise}
   \end{cases}
   \]
3. the *multiset sum* $m_1$ and $m_2$ contains all elements in the multisets $m_1$ and $m_2$ (accumulating repetitions of elements), and corresponds to the function
   \[
   m_1 \sqcup m_2 : x \mapsto m_1(x) + m_2(x)
   \]

An application

Consider

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

fun drop( [], _ ) = []
  | drop( l as h :: t, i )
      = if i > 0 then drop( t, i - 1 )
      else l;
```

and let

\[
\text{mset}([],) = \emptyset \\
\text{mset}(h::t) = \langle h \rangle \sqcup \text{mset}(t)
\]

Then, for all $\ell: \alpha \text{ list}$ and $n: \text{int}$,

\[
\text{mset}(\text{take}(\ell, n)) \sqcup \text{mset}(\text{drop}(\ell, n)) = \text{mset}(\ell)
\]
**Structural induction on trees**

Let \( P \) be a property on binary trees that we would like to prove. To establish

\[ P(t) \text{ for all } t \text{ of type } \tau \text{ tree} \]

by *structural induction*, it suffices to prove.

1. **The base case:** \( P(\text{empty}) \).

2. **The inductive step:** For all \( n \) of type \( \tau \) and \( t_1, t_2 \) of type \( \tau \text{ tree} \),

\[ P(t_1) \text{ and } P(t_2) \text{ imply } P(\text{node}(n, t_1, t_2)) \]

**Example:** No tree equals its own left subtree.

For all \( n \) of type \( \tau \) and all \( t_1, t_2 \) of type \( \tau \text{ list} \),

\[ \text{node}(n, t_1, t_2) \neq t_1. \]

---

**An application**

```
fun treemap f empty = empty
  | treemap f ( node(n,l,r) )
     = node( f n , treemap f l , treemap f r ) ;
```

Functoriality of \( \text{treemap} \).

\[ \text{treemap id} = \text{id} \]

For all \( f : \alpha \rightarrow \beta \) and \( g : \beta \rightarrow \gamma \),

\[ \text{treemap}(g \circ f) = \text{treemap}(g) \circ \text{treemap}(f) : \alpha \text{ tree} \rightarrow \gamma \text{ tree} \]

Let \( P \) and \( Q \) be properties on finitely-branching trees and forests, respectively, that we would like to prove.

To establish

\[ P(t) \text{ for all } t \text{ of type } \tau \text{ FBtree} \]

\[ Q(F) \text{ for all } F \text{ of type } \tau \text{ FBforest} \]

by *structural induction*, it suffices to prove.

1. **The base case:** \( Q(\text{empty}) \).

2. **The inductive step:** For all \( n \) of type \( \tau \), \( t \) of type \( \tau \text{ FBtree} \), and \( F \) of type \( \tau \text{ FBforest} \),

\[ Q(F) \text{ implies } P(\text{node}(n, F)) \]

\[ P(t) \text{ and } Q(F) \text{ imply } Q(\text{seq}(t, F)) \]

---

**Structural induction on finitely-branching trees**

datatype

\[ 'a FBtree = \text{node of } ' a \ast 'a FBforest \]

and

\[ 'a FBforest = \text{empty } \mid \text{seq of } 'a FBtree \ast 'a FBforest ; \]
An application

fun FBtreemap f ( node(n,F))
  = node( f n , FBforestmap f F)
and FBforestmap f empty = empty
  | FBforestmap f ( seq(t,F) )
  = seq( FBtreemap f t, FBforestmap f F );

Functoriality of FBtreemap and FBforestmap.

FBtreemap id = id
FBforestmap id = id

For all \( f : \alpha \rightarrow \beta \) and \( g: \beta \rightarrow \gamma \),

FBtreemap(g o f) = FBtreemap(g) o FBtreemap(f)
FBforestmap(g o f) = FBforestmap(g) o FBforestmap(f)

/ 221