CSE3322, 2005: ML Programming I, Comp. Sci. and Software Eng., Monash University.

In this and the next 8+/- lectures we will look at ML. ML is a good example of a functional programming (FP) language.

In this lecture we will look at basic programming with ML:

- Expressions
- Arithmetic and Boolean types
- Definitions
- Functions
- Recursion

Expressions

ML can be thought of as an up-market calculator:
– It computes the value of expressions.
– It also infers their type.

```
 % sml
Standard ML of New Jersey, Version 110.0.3, January 30, 19
- 1+2*3;
 val it = 7 : int
- 81 mod 10;
 val it = 1 : int
- ~(7-5);
 val it = ~2 : int
- it + 3;
 val it = 1 : int
```

ML uses "`it`" as the name of the current expression.

Arithmetic Expressions

There are also real numbers. These must have either a decimal point or an E exponent

```
- 2.0/6.0;
 val it = 0.333333333333 : real
- 1e~3
= ;
 val it = 0.001 : real
```

Note that an incomplete expression gives the "=" prompt.

Reals and integers have similar syntax to C.
The only important difference is that the unary minus sign is the tilde "~."

Arithmetic operations are the:

- **low precedence additive operators:** `+`, `-`.
- **high precedence multiplicative operators:**
  - `*`,
  - `/` **(real division),**
  - `div` **(integer division which rounds towards -∞) and**
  - `mod` **(integer division remainder).**
- **highest precedence:** `~` **(unary minus)**

Notice that many arithmetic operations are overloaded.

Boolean Expressions

There are also Booleans: `true` and `false`.

The usual comparison operators: =, <, >, <=, >= and <> (not equal) return Booleans and are overloaded, i.e., they work for integers, reals (except = and <>), characters or strings.

```
- 1 < 2;
 val it = true : bool
- 1 = 2;
 val it = false : bool
- 1+1 < 3 andalso 1+1 > 1;
 val it = true : bool
- not true;
 val it = false : bool
```

Boolean Expressions (Cont.)

However:
– Reals may not be compared with = or <>. Why?
– For characters and strings < means lexicographically preceds etc. (behaves like `strcmp`).

Operations on Booleans are:
– `orelse` **(logical or – like `||`)**
– `andalso` **(logical and – like `&&`)**
– `not` **(logical negation – like `!`).**

As in C, the first two are not strict, that is, they "short-cut", they might not have to evaluate their second arguments.

They have the usual precedences.

```
- 1 < 2 orelse 3 > 4;
 val it = true : bool
```

Value declarations

Alphanumeric identifiers are formed by either a letter or the character ´ followed by zero or more letters, digits, or symbols ´ and _

Most sorts of objects can be named using identifiers:

```
- val m = 3;
 val m = 3 : int

- val n = 5;
 val n = 5 : int

- m + n * n;
 val it =  28 : int

- it div 4;
 val it =  7 : int
```

The general form is:

```
val <identifier> = <value>
```

Declarations may be simultaneous:

```
- val m = 3
= and n = 5;
 val m = 3 : int
 val n = 5 : int
```

The order is immaterial.

Value declarations (cont)

You can give the type of a declaration

```
- val pi = 3.14159:real;
 val pi = 3.14159 : real
```

But you don't need to, ML infers it!

```
- val pi = 3.14159;
 val pi = 3.14159 : real
- val radius = 4.0;
 val radius = 4.0 : real
- val area = pi * radius * radius;
 val area = 50.26544 : real
```

Of course it would be better to have a function to compute the value of the area...

Defining functions

The keyword `fun` indicates a function definition.

```
- val pi = 3.14159;
 val pi = 3.14159 : real
- fun circle_area(r) = pi * r * r;
 val circle_area = fn : real -> real
- val m = circle_area(4.0);
 val m = 50.26544 : real
```

Parentheses are unnecessary. We can just type

```
- circle_area 2.0;
 val it =  12.56636 : real
```

And the definition could have been

```
fun circle_area r = pi * r * r;
```

The general form is:

```
fun <identifier> (<parameter list>) = <expression>;
```

Write a function to compute the circumference of a circle.

Defining functions (cont)

We could place our definition in a file. Imagine that `circle.ml` **contains**

```
(* useful(?) functions for dealing with circles *)
val pi = 3.14159;

fun circle_area r = pi * r * r;
fun circle_circum r = 2.0 * pi * r;
```

In ML `(* .... *)` **enclose comments. Comments nest**

An example session is:

```
- use "circle.ml";
 [opening circle.ml]
val pi = 3.14159 : real
val circle_area = fn : real -> real
val circle_circum = fn : real -> real
val it = () : unit
- circle_area 3.6;
 val it = 40.7150064 : real
```

Note that ML infers the type of `circle_area` **and** `circle_circum`**.**

If-then-else

ML has an `if-then-else`.
It is similar to C's conditional expression

```
... ? ... : ...
```

For instance, let us assume that `abs.ml` contains

```
(* computes the absolute value *)
fun abs r =
   if r > 0 then r
   else ~r;         (* NB. ‘~’ not ‘-’ *)
```

Note that the `else` must be there. An example session is:

```
- use "abs.ml";
 [opening abs.ml]
 val abs = fn : int -> int
 val it = () : unit
- abs 3;
 val it = 3 : int
- abs ~3;
 val it = 3 : int
```

How does it know that `r` is `int`?

What if we wanted `r` to be `real`?

Recursive functions

Unlike C programs, while or for loops are rare in ML programs. Why?

Instead in ML you use recursion. This is required in most useful functions.

Write a function to compute the number of digits in a given positive integer:

```
- fun digits n =
    if n < 10 then 1
    else 1 + digits (n div 10);
 val digits = fn : int -> int
- digits 7634;
 val it = 4 : int
```

Note that function application has higher precedence than all other operators so

```
1 + digits n div 10
```

means

```
1 + (digits n) div 10
```

What would happen with this code?

Recursive functions (Cont.)

Consider the expression:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

Lets compute the value of π using the above expression.
We can use a pair of recursive functions...

```
val bigreal = 1.0E5;
fun pos d = 1.0/d + neg (d+2.0)
and neg d =
  if d > bigreal then 0.0
  else ~1.0/d + pos(d+2.0);

val pi = 4.0 * pos(1.0);
```

Note the use of 'and' to handle mutual recursion.

Otherwise we would get an error since `neg` would be defined after being used.

Recursive functions (Cont.)

Write a recursive function to compute the factorial function, `fac n = n*(n-1)*...*1`.

Common Error Messages

```
- fun circle_area r = Pi * r * r;
 stdIn:25.21-25.23
 Error: unbound variable or constructor: Pi
```

The name `Pi` has not been defined.

```
- 1 + 0.5;
 stdIn:1.1-2.3
 Error: operator and operand don't agree [literal]
   operator domain: int * int
   operand:         int * real
   in expression:
     1 + 0.5
```

The + operator cannot take arguments of different kinds. It is overloaded and accepts either two integer or two real numbers, but not one of each.

Declaring a Function's Type

Because of overloaded built-in functions ML may not be able to determine types or determines an unintended type.

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

By default ML prefers `int`**s to** `real`**s.**

To make `square` **work on** `real`**s we must attach** `:real` **to one of the three occurences of** `x` **or to the overall result.**

```
- fun square (x:real) = x*x;      (* use *)
- fun square x = (x:real)*x;      (* one *)
- fun square x = x*(x:real);      (* of *)
- fun square x = x*x : real;      (* these *)
```

The brackets are necessary for precedence.

Values versus variables

Value names are not the same as variables in imperative languages.
– they cannot be updated to a new value.

A value name identifies an expression which is conceptually evaluated when the expression is first defined.
The name can be reused but this doesn't change the value defined using the previous definition.

The following merely reuses the name `pi` **and** `circle_area`**:**

```
- val pi = 3.14159;
 val pi = 3.14159 : real
- fun circle_area(r) = pi * r * r;
 val circle_area = fn : real -> real
- circle_area 2.0;
 val it = 12.56636 : real
- val pi = 0.0;
 val pi = 0.0 : real
- circle_area 2.0;
 val it =  12.56636 : real
- fun circle_area(r) = pi * r * r;
 val circle_area = fn : real -> real
- circle_area 2.0;
 val it = 0.0 : real
```

You can imagine that names and values are stored on a stack, and the most recent value for the name which is visible to the expression, is used.

Summary

Today we have looked at

- **Expressions**
- **Arithmetic and Boolean types**
- **Definitions**
- **Functions**
- **Recursion**

Homework

- **Write ML functions** `area` **and** `perim` **to respectively compute the area and perimeter of a square given its length** `l`**. Check that they work!**
- **Write a recursive ML function** `fib` **to compute the** n**th Fibonacci number (this is the** n**th element in the sequence** $1, 1, 2, 3, 5, 8, 13, ...$**).**

ML Programming II

In this lecture we will look at programming with simple data structures.

- **Characters (`char`) and strings (`string`)**
- **Built-in coercion functions**
- **Lists**

Strings

Strings are enclosed by double quotes. The syntax is similar to C.

Escape sequences are:
– `\n` **(newline),**
– `\t` **(tab),**
– `\"` **(double quote), and**
– `\\` **(backslash).**
You can also use $\backslash d_1d_2d_3$ **where** $d_1d_2d_3$ **is the ASCII code for the character.**

```
- "Hello\tworld";
 val it "Hello\tworld" x: string
```

Concatenation is the most important predefined operator. This is indicated by the "hat" (ˆ) operator:

```
- val m = "house";
 val m = "house" : string
- val n = "cat";
 val n = "cat" : string
- m^n;
 val it = "housecat" : string
```

We can also find out the size of a string:

```
- size(m^n);
 val it = 8 : int
```

Characters

The representation for characters in ML is unusual:
 # followed by a string with a single character
– `#"`x`"` represents character x.

Eg: `#"a"` **and** `#"\t"`

Type Coercion

Unlike C or C++ type coercion between basic types is never automatic.

```
- 3 + 6.4;
 stdIn:25.1-25.8
 Error: operator and operand don't agree [literal]
   operator domain: int * int
   operand:         int * real
   in expression:
     3 + 6.4
```

+ is overloaded:

```
op + : int*int -> int
op + : real*real -> real
```

only.

Coercion Functions

Built-in arithmetic coercion functions are:

- `real: int -> real`
- `floor: real -> int`
  Greatest integer no larger than argument
- `ceil: real -> int`
  Smallest integer no smaller than argument
- `round: real -> int`
  Closest integer, rounding up (abs value) for .5
- `trunc: real -> int`
  Drops digits after the decimal point

For example:

```
- real 3 + 6.4;
 val it = 9.4 : real
- floor 6.5;
 val it = 6 : int
- ceil 6.5;
 val it = 7 : int
- round 6.5;
 val it = 7 : int
- trunc 6.5;
 val it = 6 : int
```

Coercion Functions (Cont.)

Other built-in coercion functions are:

- `ord: char -> int`
  Gives integer code (ASCII) for character
- `chr: int -> char`
  Gives character coded by integer (ASCII)
- `str: char -> str.`
  Gives single character string containing the character

```
- chr 97;
 val it = #"a" : char

- ord #"b";
 val it = 98 : int

- str (chr 97);
 val it = "a" : string
```

Exercise

Write a function `upper` which takes a lowercase character and returns the corresponding uppercase character. For instance, `upper #"a"` returns `#"A"`

Using this write a function `toupper` which takes any character and returns the corresponding uppercase character if it is not already uppercase, otherwise returning the same character.

Hint:

```
- ord #"a" - ord #"A";
 val it = 32 : int
```

Lists

A list is a finite sequence of elements, all of the same type.

If the element type is `'a`, the type of the list is `'a list`.
`'a` is called a type variable.

```
- [1.0,2.0,3.0];
 val it = [1.0,2.0,3.0] : real list

- ["ab", "cde"];
 val it = ["ab","cde"] : string list

- [[1,2], [], [3]];
 val it = [[1,2],[],[3]] : int list list

- [[], []];
 val it = [[],[]] : 'a list list
```

Lists are constructed from
– `nil` (the empty list) and
– `::` (the list constructor "cons").
`[]` is shorthand for `nil`.

For example, `[1,2]` is shorthand for `1::(2::nil)`.

Exercise: **What is `[[1,2], [], [3]]` shorthand for?**

Programming with Lists

The key to programming with lists is to reason recursively about the list.
• Either the list is empty (base case) and the operation of interest should be straightforward, or
• it is non-empty (general case) and can be broken into a first element and a remaining shorter list which is dealt with recursively.

E.g. We can write a function to sum a list of integers:

What if the list `l` **is empty?**
– Return `0`**.**

What if the list `l` **has at list one element** `x`**?**
– Recursively sum the elements in the rest and add `x`**.**

The function can thus be defined as:

```
fun sumList [] = 0
  | sumList (x::xs) = x + (sumList xs);
 val sumList = fn : int list -> int

- sumList [1,3,5];
 val it = 9 : int
```

Notice the use of patterns (like "by cases" in Maths):

- **an empty list will match pattern** `[]`
- **a non-empty list will match pattern** `x::xs`

Parity of a list

Write functions `odd l` and `even l` which are true if the number of elements in the list `l` is odd or even.

The case for `odd`:
What if the list `l` is empty? Return `false`.

What if the list `l` is `x::xs`?
– Return true if `even xs` is true and vice versa.

The case for `even`:
What if the list `l` is empty? Return `true`.

What if the list `l` is `x::xs`?
– Return true if `odd xs` is true and vice versa.

Remember that, since we have mutual recursion, we need to use an `and`

```
fun even [] = true
  | even (x::xs) = odd xs
and
    odd [] = false
  | odd (x::xs) = even xs;

- use "evenoddlist.ml";
 [opening evenoddlist.ml]
 val even = fn : 'a list -> bool
 val odd = fn : 'a list -> bool
 val it = () : unit
```

Programming with Lists (Cont.)

Write a function to return the last element in a list.

What if the list `l` contains one element? Return this element.

What if the list `l` contains more than one element?
– Recursively find the last element of the list tail.

What is the actual code?

```
 val last = fn : 'a list -> 'a

- last [1,3,5];
 val it = 5 : int
- last ["abc","def"];
 val it = "def" : string
- last [];
 stdIn:25.1-25.8 Warning: type vars not generalized becaus
    value restriction are instantiated to dummy types (X1,

 uncaught exception nonexhaustive match failure
   raised at: stdIn:22.24
```

List Operations

To convert between strings and lists, use the built-ins `explode` and `implode`:

```
- explode "Bomb";
 val it = [#"B",#"o",#"m",#"b"] : char list
- implode it;
 val it = "Bomb" : string
```

The built-in infix operator `@` appends lists:

```
- [1,2] @ [4,5];
 val it = [1,2,4,5] : int list
```

The built-in function `null` tests whether a list is empty. We could define it

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

The underscores are wildcards which will match anything.

What is the point of having `null` when it is so easy to check for empty list? We will see later when discussing equality types.

List Operations (Cont.)

The function `hd` returns the first element of a list:

```
- hd [2,3,5];
 val it = 2 : int
```

The function `tl` returns the rest of the list:

```
- tl [2,3,5];
 val it = [3,5] : int list
```

We can use `hd`, `tl` and `null` instead of patterns. However, with lists we usually use pattern matching.

Reversing a List

Write a function `reverse l` **which returns the list** `l` **in reverse. You can use** `@`**.**

E.g. `reverse [1,2,3]` **should return** `[3,2,1]`**.**

What if the list `l` **is empty? Return** `[]`**.**

What if the list `l` **is** `x::xs`**?**
– reverse `xs` **;**
– add `x` **to the end of this.**

Actually ML has a library function `rev` **to reverse lists.**

Summary

We have looked at programming with:

- **Characters (`char`) and strings (`string`)**
- **Built-in coercion functions**
- **Lists**

Homework

- **Read Chapter 2 and Sections 3.1 and 3.2 of Ullman.**
- **Write a function `upperList` to take a list of characters and to return the list of corresponding uppercase characters.**
- **Write a function `upperString` to take a string and to return the string of corresponding uppercase characters.**
- **The `reverse` function of the previous slide is slow, $O(n^2)$, write a fast, $O(n)$-time, one – you should recall this algorithm from CSE2304.**

ML Programming III

In this lecture we will look at programming with:

- **Tuples**
- **Pattern-matching**

Tuples

C has type constructors for structures, unions, pointers and arrays.

ML has an even wider variety of type constructors. One of the most useful is the tuple. This is an ordered collection of values.

For example

```
- ( 1.0, "abc", 2);
 val it = (1.0,"abc",2) : real * string * int
- ( [1.0], ("abc", 2));
 val it = ([1.0],("abc",2)) : real list * (string * int)
```

Note that elements in the tuple do not need to have the same type
–like a record or struct but with no name.

A 2-tuple is called a pair.

Odd Tuples

There is no 1-tuple as such: `(pi)` **=** `pi`**.**

There is only one 0-tuple, namely `()`**. (Here we can't drop the parentheses!).**

This has a type inhabited by `()` **and** `()` **alone, namely** `unit` **(similar to C's "void").**

```
- ( 1.0 );
 val it = 1.0 : real
- ( );
 val it = () : unit
```

Passing Multiple Arguments to a Function

Recall our function to compute the area of a circle:

```
fun circle_area r = pi * r * r;
```

Now we want to write a function to compute the area of a rectangle given its width `w` and height `h`.

"Problem": **ML functions only take one argument!**

Solution: **Use a tuple.**
In effect, tuples allow us to pass more than one argument to a function.

First attempt:

```
- fun rect_area (w,h) = w * h;
```

But remember `int` v. `real` types:

```
 val rect_area = fn : int * int -> int
```

instead we need:

```
- fun rect_area (w,h) = (w * h):real;
 val rect_area = fn : real * real -> real
```

Another Example

Write a Boolean function `mem (y,l)` which returns true if `y` is a member of the list `l` and false otherwise.
E.g. `mem (2,[1,2,3])` should return `true`.

We need to use a tuple because `mem` has two arguments.

What if the list `l` is empty? Return `false`.

What if the list `l` is `x::xs`? Two cases:
– `y` is the first element `x`;
– `y` is an element of the remainder of the list `xs`.

What is the actual ML function?

(Also see "Curried" functions later.)

Insertion Sort

With tuples and lists we are now in a position to do some real programming!

Write a function `insert (x,ys)` **which appropriately inserts integer** `x` **into the sorted list of integers** `ys`**.**
E.g. `insert (2,[1,3])` **should return** `[1,2,3]`**.**

Now write a function `insertSort ys` **which returns the list** `ys` **sorted (e.g.,** `insertSort [2,1,3]` **should return** `[1,2,3]`**), and does so by using the insertion sort algorithm.**

Efficient Reverse

In a previous lecture we gave a definition of a function `reverse` that reversed a list. Unfortunately its time complexity is quadratic in the size of the input list.

It is possible to give a better definition of `reverse` which is linear in the size of the input list.

The idea is quite simple–we construct the reversed list as we traverse the original list, and when we have finished the original list we return this.

For instance consider reversing `[1,2,3]`

The ML program to do this is:

```
(* move elements from the first list to the
   second and then return this list *)
fun rev1 ([], ys) = ys
 |  rev1 ((x::xs), ys) = rev1 (xs,(x::ys));

fun reverse xs = rev1 (xs,[]);
```

Structuring Data with Tuples

Tuples are not only useful for passing more than one argument to a function, they can also be used to structure data.

Imagine that we wish to build a module for 2-D vectors. We can represent a vector by a tuple (x, y).

```
- val zerovec = (0.0, 0.0);
 val zerovec = (0.0, 0.0) : real * real

- val a = (3.0, 4.0);
 val a = (3.0, 4.0) : real * real

- fun positive (x, y) = x > 0.0 andalso y > 0.0;
 val positive = fn : real * real -> bool
- positive a;
 val it = true : bool
```

Tuples of Tuples

Now we want a function `addvec` to add two vectors. This needs to take two vectors which are themeselves tuples, i.e. tuples of tuples.

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

- addvec (zerovec, a);
 val it = (3.0,4.0) : real * real
```

Remember to always use brackets when passing arguments to a function which expects a tuple:

```
- addvec zerovec a;
 stdIn:24.1-24.17 Error: operator and operand
 don't agree [tycon mismatch]
   operator domain: (real * real) * (real * real)
   operand:         real * real
   in expression:
     addvec zerovec
```

Notice how we use pattern matching to access the tuple elements.

Returning Multiple Values from a Function

Recall our function to compute the area of a circle:

```
fun circle_area r = pi * r * r;
```

Say we want to compute both the circumference and the area...

Problem: **ML functions only return one result.**

Solution: **Use a tuple.**
In effect, tuples allow us to return more than one result from a function.

```
- fun circle_stats r = (pi * r * r, 2.0 * pi * r);
 val circle_stats = fn : real -> real * real

- circle_stats 4.0;
 val it = (50.26544,25.13272) : real * real
```

Let Expressions

Sometimes we need to create some temporary values – i.e. local variables.
the `let in end` **expression allows you to do this. The general form is:**

```
  let
    val <first variable> = <first expresion>;
          ... decs, e.g. val, fun, ...
    val <last variable> = <last expresion>
  in
    <expression>
  end
```

Each local `variable` **is visible in all subsequent** `expressions` **until** `end`**.**

```
fun circle_area r =
  let
    val pi = 3.14159;
  in
     pi * r * r
  end;
```

val circle_area = fn : real -> real

Let Expressions (Cont.)

Let expressions are also useful when a function returns a complex data structure, such as a tuple.

```
fun split [] = ([],[])
 |  split [a] = ([a],[])
 |  split (a::b::cs) =
       let
          val (M,N) = split(cs)
       in
          (a::M, b::N)
       end;

 val split = fn : 'a list -> 'a list * 'a list

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

Notice the use of pattern matching in the "variable position."

Split (cont.)

How does `split` **really work? Consider** `split [1,2,3]`**.**

The Use of Parentheses

In ML you only need to use parentheses to resolve ambiguities. So you normally write `x` or `5` rather than `(x)` or `(5)`, even in function applications: `f x`.

Parentheses are part of the syntax for tuples. So ML programmers write `plus (x,y)`, but they think of this as `plus` applied to the argument `(x,y)`, which is a tuple.

Again, you could write `plus( (x,y) )`, but why would you?

Pattern Matching

We have seen simple pattern matching for both lists and tuples. Patterns and pattern matching can be quite complex.

E.g. does `([1,2,3],5)` **match** `(x::y::zs,w)`**?**

Try matching the expression/parse trees for

```
     ,                            ,
  -------                      -------
  ::    5                      ::    w
-------                      -------
1     ::                     x     ::
   ------                       ------
   2    ::                      y    zs
      ------
      3  nil
```

Restrictions on Patterns

Variables can only occur once:

```
fun eqlist []          = true
  | eqlist [_]         = true
  | eqlist (x::x::xs) = eqlist (x::xs);
```

```
stdIn:25.6-27.41
Error: duplicate variable in pattern(s): x
```

Instead:

```
fun eqlist []          = true
  | eqlist [_]         = true
  | eqlist (x::y::xs) = x=y andalso eqlist (y::xs);
```

Where applicable, patterns should exhaust all possibilities:

```
fun prod [x] = x : int
  | prod (x::xs) = x * (prod xs);
stdIn:20.1-21.32 Warning: match nonexhaustive
          x :: nil => ...
          x :: xs  => ...

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

Indeed, `prod []` **will give a run-time error.**

Restrictions on Patterns (Cont.)

Cannot do arithmetic in patterns:

```
fun square(0) = 0
 |  square(x+1) = 1 + 2*x + square(x);

stdIn:1.5-28.39
Error: non-constructor applied to argument in pattern: +
```

In general, you cannot match function calls, including +, only data constructors.

Exercise: **Is the following legal?**

```
fun eqlist []          = true
  | eqlist [_]         = true
  | eqlist ([x,y]@xs) = x=y andalso eqlist (y::xs);
```

Up to now we have only seen the following data constructors: `(,)` **for tuples,** `nil`, `[]`, **and** `::` **for lists, and the constants for integers, reals, bools, characters, and strings.**

The rest (such as `+`, `-`, `@`, **etc, are functions). Why can't they be matched?**

Naming Patterns – `As`

Sometimes you wish to match the argument to a pattern but also to have a name for the argument for later use. `as` **allows you to do this:**

```
fun merge([],M) = M
 |  merge(L,[]) = L
 |  merge(L as x::xs, M as y::ys) =
      if x < y then x::merge(xs,M)
      else y::merge(L,ys);
```

`val merge = fn : int list * int list -> int list`

Rewrite `eqlist` **to be more efficient using “**`as`**”.**

Summary

We have looked at programming with:

- Tuples
- Pattern-matching

Homework

- Read Sections 2.4, 3.3, and 3.4 of Ullman.
- Using a tuple to represent a complex number write functions to add, subtract and multiply two complex numbers.
- What is(are) the type(s) of +?
- Using `split` and `merge` write `mergeSort`.
- Write a function to implement quicksort;
  simply use the first element of the list as the estimate of the median element.

SML Programming IV

In this lecture we will look at:

- **Matches**
- **Simple Output**
- **Exceptions**
- **Polymorphism**

Matches

We have seen pattern matching in function definitions. More generally ML provides matches. They have form

```
<pattern 1> => <expression1> |
<pattern 2> => <expression2> |
 ...
<pattern n> => <expressionn>
```

The patterns are tried in order. The result is the expression corresponding to the first pattern matched.

Matches

We can write the function `len`

```
fun len []      = 0
  | len (x::xs) = 1 + len xs;
```

or as

```
val rec len =
    fn []      => 0
     | (x::xs) => 1 + len xs;
```

The `rec` indicates that the definition is recursive, that is the definition of `len` is in terms of `len`.

More generally

```
fun f P1 = E1 | f P2 = E2 | ... | f Pk = Ek;
```

is short for

```
val rec f = fn P1 => E1 | P2 => E2 | ... | Pk => Ek;
```

Notice that `fn P1 => E1 | P2 => E2 | ... | Pk => Ek` is an expression – it is an anonymous function.

Case Expressions

Matches are also used in case expressions:

```
case <expression> of <match>
```

```
fun len L =
  case L of
    []      => 0
  | (x::xs) => 1 + len xs;
```

Note that

```
  if E1 then E2 else E3
```

is actually short for

```
  case E1 of true => E2 | false => E3
```

Print

Because the programming environment for ML is interactive we have not needed input/output. However, in practical applications programs need to read and write data to and from files.

ML provides a built-in `print` operator that writes a string to standard output.

```
- print "hello world\n";
 hello world
 val it = () : unit
```

Note the type of `print`: it always returns the value `():unit`. The fact that it has the side effect of printing a string is not reflected in its type.

To print values other than strings, we must use library functions to first convert the value into a string.

```
- fun printReal r = print (Real.toString r);
 val printReal = fn : real -> unit
- printReal 4.0;
 4.0val it = () : unit
```

Statement Lists

It is often useful to execute a sequence of two or more "statements" with side effects such as `print` expressions.

In ML this is done using

```
(<first expression> ; ... ; <last expression>)
```

The statements are executed sequentially, much like statements in a C statement block.

However in ML everything is an expression, and the value of a statement list is the value of the last expression in the list.

```
fun printListOfInt nil = ()
 |  printListOfInt (x::xs) = (
        print (Int.toString x);
        print "\n";
        printListOfInt (xs)
     );

 val printListOfInt = fn : int list -> unit
- printListOfInt [3,4,5];
 3
 4
 5
 val it = () : unit
```

Do not confuse `(p;q;r)`, `(p,q,r)` and `[p,q,r]`. What are the differences?

Exceptions

**Sometimes illegal values such as `hd []` appear during evaluation or missing patterns are encountered.
In ML these raise an exception.**

An exception is raised where the failure is discovered, and ML allows for it to be handled (caught) elsewhere—maybe far away.

```
- hd([]:real list);
 uncaught exception Empty
   raised at: boot/list.sml:36.38-36.43

- 5 div 0;
 uncaught exception divide by zero
   raised at: <file stdIn>

- 5.0/0.0;
 val it = inf : real


- (5 div 0) handle zero => 99
                (* ---------- a match *) ;
val it = 99 : int
```

Exceptions

Raising and catching exceptions is the standard approach to error handling in modern programming languages.

Exceptions were introduced in PL/1. They are provided in Ada, C++, Haskell and Java as well as ML.

Exceptions provided a structured form of jump that can be used to:
– jump out of a loop or function invocation
– pass data as part of the jump
– return to a program point that was set up to continue the computation.

Exception mechanisms provide:

- a way of raising an exception which aborts current computation and jumps to
- the exception handler which catches the exception and executes the code for that exception.

Programmer Defined Exceptions

In ML programmers can declare their own exceptions. When an exception is raised, it is transmitted by all ML functions until an exception handler detects it.

If the result of evaluating expression e is an exception E, the result of $f\ e$ is E, unless f includes a handler for E.

```
exception BadFacArg;

fun fac 0 = 1
 |  fac n =
  if n > 0 then n * fac(n-1)
  else raise BadFacArg;

- fac 3;
 val it = 6 : int
- fac ~1;

 uncaught exception BadFacArg
   raised at: stdIn:24.23-24.32
```

Programmer Defined Exceptions (Cont)

Exceptions can have arguments:

```
exception BadFacArg of int;

fun fac1 0 = 1
 |  fac1 n =
  if n > 0 then n * fac1(n-1)
  else raise BadFacArg n;

fun fac n = fac1 n
  handle
    BadFacArg n => (
      print("invalid argument to fac: ");
      print(Int.toString(n));
      print("\n");
      0
      );
```

Programmer Defined Exceptions (Cont)

```
- use "fac.ml";
 [opening fac.ml]
 exception BadFacArg of int
 val fac1 = fn : int -> int
 val fac = fn : int -> int
 val it = () : unit
- fac 3;
 val it = 6 : int
- fac 0;
 val it = 1 : int
- fac ~1;
 invalid argument to fac: ~1
 val it = 0 : int
```

Polymorphic Functions

Consider the identity function:

```
- fun id x = x;
 val id = fn : 'a -> 'a
```

This function works on all types of arguments: reals, lists, functions, etc..

It is not overloaded, it is polymorphic. Its type

```
'a -> 'a
```

is a type scheme, and `'a` is a type variable.

Some more examples:

```
 fun len []      = 0
   | len (x::xs) = 1 + len xs;
 val len = fn : 'a list -> int

 fun reverse []    = []
   |  reverse (x::xs) = (reverse xs) @ [x];
 val reverse = fn : 'a list -> 'a list

 fun fst (x, y) = x;
 val fst = fn : 'a * 'b -> 'a

- fst (("abc", 7), ("def", 6));
 val it = ("abc",7) : string * int
- fst (3.0,1);
 val it = 3.0 : real
```

Polymorphism

Polymorphic type checking is a secure yet flexible type discipline. Most ML programs need not be cluttered with type specifications, as types are deduced automatically.

ML is strongly typed:

"Well-typed programs cannot go wrong."

Once the type checker has accepted the program, no type errors can occur at run-time.

(NB. Division by zero is not a "type error".)

A polymorphic function can have different types within the same expression:

```
- len [1.0,2.0] + len ["abc","def"]
 val it = 4 : int
```

Polymorphism (Cont.)

There is a restriction with polymorphic parameters in a function definition

```
- fun twice f = f o f;
 val twice = fn : ('a -> 'a) -> 'a -> 'a

- twice (op not) true;  (* = not(not true) *)
 val it = true : bool

- twice twice (fn x => x+1) 7;
 val it = 11 : int

- hd (hd [[1,2],[3,4]]);
 val it = 1 : int

- (twice hd) [[1,2],[3,4]];
 stdIn:1.1-23.9 Error: operator and operand don't agree [c
  operator domain: 'Z list -> 'Z list
  operand:         'Z list -> 'Z
  in expression:
    twice hd
```

(The error amounts to an 'occurs check' in the type checker's unification algorithm, `f:'a->'a`**,** `hd:'b list->'b`**,** `f=hd`**, hence** `'b='b list`**.)**

Equality Types

There is a slight problem with polymorphism and equality.

```
- fun mem(x,[])     = false
   |  mem(x,y::ys) = (x = y) orelse mem(x,ys);
 val mem = fn : ''a * ''a list -> bool
```

If `''a` **is a function type or a real,** `mem` **will want to compare.**

```
- mem(3,[1,2,3]);
 val it = true : bool
- mem(3.0,[1.0,2.0,3.0]);
 stdIn:140.1-140.23 Error: operator and operand
             don't agree [equality type required]
   operator domain: ''Z * ''Z list
   operand:         real * real list
   in expression:
     mem (3.0,1.0 :: 2.0 :: <exp> :: <exp>)
```

ML will correctly not allow this since function types and reals(!) are not comaparable in ML.

Equality Types (Cont.)

The built-in function = is polymorphic `''a * ''a -> bool`.

We don't need to write a special function to test list equality:

```
- [2,3,4] = [2,3,4];
 val it = true : bool
```

The types admitting equality testing are called equality types. Type variables ranging over these are `''a`, `''b`, etc.

Equality testing is possible for most types, including tuples and lists made from equality types.

(The mechanism for equality types is greatly generalized to 'type classes' in the FP language Haskell.)

Equality Types (Cont.)

What is the difference between these three functions? Why?

```
fun len1 []      = 0
  | len1 (x::xs) = 1 + len1 xs;
```

val len1 = fn : 'a list -> int

```
fun len2 x = if x=[] then 0 else 1 + len2(tl x);
```

val len2 = fn : ''a list -> int

```
fun len3 x = if null x then 0 else 1 + len3(tl x);
```

val len3 = fn : 'a list -> int

Summary

We have looked at:

- Matches
- Print and Statements
- Exceptions
- Polymorphism

Homework

- Read Sections 4.1, 5.1, 5.2, 5.3 of Ullman.
- Add appropriate exception handling to `max`.
- Write a function to print out a list of real numbers.
- Write a polymorphic function to find the largest item in a list. If you can't do this explain why you are not stupid.

SML Programming V

In this lecture we will look at advanced data structures and types:

- **Type definitions**
- **Datatype definitions**
- **Records**

Review

Types in ML, as in most typed languages, are defined recursively with a basis of primitive types and with rules for defining more complex types from these.

Basic types: `int`, `real`, `char`, `bool`, `unit`, `exn`, `string` **(and** `instream`, `outstream` **which we will see later when dealing with I/O), plus** `'a` **and** `''a`

Product type: `'a * 'b`, **more generally** `'a * 'b * 'c *....`

Function type: `'a -> 'b`.

Type constructors: We have met `list`, **ie** `'a list`.

In this lecture we shall see how to define your own type constructor.

Type Definitions

We can define a new type in terms of an existing type. It acts as an abbreviation (renaming). Their general form is:

```
type <identifier> = <type expression>.
```

Much like a `typedef` **in C. For instance:**

```
type ints = int list;
```

Also, recall our functions for manipulating complex numbers:

```
type complex = real * real;
fun addComplex ((x1,y1),(x2,y2)) =
     (x1+x2,y1+y2): complex;
fun subComplex ((x1,y1),(x2,y2)) =
     (x1-x2,y1-y2): complex;
fun multComplex ((x1,y1),(x2,y2)) =
     (x1*x2 - y1*y2, x1*y2 + x2*y1): complex;
```

```
type complex = real * real
val addComplex = fn :
     (real * real) * (real * real) -> complex
val subComplex = fn :
     (real * real) * (real * real) -> complex
val multComplex = fn :
     (real * real) * (real * real) -> complex
val it = () : unit
```

Note that ML recognizes that `complex` **and** `real * real` **are the same here.**

Parameterized Type Definitions

More generally, we can define a type with parameters (type variables) in the declaration:

```
type (<list of type variables>) <identifier> =
 <type expression>.
```

Imagine an association list which has a domain type (the key) and a range type (the value). It "maps" domain elements to range elements, and implements a dictionary.

```
- type ('d, 'r) assoclist = ('d * 'r) list;
 type ('a,'b) assoclist = ('a * 'b) list

- val phonenumbers = [("Peter",56790345),
      ("Nicole",56790345)]:
      (string,int) assoclist;
 val phonenumbers = [("Peter",56790345),
      ("Nicole",56790345)]:
      (string,int) assoclist;
```

The `<identifier>` **in the type definition identifies a type constructor (of a given arity – the number of its parameters).**

The `<type expression>` **will be built from type constructors and from type variables appearing in** `<list of type variables>`

Datatypes

ML has a powerful mechanism for defining new types called datatypes. Their general form is:

```
datatype (<list of type variables>) <identifier> =
                  <first constructor expression> |
                  <second constructor expression> |
                   ...
                  <last constructor expression>
```

The `<identifier>` **again identifies a (data)type constructor.**

Each `<constructor expression>` **is built from a data constructor whose arguments (if any) are type variables appearing in** `<list of type variables>` **and type constructors.**

For example:

```
- datatype fruit = Apple | Pear | Grape;
 datatype fruit = Apple | Grape | Pear
```

Here `fruit` **is the new data type and** `Apple`, `Pear`, `Grape` **are data constructors with no arguments.**

Do not get confused between types, type constructors, data values and data constructors: `int` **and** `int list` **are types while** `int/0` **and** `list/1` **are type constructors,** `Apple` **and** `Apple::[]` **are data values while** `Apple/0` **and** `::/2` **are data constructors.**

Datatypes (Cont.)

```
- fun isApple x = (x=Apple);
 val isApple = fn : fruit -> bool

- isApple(Apple);
 val it = true : bool
- isApple(Pear);
 val it = false : bool
- isApple(Banana);
 stdIn:21.9-21.15 Error:
   unbound variable or constructor: Banana
```

Intutively, a variable of type `fruit` is just a tag indicating which kind of data constructor it is.
(Like an `enum` in C++).

However, data constructors can be much more powerful...

Tagged Union Datatype Example

Data constructors can have parameters. The constructor does not affect the parameter(s); it simply adds a tag to the parameter(s).

The tag allows parameters constructed in different ways to be distinguished by pattern matching.

```
type name = string;
type id = int;
type degree = string;

datatype student =
       Bachelor of name*id*degree |
       PhD of name*id |
       Master of name*id*degree;

fun name (Bachelor(n,_,_)) = n
  |   name (PhD(n,_)) = n
  |   name (Master(n,_,_)) = n;

 val name = fn : student -> name
```

Note that the extra brackets are needed if you use pattern matching on data constructors that have parameters.

Also note that while type constructors build types, data constructors build values of a type.

Falafel Rolls

Data constructors can be recursive.

A falafel roll has pita bread as its base and might have the following ingredients

- **falafel balls**
- **tabouli**
- **pickles**
- **hummus**
- **chilli**

We can use the following datatype to model falafel rolls

```
datatype falafelRoll =
  Pita |
  Falafel of falafelRoll |
  Tabouli of falafelRoll |
  Pickles of falafelRoll |
  Hommus of falafelRoll |
  Chilli  of falafelRoll;

val yummy = Chilli(Falafel(Tabouli(Falafel Pita)))
and hot   = Chilli(Chilli Pita);
```

Falafel Rolls

A real falafel roll is one with some falafel balls in it:

```
datatype falafelRoll =
      Pita |
      Falafel of falafelRoll |
      Tabouli of falafelRoll |
      Pickles of falafelRoll |
      Hommus of falafelRoll |
      Chilli  of falafelRoll;

fun realFalafel Pita = false
|    realFalafel (Falafel(r)) = true
|    realFalafel (Tabouli(r)) = realFalafel(r)
|    realFalafel (Pickles(r)) = realFalafel(r)
|    realFalafel (Hommus(r)) = realFalafel(r)
|    realFalafel (Chilli(r)) = realFalafel(r);

 val realFalafel = fn : falafelRoll -> bool

- realFalafel yummy;
 val it = true : bool
- realFalafel hot;
 val it = false : bool
```

Complex Datatypes

Generally, in datatype definitions:

- **Type variables can be used to parametrize the definition.**
- **The data constructors can take arguments.**
- **They may be recursive.**

Recall the general form:

```
datatype (<list of type variables>) <identifier> =
  <first constructor expression>   |
  <second constructor expression>  |
     ...
  <last constructor expression>
```

E.g. 1: List Datatype

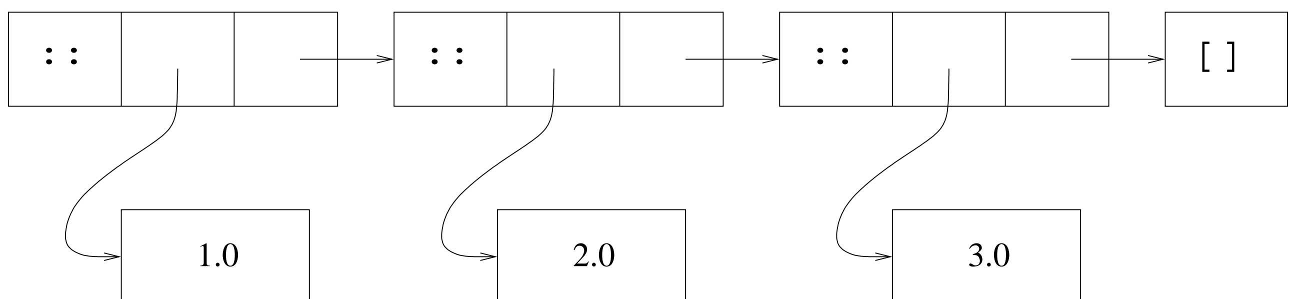
For example the `list` data type is conceptually defined by

```
datatype 'a list =
  nil |
  :: of 'a * 'a list;
```

The list `[1.0,2.0,3.0]` which is shorthand for

```
1.0 :: 2.0 :: 3.0 :: nil
```

is represented by



NB. Do not to confuse data constructors with functions:
– data constructors build data values
– functions compute data values.

E.g. 2: Parse Trees

```
datatype Uopr = uminus | knot;               (* operators *)
datatype Bopr = plus | minus | times | divide |
                eq | ne | le | lt | ge | gt |
                conj | disj ;

datatype Exp = binexp  of Exp*Bopr*Exp |  (* expressions *)
               unexp   of Uopr * Exp |
               varid   of Ide |
               numeral of int |
               expErr  of string
and      Cmd = assign   of Ide * Exp |        (* commands *)
               ifcmd    of Exp * Cmd * Cmd |
               whilecmd of Exp * Cmd |
               cmdlist  of Cmd * Cmd |
               proccall of Ide |
               write    of Exp |
               block    of Dec * Cmd |
               cmdErr   of string
and      Dec = vardec   of Ide |          (* declarations *)
               declist  of Dec*Dec |
               procdec  of Ide*Cmd |
               decErr   of string;
```

The parts of the syntax (parse trees) of a language are mutually recursive – www.csse.monash.edu.au/~lloyd/tildeFP/SML/1997/semantics.toy/

Option Datatype

Sometimes it is useful to be able to return a value if one exists or else a flag indicating that no value exists.

For instance, when reading a single character from standard input it is useful to either return the character or no character if EOF has been reached.

The built-in datatype `option` **allows this. Its definition is similar to:**

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

fun isSome NONE = false
 |  isSome (SOME(x)) = true;

fun valOf (SOME(x)) = x;
```

```
 datatype 'a option = NONE | SOME of 'a
 val isSome = fn : 'a option -> bool
 option.ml:8.1-8.24 Warning: match nonexhaustive
   SOME x => ...

 val valOf = fn : 'a option -> 'a
 val it = () : unit
```

Binary Trees

The following defines a kind of “labelled” binary tree:

```
datatype 'label btree =
  Empty |
  Node of 'label btree * 'label * 'label btree;

datatype 'a btree = Empty |
    Node of 'a btree * 'a * 'a btree

- val names = Node(Empty,"Kim",Empty);
 val names = Node (Empty,"Kim",Empty) : string btree
```

Another Example

Exercise: **Write a function to compute the height of a labelled binary tree. What is its type?**

Hint: `Int.max` **may be useful. (Also see** `Real.max`**)**

Records

A record is like a tuple, but its components (fields) are named and situated between curly brackets.

The records

```
{name = "Jones", age = 25, height = 180}
{height = 180, name = "Jones", age = 25}
```

are equal.

```
val jones_info =
  {name = "Jones",
   age = 25,
   height = 180
  };

val jones_info = {age=25,height=180,name="Jones"}
  : {age:int, height:int, name:string}
```

Selection of a field is done using #⟨*label*⟩.

```
- #age jones_info;
val it = 25 : int
- #height(jones_info);
val it = 180 : int
```

Tuples

Actually a tuple is really a record whose fields are called `#1, #2, ...`

Thus:

```
- #2 (3.6, "Select Me", 6);
 val it = "Select Me" : string
```

Records (Cont.)

We can pick fields and assign by matching:

```
- val {age = theage, height = theheight,...} = jones_info;
 val theage = 25 : int
 val theheight = 180 : int
```

Or we can use the field name as the variable itself:

```
- val {name, age,...} = jones_info;
 val age = 25 : int
 val name = "Jones" : string
```

NB. The "`...`" above is part of the ML syntax – it means "the rest of the record" (like a wild card).

Records (Cont.)

We can give the record type a name and let functions have arguments and/or results of the type:

```
 type info = {name : string,
     age : int,
     height : int
     };
  type info = {age:int, height:int, name:string}

- fun silly ({age, height,...}) = height-age;
 stdIn:55.1-55.39 Error:
  unresolved flex record (need to know the names
  of ALL the fields in this context)
   type: {age:'Y, height:'Y; 'Z}
- fun silly ({age, height,...} : info) = height-age;
 val silly = fn : info -> int
- silly jones_info;
 val it = 155 : int
```

Record Exercise

- **Give a type definition for a customer record `customer` containing an integer ID and a name (as a string).**
- **Now give a function `lt_customer` to compare two customer records based on the ID.**

Summary

We have looked at:

- Type definitions
- Datatype definitions
- Records

Homework

- Read Chapter 6 and Section 7.1 of Ullman.
- Modify the `falafelRoll` definition so that it keeps track of the number of falafel balls in the roll.
- Now write a function which returns the total number of falafel balls in the roll.
- Write a datatype which represents hamburgers. Define a function which checks that a hamburger is without cheese.
- Write a function to traverse the elements in a labelled binary tree in infix-order and return the result in a list.

SML Programming VI

In this lecture we will look at:

- **Higher-order functions**

This is a very important topic, one of the keys to functional programming.

Functions as Expressions

Recall that we could write `len` as

```
val rec len = fn
      []      => 0
    | (x::xs) => 1 + len xs;
```

Notice that `fn P1 => E1 | P2 => E2 | ...  | Pk => Ek` is an expression – it is an anonymous function.

As far as ML is concerned function definitions are just expressions and so, like any other expression they do not need a name.

```
- (fn x => x+1)(3);
 val it = 4:int

- fn x => x+1;
 val it = fn : int -> int
- it 3;
 val it = 4 : int
```

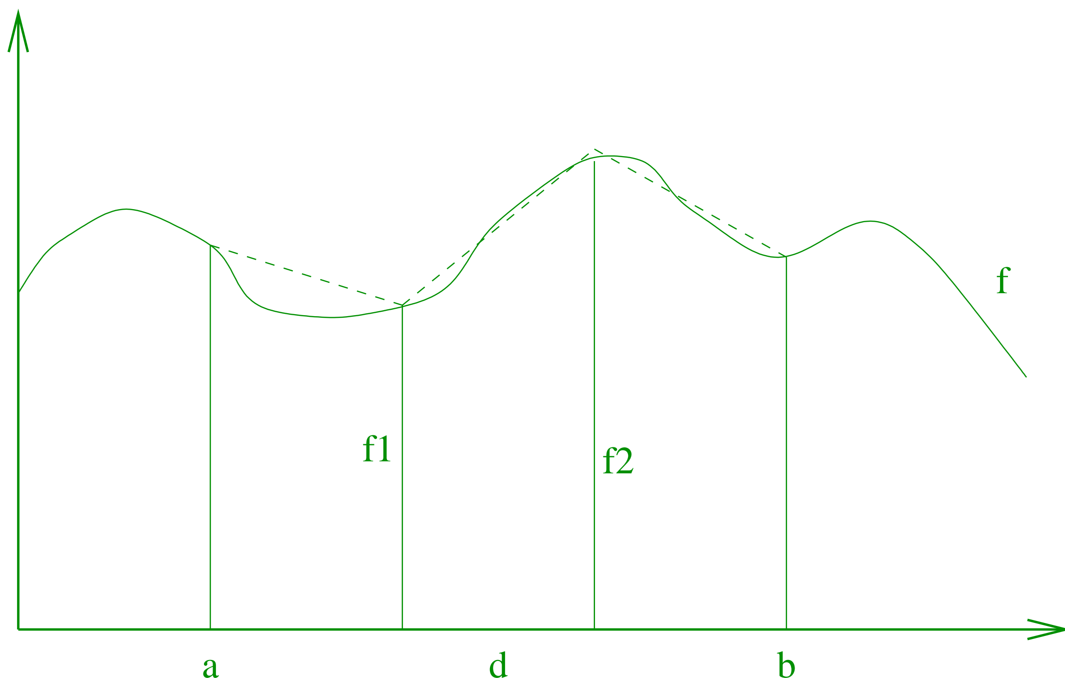
Higher Order Functions

Like any other expression, functions can be used as arguments to functions.

Functions that take functions as arguments are said to be higher-order.

Higher-order programming is easy to do in ML.

Recalling *CSE2304 Alg. and D.S.*, **one way to compute the integral of a function is to use the trapezoidal rule:**



The area of the ith trapezoid is

$$\delta \times \frac{f(a + (i - 1) \times \delta) + f(a + i \times \delta)}{2}$$

where $\delta = \frac{b-a}{n}$.

Higher Order Functions

```
(* function for integration using the trapezoid rule *)
fun trap(a,b,n,F) =
  if n<=0 orelse b-a <= 0.0 then 0.0
  else
    let
       val delta = (b-a)/real(n)
    in
       delta * (F(a)+F(a+delta))/2.0 +
       trap(a+delta,b,n-1,F)
  end;

 fun square x = x*x : real;
```

Use this to compute $\int_0^1 x^2 dx$.

```
- use "trapezoidal.ml";
 [opening trapezoidal.ml]
 val trap = fn:real * real * int * (real -> real) -> real
 val square = fn : real -> real
 val it = () : unit

- trap(0.0, 1.0, 10, square);
 val it = 0.335 : real

- trap(0.0, 1.0, 10, (fn x => x*x));
 val it = 0.335 : real
```

Another Example

Last week you were asked to write a polymorphic function to find the largest item in a list.

```
(* finds the maximum element in a list *)
exception EmptyList;
fun max [] = raise EmptyList
  | max [x] = x
  | max (x::xs) =
     let val xsmax = max xs in
       if x > xsmax then x else xsmax
     end;

 val max = fn : int list -> int

- max [4,6,3,2,6,8];
 val it = 8 : int
```

This is not possible since > is not polymorphic. However...

Another Example (Cont.)

...we can get around the problem by writing a function which takes a comparison function gt **as an argument.**

```
fun max (gt, []) = raise EmptyList
  | max (gt, [x]) = x
  | max (gt, (x::xs)) =
     let val xsmax = max (gt, xs) in
       if gt (x,xsmax) then x else xsmax
     end;

 val max = fn : ('a * 'a -> bool) * 'a list -> 'a

- val igt = fn (x:int,y) => x>y;
 val igt = fn : int * int -> bool
- val sgt = fn (x:string,y)  => x>y;
 val sgt = fn : string * string -> bool
- max (igt,[4,6,3,2,6,8]);
 val it = 8 : int
- max (sgt,["abc","def","ghi"]);
 val it = "ghi" : string
```

The Simple Map Function

The simple map function (Ullman s5.4.2) takes a function F and a list $[a_1, ..., a_n]$ and returns the list $[F(a_1), ..., F(a_n)]$.

(Similar to `mapcar` in Lisp).

```
fun simpleMap(F,[]) = []
 |  simpleMap(F,x::xs) =
       F(x)::simpleMap(F,xs);

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

- simpleMap(square,[1.0,4.0,3.0]);
 val it = [1.0,16.0,9.0] : real list

- simpleMap(~,[1,2,3]);
 val it = [~1,~2,~3] : int list
```

The Simple Map Function (Cont.)

How does `simpleMap(~,[1,2])` **execute?**

Reduce

The reduce function takes a binary function F and a list $[a_1, ..., a_n]$ and returns

$$F(a_1, F(a_2, F(..., F(a_{n-1}, a_n)))).$$

```
exception EmptyList;

fun reduce(F,[]) = raise EmptyList
 |  reduce(F,[a]) = a
 |  reduce(F,x::xs) = F(x,reduce(F,xs));

fun plus(x,y) = x+y;

 exception EmptyList
 val reduce = fn : ('a * 'a -> 'a) * 'a list -> 'a
 val plus = fn : int * int -> int

- reduce(plus,[3,4,7,10]);
 val it = 24 : int

- reduce(+,[3,4,7,10]);
 stdIn:39.8 Error: expression or pattern begins with
                   infix identifier "+"

- reduce(op +,[3,4,7,10]);
 val it = 24 : int

- reduce(fn (x,y)=>x+y, [3,4,7,10]);
 val it = 24 : int
```

The Reduce Function (Cont.)

How does `reduce(plus,[3,4])` **execute?**

`reduce` **was in APL (Iverson c1960).**

`reduce` **is similar to** `foldr` **– see later.**

Filter

The filter function takes a Boolean function P and a list $[a_1, ..., a_n]$ and returns the sublist whose elments satisfy P.

```
fun filter(P,[]) = []
 |  filter(P,x::xs) =
      if P(x) then x::filter(P,xs)
      else filter(P,xs);

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

- filter(fn(x)=> x>10, [1,10,23,45,8]);
 val it = [23,45] : int list
```

Example

Exercise: **Consider a function** `concat` **which takes a list of strings and concatenates them all.**

```
- concat ["abc","def","ghi"];
 val it = "abcdefghi" : string
```

Write a version which is recursive and write another which uses `reduce`**.**

Returning a Function

Since functions are "first class values", a function can also return a function!

Such functions are also said to be higher-order.

For example we can have a function which takes a number x and returns a function to add x on to its argument:

```
fun add x = (fn y => x+y);
```

val add = fn : int -> int -> int

Or if x is positive adds it and otherwise subtracts it:

```
fun strange x = if x > 0 then (fn y => x+y)
                  else (fn y => y-x) ;
```

val strange = fn : int -> int -> int

Returning a Function (Cont.)

How do we use these?

```
- add 3;
 val it = fn : int -> int
- it 4;
 val it = 7 : int
- (add 3) 4;
 val it = 7 : int
```

Since function application is considered left-associative we can even leave out the parentheses —

```
- add 3 4;
 val it = 7 : int
```

Currying of Functions

Consider a function with a pair as argument:

```
- fun add' (x,y) = x + y : int;
 val add' = fn : int * int -> int
```

But `add x y = add' (x,y)`**!**

What is the difference?

`add'` **requires both** `x` **and** `y` **before it can be evaluated while** `add` **only requires** `x`**, the other "argument"** `y` **can be given later.**
E.e. `add 1` **is well defined but** `add'(1,?)` **is not.**
Thus `add` **is more flexible.**

`add` **is the curried version of** `add'`**.**
(Named after the logician Haskell Curry, although he (1980) attributed the technique to Schonfinkel.)

It is often possible to avoid tuples as arguments to functions, through currying.

Currying of Functions (Cont.)

ML makes it easy to define curried functions.

We can define `add` **by**

```
fun add x y = x+y;
 val add' = fn : int -> int -> int
```

This is just shorthand for

```
fun add x = (fn y => x+y);
```

Which is just shorthand for

```
val add = (fn x => (fn y => x+y));
```

"Real" ML programmers generally use curried functions rather than tuples.

Realistic Example: Binary Search Trees

```
datatype 'label btree =
  Empty |
  Node of 'label btree * 'label * 'label btree;

fun lookup lt Empty x = false
 |  lookup lt (Node(left,lbl,right)) x =
      if lt(x,lbl) then (lookup lt left x)
      else if lt(lbl,x) then (lookup lt right x)
      else (* x=lbl *) true;

fun insert lt Empty x = Node(Empty,x,Empty)
 |  insert lt (T as Node(left,lbl,right)) x =
      if lt(x,lbl) then Node((insert lt left x),lbl,right)
      else if lt(lbl,x) then Node(left,lbl,(insert lt right
      else (* x=lbl *) T; (* already in tree *)
```

```
 val lookup = fn :
       ('a * 'a -> bool) -> 'a btree -> 'a -> bool
 val insert = fn :
       ('a * 'a -> bool) -> 'a btree -> 'a -> 'a btree
```

Binary Search Trees (Cont.)

```
- val tree = Node(Empty,"harald",Node(Empty,"peter",Empty))
 val tree =
 Node (Empty,"harald",Node (Empty,"peter",Empty)) : string
- val tree = insert (op <) tree "karen";
 val tree =
 Node (Empty,"harald",Node (Node #,"peter",Empty)) : strin
- lookup  (op <) tree "karen";
 val it = true : bool
- lookup  (op <) tree "carine";
 val it = false : bool
```

Built-In Higher Order Functions

Function Composition. Recall

$$(G \circ F)(x) = G(F(x)).$$

We can define this by

```
fun comp F G x = G(F(x));
 val comp = fn : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

- val mystery = comp square square;
 val mystery = fn : real -> real

- mystery 5.0;
 val it = 625.0 : real
```

ML provides the infix operator "o" to compose two functions.

```
- val mystery = square o square;
 val mystery = fn : real -> real
```

Built-In Higher Order Functions (Cont)

Map. ML provides a curried version of map.

```
- map;
 val it = fn : ('a -> 'b) -> 'a list -> 'b list

- val mystery = map square;
 val mystery = fn : real list -> real list

- mystery [1.0,16.0,9.0];
 val it = [1.0,256.0,81.0] : real list
```

Foldr and Foldl. More powerful curried versions of reduce. The definition of `foldr` is:

```
fun foldr F y [] = y
 |  foldr F y (x::xs) = F(x, foldr F y xs)
```

```
- val sumList = foldr op + 0;
 val sumList = fn : int list -> int
- sumList [2,4,7];
 val it = 13 : int
```

Summary

We have looked at:

- **Higher-order functions**

Homework

- **Read the rest of Chapter 5 of Ullman.**
- **Consider the following definitions for a function to concatenate three lists**

```
fun concat3a (xs,ys,zs) = xs@ys@zs;
fun concat3b xs ys zs = xs@ys@zs;
fun concat3c xs (ys,zs) = xs@ys@zs;
```

  What are the types for each and what is the difference between them?

- **Using `simpleMap`, `filter` and `reduce` write a function `nnsqsum` which sums the squares of the non-negative numbers in a list. Now write a version which is recursive and does not use any higher-order predicates.**
- **The variance of a list of reals is a measure of the "spread" from the mean. More precisely, the variance of $[a_1, ..., a_n]$ is**

$$\frac{(\Sigma_{i=1}^{n} a_i^2)}{n} - (\frac{\Sigma_{i=1}^{n} a_i}{n})^2.$$

Write a function `variance` **which uses** `reduce` **and** `simpleMap` **to compute the variance of a list of reals.**
You can use `len` **to compute the length of a list. Now write version of** `variance` **which does not use higher order programming. Again you can use** `len`**.**

- **Using ML's higher order built-ins write a function which takes a list of strings and concatenates them all.**
- **Using ML's higher order built-ins write a function which converts a list of integers into the corresponding list of reals.**

Extended Homework

Boolean expressions have form

y **and ((not** x**) or (not (**y **and false)))**

Note that x **and** y **are variables: In general variables can be arbitrary strings of characters.**

Given a valuation, i.e. a function from variable names to Boolean values, we can evaluate a Boolean expression. For example, if we evaluate the above expression with the valuation

```
fn s => if s="x" then false else true
```

we obtain $true$**.**

A Boolean expression for which there is a valuation which makes it true is said to be satisfiable,
e.g. see www.csse.monash.edu.au/~lloyd/tildeAlgDS/Wff/Wff/

Your job is to write a data type and functions in Standard ML for representing and manipulating Boolean expressions:

- **A** `datatype` **definition for** `BoolExp`**, a data type representing Boolean expressions. This should have different data constructors for the different kinds of Boolean expressions:** $true$, $false$, not, and, or **and variables.**

- **Functions for constructing a Boolean expression:**

  ```
  const: bool -> BoolExp
  var: string -> BoolExp
  lnot: BoolExp -> BoolExp
  land: BoolExp * BoolExp -> BoolExp
  lor: BoolExp * BoolExp -> BoolExp
  ```

  The function `bool` **takes a Boolean constant** *true* **or** *false* **and returns the corresponding Boolean expression,** `var` **takes the name of a variable and returns a Boolean expression variable with that name, while** `lnot`, `land` **and** `lor` **combine Boolean expressions with the operators** *not*, *and* **and** *or* **respectively. For instance,**

  ```
  val b = land( (var "y"), lor( (var "x"),
            lnot( land((var "y"), (const false)))));
  ```

  binds `b` **to the Boolean expression corresponding to the example above.**

- **A function** `BoolExpToString: BoolExp -> string` **for returning a string describing a Boolean expression. For instance** `BoolExpToString b` **should return the string**

  ```
  "(y) and ((not(x)) or (not((y) and (false))))"
  ```

- **A function** `evalBoolExp: (string -> bool) -> BoolExp -> bool` **for evaluating a Boolean expression with a valuation. For instance,**

  ```
  evalBoolExp (fn s => if s="x" then false else true) b;
  ```

  should return `true`.

- **A function** `vars: BoolExp -> string list` **for returning a list of the variables occurring in a Boolean expression. Each variable should occur exactly once in the list. For instance,** `(vars b)` **should return** `["x","y"]` **or** `["y","x"]`**. [Hint: You may want to use a slightly modified** `merge` **function from the lecture].**

- **A function** `satisfiable: BoolExp -> bool` **which determines if a Boolean expression is satisfiable. For instance,** `(satisfiable b)` **should return** `true`**. [Hint: You may want to use** `vars` **and** `eval`**].**

SML Programming VII

In this lecture we will look at ML's module system. This has three building blocks:

- Structures are collections of types, datatypes, functions, exceptions, etc. which we wish to encapsulate, i.e. a module.
- Signatures describe the types and elements in a structure, ie an interface.
- Functors are operations that take structures etc as arguments and produce new structures (!), ie a super template.

Structures

The general form of a structure definition is:

```
structure <identifier> = struct
  <elements of the structure>
end
```

Structures may be nested.

Example Structure

```
structure StringBSTree = struct
 datatype 'label btree =
   Empty |
   Node of 'label btree * 'label * 'label btree;

 val create = Empty;

 fun lt (x:string,y) = x < y;

 fun inorderTraverse Empty = []
  |  inorderTraverse (Node(lt,l,rt)) =
     inorderTraverse(lt) @ (l::inorderTraverse(rt));

 fun lookup Empty x = false
  |  lookup (Node(left,l,right)) x =
      if lt(x,l) then lookup left x
      else if lt(l,x) then lookup right x
      else (* x=l *) true;

 fun insert Empty x = Node(Empty,x,Empty)
  |  insert (T as Node(left,l,right)) x =
      if lt(x,l) then Node((insert left x),l,right)
      else if lt(l,x) then Node(left,l,(insert right x))
      else (* x=l *) T; (* already in tree *)
end;
```

```
- use "bstree.ml";
 [opening bstree.ml]
 structure StringBSTree :
   sig
     datatype 'a btree = Empty |
                         Node of 'a btree * 'a * 'a btree
     val create : 'a btree
     val inorderTraverse : 'a btree -> 'a list
     val insert : string btree -> string -> string btree
     val lookup : string btree -> string -> bool
     val lt : string * string -> bool
   end
 val it = () : unit
```

Signatures

A signature is like a type for a structure, its general form is:

```
sig <specifications> end
```

Specifications can be

- `datatype` **followed by its definition.**
- `type` **followed by an identifier, possibly with type parameters. Eg** `type foo` **or** `type (a',b') tuple`
- `eqtype`**. As above except the identifier is an equality type.**
- `exception` **followed by an exception name.**
- `val` **followed by an identifier, colon and type expression.**

Note that `functions` **are listed as** `val`**s.**

```
structure StringBSTree :
  sig
    datatype 'a btree = Empty |
                        Node of 'a btree * 'a * 'a btree
    val create : 'a btree
    val inorderTraverse : 'a btree -> 'a list
    val insert : string btree -> string -> string btree
    val lookup : string btree -> string -> bool
    val lt : string * string -> bool
  end
```

Information Hiding with Signatures

We can bind an `identifier` **to a signature by**

```
signature <identifier> =
  sig <specifications> end
```

We can use the signature to restrict the types or hide objects in a structure.

```
signature STRINGDICT =
  sig
    type 'a btree
    val create : string btree
    val insert : string btree -> string -> string btree
    val lookup : string btree -> string -> bool
  end

- structure StringDict: STRINGDICT = StringBSTree;
 structure StringDict : STRINGDICT
```

This has hidden the data constructors `Empty` **and** `Node` **and the functions** `lt` **and** `inorderTraverse` **as well as making the type information for** `create` **more precise.**

Accessing Elements in a Structure

There are two main ways to access elements in a structure.

The first is to use explicit module qualification.

```
- val dict0 = StringDict.create;
 val dict0 = Empty : string StringBSTree.btree
- val dict1 = StringDict.insert dict0 "harald";
 val dict1 = Node (Empty,"harald",Empty) :
             string StringBSTree.btree
```

The second way is to open the structure:

```
- open StringDict;
 opening StringDict
   datatype 'a btree
     = Empty | Node of 'a StringBSTree.btree *
                      'a * 'a StringBSTree.btree
   val create : string btree
   val insert : string btree -> string -> string btree
   val lookup : string btree -> string -> bool

- val dict0 = create;
 val dict0 = Empty : string btree
- val dict1 = insert dict0 "harald";
 val dict1 = Node (Empty,"harald",Empty) : string btree
```

Notice that when you open a structure, you get a list of its contents.

Opening a structure shields from view all other named with the same identifier, but it is still possible to refer to them through module qualification.

Once you have opened a structure there is no easy way to close it again!

Risk: the structure might contain declarations you are not aware of and which shield yours. Safer alternative, redefine:

```
val create = StringDict.create;
val insert = StringDict.insert;
val lookup = StringDict.lookup;
```

The rest of elements in the structure still need to be referred through module qualification.

Functors

Structures are much like any other ML values –
You can write "meta-functions" which take structures and build new structures. These are called functors.

Their general form is:

```
 functor <identifier> = (<structure name>:<signature>) [:
          sig
             ...
          end ]
     = <structure definition>
```

Ideally, we should be able to define a general version of `BSTree` **structure which is parametric in the choice of** `lt` **and yet does not require the user to always pass** `lt` **as an argument.**

Really we want it to be parametric in the element type and in `lt` **–**
First we encapsulate this in a signature:

```
signature TOTALORDER = sig
  type element;
  val lt : element * element -> bool
end;
```

Which is used to define different structures (one per type):

```
structure String: TOTALORDER =
  struct
    type element = string;
    fun lt(x:string,y) = x < y;
  end;

structure Int: TOTALORDER =
  struct
    type element = int;
    fun lt(x:int,y) = x < y;
  end;
```

Now we define the functor which takes a `TOTALORDER` structure and produces a binary search tree based on the structure.

```
functor MakeBST(Lt: TOTALORDER):
 sig
    type 'label btree;
    val create : Lt.element btree;
    val inorderTraverse :  Lt.element btree ->
                           Lt.element list
    val insert : Lt.element btree -> Lt.element ->
                           Lt.element btree
    val lookup : Lt.element btree -> Lt.element ->
                           bool
  end
=
```

```
struct
  open Lt;

  datatype 'label btree =
    Empty |
    Node of 'label btree * 'label * 'label btree;
  val create = Empty;
  fun inorderTraverse Empty = []
   |  inorderTraverse (Node(lt,l,rt)) =
      inorderTraverse(lt) @ (l::inorderTraverse(rt));
  fun lookup Empty x = false
   |  lookup (Node(left,l,right)) x =
        if lt(x,l) then lookup left x
        else if lt(l,x) then lookup right x
        else (* x=l *) true;
  fun insert Empty x = Node(Empty,x,Empty)
   |  insert (T as Node(left,l,right)) x =
        if lt(x,l) then Node((insert left x),l,right)
        else if lt(l,x) then Node(left,l,(insert right x))
        else (* x=l *) T; (* already in tree *)
end;
```

functor MakeBST : <sig>

Finally, we can "make" our binary search tree for strings by applying functor `MakeBST` **to structure** `String` **to obtain** `StringBST`**.**

```
structure StringBST = MakeBST(String);
```

Or we can make one for integers

```
structure IntBST = MakeBST(Int);
```

The general form is:

```
structure <new structure name> =
                  <functor name>(<structure argument>)
```

with an optional colon and signature before the equal sign to describe the new structure. Note that the parenthesis for the structure argument are not optional.

Exercise: How do we make a binary search tree for customer records?

Information Hiding

The ML module system facilitates the design and reuse of software. It also allows us to hide implementation details.

Some ways to hide information:

- Define a signature which does not mention the hidden elements.
- Use an opaque signature to hide certain elements.
- Use an abstract type in place of a datatype to make its data constructors invisible.
- Use of local definitions within an abstract type or structure.

We have already seen the first method and will now look at the last two.

Read the ML documentation for more details on opaque signatures.

Abstract Types

As well as hiding functions and procedures it is also important to be able to hide types.

In ML this is achieved with the abstype. The difference from a datatype is that the representation is hidden.

Suppose we want to support arbitrary precision rational numbers. One implementation is

```
datatype rat = Rat of (int*int);

val zero = Rat(0,1);
val one = Rat(1,1);
fun  ++ ( Rat(m1,n1), Rat(m2,n2) ) =
    Rat(m1*n2 + m2*n1, n1*n2);
fun  == ( Rat(m1,n1), Rat(m2,n2) ) =
    m1*n2 = m2*n1;
```

However this type also includes pairs which do not correspond to rational numbers such as $(1, 0)$ and the data constructors are visible to everyone.

Abstract Types (Cont.)

A better way is:

```
exception DenominatorIsZero;
abstype rat = Rat of (int*int)
 with
  fun  // (m,n) =
      if n=0 then raise DenominatorIsZero
      else Rat(m,n);
  fun  ++ ( Rat(m1,n1), Rat(m2,n2) ) =
      Rat(m1*n2 + m2*n1, n1*n2);
  fun  == ( Rat(m1,n1), Rat(m2,n2) ) =
      m1*n2 = m2*n1;
  fun ratToReal (Rat(m,n)) = real(m)/real(n);
 end;
infix 4 ==;  infix 6 ++;  infix 7 //;
```

```
- use "rat.ml";
 [opening rat.ml]
 exception DenominatorIsZero
 type rat
 val // = fn : int * int -> rat
 val ++ = fn : rat * rat -> rat
 val == = fn : rat * rat -> bool
 val ratToReal = fn : rat -> real
 infix 4 ==
 infix 6 ++
 infix 7 //
 val it = () : unit
```

Values of type `rat` can only be accessed and displayed using the functions declared in the abstract type declaration.

```
- 3//4 ++ 5//6;
 val it = - : rat
- ratToReal it;
 val it = 1.58333333333 : real
```

Local Definitions

Local definitions are a little bit like `let` definitions. They allow nested definitions of functions inside abstract types or structures.

```
local
 <definitions1>
in
 <definitions2>
end
```

The values declared in `<definitions1>` are not visible outside of the declaration.

```
 local
    fun itfib (n,prev,curr): int =
      if n = 1 then curr
      else itfib (n-1,curr,prev+curr)
 in
    fun fib n = itfib (n,0,1)
 end;
```

In the above example, `fib`, but not `itfib` will be available after this.

Summary

We have looked at:

- Structures
- Signatures
- Functors
- Information hiding

Homework

- Read Chapter 8 of Ullman.
- Define a signature for a generic `Mapping` structure. The mapping is a list of pairs `('d,'r)` where `'d` is the domain type and 'r the range type. For any domain value there is at most one pair in the list with the value as the first component in the pair. The structure should provide 3 functions:
  - `create` to produce the empty list;
  - `lookup` to find the range value associated with a given domain value, it should throw the `NotFound` exception if there is no associated value;
  - `insert` which takes a domain and range element `d` and `r` and makes `r` the unique range value associated with `d`.
- Modify your answer to the extended homework so that it makes use of `local` function definitions and makes the Boolean expression data type an `abstract` data type.

SML Programming VIII

In this lecture we will look at practical programming in ML. We will look at:

- **Built-in Library structures.**
- **Input/Output.**
- **Destructive update of value bindings.**

Review the functional programming paradigm.

Library Structures

ML provides a large number of built-in library structures. They include:

- `Int`
- `Word`
- `Real`. **This has a substructure** `Math`:

```
Real.Math.sqrt(4.0);
val it = 2.0 : real
```

- `Char`
- `String`
- `Substring`
- `List`
- `Array`
- `Vector`
- `OS`
- `Time` **and** `Timer`
- `General`
- `TextIO`

Open them to find the functions they provide.

Advanced I/O

ML provides functions for opening and closing files and reading and writing to files that are similar to those provided in C.

They are in the structure `TextIO`. A partial list of the functions is:

```
type vector = string
type elem = char
type instream
type outstream
val input : instream -> vector
val input1 : instream -> elem option
val inputN : instream * int -> vector
val inputAll : instream -> vector
val canInput : instream * int -> int option
val lookahead : instream -> elem option
val closeIn : instream -> unit
val endOfStream : instream -> bool
val output : outstream * vector -> unit
val output1 : outstream * elem -> unit
val flushOut : outstream -> unit
val closeOut : outstream -> unit
val inputLine : instream -> string
val openIn : string -> instream
val openString : string -> instream
val openOut : string -> outstream
val stdIn : instream
val stdOut : outstream
val stdErr : outstream
val print : string -> unit
```

Destructive Update of Value Bindings

In ML the "assignments"

```
val x = 2;
val x = 1;
```

create bindings to two different variables, each called `x`.

However sometimes for efficiency we really want to change the value of a variable. I.e. perform destructive update.

ML allows this in two ways:

(1) The `Array` library module.

```
- open Array;
...
- val A = array(5,0.0);
 val A = prim? : real array
- update(A,0,1.0);
 val it = () : unit
- sub(A,0);
 val it = 1.0 : real
- sub(A,1);
 val it = 0.0 : real
```

Destructive Update of Value Bindings (Cont.)

(2) Use of references. These are like references in C++.

```
- val x = ref 0.0;
 val x = ref 0.0 : real ref
- !x;
 val it = 0.0 : real
- x := !x + 1.0;
 val it = () : unit
- !x;
 val it = 1.0 : real
```

You also have while loops to work with destructive update.

```
- val i = ref 0;
 val i = ref 0 : int ref
- while !i < 5 do (
=  print(Int.toString(!i));
=  print(" ");
=  i := !i + 1
= );
 0 1 2 3 4 val it = () : unit
```

However, use arrays and references sparingly—they are against the spirit of functional programming!

The FP Paradigm

Functional languages have the following characteristics:

- Everything is a function or a value (in fact a function is a value).
- First-class higher-order functions.
- The underlying conceptual model is the λ-calculus (lambda).
- They are high-level languages with implicit memory management.

ML has the following characteristics:

- Strict functions (call by value).
- Pattern matching.
- Polymorphic static typing.
- A sophisticated module system.
- Exception handling.
- Provides destructive update.

Strictness

An ML function is usually strict, that is, if its argument is undefined (i.e. does not terminate), so is the result.

While strictness is natural, the eagerness to evaluate can be troublesome, e.g.:

```
fun f x = f x : int;
 val f = fn : 'a -> int

fun k x y = x;
 val k = fn : 'a -> 'b -> 'a
```

Evaluation of `k 7 (f 5)` **"should" return** 7 **but in fact it will not terminate.**

Pattern matching and its variants `if-then-else` **and** `case` **are non-strict though.**

Laziness

Many modern functional languages (starting with Miranda) use lazy evaluation instead.

In lazy functional languages
– arguments to functions are only evaluated when needed, and
– arguments to data constructors are only evaluated when needed.

As an example, in Haskell and Lazy ML (LML) we can define a function which computes the infinite list `[n,n+1,n+2,...]`**, however only as much of the list as is needed will be computed.**

```
fun from n = n :: from (n+1);
```

We can use this to lazily compute the first prime not less than `m`**:**

```
val firstprime = hd o (filter prime);

firstprime (from m)
```

Laziness is very powerful but sometimes does not fit well with side-effects like IO.

Simulating Laziness

Lazy (actually "normal order") evaluation can be simulated in ML by using higher-order functions.

In our example, the trick is to use a function to model a lazy list. When the function is called it returns the next element in the list and a function to compute the rest of the list.

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

fun head (Cons(x,_)) = x;
fun tail (Cons(_,xs)) = xs();

 val head = fn : 'a lazyList -> 'a
 val tail = fn : 'a lazyList -> 'a lazyList

fun from n =
   let fun next(x)() = Cons(x,next(x+1)) in
     next(n)()
   end;

 val from = fn : int -> int lazyList
```

Simulating Laziness (Cont.)

```
fun firstprime ns =
   let val n = (head ns);
   in
     if (prime n) then n
     else (firstprime (tail ns))
   end;
 val firstprime = fn : int lazyList -> int

- firstprime (from 2);
 val it = 2 : int
- firstprime (from 6);
 val it = 7 : int
- firstprime (from 8);
 val it = 11 : int
```

A more efficient lazy strategy, *call be need*, **can be implemented with the aid of a** `ref` **type,**
see **www.csse.monash.edu.au/ lloyd/tildeFP/SML/1997/Lazy/**

Summary

We have looked at:

- Built-in libraries
- Input/output
- Destructive update of value bindings
- The FP paradigm
- Laziness

Homework

- Read Chapter 4 and Sections 7.2, 7.3 and 9.4 of Ullman.
- Can you use `while` loops without using destructive update? For instance consider

```
- val i = 0;
-  while i < 5 do (
=   print(Int.toString(i));
=   print(" ");
=   val i = i + 1;
= );
```

- Name an application for which you would prefer to use ML to C.
- Do the assignment!