

## ML Programming I

In this and the next 8 lectures we will look at ML. ML is a good example of a **functional programming language**.

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

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

## Expressions

Basically ML is 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, 199
```

```
- 1+2*3;  
val it = 7 : int  
- 81 mod 10;  
val it = 1 : int  
- ~(7-6);  
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 have 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 down) and
  - mod (integer division remainder).
- highest precedence: ~

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.

```
- 1 < 2;  
val it = true : bool  
- 1 = 2;  
val it = false : bool  
- 1+1 < 3 and also 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 precedes 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 don't have to evaluate their second argument.

They have the usual precedences.

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

## Value declarations

Most sorts of objects can be **named**:

```
- 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;
```

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 don't need to give the type of the declaration, 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;
```

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
```

## If-then-else

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

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

For instance,

```
(* computes the absolute value *)
fun abs r =
  if r > 0 then r
  else ~r;
```

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.)

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 ints to reals.

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

```
- fun square (x::real) = x**x;  
- fun square x = (x::real)**x;  
- fun square x = x**(x::real);  
- fun square x = x**x :real;
```

The brackets are necessary for precedence.

## Values versus variables

Value names are **not** quite the same as global variables.  
– they cannot be updated to a new value.

Values are names for an expression, 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 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  
1. 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 `\d1d2d3` where `d1d2d3` is the ASCII code for the character.

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

**Concatenation** is the most important predefined operator. This is indicated by the “hat” 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 bizarre:  
# 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
```

## Coercion Functions

Built-in arithmetic coercion functions are:

- `real: int -> real`
- `floor: real -> int`  
Greatest integer no larger than argument
- `ceil: real -> int`  
Greatest integer no smaller than argument
- `round: real -> int`  
Closest integer, rounding up
- `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
```

23

## 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

- `[]` (the empty list) and
  - `:` (the list constructor “cons”).
- We can also use `nil` instead of `[]`.

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

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

24

## Recursive functions

As we have seen ML provides recursion, just like C.

A function to compute the factorial function:

```
- fun fac x = if x = 0 then 1 else x*fac(x-1);
val fac = fn : int -> int
- fac 4;
val it = 24 : int
```

## Programming with Lists

Recursion is often used for programming with lists.

The key to programming with lists is to reason recursively about the list.

- Either the list is empty and the operation of interest should be straightforward, or
- it is non-empty and can be broken into a first element and a remaining shorter list which is dealt with recursively.

We can write a function to sum the integers in a list of integers:

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

What if the list `l` is `x::xs`?

- Recursively sum the elements in the list `xs` and add `x`.

```
- 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**. (A bit like Prolog).

## Programming with Lists (Cont.)

We can 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.

```
- fun last [x] = x
= | last (x1::x2::xs) = last (x2::xs);
```

```
stdIn:1.1-22.37 Warning: match nonexhaustive
x :: nil => ...
x1 :: x2 :: xs => ...
```

```
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 because
value restriction are instantiated to dummy types (X1,X2
```

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

27

## 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 infix operator `@` appends lists:

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

The 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.

28

## 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.

## Mutual Recursion

## Summary

One complication is **mutual recursion**...

```
(* checks if the list is even *)
fun even [] = true
  | even (x::xs) = odd xs;
(* checks if the list is odd *)
fun odd [] = false
  | odd (x::xs) = even xs;
- use "evenoddlst.ml";
[opening evenoddlst.ml]
evenoddlst.ml:3:20-3:23 Error: unbound variable or
constructor: odd
```

**Need to use an and**

```
fun even [] = true
  | even (x::xs) = odd xs
and
  odd [] = false
  | odd (x::xs) = even xs;
- use "evenoddlst.ml";
[opening evenoddlst.ml]
val even = fn : 'a list -> bool
val odd = fn : 'a list -> bool
val it = () : unit
```

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 length to return the length of a list.
- 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.

## ML Programming III

## Tuples

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

ML also 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 but with no name.

A 2-tuple is called a **pair**.

In this lecture we will look at programming with:

- Tuples
- Pattern-matching

## Bizarre Tuples

A 1-tuple (`pi`) is not distinguished from `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 there is a problem:**

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

**Instead**

```
- 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?**

## 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 imagine that we want a function `addvec` to add two vectors. This needs to take two vectors which are themselves 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.

## Another Example (Cont.)

**Exercise:** Write a function `scale` to multiply a vector by a real.

**Exercise:** What is the type(s) of `+`.

## 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 Declarations

Sometimes we need to create some temporary values  
– i.e. **local variables**.

Let allows you to do this. The general form is:

```
let  
  val v1 = exp1;  
  val v2 = exp2;  
  ...  
  val vn = expn;  
in  
  expression  
end
```

**The definition of v1 is visible in exp2 etc. All vis are visible in expression.**

```
fun circle_area r =  
  let  
    val pi = 3.14159;  
  in  
    pi * r * r  
  end;  
  
val circle_area = fn : real -> real
```

## Let Declarations (Cont.)

## Split (cont.)

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

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

```
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.”

## 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 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)?`

## 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);
```

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);
```

## Naming Patterns – As

Sometimes you wish to match the argument to a pattern but also have a name for the argument. `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
```

## 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.
- Using `split` and `merge` write `mergeSort`.
- Write a function to implement quicksort.

Hint: simply use the first element of the list as the partition element.

## SML Programming IV

### 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>
```

We could write `len` as

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

The `rec` means the definition is recursive, i.e. the definition of `len` is in terms of `len`.

Actually

```
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**.

In this lecture we will look at:

- Matches
- Simple Output
- Exceptions
- Polymorphism
- Type Inference

## 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 just short for**

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

## Print

Because ML 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 void value (). The fact that 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
```

## 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.

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

```
- 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
```

## Programmer Defined Exceptions

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 n =
  if n < 1 then raise BadFacArg
  else if n=1 then 1
  else n * fac(n-1);

- fac 3;
val it = 6 : int
- fac ~1;
uncaught exception BadFacArg
raised at: stdIn:24:23-24:32
```

## Programmer Defined Exceptions (Cont)

```
exception BadFacArg of int;

fun fac n =
  if n < 1 then raise BadFacArg n
  else if n=1 then 1
  else n * fac(n-1);

fun fac n = fac1 n
  handle
    BadFacArg 0 => 1
  | BadFacArg n => (
    print("invalid argument to fac: ");
    print(Int.toString(n));
    print("\n");
    0
  );

- 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.

(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
```

## 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>)
```

Note that `=` 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.

## Equality Types (Cont.)

What is the difference between these? Why?

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

val len = fn : 'a list -> int
```

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

```
val len = fn : 'a list -> int
```

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

```
val len = fn : 'a list -> int
```

## Type Inference

How does ML perform type inference?

Consider the polymorphic function

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

1. Now the type of `len` must be an instance of  $'a \rightarrow 'b$ .
2. The body must be of type  $'c \rightarrow int$  where  $xs$  has type  $'c$ .
3. The argument must be of type  $'d list$ .
4. Thus `len` has type  $'d list \rightarrow int$ .

Types deduced by ML are **principal**, that is, as general as possible.

Interestingly, the **unification** algorithm is used for type inference. This was pioneered by Robin Milner in ML.

## More Examples

Work out the type of the following functions

```
- fun fs (x,y) = x;
- fun fst_fst z = fst (fst z);
```

```
- fun mem(x, []) = false
  | mem(x,y::ys) = (x = y) orelse mem(x,ys);
```

## Summary

We have looked at:

- Matches
- Print and Statements
- Exceptions
- Polymorphism
- Type Inference

## Homework

- Read Sections 4.1, 5.1, 5.2, 5.3 of Ullman.
- Add appropriate exception handling to `max`.
- 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:

- Higher-order functions

## 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
```

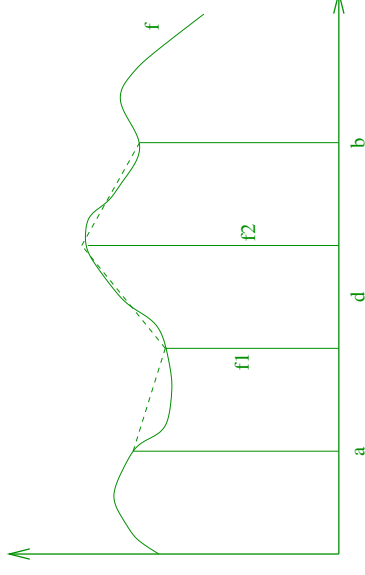
## Higher Order Functions

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.

One way to compute the integral of a function is to use the **trapezoidal rule**:



The area of the ***i*th 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;
end;

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

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

```
- use "trap.ml";
[opening trap.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
```

69

## 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.

70

## Another Example (Cont.)

However we can write a function which takes the 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 takes a function  $F$  and a list  $[a_1, \dots, a_n]$  and returns the list  $[F(a_1), \dots, F(a_n)]$ .

(Similar to `mapcar`).

```
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  $[a_1, \dots, a_n]$  and returns

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

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?

## Filter

The **filter** function takes a Boolean function  $F$  and a list  $[a_1, \dots, a_n]$  and returns the sublist whose elements satisfy  $F$ .

```
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`.**

## Example

The **variance** of a list of reals is a measure of the “spread” from the mean.

**More precisely, the variance of  $[a_1, \dots, a_n]$  is**

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

**Exercise:** 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.

**Exercise:** Write another version of `variance` which does not use higher order programming. You can also use `len` to compute the length of a list.

It should be clear that higher-order programming with lists allows you to write concise, powerful programs.

## Returning a Function

Since functions are just like any other expression 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.

Thus `add` is more flexible.

`add` is the **curried version of `add'`**.  
(Named after the logician **Haskell Curry**.)

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;
```

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.

## 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;
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.
- Using `simpleMap`, `filter` and `reduce` write a function `sumSquares` 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.
- 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.

## SML Programming VI

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 then rules for defining more complex types from these.

**Basic types:** `int`, `real`, `char`, `bool`, `unit`, `exn`, `string`, `instream`, `outstream`.

**Product type:** `T1 * T2`, more generally `T1 * T2 * ... * Tn`.

**Function type:** `T1 -> T2`.

**Type constructors:** We have met `list`, ie `T1 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.

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

Much like a typedef in C.

For instance 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.

## Parameterized Type Definitions

More generally we can pass type variables as parameters to the definition.

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

Imagine an **association list** which has a domain type and a range type. 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;
```

## Datatypes

ML has a powerful mechanism for defining new types called **datatypes**.

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

## Datatypes (Cont.)

Here `fruit` is the new data type and `Apple`, `Pear`, `Grape` are data constructors.

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

Apple
-------

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

## Complex Datatypes

More generally, in datatype definitions:

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

The general form is

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

## List Datatype

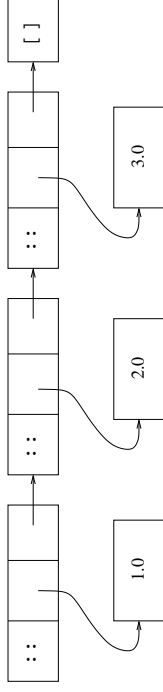
For example the `List` data type is conceptually defined by

```
datatype 'a list =  
  [] |  
  :: of 'a * 'a list;
```

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

```
1.0 :: 2.0 :: 3.0 :: []
```

is represented by



**Do not to confuse data constructors with functions:**  
– data constructors **wrap** data  
– functions **compute** things from the data.

## 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 “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
```

## 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 x))
  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

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

## Another Example

**Exercise:** Write a function to traverse the elements in a labelled binary tree in-order and return the result in a list.

## Records

**A record is like a tuple, but its components (fields) are named.**

### 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 tuples are 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
```

The `...` is part of the ML syntax—it means “the rest of the record.”

## 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
```

## Summary

We have looked at:

- Type definitions
- Datatype definitions
- Records

## Homework

- Read Chapter 6 and Section 7.1 of Ullman.
- Give a type definition for a customer record containing an integer ID and a name (as a string).
- Now give a function `lt` to compare two customer records based on the ID.
- Use the binary search tree to implement a dictionary of customer records.

## Extended Homework

A polynomial has form

$$2.0x^2y^3x^1 + 3.0 + (-2.0)y^3x^3 + 1.0x + 5.0$$

A polynomial is composed of terms, e.g.  $2.0x^2y^3x^1$ ,  $3.0$ ,  $-2.0y^3x^3$ ,  $1.0x$  and  $5.0$ . Terms are composed of a coefficient and factors. E.g.  $-2.0y^3x^3$  has coefficient  $-2.0$  and factors  $y^3$  and  $x^3$ . Note that  $x$  and  $y$  are variables: In general variables can be arbitrary strings of characters.

Given a valuation, i.e. a function from variables to their values, we can evaluate a polynomial. For example, if we evaluate the above polynomial with the valuation

```
fn v =>
  if v="x" then 2.0 else if v="y" then 1.0 else 0.0
```

we obtain 10.0.

Give type definitions for factor, term, polynomial and a valuation which respectively represent a factor, term, polynomial, and valuation.

Define functions:

```
printpoly: polynomial -> unit
add: polynomial*polynomial -> polynomial
scale: real -> polynomial -> polynomial
eval: valuation -> polynomial -> real
```

where `printpoly P` prints polynomial `P`, `add(P1,P2)` is the polynomial obtained by adding the terms in `P1` and `P2`, `scale R P` is the polynomial obtained by multiplying the coefficient of all terms in `P` by `R`, and `eval V P` is the result of evaluating `P` with `V`.

## SML Programming VII

## Structures

A structure definition has form

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

Structures may be **nested**.

```
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;
```

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, ie 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**.

## Signatures

```
- 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
```

**A signature is like a type for a structure, it has form**

```
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 vals.**

```
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.

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

## 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**.

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

`lt`—

First we encapsulate these in a structure.

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

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;
```

111

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>
```

112

Finally we can “make” our binary search tree for strings.

```
structure StringBST = MakeBST String;
```

Or we can make one for integers

```
structure IntBST = MakeBST Int;
```

## 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 = 1.5833333333333 : real
```

## Local Definitions

Local definitions are a little bit like `let` definitions. They allow nested definitions of functions.

```
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  $d$ .

## 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
```

123

## 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!

124

## The FP Paradigm

Functional languages have the following characteristics:

- Everything is a function or a value.
- First-class higher-order functions.
- The underlying conceptual model is the lambda calculus.
- 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

A 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:

```
fun f x = f x : int;  
val f = fn : 'a -> int  
  
fun zero x = 0;  
val zero = fn : 'a -> int
```

Evaluation of zero (f 5) will not terminate.

Pattern matching and its variants if-then-else and case is 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 a **lazy ML** we can define a function which computes the infinite list  $[n, n+1, n+2, \dots]$ , 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  $m$ :

```
fun firstprime (n::ns) =  
  if prime n then n  
  else firstprime ns;  
- firstprime (from m)
```

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

127

## 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!

128