Data Types in ML

int: 3, 4, 5, ~3, ~4, etc -- type int

real: 3.14, 3.56, 0.03 -- type real

bool: true, false -- type bool

string: "foo", "boo" -- type string

list: a sequence of homogeneous objects, [2,3,4] -- type int list, ["foo", "bar"] - type string list

tuple: a sequence of heterogeneous objects (abc ,3,true) -- type string * int * bool

function: fn (x, y) => x -- type int * int -> int, (or type int * real -> int, or type τ1 * τ2 -> τ1

CS603 Programming Language Organization

Lecture 18
Spring 2003
Department of Computer Science
University of Alabama

Syntax of ML

Variable definition:

```
val <var> = <expr>;
```

arithmetic operations are infix --

```
4-x
9*(y+4)
```

if expression:

```
if <expr> then <expr> else
<expr>
```

let expression:

```
let val x=3 val y=(3*5) in x*y;
```

Objectives

Provide you with some familiarity with ML. In particular, you should be able to:

- Know how to use the immediate mode to perform simple calculations
- Know how to load your own source files
- Know how to use pattern matching in your functions
- Know how to debug type errors

Starting SML Immediate Mode

Immediate mode statements must be terminated by a semicolon (;)

The result is placed in the special variable it

Notice the type inference!

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- 34 + 8;
> val it = 42 : int
- it - 20;
> val it = 22 : int
```

ML

Develop by Robert Milner circa 1975 for theorem proving.

Began as a meta language (Logic of Computable Functions, LCF), and evolved into a full programming language.

ML is a strongly typed functional language.

Type checking is done by inferring types.

Statically Type-checked

ML is polymorphic.

List Manipulators

null -- test whether a list is empty

hd -- return the first element of a list

tl -- return the list consisting of everything but the first element of its argument

:: -- x::xs is a list like xs but with x added at the beginning

@ -- infix append - join two lists together. (E.g. [x]@xs is the same thing as x::xs.)

Hello, World

Here is the Standard First Program

The unit type is like void in C/C++; it represents commands.

```
- print "Hello, World!\n\n";
Hello, World!

> val it = () : unit
-
```

More about List Types

Unlike tuples, all the elements of a list must have the same type.

```
- [3, 8.4];
! Toplevel input:
! [3, 8.4];
!     ^^^
! Type clash: expression of type
!   real
! cannot have type
!   int
-
```

Some Simple Types

```
- 20.3;
> val it = 20.3 : real
- ~27
> val it = ~27 : int
- "Hi";
> val it = "Hi" : string
- (3, "x");
> val it = (3,"x") : int * string
-
```

Variables

```
- val x = 20;
> val x = 20 : int
- x + 10;
> val it = 30 : int
- val x = 30;
> val x = 30 : int
- val y = 40 and s = "hi";
> val y = 40 : int
  val s = "hi" : string
```

The List Type

```
> val it = [2.5,3.9,4.2] : real list
- hd it;
> val it = 2.5 : real
- 3 :: [2, 4];
> val it = [3,2,4] : int list
- tl it;
> val it = [2,4] : int list
- nil;
> val it = [] : 'a list
```

The `'a list` means “a list of an arbitrary type a,” or just “a list of a's”.

Recursive Functions

```
- fun f n =
      if (n = 0)
         then 1
      else 1 + (f (n - 1));
> val f = fn : int -> int
- f 10;
> val it = 11 : int
-
```

Functions

fun name parameters = body ;

```
- fun f x = x + 4;
> val f = fn : int -> int
- fun g x y = [x,y];
> val g = fn : 'a -> 'a -> 'a list
- f 30;  (* Functions are applied
by juxtaposition! *)
> val it = 34 : int
- g 20 5;
> val it = [20,5] : int list
```

Function Definition

Syntax:

```
val [rec] <name> = fn <tuple> => <expr>
```

```
fun <name> <arg-pat> = <expr1> | …
```

Examples

```
val identity = fn x => x
val rec fact = fn x => if x <= 1
  then 1
else x * (fact (x - 1));
fun fact x = if x <= 1 then 1
  else x * (fact (x - 1));
```

⇐ equivalent ⇐

Errors

```
- fun f x = x + 4;
> val f = fn : int -> int
- f 6.3;
! Toplevel input:
! f 6.3;
!   ^^^
! Type clash: expression of type
!   real
! cannot have type
!   int
-
```

Function Application

Syntax:

`f (a, b, c)` -- equivalent to same call in C.

`f a b c` -- equivalent to `(((f a) b) c)` (called curried form)

Function application associates to the left.

Use parentheses if you want to change the association.

More Errors

```
- fun g x y = [x,y];
> val g = fn : 'a -> 'a -> 'a list
- g "hi" 4.3;
! Toplevel input:
! g "hi" 4.3;
!       ^^^
! Type clash: expression of type
!   real
! cannot have type
!   string
```

-Always always ask ``what are the types?''!

That habit will save you a lot of time!

Mystery 1

```
- fun f 0 = 0
    | f 1 = 1
    | f n = f (n-1) + f (n-2);
> val f = fn : int -> int
- (f 3, f 5, f 8);
> val it = (2,5,21) : int * int *
int
```

Pattern Matching

```
- fun f 0 = 1
    | f n = 1 + f (n-1);
> val f = fn : int -> int
- f 10;
> val it = 11 : int
-
```

Notice the similarity to a mathematical definition.

Mystery 2

```
- fun f [] = 0
    | f (x::xs) = x + f xs;
> val f = fn : int list -> int
- f [10,10,10,12];
> val it = 42 : int
-
```

Pattern Matching II

Function f takes a tuple as its argument.

Function g takes two arguments. Don't confuse them!

```
- fun f (a,b) = a + b;
> val f = fn : int * int -> int
- f (10,30);
> val it = 40 : int
- fun g a b = a + b;
> val g = fn : int -> int -> int
- g 10 20;
> val it = 30 : int
-
```

Mystery 3

```
- fun f [] = []
    | f (x::xs) =
        if (x < 5)
           then f xs
           else x :: f xs;
> val f = fn : int list -> int list
- f [1,5,10,3,4,48];
> val it = [5,10,48] : int list
-
```

Pattern Matching III

```
- fun f [] = 0
    | f (x::xs) = 1 + (f xs);
> val f = fn : 'a list -> int
- f [10,20,30,40];
> val it = 4 : int
-
```

Can you explain what this function is doing? What is the type of x? What is the type of xs?

Recursion in ML

In functional languages, repetition is accomplished by recursion.

```
fun gcd m n =
    if m = n then m
    else if m < n then gcd m (n % m)
    else gcd n m;
```

Mystery 4

```
- fun f 0 a = []
    | f n a = a :: (f (n-1) (a-1));
> val f = fn : int -> int -> int list
- f 6;
> val it = fn : int -> int list
- it 10;
> val it = [10,9,8,7,6,5] : int
list
-
```

Lists

Recursive data structure:

A ♦ list (a list whose elements are of type ♦) is either empty or a ♦ joined to a ♦ list.

If L = x joined to M, then x is the head of L and M is the tail of L

Bonus!

Higher Order Functions!

```
- fun twice f x = f (f x);
> val twice = fn : ('a -> 'a) -> 'a -> 'a
- fun inc n = n + 1;
> val inc = fn : int -> int
- twice inc 20;
> val it = 22 : int
-
```

Lists in ML

Constants:

`[1,2,3]: int list`

`[true, false]: bool list`

Operations:

`hd [1,2,3] = 1`

`tl [1,2,3] = [2,3]`

`null [] = true`

`null [1,2] = false`

`1::[2,3] = [1,2,3]`

One more thing....

```
- use "myprogram.sml";
```

Immediate mode is fun, but it can be counter-productive....

Example: merge sort

```
fun msort L =
let val halves = split L
in merge (msort (hd halves))
         (msort (hdtl halves))
end

fun split [] = [[], []]
  | split [a] = [[a], []]
  | split (a::b::t) =
       let val splittltl = split t
       in [a::(hd splittltl),
           b::(hdtl splittltl)]
       end;
```

Types of ML lists and operations

Lists can contain other lists, but are homogeneous.

```
[[1,2], [], [4,5,2]]: (int list) list
```

But `[1, [2,3]]` is not legal.

List operations have polymorphic types:

hd: ☋ list -> ☋

tl: ☋ list -> ☋ list

:: : ☋ * ☋ list -> ☋ list

null: ☋ list -> bool

Algebraic Data-types

```
datatype 'a tree = Empty
                 | Node of 'a tree * 'a * 'a tree

fun height Empty = 0
  | height (Node (lft, _, rht)) = 1 + max (height lft, height rht)
```

Simple examples

```
fun tltl L = (tl (tl L));
fun hdtl L = hd (tl L);
fun incrhd L = (1+(hd L))::(tl L);
fun swaphd L =
    (hdtl L) :: (hd L) :: (tltl L);
fun length L = if null L then 0
         else 1+(length (tl L));
fun append L M = if null L then M
   else (hd L)::(append (tl L) M);
fun concat L M = if null L then L
     else (append (hd L) M);
```

Resources

"Programming in Standard ML"

http://www-2.cs.cmu.edu/~rwh/smlbook/offline.pdf

Source code from above book

http://www-2.cs.cmu.edu/~rwh/smlbook/examples/

Moscow ML

http://www.dina.dk/~setsoft/mosml.html

Pattern-matching in function definitions

```
fun f [] = ...
 |  f (x::xs) = ...x...(f xs)...
fun f [] M = ...M...
 |  f L [] = ...L...
 |  f (x::xs) (y::ys) =
          ...x...y...(f xs ys)...
fun f [] = ...
 |  f [x] = ...x...
 |  f (x::y::xs) =
          ...x...y...(f (y::xs))...
```