ML Lecture

Oct. 26 2004

Acknowledgments:
1. Standard ML of New Jersey website: www.smlnj.org
2. Programming in Standard ML. by Robert Harper.
3. Concept in Programming Lang. by John C. Mitchell

Function Declarations (recap)

Sheila will talk more about functions on Thursday

Recall: A function maps a type to another one: accepts only a single argument.

multiple arguments? use an n-tuple instead!

```
fun fname (pattern1) = exp1   (* () optional *)
  | fname (pattern2) = exp2
  ...
  | fname (patternN) = expN;
```

Lazy evaluation: when called, the expi associated with the first matched pattern will be chosen.

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

-sum (2,3);
val it = 5 : int
```

Function examples...

```
-fun len (nil) = 0  (* nil or [], can drop ()
   | len (h::rest) =    (*need () in in arg.*)
                   1+len(rest);
val len= fn: 'a list -> int

-len ([5]);
val it = 1: int
-len ["Alice", "John"];
val it = 2: int
```

Note1: no variable can occur twice in each pattern!

```
fun eq(x,x)=true
  | eq(x,y)=false;
Error: duplicate variable in pattern(s)
```

Note2: If the patterns don't exhaust all the possible values, we get a warning: "Warning: match nonexhaustive"

Type Synonym

We can give existing types new names.

Syntax: `type` *tycon* = *ty*

tycon becomes an alias (synonym) for the <u>existing</u> type *ty*.

```
1   -type float = real;
    type float = real

2   -type count = int  and  average=real;
    type count = int
3   type average = real
    ??
4   -val f: float = 2.3;
    val f=2.3: float
5   -val i:count = 3;
   val i = 3: count
```

Type Synonym (continue...)

But notice `float`, `real`, and `average` are all of the same base type, i.e. `real`!

```
 6  -val a:average = f;
    val a = 2.3: average

 7  -val res = a+f;
    val res = 4.6: average
```

Type synonyms make program more readable.

```
 8 -type car= {make:string, built:int};
    ??

 9 -val c1: car = {make="Toyota", built=2001};
   ??

10 -fun nextModel ({make=n,built=y}:car)= y+1;
   val newxtModel = fn : car -> int

11 - nextModel c1;
   ??
```

User defined datatypes

General Syntax:

```
datatype tycon = cons1 of ty1
               |cons2 of ty2
               ...
               |consn of tyn
```

- Defines a **new** type called `tycon`.

- `tyi`'s are previously defined types..

- `consi`'s are *constructors*. They are used to create a value of `tycon` type

Note: "`of tyi`" is omitted if a constructor does not need any argument (such constructors are called *constants*).

Enumerated Types

When all constructors are constants (no argument).

Example:

```
1  -datatype color = Red|Blue| Green;
   datatype color = Blue | Green | Red

2  -val c=Red;  (*calling constructor Red*)
   val c=Red: color;

3  -fun colorStr(Red)= "Red"
4     | colorStr(Blue)= "Blue"
5     | colorStr(Green)= "Green";
   val colorStr = fn : ??

6  -colorStr(c);
   val it= ??
```

Variant Types

Can create union of different types:

```
1  -datatype number = r of real
2                    | i of int;
   datatype number= i of int | r of real

3  -val n1 = i 2;
   val n1 = i 2 : number

4  -val n2 = r 3.0;
   ??

5  -val lst=[r 2.2, i 3, i 4, r 0.1];
   val lst = [r 2.2, i 3, i 4,r 0.1]: ??

6  -fun sumInts ([])=0
7      |sumInts (i x::rest)= x+ sumInts rest
8      |sumInts (r x::rest)= sumInts rest;
   val sumInts = fn : ??

9  -sumInts lst;
   ??
```

Recursive Types

A datatype can be recursive: e.g. **linked list**.

```
1  -datatype llist= Nil | Node of int*llist;
   datatype llist = Nil | Node of int*llist

2  -val x = Nil;
   val x=Nil: ??

3  -val y = Node (5, Nil);
   ??
4  -val z = Node(3, Node(2,Node(1,Nil)));
   ??
 (*computing the length of a linked list*)
5  -fun len Nil =0
6       |len(Node(_,rest))= 1 + len rest;
  val len = fn : ??

7  -len z;
   ??
```

Recursive Types (continue...)

Example: a *polymorphic* linked list

```
1 -datatype 'a llist= Nil|Node of 'a*('a llist);

2  -val x = Nil;
   val x=Nil: ??

3  -val y = Node (5, Nil);
   val y = Node (5,Nil) : ??

4  -val z = Node("Test", Node("B",Nil));
     ???
```

A binary tree where only leaves have data:

```
6  -datatype 'a tree= L of 'a
                    | N of ('a tree)*('a tree);
7  -val mytree= N(L(1),N(L(2),L(3)));

8  -fun max (x,y)= if x>y then x else y;
9  -fun depth(L _)=0
10     |depth(N(ltree,rtree))=
            1+max (depth ltree, depth rtree);
```

Mutual Recursive Types

Want to represent a tree with arbitrary #of branches.

See the diagram first ...

Defining mutually recursive datatypes (using **and**).

```
1 -datatype tree = Empty| Node of int*forest
2      and forest= Nil  | Cons of  tree*forest
  datatype tree = Empty | Node of int * forest
  datatype forest = Cons of tree * forest | Nil

3 -val t1=Node(2,Nil);
  ??
4 -val t2=Node(3,Nil);
  ??
5 -val t3=Node(7,Cons(t1,Cons(t2,Nil)));
  ??
6 -val t4=Node(5,Nil);
  ??
7 -val t5=Node(1,Nil);
  ??
8 -val t6=Node(2,Cons(t5,Cons(t4,Cons(t3,Nil))));
  ??
```

Mutual Recursive Types: function example...

We want to count how many nodes are in a tree.

solution: 1+ #of nodes in its subtrees (i.e. forest)

```
-fun numnodeT (Empty)=0
    | numnodeT (Node(data,f))= 1+ numnodeF(f)
  and
        numnodeF(Nil) = 0
       |numnodeF(Cons(t,f))= ???


  val numnodeT = fn : tree -> int
  val numnodeF = fn : forest -> int

(* Note that numnodeT and numnodeF are
  mutually recursive.*)

  -numnodeT(t6)
  ??
```

SML Exceptions

ML's *exceptions* (similar to the ones in C++, Java) provide a uniform way to handle errors, and eliminate the need for ad hoc, special exceptional return values from functions.

When encountering an unexpected error: *raise* an exception instead of writing ugly codes! The caller will *catch* the exception and will take care of it.

SML has primitive exceptions:

```
3 div 0  (* this will raise Div   *)
hd nil   (* this will raise Match  *)
2*maxint (* this will raise Overflow *)
```

We can define our own exceptions:

```
exception factNeg;
fun robust_fact n=
    if n<0 then
        raise factNeg  (*note here!*)
    else fact n;
```

SML Exceptions (continue...)

Exceptions can carry data:

```
exception factNeg of int;
fun robust_fact n=
     if n<0 then
         raise (factNeg n) (*note here!*)
     else fact n;
```

Handling exceptions:

```
Syntax:   exp handle match
Note: return type of match must be same as exp.
```

```
-fun print_fact n=
   print (Int.toString(robust_fact n))
 handle factNeg(x) => print "n must be>-1!";
val print_fact = fn : int -> unit
```

```
(*return type of fun&exception are both unit*)
```

```
-print_fact ~3;
```

Defining infix operators

Syntax: `infix prec opr` where $0 \le$ `prec` ≤ 9.

Example: `infix 5 ++`

Default: it is left associative.

To use the infix `++` as a normal prefix function use **op++**. You need this when specifying it as a function or passing it to another function as an argument.

Example:

```
1  -fun op++ (x,y) = x+y+1;
   val ++ = fn : int * int -> int

2  -val a= 2 ++ 10;
   val a=??  : ??
```

To make the operator back to prefix: `nonfix ++`

Infix operators: Examples

```
1  -infix 5 --;

2  -fun op-- ((x1,y1),(x2,y2))= (x1-x2,y1-y2);
   val -- = fn :(int*int)*(int*int) -> int*int

3  -val x= (1,2)--(3,7)--(12,5);
   val x = ?? : ??

4  -infix 6 **;

5  -fun op** ((x1,y1),(x2,y2))= (x1*x2,y1*y2);
   val ** = fn: ??

6  -val y = (40,100)--(2,5)**(5,10);
   val y =  ??   :  ??

(*what if we make it a prefix operator!*)
7  -nonfix **;   (*after this ** is prefix*)
8  -(1,1) ** (2,2);
   ????
9  - **( (1,1),(2,2));
   ???
```