Tutorial 7

Week of October 31, 2005 (Boo!)

1 Anonymous functions

The syntax for anonymous functions is:

```
fn <argument> => <body>;
```

Examples:

```
- fn x => x;
val it = fn : 'a -> 'a

- (fn x => x) 5;
val it = 5 : int

- (fn x => x) "foo";
val it = "foo" : string

- fn (x, y) => [2*y, 2*x];
val it = fn : int * int -> int list

- fn (x,y) => x^y^"!";
val it = fn : string * string -> string
```

More complicated example:

Write a function `double` that takes a list of tuples of integers and strings and doubles the integer in every tuple.

```
fun double [] = []
|   double ((first:int*string)::rest) = (2 * #1first, #2first)::double(rest);

val double = fn : (int * string) list -> (int * string) list

- double [(1, "abc"), (2, "def")];
val it = [(2,"abc"),(4,"def")] : (int * string) list

BUT:

fun double [] = []
|   double (first::rest) = (2 * #1first, #2first)::double(rest);

=> Error: unresolved flex record (need to know the names of ALL the fields
   in this context)
```

2 Functions as parameters

Write a function `applyall` that takes a list of functions and a value and returns the list, the elements of which are the results of applying every function in the input list to the input value, in order.

```
fun applyall ([], _) = []
|   applyall (first::rest, x) = (first x)::applyall(rest,x);

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

fun positive n = n>0;
fun nonnegative n = n>=0;
fun id n = n;
fun double n = 2*n;
fun listoftwo L = length(L)=2;

applyall([positive, nonnegative], 0);
val it = [false,true] : bool list

applyall([id, double], 1);
val it = [1,2] : int list

applyall([null, listoftwo], [1,2]);
val it = [false,true] : bool list
```

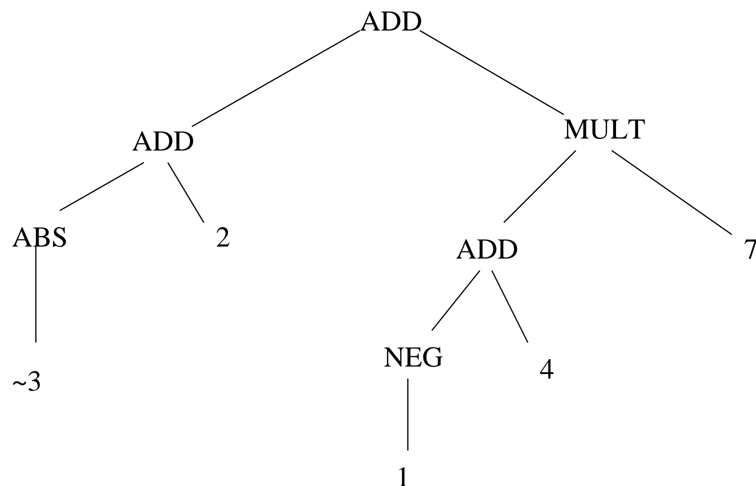
3 Functions and their types

Any ML function accepts **one** argument. The type of a function is completely determined by the types of its argument and its return value.

```
- fun switch(x,y) = (y,x);

val switch = fn : 'a * 'b -> 'b * 'a

- fun add_dummy (x, y, z) = (x, y, z, "dummy");

val add_dummy = fn : 'a * 'b * 'c -> 'a * 'b * 'c * string

- fun add_dummy x = "dummy"::x;

val add_dummy = fn : string list -> string list

-fun double [] = []
 |   double ((first:int*string)::rest) = (2 * #1first, #2first)::double(rest);

val double = fn : (int * string) list -> (int * string) list

-fun applyall ([], _) = []
 |   applyall (first::rest, x) = (first x)::applyall(rest,x);

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

-fun applytwice (func, value) = func (func value);

val applytwice = fn : ('a -> 'a) * 'a -> 'a

-fun applyboth (func1, func2, value) = func1 (func2 value);

val applyboth = fn : ('a -> 'b) * ('c -> 'a) * 'c -> 'b
```

4 Variant Types

```
(* A new type: dollars, which is either US dollars or Canadian dollars. *)
datatype dollars = USD of real |
                   CAD of real;

(* euro = fn: dollar -> real
 * return the equivalent in EURO *)
fun euro (CAD x) = 0.75 * x
|   euro (USD x) = 0.85 * x;

(* A new type: account.
    chequing: amount, interest rate, service charges per year
    savings:  amount, interest rate
    invest:   amount, interest rate, minimum balance *)
datatype account = chequing of  dollars*real*dollars |
                   savings of   dollars*real         |
                   invest of    dollars*real*dollars;

(* calculate = fn : account -> real
 * return the bank balance in euro after 1 year. *)
fun calculate (chequing (amt, rate, charge)) =
     (1.0+rate)*euro(amt) - euro(charge)

|   calculate (savings  (amt, rate)) =
     (1.0+rate)*euro(amt)

|   calculate (invest   (amt, rate, min)) =
         if euro(amt) < euro(min) then euro(amt)
         else (1.0+rate)*euro(amt);

calculate(chequing (100.0, 0.1, 5));
- Error: operator and operand don't agree [tycon mismatch]

calculate(chequing (dollars 100.0, 0.1, dollars 5));
- Error: unbound variable or constructor: dollars

calculate(chequing (CAD 100.0, 0.1, CAD 5.0));
val it = 68.25 : real

calculate(invest (CAD 100.0, 0.25, USD 50.0));
val it = 81.25 : real
```



5 Recursive Types

```
datatype mathTree = leaf of int |
                    unary of (int -> int) * mathTree |
                    binary of (int * int -> int) * mathTree * mathTree;

fun add (x,y) = x + y;
fun mult (x,y) = x * y;
fun neg (x) = ~x;
fun abs (x) = if x >= 0 then x else ~x;
```

Create the tree in the figure above.

```
val myMathTree =   binary(add,
                          binary(add,
                                 unary(abs,leaf(~3)),
                                 leaf(2)),
                          binary(mult,
                                 binary(add,
                                        unary(neg,leaf(1)),
                                        leaf(4)),
                                 leaf(7)));
val myMathTree =
  binary (fn,binary (fn,unary #,leaf #),binary (fn,binary #,leaf #)) : mathTree
```

```
(* eval = fn : mathTree -> int
 * evaluate the mathTree *)
fun eval (leaf(n)) = n
|   eval (unary (f,T)) = f (eval T)
|   eval (binary(f,L,R)) = f (eval(L),eval(R));

- eval myMathTree;
val it = 26 : int
```

What about just a binary tree? I.e., *any* binary tree?

```
datatype 'a tree = leaf of 'a |
                   node of ('a tree) * ('a tree);


(* count = fn : 'a tree -> int
 * return the number of leaves *)

fun count (leaf _) = 1
  | count (node (L,R)) = count L + count R;
```