Trees

Lists are a very useful data structure, but not ideal for everything. For many applications, using **trees** will lead to asymptotically faster programs.

ML allows us to define **new** data structures including trees with a `datatype` declaration.

Example: binary trees with values at the internal nodes.

```
- datatype intbtree =
    Leaf
  | Node of intbtree * int * intbtree
datatype  intbtree
  con Leaf : intbtree
  con Node : intbtree * int * intbtree -> intbtree
```

This **recursive** declaration has two effects:

- It declares a brand-new type called `intbtree`.
- It declares two **data constructors** for that type.

These constructors can be used (like functions) to **construct** values of the datatype and can be used in patterns to **analyze** values.

```
- fun insert x Leaf = Node(Leaf,x,Leaf)
    | insert x (Node(left,y,right)) =
                if x < y then
                  Node(insert x left,y,right)
                else
                  Node(left,y,insert x right)

- val b = insert 6 a;
val b =
  Node
    (Node (Leaf,1,Leaf),2,
     Node (Node (Leaf,4,Leaf),5,
           Node (Node (Leaf,6,Leaf),8,Leaf))) : intbtree

- lookup 5 b;
val it = true : bool
- lookup 6 a;
val it = false : bool
```

Operations on trees

Suppose using **ordered** `intbtree` to represent sets.

```
- val empty = Leaf;
val empty = Leaf : intbtree

- val a =
   Node(Node(Leaf,1,Leaf),2,
        Node(Node(Leaf,4,Leaf),5,Node(Leaf,8,Leaf)));
val a = Node (Node (Leaf,1,Leaf),2,
              Node (Node #,5,Node #)) : intbtree
- System.Control.Print.printDepth := 100;  (* sml/nj *)
val it = () : unit
- a;
val it =
  Node (Node (Leaf,1,Leaf),2,
        Node (Node (Leaf,4,Leaf),5,Node (Leaf,8,Leaf)))
  : intbtree

- exception NotFound
- fun lookup x Leaf = false
    | lookup x (Node(left,y,right)) =
              x = y
              orelse (x < y andalso lookup x left)
              orelse (x > y andalso lookup x right)
```

Equivalent C code

```
typedef struct intbtree {
        struct intbtree *left;
        int value;
        struct intbtree *right;
        } *Intbtree;

Intbtree NODE(Intbtree l,int v,Intbtree r) {
   Intbtree t = (Intbtree)
                  (malloc(sizeof(struct intbtree)));
   assert(t);
   t->left = l;
   t->value = v;
   r->right = r;
   return t;
};
```

(Assume `NULL` pointer corresponds to `LEAF`.)

```
int lookup(int x,Intbtree t) {
   return
    (t != NULL) &&
    ((x == t->value)
     || (x < t->value && lookup(x,t->left))
     || (x > t->value && lookup(x,t->right)));
}
```

Polymorphic trees

Can define a binary tree datatype that carries values of any type:

```
- datatype 'a bintree =
       Leaf
     | Node of 'a bintree * 'a * 'a bintree;
datatype  'a bintree
  con Leaf : 'a bintree
  con Node : 'a bintree * 'a * 'a bintree -> 'a bintree
```

Here `bintree` is really a **type constructor**, rather than a type, because it can be applied to arbitrary types `'a` to construct new types `'a bintree`.

As expected, we can define polymorphic functions over such types.

Datatypes in General

The tree definitions we've seen are just a special case of a very general mechanism for defining new type (constructors).

In general, a datatype declaration may include:

- any number (> 0) of data constructors, each with any number (≥ 0) of arguments;
- any number (≥ 0) of type variables;
- any number (> 0) of mutually-recursive type constructors.

Many familiar types (some of them built-in) are really just special cases of `datatypes`.

- Lists. Built-in lists behave just as though they were defined like this:

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

- Discriminated union types in general.

```
- fun size Leaf = 1
    | size (Node(left,y,right)) =
                (size left) + 1 + (size right);
val size = fn : 'a bintree -> int

- fun sum Leaf = 0
    | sum (Node(left,y,right)) =
                (sum left) + y + (sum right);
val sum = fn : int bintree -> int

- fun reflect Leaf = Leaf
    | reflect (Node(left,y,right)) =
                Node(right,y,left);
val reflect = fn : 'a bintree -> 'a bintree

- fun map f Leaf = Leaf
    | map f (Node(left,y,right)) =
                Node(map f left,f y, map f right);
val map = fn : ('a -> 'b) -> 'a bintree -> 'b bintree

- fun reduce f a =
    let fun g Leaf = a
          | g (Node(left,y,right)) = f(g left,y,g right)
    in g
    end;
val reduce =
   fn : ('a * 'b * 'a -> 'a) -> 'a -> 'b bintree -> 'a

- val size' = reduce (fn(l,y,r) => l + 1 + r) 1;
val size' = fn : 'a bintree -> int
- val sum' = reduce(fn(l,y,r) => l + y + r) 0;
val sum' = fn : int bintree -> int
```

- Booleans. Built-in booleans act just like this:

```
datatype bool = false | true
```

- Enumerated types in general:

```
datatype day =
     Sun | Mon | Tues | Wed | Thurs | Fri | Sat
```

- Option type.

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

fun search x nil = NONE
  | search x ((k,v)::t) = if x = k then
                            SOME v
                          else search x t
val search = fn : ''a -> (''a * 'b) list -> 'b option
```