CS 312 Recitation 6
Structures, signatures & more functional examples

When we try to use SML to build larger programs, and particularly when the software is being developed by a team of programmers, more language features become handy. One such feature is the *module*, which is a collection of datatypes, values, and functions that are grouped together as one syntactic unit. A well designed module is reusable in many different programs. Modules also provide a good way to structure group development of software, because they make a convenient way to cut up the program and assign responsibilities to different programmers. Modules are even useful for sufficiently large single-person software projects, because they reduce the amount of information that the programmer needs to remember about the parts of the program that are not currently under development. We've already been using modules when we write qualified identifiers of the form ModuleName.id to access Basis Library functionality. Now we'll see how we can write our own modules.

Structures

Modules in SML are implemented by `structure` declarations that have the following syntax:

```
structure StructureName = struct
    declarations
end
```

By convention, `structure` names always start with capital letters. Examples of the declarations that can go inside of a `structure` are:

- `open List`
  Puts all the declarations that are in the `List` structure in this structure, so that you do not need to say `List.partition` but can instead type `partition`.
- `type myInt = int`
  Tells the interpreter that anytime it sees the type `myInt` that we actually mean the type `int`.
- `datatype myNum = Int of int | Real of Real`
- `val meaningOfLife = 42`
- `fun askQuestion1 = "What is your name?"`
- `structure L = List`
  Inside the structure, you may now refer to `List.partition` as `L.partition` as well.
- `exception WrongAnswer`
  `exception Fail of string`
  Defines an exception that can be raised. Exceptions can also carry values with them (see the `Fail` exception for example).

Accessing `structure` members is done in much the same way that you access Java class members, `StructureName.declaration`. This works for all of the declarations above, including the `type, datatype, exception, and structure` declarations.

Signatures

To successfully develop large programs, we need more than the ability to group related operations together, as we've done. We need to be able to use the compiler to enforce the separation between different modules, which prevents bad things from happening. Signatures are the mechanism that enforces this separation.

A signature declares a set of types and values that any module implementing it must provide. It might look something like the following:

```
signature SIG_NAME = sig
    exception WrongAnswer
    type myInt
    datatype myNum = Int of int
    val x:int
    val add:int * int -> int
end
```

And a `structure` defines the `signature` that it implements as follows:

```
structure StructureName :> SIG_NAME = struct
      declarations
end
```

Note that `signature` names are all-caps by convention. If a `structure` implements a `signature`, then anyone using the `structure` may not see or use any values, types, or exceptions other than what is defined in the signature.

Maps

Often in computer science we need a mapping from one kind of value to another. For example, if we wanted to store a telephone book, we would need a mapping from names (strings) to phone numbers (ints). There are many ways to implement maps, but ML allows us a truly creative and unique representation that languages like Java do not. We will represent maps as functions.

```
type ('a,'b) map = 'a->'b
```

The empty map will be a function that returns an error.

```
exception EmptyMap
val empty = fn _ => raise EmptyMap
```

If you're confused now, the lookup function will confuse you even more.

```
fun lookup (m:('a,'b) map) (k:'a) = (m k)
```

What we are doing here is storing the lookup function as the map itself. ML allows us to do this because functions are first-class objects. When we insert a key/value pair, we create a new lookup function that returns the value given a key.

```
fun insert (m:('a,'b) map) (k:'a,v:'b) : ('a,'b) map =
  fn x => if x=k then v else (m k)
```

Or, with curried syntax,

```
fun insert (m:('a,'b) map) (k:'a,v:'b) (x:'a) =
  if x=k then v else (m k)
```

Of course, this implementation does not allow removal, so its uses are limited. However, if you can understand this code, you have a good feel for many key ideas in functional programming.

Technical subtlety: If you actually type in the above code for insert, ML will complain about a type mismatch. This has to do with the way ML handles polymorphic functions using the equality operator. The idea is that a polymorphic function should work at all types, but equality is not defined at all types; ML therefore requires us to use another kind of type variable, which will range only over those types where equality is defined. Those variables are written: `''a`. Therefore, if you want to write code which actually compiles, you should use the following:

```
fun insert (m:(''a,'b) map) (k:''a,v:'b) : (''a,'b) map =
  fn x => if x=k then v else (m k)
```

Or, with curried syntax,

```
fun insert (m:(''a,'b) map) (k:''a,v:'b) (x:''a) =
  if x=k then v else (m k)
```

For more information on this, see section 2.9 (pp. 38-40) of Riccardo Pucella's **notes.**

Implementing Binary Trees with Tuples

In recitation, we saw an example of using a datatype to define integer lists in terms of an empty list (*Nil*) and cons cells (a head containing an integer and a tail consisting of another list). We were able to iterate over the list, doing various manipulations on the data, and we were able to represent this concisely using higher-order functions. Today we're going to start by doing the same thing with binary trees to make sure everyone is very comfortable with pattern matching in **case** expressions.

The obvious way to start is with the following datatype, which we saw at the end of the last lecture:

```
datatype inttree = Leaf | Branch of (int * inttree * inttree)
```

This defines a type `inttree` to be either a leaf node (containing no data) or a branch node (containing an int and left and right subtrees). We could have defined a leaf note to contain an integer and no subtrees (some people do this), but then we'd need another constructor to represent the empty tree. Consider the representation of a generic tree.

The first logical function to write is `is_empty`:

```
fun is_empty (xs:inttree) : bool =
    case xs of
        Leaf => true
      | _ => false
```

Then, just as we computed the length of a list, we can count the non-leaf nodes in a tree:

```
fun size (xs:inttree) : int =
    case xs of
        Leaf => 0
      | Branch(_, left, right) => 1 + size(left) + size(right)
```

The pattern matching done in this function is very powerful. (If you don't see the power yet, you certainly will when the datatypes become as complicated as our definition of expressions in ML.) We can make very trivial changes to this function to compute several other interesting values:

- How could we modify this function to count the leaf nodes?
- How could we modify this function to determine the depth of a tree?
- How could we modify this function to sum the elements in a tree?
- How could we modify this function to reflect a tree about its center?

Implementing Binary Trees with Records

For both lists and trees, we've been using tuples to represent the nodes. But with trees, there may be some confusion with respect to the order of the fields: does the datum come before or after the left subtree? We can solve this problem using a record type. We can define it as

```
datatype inttree = Leaf | Branch of { datum:int, left:inttree, right:inttree }
```

(Note: Binary trees are simple enough that this probably would not be adequate motivation to use records in a real program, since we now have to remember the field names and spell them out every time we use them. Skipping to the next section on polynomials is fine if running low on time.)

Using this new representation, we can write *size* as

```
fun size (xs:inttree) : int =
    case xs of
        Leaf => 0
      | Branch{datum=i, left=lt, right=rt} => 1 + size(lt) + size(rt)
```

We've written several functions to analyze trees, but we don't yet have a way to generate large trees easily, so if you want to try these functions in your compiler, you'd have a lot of typing to do to spell out a tree of depth 10. Let's get the compiler to do it for us.

```
fun complete_tree (i:int, depth:int) : inttree =
    case depth of
        0 => Leaf
      | _ => Branch{datum=i,
                    left=complete_tree(2*i, depth-1),
                    right=complete_tree(2*i+1, depth-1)}
```

This function will take an integer *i* and a depth and recursively create a complete tree of the given depth whose nodes are given distinct indices based on *i*. If we start with *i*=1, then we get a complete tree whose preorder node listing is 1, 2, 3, etc. Consider the example given by

```
val test_tree = complete_tree(1,3)
```

Now that we have an example tree to work on, we need a cleaner way to visualize the tree than looking at the compiler's representation of records. Let's write a function to print the contents of a tree in order:

```
fun print_inorder (xs:inttree) : unit =
    case xs of
        Leaf => ()
      | Branch{datum=i, left, right} => (print_inorder(left);
                                         print(" " ^ Int.toString(i) ^ " ");
                                         print_inorder(right))
```

Notice that here we did not provide names for binding the left and right subtrees. Actually, the use of record labels only is just syntactic sugar for binding the same name to its value, so we could have written "*datum=i, left=left, right=right*". Anyway, our function behaves as follows on our test tree:

```
- print_inorder(test_tree);
 4  2  5  1  6  3  7 val it = () : unit
```

We could have applied many other functions to each element of the tree. A standard data structure operation is *apply*, which executes a given function on every element. The function is evaluated for side-effects only; the return value is ignored. How could we write *apply_inorder* for our trees?

```
fun apply_inorder (f:int->unit, xs:inttree) : unit =
    case xs of
        Leaf => ()
      | Branch{datum, left, right} => (apply_inorder(f,left);
                                       f(datum);
                                       apply_inorder(f,right))
```

Using this, we can write a very short version of *print_inorder*:

```
fun print_inorder (xs:inttree) : unit =
    apply_inorder(fn (i:int) => print(" " ^ Int.toString(i) ^ " "), xs)
```

Another common operation is *map*, which generates a copy of the data structure in which a given function has been applied to every element. We can write *apply_inorder* as

```
fun map_tree (f:int->int, xs:inttree) : inttree =
    case xs of
        Leaf => Leaf
      | Branch{datum=i, left, right} => Branch{datum=f(i),
                                               left=map_tree(f,left),
                                               right=map_tree(f,right)}
```

How could we use this to square a tree?

```
val tripled_tree = map_tree(fn (i:int) => i*3, test_tree)
```