CSC/MAT-220: Lab 2

Due: 9/10/2018

At its core, ML consists of expressions to be evaluated. Each expression has three characteristics: type, value, and effect. Some examples of types are int, real, char, string, bool, and the functional type. Recall that ML does not perform implicit conversions, like Python does, and mixed types will cause an ill-typed expression that will not be evaluated. A well-typed expression is evaluated to determine its value. The first expression bellow evaluates to a value of 5, and the second expression evaluates to a value of (). Note that () is called a unit, which is an empty tuple that acts as a trivial placeholder.

```
(* SML Example 1 *)
2 + 3;
print("Hello World\n");
```

Much of what follows has been adapted from [1, Chapter 5] and [2, Chapter 4].

Bindings, Scope, and Functions

Just as in other languages, in ML we can store values in variables and then use these variables in proceeding expressions. However, in contrast to other languages, variables in ML do not vary. It is for this reason that we adjust our terminology and say that a value is bound to a variable. For example, the expression below will bind the value 2 to the variable x.

```
(* SML Example 2 *)
val x = 2;
```

This type of binding is called a *value binding*. In addition, we may bind types, see the expression below.

```
(* SML Example 3 *)
type count = int;
```

The power in *type bindings* will become more clear when we create our own types.

At this point, we note that the value binding above forces the compiler to implicitly determine the variable's type. We can explicitly declare this for the compiler, see the example below.

```
(* SML Example 4 *)
val pi:real = 3.14 and e:real = 2.71;
```

Note that, *and* is a logical operator for evaluating two expressions. Whereas, *andalso* is an operator for combining two boolean variables.

The purpose of a binding is to make a variable or type available for use within a particular scope. In ML, scope is static; therefore, the scope of a binding is determined by context and not the order of evaluation. In the previous examples, we assumed global scope; in what follows we discuss how to limit scope.

The scope of a binding may be limited using *let* and *local* expressions. The form of these expressions are as follows.

let *dec* in *exp*
The scope of the declaration *dec* is limited to the expression *exp*.

local *dec* in *dec'*
The scope of the declaration *dec* is limited to the declaration *dec'*.

To clarify when you should use *let* vs. *local*, note that the former results in a expression, whereas the latter results in a declaration. The *local* expression is especially useful when declaring temporary helper functions. To gain an appreciation for the power of limiting scope, consider the following example.

```
(*SML Example 5 *)
val m:int = 2;
val r:int =
        let
                val m:int = 3
                val n:int = m*m
        in
                m*n
        end*m;
```

Before moving on, go do Problem I.

Recall that the function data type in ML is of the form $type \rightarrow type'$, and allow us to abstract the data from a calculation. Furthermore, functions in ML are first-class data types. Meaning, they may be computed as the value of an expression, bound to a variable, and passed as an argument of a function. Consider the following example.

```
(* SML Example 6 *)
val x:real = 1.0;
val y:real = 2.0;
val f:real->real = fn x:real => x+y;
val g:real->real = fn y:real => x+y;
val h:real*real*(real->real)*(real->real)->real =
        fn (x:real,y:real,f:real->real,g:real->real) =>
                let
                        val x:real = 3.0
                in
                        f(x)
                end*g(y)*f(x);
```

This is a rather complex example. The occurrence of x in the body of the function f refers to the parameter of f, whereas the occurrence of x in the body of the function g refers to the *val* binding. To make things more interesting, the function h has a local *val* binding which is shadowing the parameter x. Before proceeding, go do Problem II.

Aggregate Data Structures

An aggregate data structure is any type of data that can be referenced as a single entity, and yet consist of more than one piece of data. A distinguishing feature of ML is that aggregate data structures, such as tuples and lists, may be created and manipulated with ease. It is not necessary to concern yourself with the allocation or deallocation of data structures, such as in Fortran, nor with any particular representation strategy involving pointers, such as in C.

Tuples

An *n-tuple* is a finite ordered sequence of values of the form

$$(val_1,\ val_2, \ldots, val_n)\,,$$

and is of a *product type* of the form

$$type_1 * type_2 * \cdots * type_n.$$

Note that each type can differ, even within the same tuple.

The *0-tuple*, also known as the *null tuple*, is the empty sequence of values (). In ML, it is a value of type *unit*. The null tuple type is surprisingly useful, especially in the presence of effects. On the other hand, there seems to be no use for the *1-tuple*, so they are absent from ML. A *2-tuple*, or ordered pair, is defined in the following example

```
(* SML Example 7 *)
val pair:int*int = (2,3)
```

Similarly, the following example has well formed bindings for a triple and pair of pairs.

```
(* SML Example 8 *)
val triple:int*real*string = (2,2.0,"2");
val pair_of_pairs:(int*int)*(real*real) = ((2,3),(2.0,3.0));
```

Note that the nesting of the parentheses matters. A pair of pairs is not the same as a quadruple. A very interesting and powerful feature of ML is the use of *pattern matching* to access specific elements of an aggregate data structure. See the following example

```
(* SML Example 9 *)
val (x:int,y:real,z:string) = triple;
val ((_,_),(x:real,_)) = pair_of_pairs;
```

In the first binding, we are assigning the corresponding values in *triple* to the variables x, y, and z, respectively. In the second binding, the underscores indicate positions in the pattern that we are ignoring, and the result of the expression is the value 2.0 is bounded to the variable r. Before proceeding, go do problem III.

Lists

Tuples are powerful because they allow for mixed types; however, they have a fixed size. Lists, on the other hand, cannot have mixed types, but allow for a dynamic size. The example below shows two lists of varying size, but both of type *int list*.

```
(* SML Example 10 *)
val list1 = [1,2,3];
val list2 = [~3,~2,~1,0,1,2,3];
```

Lists are an example of a recursive data structure and will discuss this in more detail in a future lab. For now, it is important to note the difference between the data structure of a tuple and a list. An n-tuple has n components, and n is a fundamental aspect of the type. A list on the other hand always has two components: a *head* and a *tail*. For instance, the head of list1 is 1 and the tail is $[2, 3]$. Note that the head is of type int and the tail is of type int list. A list can always be composed from a head and a tail, and a list can always be decomposed based on its head and its tail. For instance, see the example below.

```
(* SML Example 11 *)
val list = 1::[2,3];
val h::t = list;
```

In the previous example, the binary constructor :: constructs a non-empty list from the value 1 and $[2, 3]$. The constructed list is equivalent to list1 from Example 10. Furthermore, we can use the constructor :: in pattern matching to decompose the head and the tail of the list and assign them to the variables h and t, respectively. Since the list is a recursively defined data structure, the tail of every list can also be broken into a head and tail, and so on until the tail is the empty list. Consider the following example.

```
(* SML Example 12 *)
val list = [1,2,3,4,5];
val _::h::t = list;
```

In the previous example, we use the underscore to ignore the initial head of the list. Thus, h will have value 2 and t will have value $[3, 4, 5]$.

Assignment

At the top of your file include a commented line with your name, lab number, and date. In addition, be sure to comment generously throughout so I can follow your steps. Finally, when you are done upload your source code to Dropbox. The file you you turn in should be labeled as follows: *yourname_labnumber.sml*.

I. Without writing the code in SML, determine the value of the variable r in Example 5. Provide an explanation for what is happening.

II. Without writing the code in SML, determine the value of the following expression

```
h(~1.0,3.0,f,g);
```

where f, g, h are as defined in Example 6. Provide an explanation for what is happening.

III. Use tuples to write a function that evaluates to a pair of type real*real that contains the roots of a quadratic polynomial: $ax^2 + bx + c$, where a, b, c are real numbers.

IV. Write a function that takes on an int list of length at least 2, then takes the first two elements of the list and appends the first element to the back of the list and the second element to the front of the list. Perform testing on a list of length 2, 3, 4, and 5.

Note that when appending to the front, the use of the :: constructor will suffice, but when appending to the back you may need a function from SML's list structure. Be sure that this is the only time you use a function from SML's list structure on this assignment, we will talk more about these functions in a future lab.

V. Write a function that takes on a list of the first n Fibonacci numbers and returns a list with the first $(n + 1)$ Fibonacci numbers. Perform testing on a list of length 2, 3, 4, and 5.

References

[1] R. Harper, *Programming in Standard ML*, Carnegie Mellon University, Pittsburgh, PA, 2011.

[2] T. VanDrunen, *Discrete mathematics and functional programming*, Franklin, Beedle and Associates, Wilsonville, OR, 2012.