# ML, Very Briefly.

For students who have not purchased or rented a copy of "Elements of ML Programming" by Jeffrey D. Ullman, this is a *very* brief introduction to ML.

## Comments

Comments are delineated by two markers, starting and ending a comment: ***(*** and ****)***. Oddly, these can be nested, as in the start and end markers are paired, just as parentheses are in expressions.

## How to Start the Interpreter and Escape It!

To get the ML interpreter on the csx server up and running, log into your csx account, and type ***sml*<*enter*>**. You will see a line that says, "Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM; autoload enabled]", then a command prompt that is a single hyphen (or dash). To exit the interpreter enter **<*control*>*d***.

## What Numbers (Constants) Look Like

Rather than do things the way most other languages do them, ML has some "peculiarities". Perhaps the oddest is that "-n" as "negative n" is wrong. Instead you use **~*n***. This eliminates the confusion that might arise between instances of the unary minus operator and the subtraction operator ("-"), but it cause lots of errors, particularly when one doesn't use ML much, or hasn't used it in a while.

Fortunately, in other regards, integers and real numbers look like you expect them to. Typing "1331" gives you an integer that is the cube of 11. Typing "3.14159" gives you the six digit approximation of π as a real number. So, apart from an odd unary operator, constants pretty much look like constants in other languages.

## Coercion

ML does not allow automatic conversions (coercion) between integers and reals. You must explicitly convert numbers from one type to another, if conversion is required in your code. There are five functions for handling these conversions between real and integer types.

***real(n)*** – accepts an int as its argument and returns a real.

***floor(x)*** – accepts a real as its argument and returns the largest integer less than or equal to x.

***ceil(x)*** – accepts a real as its argument and returns the smallest integer greater than or equal to x.

***round(x)*** – is the usual rounding, returning the integer closest to x, with 0.5 rounded up.

***trunc(x)*** – accepts a real as its argument and returns integer with the portion to the right of the decimal removed.

## Arithmetic Operators

If you are going to make negation peculiar, why not make one or two arithmetic operators a bit peculiar, too? ML does that. The additive operators are the + and - we expect, as well as * for multiplication, but there are differences between reals and integers when it comes to division. The / character does the job for real numbers, but integers use ***div***. For integers, you also have ***mod*** for the "modulos" function, which isn't really the modulus of math but the remainder in integer division.

Integer division "rounds down" towards negative infinity. To see this, try using the following two commands in the sml interpreter.

```
~3 div 2;
3 div 2;
```

What you get is the following evaluations: ***val it = ~2 : int*** and ***val it = 1 : int***, respectively.

Remember, you cannot mix integers and reals in an arithmetic expression.

## We Have *true* and *false* Answers …

For convenience, the Boolean values ***true*** and ***false*** are predefined.

## Stringing Things Along with Characters

The designers of ML graciously decided to make strings in ML look like strings in many other languages. Thus **"This is a string."** is the string that says it is a string. Other good news is that the ***n*** and ***t*** from C and Java mean the same thing in ML: new line and horizontal tab. That \ character is an escape character, so you can use \" in a string to put a double quote in it, without terminating the string. Further, like C, etc., if you put a \ in a string and follow it with three digits in the range 0 to 7, inclusive, you are specifying a character in the ASCII character set by its octal (base 8) representation. To put a \ in a string when you really mean "put a backslash right here," you escape the escape by using "\\".

Characters are different from strings, though, if you want to specify a character not in a string. **#"z"** means the character z, while **"z"** means the string of one character length.

To concatenate two strings together, you use the ^ character. If you type ***"dog" ^ "fish";*** at the sml command prompt, you get ***"val it = "dogfish" : string"*** as a response.

## Comparison Operators

ML does not let you test for equality between real numbers directly. Why? Because of the errors that build up over multiple operations using real numbers. So, keep that in mind when you consider the use of "=" and "<>", the equality and inequality operators.

The operators are rather obvious, so here they are: =, <, >, <=, >= and <>.

The interesting thing here is that these may be used to compare integers, reals, characters and strings, with the notable exceptions of = and <> for reals. In the cases of characters and strings, the operators test lexicographical relationships. Thus, **#"a" < #"b"** evaluates to true, as does the expression **"albatross" < "zebra"**.

## Logical Operators

Just when you thought things might not be so confusing, you have to deal with logic. Logical operators re the following: **not**, **andalso**, and **orelse**. The **not** keyword is the usual unary operator, while **andalso** simply means "and" and **orelse** (preferably said in a non-threatening tone) simply means "or". Evidently, the creators of ML think people just don't type enough – at least when using the rest of ML.

Both **andalso** and **orelse** evaluate the right-hand expression only if the left-hand expression evaluates to **true**.

## The *if-then-else* Expression.

As one might expect, since everything in ML is an expression to be evaluated, so is an **if-then-else** construct. If the logical expression of the **if**'s logical expression evaluates as true, the **then** clause is evaluated and the result is evaluation the **if-then-else** construct. Otherwise the expression in the **else** portion is evaluated and used as the evaluation of the **if-then-else**.

From the Ullman book we have the following example.

```
if 1<2 then 3+4 else 5+6;
```

Since 1 is less than 2, the expression between the **then** and **else** is evaluated, producing a result of 7, which is then the evaluation of the **if-then-else**.

## Lists!

ML has a datatype called a list. Lists are collections of elements of the same type. A list of the first eight Fibonacci numbers looks like this: **[0, 1, 1, 2, 3, 5, 8, 13]**.

Lists are one of the most common constructs in ML, as they server in place of arrays and vectors, and ML is pretty much designed to handle lists efficiently (so long as you think recursively).

There are some important and useful list operators and functions. The **@** operator is used to concatenate lists. For example, the expression **[0, 1, 1, 2] @ [3, 5, 8, 13]** evaluates as the list **[0, 1, 1, 2, 3, 5, 8, 13]**. But, if you want to add an element to the head of a list, you can use the **::** operator (called the *cons* operator) like in this example for an integer and integer list:

**1::[2,4,8,16]** evaluates as **[1,2,4,8,16]**.

You can use empty lists, which are expressed as **[]**.

The first element of a list is referred to as the head of the list, and the remainder following the head is call the tail. The function **hd(L)** returns the first element (or head) of list L, while **tl(L)** returns the tail of list L.

Something you should learn to use when declaring functions is that the cons operator (::) can by used to simplify the declaration considerably. For example, assume you will be passing a list to a function, and you want to be able to refer to the list as a whole, as well as its head and tail. You could say something like the following.

```
fun myFun(L:real list as h::T) = ...
```

You can even use more than one cons operator in such a situation where you are performing operations on the first two elements of the list, as shown here.

```
fun myfun(L:real list as h::s::T) = ...
```

You can also create lists from strings using the **explode(S)** function. If passed a string of characters, it returns a list of characters. For example, **explode("Hello, World!")**; evaluates as [#"H",#"e",#"l",#"l",#"o",#",",#" ",#"W",#"o",#"r",#"l",#"d",#"!"]. There is an inverse operation for this, provided by the function **implode(L)**, which takes a list of characters and returns a string.

## Tuples

A tuple is like a list, except it isn't a list. It is a collection of things. Where a list is created using square brackets ([]), tuples are created using parentheses. The elements of a tuple do not have to be of the same type. You can mix the members, as far as type, rather freely. Here's an example of a list with multiple datatypes in it.

```
val d = (1,2,#"A",(3.1,"string!",[0,1,1,2],("Fibonacci","seq.")),
true, false);
```

You can get the individual elements of a tuple using the **#** operator/function. The evaluation of **#3d** is **#"A"**. If you have multiple levels of tuples (such as above), you can nest the references such as **#4(#4d)**, which evaluates as **("Fibonacci","seq.")**.

## Declaring Variables

As shown above, you can declare variables and give them values. Declarations are quite simple, using the **val** key-word as shown below.

```
val fibseq8 = [0,1,1,2,3,5,8,13];
val myint = 1331;
val mypi = 3.14159;
```

The names/symbols for variables are strings of letters and digits, starting with letters.

## Declaring Functions

Declaring a function is easy, and permits considerable flexibility, since you can deal with much of the conditional character of a function (that would otherwise be dealt with using more complex logic) in the declaration. The following function is a recursive function to determine the number of elements in a list of integers.

```
fun myLLen([]) = 0
|   myLLen(L:int list as h::T) = 1 + myLLen(T);
```

First, notice that the function explicitly specifies the type of the list using **:int list**. Since ML is strongly typed, conflicts in type are common. You must often explicitly state, and properly specify the types of arguments when the operations performed within a function declaration are type sensitive – which is most of the time.

If you need to declare variables within a function, you can do so by using a **let** clause.

```
fun hundredpowX4(x:real) =
    let
        val a = 4.0;
        val four = x*x*x*x;
        val twenty = four*four*four*four*four;
    in
        a*twenty*twenty*twenty*twenty*twenty;
    end;
```

Notice we created sub-expressions as well as a variable.

## How to invoke a function

Using the function **myLLen()** above, assume you have an int list **A**. You invoke the function to determine the length of **A** by typing the following command.

```
myLLen(A);
```

You can create a variable to hold this returned length as follows.

```
val lenA = myLLen(A);
```

## Loading an Existing File of ML Code

Assume you have a file containing ML code, and the file is named "mycode.ml". You can load it by means of the **use** command.

```
use "mycode.ml";
```