CSC/MAT-220: Lab 1

Due: 8/27/2018

Installing LaTeX

The ability to use LaTeXis available on every school computer. If you are a Windows user, then open up the application TeXworks. If you are a Mac user, then open up the application TeXShop. Both environments are similar and you will be given a text editor for which you type your document. You can compile your document by pressing the green button (TeXworks), or by pressing Typeset (TeXShop). You can also compile your files in the terminal using the *pdflatex* command, see me if you are interested.

If you would like to use LaTeXon your own computer, then consider the following links:

TeXWorks: https://www.tug.org/texworks/
TeXShop: https://tug.org/mactex/mactex-download.html

For a brief introduction to using LaTeX, please review the Mathematical Writing Handout posted on our class website.

Installing SML/NJ

Standard ML of New Jersey (abbreviated SML/NJ) is a compiler for the Standard ML '97 programming language with additional libraries, tools, and documentation. To install SML/NJ on your computer, go to https://www.smlnj.org and find the appropriate download link for your operating system.

Windows: After running the installer, an item for SML of New Jersey will be added to your start menu and the command *sml* can be used in the command line.

Mac: The standard installation location is /usr/local/smlnj, if you changed that location then take note below. Open the terminal and enter the following command:

export PATH="$PATH:/usr/local/smlnj/bin"

Now you can use the command *sml* in the terminal.

Note that there is a Homebrew formula for easy install of SML/NJ, if you have Homebrew installed on your Mac then I highly recommend this avenue of installation.

Using SML

To begin a SML interactive session, simply enter the command *sml* at the command line/terminal. If successful, you will receive a message stating the name of the compiler, and you will be prompted with a

hyphen. The hyphen indicates that SML is awaiting an expression to *evaluate*, an equal sign indicates the expression is still in progress. Much of what follows has been adapted from [1, Chapter 2] and [2, Chapter 2].

The SML Basics

Computation in ML consists of evaluation of expressions. Each expression has three characteristics:

- It may or may not have a *type*.
- It may or may not have a *value*.
- It may or may not cause an *effect*.

Type: Every expression is required to have at least one type; those that do are said to be well-typed, those that don't are said to be ill-typed. Any ill-typed expression is considered ineligible for evaluation. The type checker determines whether or not an expression is well-typed, rejecting with an error those that are not. For example, the first expression below is well-typed, while the second expression is ill-typed.

```
(* SML Example 1 *)
3 + 4;
3.5 + 4;
```

Value: A well-typed expression is evaluated to determine its value, it is has one. For instance, the first expression has a value, whereas the second expression does not (actually it has type *unit*, more on that latter).

```
(* SML Example 2 *)
val x = 2;
print "Hello World";
```

If the expression has a value, the form of that value is predicted by its type. For example, the first expression above has type *int* and the type of the expression below is *bool* (note only one equal sign).

```
(* SML Example 3 *)
x = 3;
```

Effect: Effects include raising an exception, modifying memory, performing input or output, and sending a message on the network. Note that the type of an expression does not indicate what possible effects might take place. While we will not focus heavily on the effects of evaluation in this course, this is a good time to note a stark contrast to programming in ML as compared with imperative programming (such as in C or Java). In imperative programming, you are often working with mutable memory and commands are issued to retrieve or modify this memory. In ML, bindings are static and the value of a variable does not change while evaluating within the scope of that variable. This is a major aspect of the functional programming paradigm which seeks to avoid changing state and mutable data.

Types

Generally speaking types are defined by specifying three things: a *name* for the type, potential *values* for the type, and *operations* that may be performed on values of the type. Each type has a signature which can

be found online. For instance try the Google search *SML real*. Below we list some common types and their basic attributes.

- Type name: int
  - Values: ..., ~3, ~2, ~1, 0, 1, 2, 3, ... (note that tilde denotes a negative number in SML!)
  - Operators: +, -, *, /, =, <, ...
- Type name: real
  - Values: 3.14, ~2.17, ...
  - Operators: +, -, *, /, =, <, ...
- Type name: char
  - Values: #"a", #"b"
  - Operators: ord, =, <, ...
- Type name: string
  - Values: "abc", "123", ...
  - Operators: ^ , size, =, <, ...
- Type name: bool
  - Values: true, false
  - Operators: if *expr1* then *expr2* else *expr3*, andalso, orelse, not

Note that some types share the same operators, this is known as *overloading*. For example, in an expression involving addition the type checker attempts to determine which form of addition (fixed point or floating point) should be used. Note that SML does not perform any implicit conversions between types. Therefore, if the expression has mixed types then it is ill-typed.

Functions

Like all functional programming languages, a key feature of ML is the function. So far, we have the ability to calculate the values of expressions and to bind these values to variables. Functions can be used to abstract the data from a calculation, leaving behind the skeleton of the calculation.

A function in ML is a value of function type of the form $type \to type'$. The *type* is the domain type, and the $type'$ is the co-domain type. For example, Math.sqrt is a primitive function of type $real \to real$. With the square root function, we can build a function expression for the fourth root:

```
(* SML Example 4 *)
fn x : real => Math.sqrt (Math.sqrt x);
```

This function may be applied to an argument by writing an expression such as

```
(* SML Example 5 *)
(fn x : real => Math.sqrt (Math.sqrt x)) 16.0;
```

This calculation proceeds by binding the variable x to the argument 16.0, then evaluating the expression Math.sqrt (Math.sqrt x) in the presence of this binding. When the evaluation is complete, the binding of x is dropped.

Writing the function expression and argument at the same time can become tedious. So, we bind the function expression to a variable as follows.

```
(* SML Example 6 *)
val fourthroot : real -> real =
        fn x : real => Math.sqrt (Math.sqrt x);
```

This notation for defining functions can be cumbersome. Therefore, SML provides a more concise syntax for defining functions.

```
(* SML Example 7 *)
fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x);
```

The meaning in both definitions are the same, we are binding *fn x : real => Math.sqrt (Math.sqrt x)* to the variable fourthroot. Personally, I prefer the more explicit syntax of Example 7, especially for my students who are learning SML for the first time.

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. Go through each SML Example in the reading. For each example, provide a few sentences on why it was included and what you learned.

II. Is *ceil(2.3)* a well-typed expression? What about *ceil(2)*? Why or why not?

III. Mercury orbits the sun in 87.969 days. Write a function that computes the corresponding Mercury age given an Earth age.

IV. Finish the following function:

```
val min : int*int*int -> int =
    fn (x:int,y:int,z:int) =>
```

which is intended to evaluate to the minimum of the triple $(x : int, y : int, z : int)$. Hint: Make use of the boolean operators.

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.