COMP 212 Spring 2020
Lab 1

1 Getting Started

Install Standard ML according to the resources page of the course web site. Once you have followed those directions, you should be able to type

```
$ smlnj
```

When you run SML, you should get something that looks like:

```
$ smlnj
Standard ML of New Jersey v110.96
-
```

This is the SML *REPL* (read-eval-print-loop): it *reads* the programs you enter, *evaluates* them, *prints* the result, and then waits for more input. (I.e. it is equivalent of `coin` for C0.)

To quit the REPL, type `Control-d`.

2 Arithmetic

From here, we can type in SML expressions for sml/nj to evaluate. For example, if we want to add $2 + 2$, we write `2 + 2;`.

Task 2.1 Type

```
2 + 2;
```

into the REPL and press Enter. What is SML's output?

The output line, `val it = 4 : int`, is the result of evaluating the expression that you gave it. `it` is the default name for variables if a name is not provided, `4` is the actual result, and `int` is the type of the expression - in this case, an integer. SML uses types to ensure at compile time that programs cannot go wrong in certain ways; the phrase `4 : int` can be read "4 has type int".

Notice that the expression must be terminated with a semicolon; if we do not do this, the REPL does not know to evaluate the expression and expects more input.

Task 2.2 Type

```
2 + 2
```

(note there is no semicolon) into the REPL and press Enter. What is SML's output?

After doing that, type just a semicolon. What happens now?

As you can see, it is possible to put the semicolon on the next line and still get the same result.

2.1 Parentheses

In a math class a long time ago, you probably learned the rules of operator precedence - for example, you multiply before you add, but anything grouped in parentheses gets evaluated first. SML follows the exact same rules of precedence. Also note that you can add parentheses to expressions to change the order of evaluation.

Task 2.3 Type

```
1 + 2 * 3 + 4;
```

into the REPL. What would you expect the result to be? What is the actual result?

Now, type

```
(1 + 2) * (3 + 4);
```

into the REPL. Is the result the same?

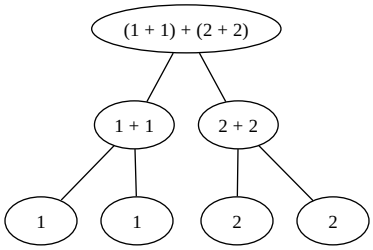
3 Evaluation

We can run SML programs by hand, calculating the value of an expression step-by-step. For example, `(1 + 1) + 1` steps to `2 + 1`, which steps to `3`, which is a value, at which point evaluation stops. The (sequential) running-time of the program is just the number of steps (for our purposes, we say that all arithmetic operations take exactly one step of computation and numbers take zero steps).

Task 3.1 Figure out how many steps it takes to evaluate `(1 + 2) * (3 + 4)` sequentially, writing out each intermediate step.

We can also evaluate multiple parts of an expressions in parallel, when those parts do not depend on each other. For example, suppose we have some expression `(...) + (...)`, where each set of parentheses contains a large subexpression. Instead of arbitrarily choosing a side to evaluate first, in a parallel setting it is possible to evaluate both sides at once!

It is easy to see what parts of a computation are independent if we draw the expression as a *tree* (trees grow down the page in computer science). For example, the tree for `(1 + 1) + (2 + 2)` looks like



The leaves represent values that do not need any computation to evaluate. In this case, the only other nodes in the tree are arithmetic operations, which we have defined to have a cost of 1.

Task 3.2 Write the tree for `(1 + 2) * (3 + (4 * 5))`. You may want to use another sheet of paper.

The *work* or sequential complexity of a program is the total number of steps it takes to evaluate it. The *span* or idealized parallel complexity is the length of the longest data dependency in the program—for example in the expression `(1 + 1) + 2` the outer `+ 2` depends on the inner `1+1` because we cannot add 2 to something until we know what the something is.

Once we have written an expression as a tree, the work is the *size* of the tree (in particular, the number of *internal* nodes, those not in the bottom row), and the span is the *depth* of the tree (the number of internal nodes on the longest path from the top to the bottom).

Work For example, in the case of `(1 + 1) + (2 + 2)`, the work is 3, since there are three additions that need to be made.

Task 3.3 What is the work associated with the tree for `(1 + 2) * (3 + (4 * 5))`?

Span The span of `(1 + 1) + (2 + 2)` is 2.

Task 3.4 What is the span associated with the tree for `(1 + 2) * (3 + (4 * 5))`?

4 Types

There are more types than just `int` in SML. For example, there is a type `string` for strings of text.

Task 4.1 Type

```
"foo";
```

into the REPL. What is the result?

Strings, then, behave just like integers - instead of seeing a number as the output, you see the string. It is also possible to concatenate two strings, using the `^` operator. This can be used just like `+` is used on integers.

Task 4.2 Type

```
"foo" ^ " bar";
```

into the REPL. What is the result?

We can write a program that is not well-typed to see what SML does in that situation. For example, you can only concatenate two strings.

Task 4.3 What happens when you try to type the expression

```
3 ^ 7;
```

into the REPL?

This is an example of one of SML's error messages - you should start to familiarize yourselves with them, as you will be seeing them quite a lot this semester!

5 Variables

Above, we mentioned that the results of computations are bound to the variable `it` by default. This means that once we have done one computation, we can refer to its result in the next computation:

Task 5.1 Type

```
2 + 2;
```

into the REPL. Then, type

```
it * 2;
```

into the REPL. What is the result?

As you can see, `it` stands for the result of the previous expression. Of course, `it` is not the only name possible for a variable. We can choose which name the REPL gives to a variable with the keyword `val`. Similar to the REPL's output, we say `val <varname> : <type> = <exp>` to bind the result of `<exp>` to `<varname>`.

Task 5.2 Type

```
val x : int = 2 + 2;
```

into the REPL. What is the result? How does it differ from just writing `2 + 2;`?

As you can see, that declaration binds the value of `2 + 2` to the variable `x`. We can now use the variable:

Task 5.3 Type

```
x;
```

into the REPL; what is the result?

Task 5.4 Type

```
val y : int = x * x;
```

into the repl. What is the result?

Task 5.5 Try to type

```
val z : int = "3";
```

into the REPL. What happens? Why?

Task 5.6 After that, try to type

```
z * z;
```

into the REPL. What happens? Why?

As you can see, trying to define a variable with the wrong type is an error, as is trying to refer to a variable that is not defined. It is also important to keep in mind that variables in SML are different from variables in C_0 and other imperative programming languages. Each time a variable is declared, SML creates a fresh variable. If the name was already taken, the new definition *shadows* the previous definition: the old definition is still around, but uses of the variable refer to the new definition.

Task 5.7 Write the following in the REPL:

```
val x : int = 3;
val x : int = 10;
val x : string = "hello, world";
```

What are the value and type of `x` after each line?

6 Using Files

Quit the SML REPL by typing `Control-D`.

Make a directory named `comp212`, by typing (in the terminal)

```
mkdir comp212
```

This will create a new directory (on a Mac: in `/Users/<yourname>/comp212`
on Windows, in `C:\cygwin\<yourname>\comp212`).

Download `lab01.sml` from the course web site, and save it into the directory you just made.

Back **in the terminal**, type

```
cd comp212
```

to go into your `comp212` directory, and then start `smlnj` again. Now that you are back in SML, to load the file into the REPL, type `use "lab01.sml";`. The output from SML should look like

```
- use "lab01.sml";
[opening lab01.sml]
...
val it = () : unit
-
```

Now that you have done this, you have access to everything that was defined in `lab01.sml`, as if you had copied and pasted the contents of the file into the REPL.

7 Functions

7.1 Applying functions

In this file, notice that there are functions defined. For example, there is

```
(* takes an int and returns the corresponding string *)
val intToString : int -> string
```

In this case, the function can be invoked by writing `intToString(37)`. However, the parentheses around the argument are actually unnecessary. It doesn't matter whether we write `intToString 37` or `(((intToString) (37)))` – both are evaluated exactly the same.

Task 7.1 Type

```
(intToString 37) ^ " " ^ (intToString 42);
```

into the REPL. What is the result?