15-150 Spring 2016
Lab 1

January 13, 2016

Welcome to 15-150's first lab! In each lab we give you problems to work on with the assistance of the TAs. This week we cover some of the basics, so you start to know SML a little.

We are recording attendance each week. Please show up on time! Don't worry if you run out of time. You will still get credit for the lab; credit is based on participation, not completeness.

1 Getting Started

1.1 Setting Your Path

If you are logged in to a cluster machine running Linux or OSX, or if you are ssh'd into a UNIX machine, we provide a macro called `smlnj` that you can use at

```
/afs/andrew/course/15/150/bin/smlnj
```

To be able to conveniently use this without typing the entire path each time, you will need to modify your `$PATH` environment variable as follows

Task 1.1 First, determine which shell you are running with the command `echo ${SHELL}`.

- If `$SHELL` is `/bin/bash` (or something ending in `bash`), add the line

  ```
  PATH="${PATH}:/afs/andrew/course/15/150/bin"; export PATH
  ```

  to the file `.bashrc` in your home directory.

  To actually bring this new definition for `PATH` into effect, you can either start a new shell or type

  ```
  source ~/.bashrc
  ```

- If `$SHELL` is `/bin/csh` (or something ending in `csh`), add the line

```
set path = ( $path /afs/andrew/course/15/150/bin/ .)
```

to the file `.cshrc` in your home directory. If you get an error when you try to run `smlnj` after adding this line to your `.cshrc`, ask a TA to look at your `.cshrc`.

To actually bring this new definition for `PATH` into effect, you can either start a new shell or type

```
source ~/.cshrc
```

Now that you have done this, you can run SML with the command `smlnj`.

1.1.1 Additional SML implementations

If you want to run SML from your own machine, you have a few options. If you have SML installed locally, you just type `sml`. You may notice that this does not allow you to conveniently navigate the interface with the arrow keys to access other places on the line or your command history; if you want to do this, you have to run `rlwrap sml`, where `rlwrap` is part of GNU Readline. You should be able to install this with your package manager on a Linux distribution, or with MacPorts on OSX. Also note that `smlnj` is just a macro we defined for `rlwrap sml`, so you may find it beneficial to do the same thing on your local machine. There are installation instructions for SML on the course website.

Keep in mind that you can access the Unix machines from any machine with an internet connection by sshing to `unix.andrew.cmu.edu`; this may be more convenient than setting up SML to run locally.

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

```
hqbovik@unix14 hqbovik % smlnj
Standard ML of New Jersey v110.76 [built: Mon Sep  9 21:09:47 2013]
-
```

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.

To get out of the REPL, type `Control-d`.

1.2 Cloning The `git` Repository

`git` is a popular version control system. We will be using a `git` repository to distribute all of the files for your homeworks and labs to you for this semester. It will also include compiling versions of code from lecture for you to play with, and various other helpful documents as the course progresses.

Task 1.2 Follow the instructions in Section 2 of

`http://www.cs.cmu.edu/~15150/resources/handouts/git.pdf`

and check out the `git` repository to get the files for this lab.

Once the `git` repository has been copied over, you should change into the code directory for this lab. Assuming you named your copy of it `15150`, that means you should do

```
cd 15150/lab/01/code/
```

The whole process of checking out the `git` repository should look like the following transcript:

```
hqbovik@unix14 hqbovik % ls
oldfiles  private  public  www
hqbovik@unix14 hqbovik % cd private
hqbovik@unix14 private % ls
hqbovik@unix14 private % git clone /afs/andrew.cmu.edu/course/15/150/handout 15150
Initialized empty Git repository in
/afs/andrew.cmu.edu/usr12/hqbovik/private/15150/.git/
hqbovik@unix14 private % ls
15150
hqbovik@unix14 private % cd 15150/
hqbovik@unix14 15150 % ls
hw  lab  src
hqbovik@unix14 15150 % cd lab/01/code
hqbovik@unix14 code % ls
lab01.sml
hqbovik@unix14 code % smlnj
Standard ML of New Jersey v110.76 [built: Mon Sep  9 21:09:47 2013]
-
```

2 Expressions

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

Task 2.1 Enter the text

```
2 + 2;
```

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

The output line, `val it = 4 : int`, indicates the type and the result of evaluating the expression. `it` is used as a default name for the value if a name is not provided by you, `4` is the value or result, and `int` is the type of the expression - in this case, meaning 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 was 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 Enter the text

```
2 + 2
```

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

After doing that, enter 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 an arithmetic class long ago, you probably learned some standard rules of operator precedence – for example, multiply before you add, but anything grouped in parentheses gets evaluated first. SML follows the exact same rules of precedence. You can, of course, insert parentheses into expressions to force a particular order of evaluation.

Task 2.3 Enter the text

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

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

Now, enter

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

into the REPL. Is the result the same? Why?

3 Evaluation

As you were determining how your expressions evaluated, you may have gone step-by-step, evaluating each of the arithmetic operations one at a time. As it turns out, this is how we can determine the runtime of an expression. For example, `(1 + 1) + 1` steps to `2 + 1`, which steps to `3`, which is a value, at which point evaluation stops. In this case, then, evaluation takes two steps to complete. For our purposes, we assume that all arithmetic operations take exactly one step of computation and we assume that numbers take zero steps. Also remember that arithmetic operators like `+` and * evaluate from left to right. We introduce the notation `e` $\Longrightarrow^1$ `e'` to mean that evaluation of `e` steps to `e'` in a single step, and `e` $\Longrightarrow$ `e'` for any finite number of steps. So for example,

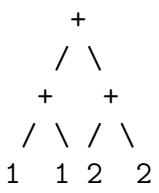
`(1+1)+1` $\Longrightarrow^1$ `2+1` $\Longrightarrow^1$ `3`
`(1+1)+1` $\Longrightarrow$ `3`

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

3.1 Computation Trees

An important property of additions is that the final result of a bunch of additions does not depend on the order in which the adds were performed. Suppose we have some expression `(...) + (...)`, where each parenthesized sub-expression is built from additions and numerals. Instead of arbitrarily choosing a side to evaluate first, in a parallel setting it is possible to evaluate both sides at once, and the result will always come out the same as if we had evaluated sequentially.

It is useful to draw an expression as a *computation tree* (later, we will see that these expression trees are a special case of what is called a *cost graph*). 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.

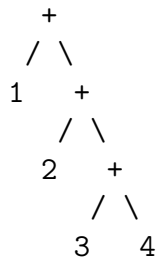
Task 3.2 Draw the tree for `(1 + 2) * (3 + (4 * 5))`.

3.2 Work and Span

Given an expression, we can determine its *work*, which is the total number of operations needed to evaluate the expression, and its *span*, which is the length of the longest *critical path* in the computation tree; a critical path is a sequence of operations that each depend on the results of the previous one. The work is the number of steps it takes to evaluate the expression sequentially, on a machine with only one processor. The span is the best possible number of steps for parallel evaluation, assuming enough processors—if there are not enough processors, the cost would be somewhere between the work and the span. We will do a bunch of work and span analysis this semester.

Once we have written the expression as a computation tree, the work is the *size* (number of non-leaf nodes) of the tree, and the span is the *depth* (length of the longest path) of the tree.

For example, the expression `(1 + (2 + (3 + 4)))` , with computation tree



has work 3, because there are three additions that need to be made. It also has a span of 3, since the result of the outermost add cannot be done until the result of the other two adds has been found, and the result of the middle add needs the result of the innermost add.

On the other hand the expression `(1 + 2) + (3 + 4)` has work 3 but span 2.

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

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 Enter the text

```
"foo";
```

into the REPL. What is the result?

Instead of seeing a number as the output, you see a string here. It is possible to concatenate two strings, using the infix `^` operator. This can be used just like `+` is used on integers.

Task 4.2 Enter the text

```
"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 enter 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, at least until you get used to types!

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 Enter

```
2 + 2;
```

into the REPL. Then, enter

```
it * 5;
```

into the REPL. What is the result?

As you see, before the second evaluation the value bound to `it` was 4 (the value of `2+2`), and now `it` is bound to 20, the result of the most recent expression evaluation (the value of `it * 5` with `it` bound to `4`).

Of course, you shouldn't get into the habit of using `it` like this! The SML runtime system only uses it (i.e., `it`) as a convenient default and a way to help you debug code. Usually you will want to choose a more mnemonic name by which to refer to a value, and the SML syntax for *declarations* allows you to do this.

A simple form of declaration has the syntax

```
val <varname> = <exp>
```

This declaration binds the value of `<exp>` to `<varname>`. If we want to mention the desired type of this variable explicitly we may write

```
val <varname> : <type> = <exp>
```

We can use a `val` declaration at the prompt in the SML runtime system, or as part of a `let`-expression with syntax

```
let val <varname>=<exp1> in <exp2> end
```

The value of this `let`-expression is obtained by evaluating `<exp1>`, binding its result to the variable named `<varname>`, and using this binding while evaluating `<exp2>`. This binding is not available for use outside of `<exp2>`, and we say that the *scope* of the binding is this expression.

The SML declaration syntax is much more general than we have indicated here, but this gives us enough to work with for now and also introduces the key notions of scope and binding.

Task 5.2 Enter the declaration

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

into the REPL. What is the result? How does it differ from just typing `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 Enter

```
x;
```

into the REPL; what is the result?

Task 5.4 Now enter

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

into the REPL. What is the result?

Task 5.5 How about

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

What happens? Why?

Task 5.6 After that, enter

```
z * z;
```

into the REPL. What happens? Why?

Task 5.7 Variables in SML refer to values, but are not *assignable* like variables in imperative programming languages. Each time a variable is declared, SML creates a fresh variable and binds it to a value. This binding is available, unchanged, throughout the *scope* of the declaration that introduced it. 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. We'll talk about this more in lecture and subsequently.

Type 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

Now that we have written some basic SML expressions, we can take a look at something a little more interesting: getting input from files. We have provided the file `lab01.sml` for you in the git repo you cloned in task 1.1. under `lab/01/code/`

Task 6.1 In 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

```
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 Enter

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

into the REPL. What is the result?

7.2 Defining functions

We will generally require you to use a simple standard format for commenting function definitions: the SML code for a function definition should be preceded by comments that specify the function's type, a "requires"-condition, and an "ensures"-condition that describes how the function behaves when applied to arguments that satisfy the pre-condition. We may waive this requirement when the function is part of the SML basis, or has been specified earlier. When the function has no pre-conditions, we use the notation "`REQUIRES: true`" (i.e. all arguments of the correct type satisfy the pre-condition).

For example, here is a function from class, treated in this style.

```
(* sum : int list -> int   *)
(* REQUIRES:  true         *)
(* ENSURES:   sum(L) evaluates to the sum of the integers in L. *)

fun sum [] = 0
  | sum (x::L) = x + (sum L);
```

We read this specification as saying that: *For all values* `L : int list`, `sum(L)` *evaluates to the sum of the integers in* `L`.

Task 7.2 In `lab01.sml`, complete the following template for an SML function `mult` that multiplies the integers in a list. Choose a sensible base case for the empty list.

```
(* mult : int list -> int    *)
(* REQUIRES:  true           *)
(* ENSURES:   mult(L) evaluates to the product of the integers in L. *)

fun mult [] =     (* FILL IN *)
  | mult (x::L) = (* FILL IN *);
```

Task 7.3 Complete the specification for the following function:

```
(* mult' : int list * int -> int *)
(* REQUIRES: true              *)
(* ENSURES: mult'(L, a) . . . (* FILL IN *)     *)

fun mult' ([], a) = a
  | mult' (x::L, a) = mult' (L, x*a);
```

Task 7.4 Using `mult`, define an SML function

```
Mult : int list list -> int
```

such that for all lists of integer lists `R`

```
Mult(R)
```

evaluates to the product of all the integers in the lists of `R`.

You can do this by filling in the following template:

```
fun Mult [] =

  | Mult (r::R) =
```

Task 7.5 Using `mult'`, define an SML function

```
Mult' : int list list * int -> int
```

such that for all lists of integer lists `R` and all integers `a`,

```
Mult' (R, a)
```

evaluates to the product of `a` with the product of all the integers in the lists of `R`.

You can do this by filling in the following template:

```
fun Mult' ([], a) =

  | Mult' (r::R, a) =
```