15-150 Fall 2011
Lab 1

August 31, 2011

Welcome to 15-150's first lab! Each Wednesday, we will give you some problems to work on with the assistance of the TAs. Since you are just getting started, this week will cover some of the basics. Think of this lab as a first date, where you and SML are just getting to know each other a little.

1 Getting Started

If you are logged in to a cluster machine running Linux or OS X, or ssh'd into one of the Unix machines, we have provided a macro called `smlnj` that you can use in

```
/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. First, determine which shell you are running with the command `echo ${SHELL}$`. If the result is `/bin/bash` (or something ending in `bash`), add the line

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

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

If the result 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`.

Now that you have done this, you can run SML with the command `smlnj`. 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` to get to the repl. 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 OS X. 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:

```
[zsparks@unix13:~]$ rlwrap sml
Standard ML of New Jersey v110.69 [built: Wed Apr 29 12:25:34 2009]
-
```

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.

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

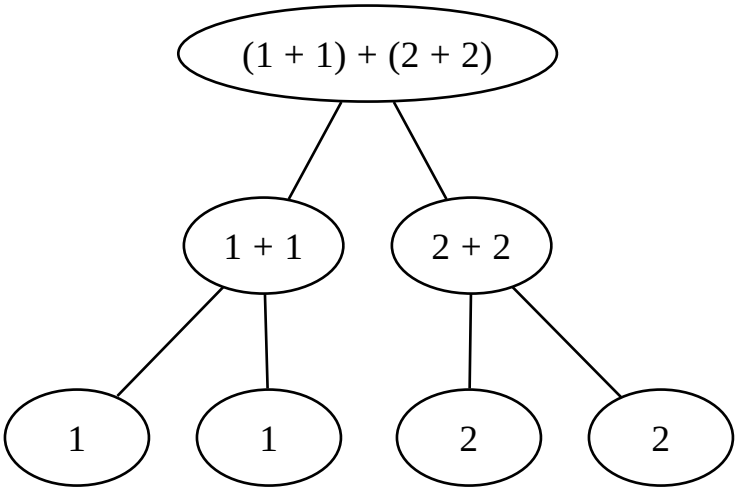
As you were determining how your expressions evaluated, you may have gone step-by-step, evaluating each of the arithmetic operations one at a time. As it turns out, this is how we can determine the runtime of an expression. For example, `(1 + 1) + 1` steps to `2 + 1`, which steps to `3`, which is a value, at which point evaluation stops. In this case, then, evaluation takes two steps to complete. For our purposes, we 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)`, writing out each intermediate step.

3.1 Expression Trees

Another important property of these additions is that there are no data dependencies. 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!

Instead of just listing the intermediate values, then, it is possible to draw a tree that represents the cost graph of the evaluation. 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.

3.2 Work and Span

Given an expression, we can determine its *work*, which is the total number of operations that need to be done to evaluate the expression, and its *span*, which is the length of the longest *critical path*—a sequence of operations that each depend on the results of the previous one. For example, the value of `(1 + 1) + (2 + 2)` depends on the value of `2 + 2`.

Once we have written an expression as a 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.

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. This is the same as the cost of evaluating the expression sequentially, on a machine with only one processor.

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

Span Span, on the other hand, is the length of the longest path from the root to a leaf. This is also the optimal cost of parallel computation, assuming enough processors. For example, the depth of `(1 + 1) + (2 + 2)` is 2.

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

Keep in mind that this is the *optimal* cost - if there are not enough processors, the cost could be somewhere between the work and the depth. We will do a bunch of work and span analysis this semester.

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 overrides the previous 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

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 `lab1.sml` for you; please copy it from the lab AFS directory by running the following command:

```
cp /afs/andrew.cmu.edu/course/15/150/asgn/lab/01/lab1.sml lab1.sml
```

To load it into the REPL, type `use "lab1.sml";`. The output from SML should look like

```
- use "lab1.sml";
[opening lab1.sml]
...
val it = () : unit
-
```

Now that you have done this, you have access to everything that was defined in `lab1.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?

7.2 Defining functions

Now, recall Tuesday's lecture, in which we discussed ways to compute the number of students who had taken 15-122 in parallel, where we compute each row individually. We represented this with a data structure called a *sequence*, which we will discuss in more detail later in the course. It is somewhat straightforward to construct a sequence, though, using the function `cons`. For example, if we represent the class as two rows of three students each:

```
row1 : yes yes no
row2 : no  yes no
```

the corresponding values are

```
val row1 : row  = cons (1 , cons (1 , cons (0 , emptyRow)))
val row2 : row  = cons (0 , cons (1 , cons (0 , emptyRow)))
val classroom : students = cons (row1 , cons (row2 , emptyRoom))
```

Note that we do need parentheses around the arguments to `cons` in this case, since it takes multiple arguments - an element, and a sequence to add it to. `cons (x,S)` returns the result of putting the element `x` onto the beginning of sequence `S`.

Task 7.2 Write expressions `row1`, `row2`, `row3`, and `classroom` to represent the class

```
row1 : yes no  yes yes
row2 : no  no  no  yes
row3 : yes yes no  no
```

Task 7.3 Write expressions `row4`, `row5`, `row6`, and `classroom2` to represent the class

```
row4 : yes yes yes yes
row5 : yes yes yes yes
row6 : yes yes yes yes
```

Also note the function `count` in `lab1.sml`. This determines the number of students in a class who have taken 15-122 already by first counting the number of students in each row (potentially in parallel), then summing up the results of the rows. There is also a function

```
(* returns 1 if everyone in the row has taken 122, returns 0 otherwise *)
val allInRow : row -> int
```

Task 7.4 Using `allInRow`, write a function `allInClassroom` that returns `1` if everyone in the class has taken 15-122 and `0` otherwise. You should copy the code for `count` and only replace what needs to be replaced. The code for `count` can be found in the file `lab1.sml` that you copied into your directory, which you can open in any text editor. This code is also discussed in the Lecture 1 notes, which are available from the web page.

Test your function by evaluating `allInClassroom classroom` (which should return `0`) and `allInClassroom classroom2` (which should return `1`).