15-150 Fall 2014
Lab 01

Thursday 28th August, 2014

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.

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, because credit is based on participation. If you do not manage to complete it, we strongly encourage to do so on your own time.

1 Getting Started

1.1 Running SML

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`. Otherwise, you can access the unix machines from any machine with an internet connection by ssh'ing to `unix.qatar.cmu.edu`.

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

```
[iliano@linux13 ~]$ sml
Standard ML of New Jersey v110.74 [built: Mon Aug 13 21:05:32 2012]
-
```

This is the SML prompt (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 SML/NJ, type `Control-d`.

1.2 Setting up Emacs/Vim

SML is best written in a text editor. There are lots of editors that support SML (and many other programming languages). Here we mention two of them, Emacs and Vim, but feel free to use any editor you want (as long as it gives you syntax highlighting and a way to run code directly from inside the editor). Examples include Notepad++ (Windows), TexMate (Mac), Sublime (any OS), and even Eclipse (this may be a bit heavy weight).

Emacs contains an excellent mode specifically for editing SML. To install Emacs sml-mode, see Section 1.2 of

http://www.smlnj.org/doc/Emacs/sml-mode.html

This will make Emacs open all files ending in `.sml` in sml-mode, giving you syntax highlighting and indentation support.

To start SML as a subprocess of Emacs, enter the command

```
M-x run-sml
```

This will load the SML/NJ prompt as a buffer in Emacs which you can then interact with in the same way would interact with the prompt when running it stand-alone. To load the current buffer into SML, enter the command

```
C-c C-b
```

(that's `Control-c Control-d`).

If you have experience using Vim and prefer that over Emacs feel free to continue using it. If you have not done so already, you should add some settings to your `.vimrc` file for things like smart tab indentation and parenthesis matching. There is plenty of information on the web about how to set up Vim.

1.3 Getting Labs

We will be distributing the text and any starter code for the labs using Autolab. Each week's lab will start being available at the beginning of class.

On the Autolab page for the course, the current lab is the last entry under `Lab`. Say it is called "`labnn`". Click on this link. There, two links matter

- View writeup: this is the text of the lab in PDF format.
- Download handout: this is the starter code, if any, for the lab. It will always be distributed as a compressed archive (in `.tgz` format). Uncompressing it will create the following directories:

`lab`*nn*`/`	Directory for lab *nn*
`code/`	Code directory for lab *nn*
`*.sml`	Starter files for lab *nn*
`handout.pdf`	Copy of writeup for lab *nn*

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;
```

at the SML prompt 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 SML interpreter does not know to evaluate the expression and expects more input.

Task 2.2 Enter the text

```
2 + 2
```

(no semicolon) into at the SML prompt 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;
```

at the prompt. What would you expect the result to be? What is the actual result?

Now, enter

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

at the prompt. 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 numbers take zero steps. Also remember that arithmetic operators like `+` and * evaluate from left to right. We introduce the notation `e` $\mapsto$ `e'` to mean that evaluation of `e` steps to `e'` in a single step, and `e` $\mapsto^*$ `e'` for a finite number of steps. So for example,

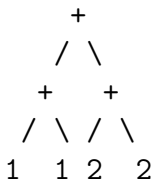
`(1+1)+1` $\mapsto$ `2+1` $\mapsto$ `3`
`(1+1)+1` $\mapsto^*$ `3`

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 this:



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.

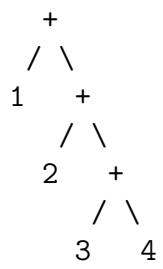
Task 3.2 Draw the tree for `(1 + 2) * (3 + (4 * 5))`.

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";
```

at the SML prompt. 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";
```

at the SML prompt. What is the result

We can write a program that is not well–typed to see what SML does in that situation.

Task 4.3 What happens when you enter the expression

```
3 ^ 7;
```

at the SML prompt?

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 Functions

5.1 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 5.1 Enter

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

at the SML prompt. What is the result?

5.2 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. You may omit the requires-condition when it there is none (i.e., it requires `true`).

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

```
(* factorial n ==> m
   REQUIRES: n >= 0
   ENSURES: m is the product of all integers between 1 and n
 *)
fun factorial (0: int): int = 1
  |  factorial n = n * (n-1);
```

We read this specification as saying that:

> *For all values* `n : int` *such that* $n \geq 0$, `factorial(n)` *evaluates to the product of the integers in from 1 to* n.

Task 5.2 Complete the following template for an SML function `summorial` that calculates the sum of all integers from 0 to n.


```
(* summorial n ==> m
   REQUIRES: n >= 0
   ENSURES: m is the sum of all integers between 0 and n
 *)
fun summorial (0: int): int = raise Fail "Fill me in"
  | summorial n = raise Fail "Fill me in"
```

Task 5.3 Complete the specification for the following function:

```
(* summorial' (n,a) ==> m
   REQUIRES: n >= 0
   ENSURES: (* FILL ME IN *)
 *)
fun summorial' (0: int, a: int): int = a
  | summorial' (n, a) = summorial' ((n-1), n+a)
```

Checkout point!

Completing everything up to here in the lab assignment will guarantee credit for this lab.

Click here or go to the class schedule and click on a `Check me in` button.

6 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 6.1 Enter

```
2 + 2;
```

at the SML prompt. Then, enter

```
it * 2;
```

at the SML prompt. 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 8, the result of the most recent expression evaluation (the value of `it * 2` 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 can write

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

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 6.2 Enter the declaration

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

at the SML prompt. 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 6.3 Enter

```
x;
```

at the SML prompt; what is the result?

Task 6.4 Now enter

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

at the SML prompt. What is the result?

Task 6.5 How about

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

What happens? Why?

Task 6.6 After that, enter

```
z * z;
```

at the SML prompt. What happens? Why?

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.

Task 6.7 Type the following at the SML prompt:

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

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

7 Using Files

Now that we have written some basic SML expressions, we can take a look at something a little more interesting: loading a program 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 7.1 At the SML prompt, 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 SMl/NJ.

