15-150 Spring 2012
Lab 2

25 January 2012

The goal for the second lab is to make you more comfortable writing functions in SML using the methodology discussed in the last couple of lectures, and doing inductive proofs. Please take advantage of this opportunity to practice with the assistance of the TAs and your classmates: You are encouraged to collaborate with your classmates and to ask the TAs for help!

1 Introduction

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Setting up Emacs/Vim

SML is best written in a text editor. Emacs and Vim are the two clear choices for text editors in a modern UNIX environment. Emacs specifically contains an excellent mode specifically for editing SML. To install Emacs sml-mode on your Andrew Emacs setup, simply run

```
emacs_setup
```

from a terminal at your cluster machine or by SSH-ing into one of the Andrew UNIX time-share servers. 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 REPL as a buffer in emacs which you can then interact with in the same way would interact with the REPL when running it stand-alone. To load the current buffer into SML, enter the command

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

More extensive documentation on emacs sml-mode can be found at

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

If 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 are various useful links about setting up and using Vim on the course website, and Google is always your friend as well. If you use Vim, or are uncomfortable using multiple processes inside emacs, you should open two terminals or SSH sessions so you can be editing your file in Vim in one and interacting with the SML REPL in the other. This will save a tremendous amount of time and effort.

As always, please ask your TAs or those around you for help if you'd like it.

1.3 Methodology

Recall from lecture the five step methodology for writing functions:

1. Write the type of the function. For example, the first step we took in writing the function `double` was to write the declaration:

   ```
   fun double (n :  int) :  int = ...
   ```

   which specifies that `double` has type `int -> int`.

2. Write the purpose or specification of the function. This should appear in a comment above the body of the function.

3. Write a few examples of how the function should transform a value of the argument type into a value of the result type. This should also appear in a comment above the body of the function.

4. Write the body of the function. The purpose and the examples will help you with this step. The body will often follow a pattern of recursion like the ones given later in this lab.

5. Finally, test the function. The examples can be turned into tests. We can write these tests using pattern matching since constants are patterns in SML.

Putting these five steps together results in code that looks like this:

```
(* Purpose: double the number n
 * Examples:
 *  double 0 ==> 0
 *  double 2 ==> 4
```

```
 *)
fun double (n : int) : int = <... body of double ...>

(* Tests for double *)
val 0 = double 0
val 4 = double 2
```

Make sure to use this five step methodology for all of the functions you write.

2 Recitation Time: `let`, Pairs, and Recursion

The TAs will introduce a few new things: the `let` construct, pairs, and alternate recursion schemes. See the end of the Lecture 3 notes to review this material.

3 Recursion on the Natural Numbers

We will write several recursive functions over the natural numbers.

3.1 Structural Recursion

The bodies of the first two of these functions will follow the basic pattern of *structural recursion* that we discussed in lecture. To review: They will consist of a `case` statement on the argument that has two branches. The first branch will specify the base case when the argument is zero. The second branch will specify the induction case when the argument is greater than zero. The induction case will include a recursive application of the function to an argument that is one less. So the definitions of the first two functions will match the pattern:

```
fun f (x : int) : int =
  case x of
    0 => (* base case *)
  | _ => ... f (x - 1) ...
```

with the base case and ellipses filled in appropriately based on the purpose of the function.

Summorial We begin by writing a recursive function that takes a natural number, `n`, and calculates the sum of the numbers from `0` to `n`:

$$\texttt{summorial n} \texttt{==>} 0 + 1 + 2 + \ldots + n$$

Task 3.1 Define the `summ` function such that `summ n` equals the sum of the natural numbers from `0` to `n`. Remember to follow the steps of the methodology. What should the type of

`summ` be? Write a purpose for `summ` and a few examples. Write the body of the `summ` function and as you write it, attempt to justify its correctness to yourself. After you write the body of the function, write a few tests based on your examples.

Squaring We will now write a recursive function with a more complicated induction case.

Task 3.2 Define the `square` function such that `square n` returns the product of `n` with itself. Remember to follow the five step methodology. Do not use integer multiplication in the body of `square`. *Hint:* You may apply the `double` function in the body of `square`, and use the following identity:

$$n^2 = (n-1)^2 + 2n - 1$$

Have the TAs check your work before proceeding!

3.2 More Advanced Patterns of Recursion

Odd Recall the `evenP` function of type `int -> bool` that transforms a natural number, `n`, into `true` if and only if it is even (the definition of `evenP` is given in `lab2.sml`). This definition uses a different pattern of recursion:

To define a function on all natural numbers, it suffices to give cases for

- `0`
- `1`
- `2 +` n, using a recursive call on n

Therefore, the `case` statement in the body of `evenP` has three branches rather than two. The first two branches give the base cases, and the third branch includes a recursive application of the function to the natural number that is two less than the argument:

```
fun g (x : int) : bool =
 case x of
   0 => (* base case 0 *)
 | 1 => (* base case 1 *)
 | _ => ... g (x - 2) ...
```

with the base cases and ellipses filled in appropriately based on the purpose of the function.

We will now define the `oddP` function using this pattern.

Task 3.3 Define the `oddP` function of type `int -> bool` that transforms a natural number `n` into `true` if and only if it is odd. Once again, remember to follow the five step methodology. Do not call `evenP` in the definition of `oddP`. *Hint:* How do the base cases of `evenP` and `oddP` differ?

Divisible by Three Next, you will define a function `divisibleByThree : int -> bool` such that `divisibleByThree n` evaluates to `true` if n is a multiple of 3 and to `false` otherwise. Do not use the SML `mod` operator for this task.

Task 3.4 Define this function, following the five-step methodology. *Hint:* You will need a new pattern of recursion to define this function. Explain the pattern here:

To define a function on all natural numbers, it suffices to give cases for

Have the TAs check your work before proceeding!

4 Two-argument Functions

Suppose we didn't have `+` built in; how could we define it?

So far we have only defined functions with argument type `int`. In this problem, you will define a *two-argument function*

```
fun add (x : int, y : int) : int = ...
```

that computes the sum of `x` and `y`. *Hint:* The body of the `add` function should start with a `case` statement on `x`. The base case will give the sum of `0` and `y`, and the induction case will use the sum of `x-1` and `y` to compute the sum of `x` and `y`. That is, `add` should follow the pattern of *structural recursion on* `x`.

Task 4.1 Define the `add` function that computes the sum of a pair of natural numbers. You may use SML addition and subtraction of `int` constants in the definition of `add` (e.g. `+ 1` and `- 1`), but you may not add two variables. Remember to follow the five step methodology.

5 Induction

Ask a TA to check your definition of `add` before doing this task!

If you defined `add` correctly, then it is a simple calculation to see that for any n, `add (0,n)` $\cong$ `n`.

However, to show that for all natural numbers `m`, `add (m,0)` $\cong$ `m`. requires an inductive proof. In this proof, we do induction on only the first argument m, leaving the second argument alone.

Task 5.1 Fill in the proof on the following page.

Theorem 1. *For all natural numbers m,* `add (m,0)` $\cong$ `m`.

The proof is by induction on m.

- **Case for** 0

  To show:

  Proof:

- **Case for** $1 + k$

  Inductive hypothesis:

  To show:

  Proof: