15-150 Spring 2016
Lab 2

18 May 2016

This lab will give you practice writing code and writing proofs. Enjoy!

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 in particular 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 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 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

You must use the five step methodology discussed in class for writing functions, for **every** function you write this semester. This is the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify in a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify in an `ENSURES` clause what the function computes (what it returns) when applied to arguments satisfying the REQUIRES clause.
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.
   You have enough test cases when you have throughly tested the different possible behaviors of the function. Often this involves a test for each base case and each recursive case.

For example, for the factorial function, the following would receive full credit :

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
 *)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 Recursion on the Natural Numbers

We will write several recursive functions over the natural numbers.

2.1 Basic Recursion

The bodies of the first two of these functions will follow the basic pattern of *recursion* that we discussed in lecture. 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) ...
```

when using a case statement or

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

when using clauses 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`:

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

Task 2.1 Define the `summ` function such that for $\texttt{n} \geq 0$, `summ n` equals the sum of the natural numbers from `0` to `n`. What should the type of `summ` be? Write the body of the `summ` function and as you write it, attempt to justify its correctness to yourself. For practice, try writing two versions of this function, one using a case statment and the other using clauses. 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 2.2 Define the `square` function such that for all $\texttt{n} \geq 0$ `square n` returns the product of `n` with itself. Do not use integer multiplication in the body of `square`. Again, try writing

the function both ways.
Hint: You may apply the `double` function in the body of `square`, and use the following identity:

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

Divisible by Three Next, you will define a function `divisibleByThree : int -> bool` such that for `n` ≥ 0 `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 2.3 Define this function. *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!

3 Another Pattern 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 `lab02.sml`). This definition uses a different pattern of recursion:

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

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

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 evenP (x : int) : bool =
 case x of
   0 => (* base case 0 *)
 | 1 => (* base case 1 *)
 | _ => ... evenP (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 3.1 Define the `oddP` function of type `int -> bool` that transforms a natural number `n` into `true` if and only if it is odd. Do not call `evenP` or `mod` in the definition of `oddP`.

4 GCD

Below is a `GCD` function for computing the greatest common divisor of two non-negative numbers. The g.c.d. of `m` and `n` is the largest integer that divides both `m` and `n`, with zero remainder.

```
(* GCD: int * int -> int *)
(* REQUIRES m, n >= 0 *)
(* ENSURES GCD (m, n) evaluates to the g.c.d. of m and n *)

fun GCD (m : int, 0) : int = m
  | GCD (0, n : int) : int = n
  | GCD (m : int, n : int) : int =
        case m > n of
            true => GCD(m mod n, n)
          | false => GCD(m, n mod m)
```

Task 4.1 Let's prove that the code meets the specification, using complete induction on the product `mn`.
You may use the following lemmas in your proof as facts, but be sure to cite them where you do.

Lemma 1: `m mod n = m - (m div n) * n` for all natural numbers `m` and `n`, `n` $\neq$ `0`.

Lemma 2: The g.c.d. of m and n is the same as the g.c.d of (m mod n) and n, when $m > n$

Theorem:

Proof: By _____ (method) on _____ (expression)

Base Case:

NTS (Need to Show):

Inductive Step:

IH (Inductive Hypothesis):

NTS (Need to Show):

Task 4.2 Why can't you use simple induction to prove GCD correct?

Task 4.3 What happens when you evaluate `GCD(~1, 0)`?

Have the TAs check your work before proceeding!

5 Stein's Algorithm (Bonus)

Stein's algorithm for g.c.d. checks the *parity* (even or odd) of `m` and `n`, and takes out a common factor of 2 when possible. Here is the spec, and an implementation that uses a plethora of case statements.

```
(* stein : int * int -> int *)
(* REQUIRES: m, n >= 0      *)
(* ENSURES: stein(m,n) returns the g.c.d. of m and n. *)
fun stein(m,n)=
  case (m=0) of
       true => n
     | false =>
         (case (n = 0) of
              true => m
            | false =>
                (case m = n of
                     true => m
                   | false =>
                       (case (m mod 2 = 0) of
                            true => (case (n mod 2 = 0) of
                                         true => 2 * stein(m div 2, n div 2)
                                       | false => stein(m div 2, n))
                          | false => (case (n mod 2 = 0) of
                                           true => stein(m, n div 2)
                                         | false =>
                                             (case (n>m) of
                                                  true => stein(m, (n-m) div 2)
                                                | false => stein((m-n) div 2,n))))))
```

Task 5.1

(i) Here we used a plethora of `case`'s, but this is bad style. Write the `stein'` function in the `lab02.sml` file using fewer case statements.

(ii) What happens when you evaluate `stein(~1, 0)`? Is `stein` extensionally equivalent to `GCD`?

(iii) What happens if we try using pattern matching, for example:

```
fun stein (m, m) = m
  | stein (m, n) = ...
```