15-150 Fall 2013
Homework 02

Out: Wednesday, 4 September 2013
Due: Tuesday, 10 September 2013 at 23:59 EST

1 Introduction

In this assignment, you will go over some of the basic concepts we want you to learn in this course, including defining recursive functions and proving their correctness. We expect you to follow the methodology for defining a function, as shown in class.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/02` directory should contain a file named exactly `hw02.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/02` directory (that contains a `code` folder and a file `hw02.pdf`). This should produce a file `hw02.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw02.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your PDF is valid; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

Your `hw02.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 10 September 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall 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 via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function:

```
(*  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 Basics

The built-in function

```
    real : int -> real
```

returns the `real` value corresponding to a given `int` input; for example, `real 1` evaluates to `1.0`. Conversely, the built-in function

```
    trunc : real -> int
```

returns the integral part (intuitively, the digits before the decimal point) of its input; for example, `trunc 3.9` evaluates to `3`. Feel free to try these functions out in `smlnj`.

Once you understand these functions, you should solve the questions in this section in your head, *without* first trying them out in `smlnj`. The type of mental reasoning involved in answering these questions should become second nature.

2.1 Scope

Task 2.1 (4%). Consider the following code fragment:

```
fun double (x : real) : real = x + x
fun double (y : int) : int = y + y
fun doubleplus (z : real) : real = double (z + 1.0)
```

Does this typecheck? Briefly explain why or why not.

Solution 2.1 It does not typecheck. In the declaration for `doubleplus`, the function call on `double` is bound by `fun double (y : int) : int = y + y` on the previous line. Thus, `double` is of type `int -> int` and, because `z + 1.0` is a real, `double` cannot accept the argument `z + 1.0`.

A more formal way of saying this would be that the declaration of `double` on the second line ”shadows” the declaration of `double` on the first line, which effectively hides (or ”overwrites”) the first declaration.

2.2 Pattern-Matching

Task 2.2 (6%). For each of the following patterns, state what kinds of values of type `int list` can be matched to it successfully. For example, the pattern `[ ]` only matches the empty list and the pattern `[true]` matches no values of type `int list`.

(i) Pattern `x::L`

(ii) Pattern `_::_`

(iii) Pattern `x::(y::L)`

(iv) Pattern `(x::y)::L`

(v) Pattern `[x, y]`

Solution 2.2

(i) Non-empty int lists.

(ii) Non-empty int lists.

(iii) Non-empty int lists of size 2 or greater.

(iv) No values of type `int list` satisfy this pattern. Note that the pattern `(x::y)` must follow that of a list, so the entire pattern `(x::y)::L` is a list of lists. A list of lists cannot be of type `int list`, so no values of type `int list` will match this pattern.

(v) A list of exactly 2 elements.

Task 2.3 (5%). For each of the following sets of values, give a single pattern that would match only with the values in the set. If such a pattern cannot be made, explain why.

(i) Lists of length 3

(ii) Lists of length 2 or 3

(iii) Non-empty lists of pairs

(iv) Pairs with both components being non-empty lists

Solution 2.3

(i) `[x, y, z]`

(ii) Not possible, because we cannot pattern-match on two specifically different types of values.

(iii) `(x, y) :: L`

(iv) `(x :: xs), (y :: ys)`

Let Bindings In Lecture 2, we went over SML's syntax for let-bindings. It is possible to write `val` declarations in the middle of other expressions with the syntax `let ... in ... end`.

Task 2.4 (8%). Consider the following code fragment (the line-numbers are for reference; they are not part of the code itself):

```
 (1)  val x : int = 3
 (2)  val temp : int = x + 1
 (3)  fun assemble (x : int, y : real) : int =
 (4)    let val g : real = let val x : int = 2
 (5)                           val m : real = 6.2 * (real x)
 (6)                           val x : int = 9001
 (7)                           val y : real = m * y
 (8)                       in y - m
 (9)                       end
(10)    in
(11)      x + (trunc g)
(12)    end
(13)
(14)  val z = assemble (x, 3.0)
```

Note that in the declaration of `temp` in line (2), the binding ⟦ `x:3` ⟧ is introduced for the variable `x` in that line. The value of the binding is of type `int`.

(a) What binding (and type) does the declaration in line (4) introduce for the variable `x`? Briefly explain why.

(b) What binding (and type) does the declaration in line (5) introduce for the variable `m`? Briefly explain why.

(c) What binding (and type) does the declaration in line (6) introduce for the variable `x`? Briefly explain why.

(d) What value does the expression `assemble (x, 3.0)` evaluate to in line (14)?

Solution 2.4

(a) The binding introduced is ⟦ `x:2` ⟧ of type `int`, because the nearest enclosing binding is `val x : int = 2` on line (4).

(b) The binding introduced is ⟦ `m:12.4` ⟧ of type `real`, because the nearest declaration for `m` is on line (5). Note that `x` = `2` in the evaluation for `m`.

(c) `x` is bound to `9001:int`, as the nearest binding is `val x : int = 2` on line (6).

(d) `27` of type `int`.

2.3 Evaluational and Equational Reasoning

It is important to understand the difference between expression evaluation using $\Rightarrow^*$and extensional equivalence (equational reasoning) using $=$. To test your understanding of each of these symbols and their definitions and properties, we ask you in this section to determine whether the following statements are extensionally equivalent and whether one expression evaluates to the next.

Task 2.5 (8%). For the following statements, state whether it is true or false (make sure to note which symbol is being used). Next, give enough detail to show that you understand what the notation means. This will involve using equational laws and equational reasoning, or evaluation rules and evaluational reasoning, as appropriate. Laws, rules, and examples of such reasoning were shown in class. You can consult the lecture notes for more examples.

(a) `(fn x:int => x+(2-1))` $=$ `(fn x:int => x+1)`

(b) `(fn x:int => x+(2-1))` $\Rightarrow^*$`(fn x:int => x+1)`

Solution 2.5

(a) True. To prove that this statement is true, we just need to show that for every input to both functions on each side, the same output results. This is fairly simple; we just note that `2-1 = 1` by basic arithmetic and apply it to the function on the left to get the function on the right. Formally, we would write this out explicitly.

(b) False. In SML, this function does not evaluate in any number of steps to the value on the right side. The function on the left side will not be simplified any further, because, in SML, lambda functions always stay the same when created until it is supplied with an argument (after which it will be evaluated).

Now look carefully at the following function, `decimal`, as shown in lecture:

```
fun decimal (n:int) : int list =
    if n<10 then [n]
    else (n mod 10) :: decimal(n div 10)
```

Remember, the symbols $=$ and $\Rightarrow^*$for extensional equivalence and expression evaluation, respectively, have different meanings. With that in mind:

Task 2.6 (4%). Using equational reasoning (with the $=$ symbol), show the following:
`decimal (5 + 5)` $=$ `[0, 1]`

Solution 2.6

```
decimal (5 + 5) = decimal 10
(* Addition *)
                          = if 10<10 then [10]
                              else (10 mod 10) :: decimal(10 div 10)
(* Substitution into decimal *)
                          = 0 :: decimal 1
(* 10<10 is False *)
                          = 0 :: (if 1<10 then [1]
                                     else (1 mod 10) :: decimal(1 div 10))
(* Substitution into decimal *)
                          = 0 :: [1]
(* 1<10 is True *)
                          = [0, 1]
(* Definition of Cons on Lists *)
```

Do **not** use the $\Rightarrow^*$notation for evaluation. You must use equational laws about function application, substitution, and conditional expressions as shown in lecture. Give enough detail to show that you understand how to use these equational laws, making sure to only use an equation when it is applicable to the problem at hand. For example, the substitution law for function application requires that the argument expression is a value.

Task 2.7 (4%). Using evaluational reasoning (with the $\Rightarrow^*$symbol), show the following:

`decimal (5 + 5)` $\Rightarrow^*$`[0, 1]`

Do **not** use the equational reasoning that you used in the previous task. You must use evaluational facts as shown in lecture. Give enough detail to show that you understand how SML evaluates expressions, and make sure to clearly explain the order in which sub-expressions get evaluated.

Solution 2.7 Note that the square brackets refer to variable bindings.

```
decimal (5 + 5) ⇒* decimal 10
                ⇒* [n:10] (fn n => if n<10 then [n]
                     else (n mod 10) ::  decimal(n div 10))
                ⇒* if 10<10 then [10]
                     else (10 mod 10) ::  decimal(10 div 10)
                ⇒* (10 mod 10) ::  decimal (10 div 10)
                ⇒* 0 ::  decimal (10 div 10)
                ⇒* 0 ::  decimal 1
                ⇒* 0 ::  [n:1] (fn n => if n<10 then [n]
                     else (n mod 10) ::  decimal(n div 10))
                ⇒* 0 ::  if 1<10 then [1]
                     else (1 mod 10) ::  decimal(1 div 10))
                ⇒* 0 ::  [1]
                ⇒* [0, 1]
```

3 Recursive Functions

3.1 Addition and Subtraction

The following function adds two natural numbers recursively by repeatedly adding 1:

```
(* add : int * int -> int
   REQUIRES:  x, y >= 0
   ENSURES:  add(x,y) = x+y
*)
fun add (0 : int, y : int) : int = y
  | add (x : int, y : int) : int = 1 + add(x-1, y)
```

(Recall that a *natural number* is a non-negative integer.)

In the next part, let us define a function `leq : int * int -> bool` that accepts a pair `(x, y)` of two natural numbers `x` and `y` and returns true if `x` $\leq$ `y` and false otherwise.

Task 3.1 (5%). In `hw02.sml`, write and document a recursive function

```
leq : int * int -> bool
```

that satisfies the specification above. Your implementation has a specific set of restrictions, namely that it:

1. **must** use recursion;
2. can only decrement values by 1;
3. must use only pattern-matching on integer values (no conditional expressions, no use of `<`, and no boolean case expressions!).

3.2 Going Halfway Forwards

A famous series in mathematics that can be used to represent one of Zeno's paradoxes is of the following form:

$$\sum_{i=1}^{\infty} \frac{1}{2^i} \quad = \quad \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$$

This series converges to the real value `1.0`, because the "partial sum" of the first n terms of this series gets closer and closer to the value `1.0` as n gets larger. We will be exploring the partial sum of the first n terms in this series, which we will denote as S_n, for $n \geq 0$:

$$S_n = \sum_{i=1}^{n} \frac{1}{2^i}$$

Recall that, by definition, $\sum_{i=k}^{n} f(i)$ is 0 if $n < k$.

Task 3.2 (4%). What are S_0, S_1, S_2, S_3?

$\boxed{\textbf{Solution 3.2}}$

$$S_0 = 0 \qquad S_1 = 0.5 \qquad S_2 = 0.75 \qquad S_3 = 0.875$$

As defined in the homework, $S_0 = \sum_{i=1}^{0} \frac{1}{2^i}$, the sum of an empty sequence, so it is 0.

Task 3.3 (8%). In `hw02.sml`, write and document a recursive function

```
    halfSum : int -> real
```

such that `halfSum n` calculates S_n for any natural number n. For this problem you may use any functions or operators you wish. In particular you may want to use the built-in function `real` discussed earlier in this assignment, although it is not crucial (there are other ways to write halfSum :). Please only use recursion; do not look for a closed form for S_n. Instead, look for a way to compute S_n recursively.

Note: When testing your code, it is somewhat fragile to compare real values for equality, because computations on `reals` are prone to rounding errors. For this reason, SML does not allow pattern-matching on values of type `real` (floating point numbers), so you cannot do a test like `val 0.5 = halfSum 1`. Instead, you need to use an explicit equality test, like the following:

```
val true = Real.==(halfSum 1, 0.5)
```

Methodologically, it is usually better to check that two real values differ by a small ϵ, rather than checking for exact equality with `Real.==`. However, `Real.==` should suffice for writing tests in this assignment. That said, if you encounter an unexpected failing test, it may be because `Real.==` does exact floating point comparison, and your calculation does not come out to exactly the value you anticipate.

3.2.1 Going Halfway Forwards and Back

A related sequence is the alternating version of this series, in which every other term has negative sign:

$$\sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{2^i} \quad = \quad \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \cdots$$

For any natural number n, we will call the partial sums of this alternating series I_n:

$$I_n = \sum_{i=1}^{n} \frac{(-1)^{i+1}}{2^i}$$

Like the previous series, this alternating series converges; as n approaches infinity, I_n approaches $\frac{1}{3}$.

Task 3.4 (4%). What are I_0, I_1, I_2, I_3?

Solution 3.4

$$I_0 = 0 \qquad I_1 = 0.5 \qquad I_2 = 0.25 \qquad I_3 = 0.375$$

Task 3.5 (8%). In `hw02.sml`, write and document a recursive function

```
altHalfSum : int -> real
```

such that `altHalfSum n` calculates I_n, according to the specifications above. Again, please use recursion. Hint: find a way to calculate I_n recursively. Don't look for a closed form for I_n, and don't try to use `halfSum` as a helper function!

3.3 Primality Test

You have probably heard about prime numbers in the past. Here, we will be utilizing recursion to determine if a natural number is prime. Recall the definition for prime numbers:

Theorem 1. *A natural number $n > 1$ is prime if and only if it is divisible by only itself and 1.*

Given the input number n, a simple test for primality is to go through all of the numbers from 2 to $n - 1$ and check if each divides into n. This can be done with recursion, using an extra argument that keeps track of which potential divisors we have already checked. We can test for divisibility using `mod`, because n is divisible by m if and only if n `mod` $m = 0$.

Of course, we only really need to check for divisibility by 2 through $\sqrt{n}$, but we will not penalize you if your code checks for divisibility by 2 through $n - 1$.

Task 3.6 (12%). Write an ML function

```
is_prime : int -> bool
```

in `hw02.sml` such that for all natural numbers `n`, `is_prime` returns true if `n` is prime and false otherwise.

Hint: Use a recursive helper function of a suitable type, as outlined above. Make sure you give proper documentation for any helper function(s) that you define, including type and specification.

4 Induction

4.1 All-Zero Lists

Recall the function `eval : int list -> int` as shown in lecture:

```
  fun eval [ ] = 0
   |  eval (x::L) = x + 10 * eval L
```

An integer list is called *all zeroes* if each of its members is 0. Trivially, the empty list has this property, and a non-empty list `x::R` has this property if and only if `x = 0` and `R` is *all zeros*.

Task 4.1 (10%). Prove that for all integer lists `L` such that `L` is *all zeros*, `eval L` $\Rightarrow^*$`0`. Make sure to use evaluational reasoning accurately, and state clearly what method of induction you use. You can find some helpful proof templates in `code/hw02-proof-templates.tex`. You should structure your proof using the appropriate template for the proof method you use.

$\boxed{\textbf{Solution 4.1}}$

Theorem: For all integer lists `L` such that `L` is *all zeroes*, `eval L` $\Rightarrow^*$`0`.

Proof. The proof is by structural induction on `L`.

Base Case: `L` $=$ []:
Need to show: `eval []` $\Rightarrow^*$ `0`.
Showing:

`eval []` $\Rightarrow^*$ `0`	First clause of `eval` is relevant since `L=[]`

Inductive Step: Prove for `L = x::R`:
Inductive Hypothesis: `eval R` $\Rightarrow^*$`0`.
Need to show: `eval (x::R)` $\Rightarrow^*$`0`
We note that since `L` is *all zeroes*, each element in `L` is `0`. Thus, `x` $=$ `0`, and we try to show that `eval 0::R` $\Rightarrow^*$`0`. The binding for $[\![$ `x:0` $]\!]$ will be shown for evaluation purposes.
Showing:

`eval (0::R)` $\Rightarrow^*$ $[\![$ `x:0` $]\!]$ `(x + 10 * eval R)`	By the 2nd clause of `eval`, as `L` is non-empty
$\Rightarrow^*$ `0 + 10 * eval R`	Substitution of `x = 0`
$\Rightarrow^*$ `0 + 10 * 0`	By IH
$\Rightarrow^*$ `0 + 0`	Multiplication
$\Rightarrow^*$ `0`	Addition

By the Base Case and Inductive Step, the claim is true.

□

4.2 Summation of Odd Numbers

Task 4.2 (10%). Look at the following function carefully.

```
fun sumOdd (0 : int) : int = 0
  | sumOdd (n : int) : int = (2*n - 1) + (sumOdd (n - 1))
```

Prove the following theorem about `sumOdd`:

Theorem 2. *For all natural numbers* `n`*,* `sumOdd n` $=$ `n * n`*.*

The proof is by induction on the natural number `n`.

Follow the same requirements as in the previous induction proof. You may assume that `n*n + 2*n + 1` $=$ `(n+1) * (n+1)`. Cite this as fact (i).

$\boxed{\textbf{Solution 4.2}}$

Theorem: For all natural numbers `n`, `sumOdd n = n * n`.

Proof. The proof is by mathematical induction on `n`.

Base Case: `n=0`.
Need to Show: `sumOdd 0 = 0`.
Showing:

`sumOdd 0` $=$ `0`	First clause of `sumOdd` is relevant since `n=0`

Induction Step: Prove for `n+1`, with `n` $\geq$ `0`:
Inductive Hypothesis: `sumOdd (n) = n * n`.
Need to show: `sumOdd (n+1) = (n+1) * (n+1)`.
Showing:

`sumOdd (n+1)`	$=$ `2*(n+1)-1 + sumOdd((n+1)-1)`	2nd clause of `sumOdd` is relevant, as `n+1 > 0`
	$=$ `2*n+1 + sumOdd(n)`	Distribution and Arithmetic
	$=$ `2*n+1 + n*n`	By IH
	$=$ `n*n+ 2*n+1`	Commutativity of Addition
	$=$ `(n+1) * (n+1)`	Factoring (Fact (i))

Thus we have shown for all n, `sumOdd(n) = n * n`.

□