15-150 Spring 2013
Homework 02

Out: Wednesday, 23 January 2013
Due: Tuesday, 29 January 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, 29 January 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 presented in lecture:

```
(*  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 4` evaluates to `4.0`. 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 2.7` evaluates to `2`. 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 (3%). Consider the following code fragment:

```
fun square (x : real) : real = x * x
fun square (x : int) : int = x * x
val z : real = square 7.0
```

Does this typecheck? Briefly explain why or why not.

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 2.2 (8%). Consider the following code fragment (the line-numbers are for reference, not part of the code itself):

```
 (1)  val x : int = 9
 (2)
 (3)  fun assemble (x : int, y : real) : int =
 (4)    let val q : real = let val x : int = 3
 (5)                           val p : real = 5.4 * (real x)
 (6)                           val y : real = p * y
 (7)                           val x : int = 123
 (8)                       in p + y
 (9)                       end
(10)    in
(11)      x + (trunc q)
(12)    end
(13)
(14)  val z = assemble (x, 2.0)
```

(a) What value gets substituted for the variable `x` in line (5)? Briefly explain why.

(b) What value gets substituted for the variable `p` in line (8)? Briefly explain why.

(c) What value gets substituted for the variable `x` in line (11)? Briefly explain why.

(d) What value does the expression `assemble (x, 2.0)` evaluate to in line (14)?

2.2 Evaluation

Task 2.3 (6%). Consider the following code fragment:

```
fun square (x : int) : int = x * x
val z : int =
  let
    val x : real = real (square 6)
  in
    3 + (trunc x)
  end
```

Provide a step-by-step sequential evaluation trace of the right-hand-side of the declaration of `z` (that is, `let val x : real = real (square 6) in 3 + (trunc x) end`). You may assume that, for values `i : int`, the expression `real i` evaluates in one step to the corresponding `real` value, and similarly for `trunc r` given a value `r : real`.

3 Extensional Equivalence and Referential Transparency

Consider the function `fact` of type `int -> int`, given by:

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

The form of this recursive function definition gives us the following equivalences:

(1) `fact(0)` $\cong$ `1`

(2) `fact(n)` $\cong$ `n * fact(n-1),` for any integer `n` with `n>0`.

In particular, the following two instances of (2) are obtained by picking `n=1` and `n=2`:

- `fact(1)` $\cong$ `1 * fact(1-1)`
- `fact(2)` $\cong$ `2 * fact(2-1)`

Recall that *referential transparency* implies that the value of any expression is unchanged if we replace a sub-expression by another expression with the same value. Hence, by referential transparency and the facts that `1-1 = 0` and `2-1 = 1`, we can deduce the equivalences

(a) `fact(1)` $\cong$ `1 * fact(0)`

(b) `fact(2)` $\cong$ `2 * fact(1)`

Task 3.1 (5%). Using extensional equivalence and referential transparency, show that

$$\texttt{fact(3)} \cong \texttt{6}$$

Your answer must *not* use the $\Longrightarrow$ notation (for evaluation). You should instead properly use the equivalences (1) and (2) as well as (a) and (b) mentioned above. Cite these equivalences to justify your reasoning. Write your proof mathematically, line by line, justifying each step. Do not write an English paragraph.

Task 3.2 (5%). Using extensional equivalence and referential transparency, show that

$$\texttt{fact(fact(1000))} \cong \texttt{fact(1000)} * \texttt{fact(fact(1000)} - \texttt{1)}$$

You may assume that `fact(1000)` returns a positive integer. As before, do not perform any evaluations.

Task 3.3 (3%). Define

```
fun f (x : int) : int = f x
```

Are the following two expressions extensionally equivalent?

$$\texttt{fact(}\sim\texttt{1)} \overset{?}{\cong} \texttt{f 10}$$

Explain why or why not.

4 Recursive Functions

4.1 Multiplication

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

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

(Recall that a *natural number* is a nonnegative integer.)

Task 4.1 (5%). In `hw02.sml`, write and document the function

```
mult : int * int -> int
```

such that `mult (m, n)` recursively calculates the product of `m` and `n`, for any two natural numbers `m` and `n`. Your implementation may use the function `add` mentioned above and `-` (subtraction), but it may not use `+` or `*`.

4.2 Harmonic Series

In mathematics, the harmonic series is the series

$$\sum_{i=1}^{\infty} \frac{1}{i} \quad = \quad 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + ...$$

Although this series ultimately diverges, it does so very slowly. The partial sum of the first n numbers in this series is called the nth harmonic number, H_n, defined for all $n \geq 0$:

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

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

Task 4.2 (3%). What is H_0? Give a few more examples of elements in the harmonic series.

Task 4.3 (7%). In `hw02.sml`, write and document the function

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

such that `harmonic n` recursively calculates H_n, the nth harmonic number, for any natural number n. For this problem you may use any functions or operators you wish. In particular you may need to use the built-in function `real` discussed earlier in this assignment.

Note: it is somewhat fragile to compare floating point numbers for equality, because computations on `real`s are prone to rounding errors. In many circumstances, two floating point numbers that you think should be equal will actually be slightly different. For this reason, SML does not allow pattern-matching on floating point numbers, analogous to `val 120 = fact 5`. Instead, you need to use an explicit equality test:

```
val true = Real.==(harmonic 1, 1.0)
```

Methodologically, it is usually better to check that two floats 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.

4.2.1 The Alternating Harmonic Series

A related sequence is the alternating harmonic series, in which every other term has negative sign,

$$\sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{i} \quad = \quad 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - ...$$

For any natural number n, the partial sums of the series are the alternating harmonic numbers I_n:

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

Unlike the harmonic series, the alternating harmonic series converges; as n approaches infinity, I_n approaches $\ln(2)$ (the natural log of 2).

Task 4.4 (3%). What is I_0? Give a few more examples of elements in the alternating harmonic series.

There are multiple ways of calculating the alternating harmonic numbers, depending on how one determines the sign of the terms.

Version 1: The most straightforward is, at each step, to check whether the index of the term is even or odd, and choose the sign appropriately.

Task 4.5 (6%). In `hw02.sml`, write and document the function

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

such that `altharmonic n` recursively calculates I_n, determining at each step whether the term being added is positive or negative. You will need to use `real : int -> real` and also `evenP` and/or `oddP`, both of type `int -> bool`, provided for you in `hw02.sml`.

Version 2: Alternatively, we can pass along information that tells us whether `n` is even or odd. To do this, we need a *helper function*

```
altharmonicHelper : int * bool -> real
```

`altharmonicHelper (n : int, even : bool)` takes two arguments, and recursively calculates I_n, **assuming that** `even` is `true` iff `n` is even (so `false` iff `n` is odd).

Task 4.6 (6%). In `hw02.sml`, write the two-argument function

```
altharmonicHelper : int * bool -> real
```

and then write

```
altharmonic2 : int -> real
```

using `altharmonicHelper`, filling out documentation for both of them.

`altharmonicHelper` should **not** call `evenP` or `oddP`, though `altharmonic2` may do so.

4.3 Modular Arithmetic

We have already implemented addition and multiplication as recursive algorithms, but what about subtraction and division? Subtraction is (mostly) straightforward, but division is a little bit trickier. For example, $\frac{8}{3}$ isn't a whole number – you could claim that the answer is 2, but you still have a remainder of 2 left over since 8 isn't exactly a multiple of 3. This means that in order to write a version of division that does not lose any information, we must return two things: the quotient, and the remainder of the division.

Fortunately, this is very straightforward to do! Just as we can write functions that take two arguments, we can write functions that evaluate to a pair of results.

The algorithm is fairly simple: subtract *denom* from *num* until *num* is less than *denom*, at which point *num* is the remainder, and the number of total subtractions is the quotient. (Note that this is somewhat dual to multiplication!)

Task 4.7 (10%). Write the function

```
divmod : int * int -> int * int
```

in `hw02.sml`.

Your function should meet the following spec:

> For all natural numbers `n` and `d` such that `d > 0`, there exist natural numbers `q` and `r` such that `divmod(n, d)` $\cong$ `(q, r)` and `qd + r = n` and `r < d`.

If `n` is not a natural number or `d` is not positive, your implementation may have any behavior you like.

Integer division and modular arithmetic are built in to SML (`div` and `mod`), but **you may not use them for this problem**. The point is to practice recursively computing a pair.

Sum Digits Having defined `divmod`, we can proceed to write some functions that do interesting things with modular arithmetic. For example, it is fairly straightforward to compute the sum of all the digits in a base 10 representation of a number. First, check to see if the number is zero. If it isn't, add the remainder of dividing the number by 10 to the result of recursively applying the function to the number divided by 10. This adds the least significant digit to the total, then "chops off" that least significant digit and proceeds recursively on the result, ending when the number has been completely truncated. For example, applying this algorithm to 123 adds 3 to the sum of the digits in 12, which adds 2 to the sum of the digits in 1, which is just 1, so the total result is 6.

Of course, this can also be generalized to an arbitrary base by dividing by the base b instead of 10 each time. Thus, we can write a function in SML

```
    sum_digits : int * int -> int
```

such that for any natural numbers `n` and `b` (where `b > 1`) `sum_digits (n, b)` evaluates to the sum of the digits in the base `b` representation of `n`.

Task 4.8 (10%). Write the function

```
    sum_digits : int * int -> int
```

in `hw02.sml`.

5 Induction

5.1 Correctness of Double

Consider the following function:

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

Task 5.1 (10%). In this problem, you will prove the following specification:

Theorem 1. *For all natural numbers n, `double n` $\cong$ `2*n`.*

This is intentionally a very simple theorem about a very simple piece of code. The goal of this problem is for you to practice getting the form of an inductive proof exactly right. **Your proof must follow the template for mathematical induction given in Lecture 3.** First, you should state the theorem you are trying to prove. Then, you need to state what *method of induction* you are using and what the *induction variable* is. Next, you should say what the *base case* is, what you are trying to show in the base case, and then show it. Finally, you need to say what the the induction step is, what the *inductive hypothesis* is, and what you need to show in the induction step. And then show it. As you do that, you must cite precisely where you use the inductive hypothesis and you must justify any reduction or equivalence steps that aren't instantly obvious from the code.

5.2 Correctness of Summorial

Task 5.2 (10%). As you may recall, the closed form for the sum of the natural numbers from 0 to n is

$$0 + \ldots + n = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

We will use this closed form to prove the correctness of the `summ` function that you implemented in lab:

```
fun summ (0 : int) : int = 0
  | summ (n : int) : int = n + (summ (n - 1))
```

Theorem 2. *For all natural numbers* `n`*,* `summ n` $\cong$ `(n*(n+1)) div 2`*.*

The proof is by induction on the natural number `n`.
Follow the same requirements as in the previous induction proof.

You may assume the following fact:

For all natural numbers n,

$$(n+1) + \frac{n(n+1)}{2} = \frac{(n+1)((n+1)+1)}{2}$$

.