15-150 Spring 2016
Homework 02

Out: Wednesday, 20 January 2016
Due: Tuesday, 26 January 2016 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.andrew.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 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, 26 January 2016 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
```

1.5 Style

In this and future homeworks, we will also grade your code for style. If your code for a task does not adhere to the style guidelines on the course website, you will temporarily receive a score of 0 for that task. You may then fix the problems with your code style and show it to a TA within two weeks of getting your graded assignment back. If you do this, you will get the number of points you would get if your original submission had proper style. If you do not do this, you will keep the grade of 0 for that task.

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 (6 pts). Consider the following code fragment:

```
fun squareit (a : real) : real = a * a
fun squareit (b : real) : int = trunc b * trunc b
fun bopit (c : real) : real = squareit (c + 1.0)
```

Does this typecheck? Briefly explain why or why not.

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 2.2 (11 pts). Consider the following code fragment (the line-numbers are for reference; they are not part of the code itself):

```
 (1)  val r : int = 4
 (2)  val i : real = 2.0
 (3)  val p : real = 3.0
 (4)  val temp : real = p - 1.0
 (5)  fun generate (p : int, r : int, q : real) : int =
 (6)    let
 (7)      val g : real =
 (8)        let
 (9)          val i : real = 5.0
(10)          val w : real = i * q
(11)          val a : real = temp * (real r)
(12)          val t : int = 30
(13)          val a : real = a - real p
(14)        in
(15)          w + (real t) - a
(16)        end
(17)    in
(18)      trunc i + trunc g
(19)    end
(20)
(21)  val life = generate (r, trunc i, temp)
```

Note that for evaluating the declaration for `temp` in line (4), the binding `[3.0/p]` is used for the variable `p`. The value of the binding is of type `real`. Answer the following questions within the context of evaluating line (21).

(a) What value gets substituted for the variable `i` in line (10)? Briefly explain why. What is its type?

(b) What value gets substituted for the variable `p` in line (13)? Briefly explain why. What is its type?

(c) What value gets substituted for the variable `a` in line (15)? Briefly explain why. What is its type?

(d) What value does the expression `generate (r, trunc i, temp)` evaluate to in line (21)?

2.2 Evaluation

Task 2.3 (9 pts). Consider the following code fragment:

```
fun double (t : int) : int = 2 * t
val r : int =
  let
    val a : real = real (double 3)
  in
    ~5 + (trunc a)
  end
```

Provide a step-by-step sequential evaluation trace of the right-hand-side of the declaration of `r` (that is, `let val a : real = real (double 3) in ∼5 + (trunc a) 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 x` given a value `x : real`. Make sure to include variable bindings where appropriate!

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 (8 pts). 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 (6 pts). Define

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

Are the following two expressions extensionally equivalent?

$$\texttt{fact(}\sim\texttt{5)} \overset{?}{\cong} \texttt{f }\sim\texttt{5}$$

Explain why or why not.

4 Induction

4.1 Summation of Even Numbers

Task 4.1 (12 pts). Look at the following function carefully.

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

Prove the following theorem about `sumEven`:

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

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 + 3*n + 2` $\cong$ `(n+1) * (n+2)`. Cite this as fact (A).

4.2 Proof Check

Consider the following two functions:

```
fun exp2 (n : int) : int =
  case n of
    0 => 1
  | _ => 2 * exp2 (n-1)

fun g (n : int) : int =
  case n of
    1 => 1
  | 2 => 2
  | _ => g (n-1) + 2 * g (n-2)
```

Task 4.2 (6 pts). Prove or disprove the following theorem:

Theorem 2. *For all natural numbers $n \geq 1$,* ***exp2 n*** $\cong$ ***g n***

If the theorem is true, your proof should follow one of the templates for induction given in Lecture. If the theorem is false, show a counter example.

5 Recursive Functions

5.1 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 5.1 (6 pts). 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 `*`.

5.2 Pascal's Triangle

Consider Pascal's triangle, in which each element is formed by the two elements above it (and one to the left) and the uppermost element is 1. The first five rows of the triangle follow:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Specifically, the element at position (i, j) is defined as the sum of the elements at $(i-1, j-1)$ and $(i-1, j)$ for all $i, j > 0$. Elements in the zeroth column or in a position where the row and column are equal are defined to be 1.

Task 5.2 (12 pts). Define the function

```
pascal : int * int -> int
```

that when applied to a zero-indexed row-column tuple of ints `i,j` evaluates to an integer representing the element of Pascal's triangle for position (i, j). Note that the row is the first element of the tuple, and that while there is a closed form for Pascal's triangle, you must use a recursive solution.

5.3 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 5.3 (12 pts). 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.

5.4 Primality Test

Here, we will be utilizing recursion to determine if a natural number is prime. Recall the definition for prime numbers:

Theorem 3. *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 the current divisor that we should check next. 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 5.4 (12 pts). 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.