# 15-150 Spring 2012
# Homework 2

Out: Tuesday, 24 January
Due: Wednesday, 1 February, 0900

## 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 five-step 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

To submit your solutions, place your `hw02.pdf` and modified `hw02.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw02/
```

Your files must be named exactly `hw02.pdf` and `hw02.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/02/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `hw02.sml` file must contain all the code that you want to have graded for this assignment and 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 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose, examples, or tests even if the implementation of the function is correct.

## 1.4 Due Date

This assignment is due on Wednesday, 1 February 2012, at 09:00 EST. Remember that this deadline is final and that we do not accept late submissions.

# 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.

> **Solution 2.1** It does not typecheck. In `val z : real = square 7.0`, the identifier `square` is bound by `fun square (x : int) : int = x * x` on the previous line, and is thus of type `int -> int`, so cannot accept an argument `7.0` of type `real`.

In lab, 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`.

See the end of the Lecture 3 notes for more details.

**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 = 12
(2)
(3)  fun assemble (x : int, y : real) : int =
(4)    let val q : real = let val x : int = 3
```

```
 (5)                            val p : real = 5.2 * (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 gets substituted for the variable `x` in line (5)? Briefly explain why.

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

(c) What 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)?

**Solution 2.2**

(a) `x` is bound to `3` on line (5), because the nearest enclosing binding is `val x : int = 3` on line (4).

(b) `p` on line (8) is bound to the result of `5.2 * (real x)` on line (5), that being its nearest enclosing binding.

(c) `x` on line (11) is bound to the argument `x` of the function `assemble`, that being its nearest enclosing binding. In particular, it is not bound to `123` by the binding on line (7), since that binding does not enclose its usage point.

(d) `58`.

## 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`.

**Solution 2.3**

```
    let val x : real = real (square 6)
    in 3 + (trunc x)
    end
|-> let val x : real = real (6 * 6)
    in 3 + (trunc x)
    end
|-> let val x : real = real 36
    in 3 + (trunc x)
    end
|-> let val x : real = 36.0
    in 3 + (trunc x)
    end
|-> 3 + (trunc 36.0)
|-> 3 + 36
|-> 39
```

## 2.3 Equivalence

Recall the `fact` function from Lecture 3.

**Task 2.4** (4%). Define

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

Are the following two expressions equivalent?

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

Explain why or why not.

**Solution 2.4** These two expressions are equivalent. This is because they are both infinite loops, and all infinite loops are equivalent (see Lecture 3 notes).

To show that they are infinite loops, try evaluating them. To evaluate `f 10` we first evaluate `10`, which is already a value. We then substitute `10` into `f`, which gives us `f 10` again. No matter how many times we perform this substitution, we keep getting `f 10` which is not a value - hence an infinite loop.

Similar logic holds for `fact (~ 1)`. As we keep evaluating it the expression does change (e.g. from `fact (~ 1)` to `fact (~ 2)`), but the result is always another call to `fact` and never a value.

**Task 2.5** (4%). Prove the following equivalence:

$$\texttt{fact(fact 3)} \cong \texttt{(fact 3) * fact((fact 3) - 1)}$$

Be careful: because ML is call-by-value, you can only expand a function definition when a function is applied to a value. For this problem, you may use only the rules of equivalence from Lecture 3 (not the valuability-based rules covered in Lecture 4—this task is asking you to prove an instance of the valuability rules, to get a sense for why they are true).

**Solution 2.5** This proof is provided in detail because you should write your proofs for this class carefully until you become more confident.

We cannot expand the definition of `fact` for the outer call in `fact (fact 3)` because `fact 3` is not a value. However, we can expand the inner call because 3 is a value. Repeating this expansion will give us a value equivalent to `fact 3` which we can then use to prove the given equivalence.

Note that whenever we expand a function definition, we justify it by verifying that the argument is a value. Also note that irrelevant code may be abbreviated with ellipsis dots, but please take care that you do not abbreviate relevant code.

`fact 3`

| | | |
|---|---|---|
| $\cong$ | `(case 3 of 0 => ... \| _ => 3 * fact (3-1))` | 3 is a value |
| $\cong$ | `3*fact (3-1)` | Case analysis |
| $\cong$ | `3*fact 2` | Arithmetic |
| $\cong$ | `3*(case 2 of 0 => ... \| _ => 2 * fact (2-1))` | 2 is a value |
| $\cong$ | `3*(2*fact (2-1))` | Case analysis |
| $\cong$ | `3*(2*fact 1)` | Arithmetic |
| $\cong$ | `3*(2*(case 1 of 0 => ... \| _ => 1 * fact (1-1)))` | 1 is a value |
| $\cong$ | `3*(2*(1*fact (1-1)))` | Case analysis |
| $\cong$ | `3*(2*(1*fact 0))` | Arithmetic |
| $\cong$ | `3*(2*(1*(case 0 of 0 => 1 \| _ => ... )))` | 0 is a value |
| $\cong$ | `3*(2*(1*1))` | Case analysis |
| $\cong$ | `3*(2*1)` | Arithmetic |
| $\cong$ | `3*2` | Arithmetic |
| $\cong$ | `6` | Arithmetic |

That is, `fact 3` $\cong$ 6. Because equivalence is a congruence, we can substitue 6 for `fact 3`. From there we can expand `fact 6` and prove our equivalence:

`fact (fact 3)`

| | | |
|---|---|---|
| ≅ | `fact 6` | Substitute `(fact 3)` ≅ 6 |
| ≅ | `case 0 of .. \| _ => 6 * fact (6-1)` | 6 is a value |
| ≅ | `6 * fact (6-1)` | Case analysis: $6 \neq 0$ |
| ≅ | `(fact 3) * fact ((fact 3) - 1)` | Substitute 6 ≅ `(fact 3)` |

Or in summary, `fact (fact 3)` ≅ `(fact 3) * fact ((fact 3) - 1)`.

# 3 Recursive Functions

## 3.1 Multiplication

In lab, we defined a function

```
add : int * int -> int
```

that implements a recursive definition of addition on the natural numbers. It is also possible to define multiplication in a similar way, in terms of addition.

**Task 3.1** (5%). In `hw02.sml`, write 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 `*`.

## 3.2 Harmonic Series

In mathematics, the harmonic series is the series

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

Although this series ultimately diverges, it does so very slowly. The partial sum of the first $n$ numbers in this series is called the $n$th harmonic number, $H_n$:

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

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

**Task 3.2** (3%). What is $H_0$? Give a few more examples of elements in the harmonic series.

**Solution 3.2**

$$H_0 = 0 \qquad H_1 = 1 \qquad H_3 = 1.8333...$$

As defined in the homework, $H_0 = \sum_{i=1}^{0} \frac{1}{i}$, the sum of an empty sequence, so it is 0.

**Task 3.3** (7%). In `hw02.sml`, write and document the function

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

such that `harmonic n` recursively calculates $H_n$, the $n$th 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 $\epsilon$, 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 The Alternating Harmonic Series

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

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

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 3.4** (3%). What is $I_0$? Give a few more examples of elements in the alternating harmonic series.

**Solution 3.4**

$$I_0 = 0 \qquad I_1 = 1 \qquad I_2 = 0.5$$

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 3.5** (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** This method of calculating the alternating harmonic numbers is rather inefficient because we defined `evenP` and `oddP` recursively, taking time proportional to the input value $n$. Since one is invoked at each step, executing `altharmonic` will take a number of steps that is quadratic in the input number.

There is a faster method:[1] 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 and `false` iff `n` is odd.

**Task 3.6** (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. You may find the function

```
not : bool -> bool
```

to be helpful for `altharmonicHelper`. Executing `altharmonic2` should take a number of steps that is linear in the input number.

[1] There is another faster method: use integer modular arithmetic to implement `evenP` in constant time. However, the method we suggest here would work even for non-fixed-size integers, where `mod` is not constant time. Moreover, it gets you to practice writing a function whose argument is a pair.

## 3.3 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 straighforward to do! Just as we can write functions that take two arguments, we can write functions that evaluate to a pair of results. See the `geom` example from lab/the end of the Lecture 3 notes.

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 3.7** (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 ML (`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 recursing on the number divided by 10. This adds the least significant digit to the total, then "chops it off" of the end and recurses 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 one, 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 3.8** (10%). Write the function

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

in `hw02.sml`.

# 4 Induction

## 4.1 Correctness of Double

Recall the `double` function from lecture:

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

**Task 4.1** (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 structural induction on a natural number**; see the Lecture 3 notes. Your equality reasoning should include each step of evaluation necessary to prove the equivalence, analogously to the proof of correctness of `exp` in the notes. You may assume basic properties of arithmetic (associativity, distributivity of $*$ over $+$, commutativity, etc.).

**Solution 4.1**

*Proof.* The proof is by induction on the natural number $n$. The predicate to be proved is `double x` $\cong$ `2*x`.

**Case for** 0: To show: `double 0` $\cong$ `2 * 0`.
Proof:

| | |
|---|---|
| `double 0` | |
| $\cong$`case 0 of 0 => 0 \| _ => 2 + double(0-1))` | Step - `0` is a value |
| $\cong$`0` | Step |
| $\cong$`2*0` | Arithmetic |

Thus `double 0` $\cong$ `2*0`.

**Case for** $k+1$: Inductive Hypothesis: `double k` $\cong$ `2*k`.
To Show: `double(k+1)` $\cong$ `2*(k+1)`
Proof:

| | |
|---|---|
| `double(k+1)` | |
| $\cong$`case (k+1) of 0 => 0 \| _ => 2 + double((k+1)-1)` | Step - `k+1` is a value |
| $\cong$`2 + double((k+1)-1)` | Step |
| $\cong$`2 + double k` | Arithmetic |
| $\cong$`2 + (2*k)` | I. H. |
| $\cong$`2 * (k+1)` | Arithmetic |

By induction, `double n` $\cong$ `2*n` for all natural numbers $n$. □

## 4.2 Correctness of Summorial

**Task 4.2** (15%). 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 (n : int) : int =
    case n of
      0 => 0
    | _ => 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 template as above. Your equality reasoning should include each individual step of evaluation necessary to prove the equivalence.

In the inductive case, you will need to do some algebraic manipulation. Break this out as a separate lemma, and prove it; you may assume basic properties of arithmetic (associativity, distributivity of $*$ over $+$, commutativity, etc.).

**Solution 4.2**

*Proof.* The proof is by induction on the natural number $n$. The predicate to be proved is `summ x` $\cong$ `(x*(x+1)) div 2`.

**Case for** 0.
To Show: `summ(0)` $\cong$ `(0*(0+1)) div 2`.
Proof:

| | |
|---|---|
| `summ 0` | |
| $\cong$`case 0 of 0 => 0 \| _ => 0 + (summ(0-1))` | Step - `0` is a value |
| $\cong$`0` | Step |
| $\cong$`(0*(0+1)) div 2` | Arithmetic |

Thus `summ(0)` $\cong$ `(0*(0+1)) div 2`.

**Case for** $k+1$.
Inductive hypothesis: `summ (k)` $\cong$ `(k*(k+1)) div 2`.
To show: `summ (k+1)` $\cong$ `((k+1)*((k+1)+1)) div 2`.
Proof:

| | |
|---|---|
| `summ(k+1)` | |
| $\cong$`case (k+1) of 0 => 0 \| _ => (k+1) + (summ((k+1)-1))` | `Step - k+1 is a value` |
| $\cong$`(k+1) + summ((k+1)-1)` | Step |
| $\cong$`(k+1) + summ(k)` | Arithmetic |
| $\cong$`(k+1) + (k*(k+1)) div 2` | I. H. |

We now need to show that for a natural number $k$, $(k+1)+\frac{k(k+1)}{2} = \frac{(k+1)((k+1)+1)}{2}$. This is true due to Lemma 1. That shows `summ(k+1)` $\cong$ `((k+1)*((k+1)+1)) div 2`, which completes the proof.

Note: We chose to write this lemma in math notation because it is algebra-heavy, but it is equally valid to reason in SML notation instead. □

**Lemma 1.** *For all natural numbers $n$,*

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

*Proof.* The proof performs standard algebraic manipulations. We give it in excruciating detail because you should write your proofs for this class carefully until you become more confident. Note that the justification for each step is listed in the righthand column.

$$
\begin{aligned}
(n+1)+\frac{n(n+1)}{2} &= (n+1)+\frac{(n*n+1*n)}{2} && \text{Distributivity}\\
&= (n+1)*(\frac{2}{2})+\frac{n*n+1*n}{2} && \text{Mult. Identity}\\
&= \frac{((n+1)*2)}{2}+\frac{n*n+1*n}{2} && \text{Associativity}\\
&= \frac{(n*2+1*2)}{2}+\frac{n*n+1*n}{2} && \text{Distributivity}\\
&= \frac{(n*2+1*2)+n*n+1*n}{2} && \text{Distributivity}\\
&= \frac{n*2+1*2+n*n+1*n}{2} && \text{Associativity}\\
&= \frac{n*n+1*2+n*2+1*n}{2} && \text{Commutativity}\\
&= \frac{n*n+1*2+2*n+1*n}{2} && \text{Commutativity}\\
&= \frac{n*n+1*2+n(2+1)}{2} && \text{Distributivity}\\
&= \frac{n*n+1*2+n*3}{2} && \text{Arithmetic}\\
&= \frac{n*n+2+n*3}{2} && \text{Arithmetic}\\
&= \frac{n*n+n*3+2}{2} && \text{Commutativity}\\
&= \frac{(n+1)(n+2)}{2} && \text{Distributivity}\\
&= \frac{(n+1)(n+1+1)}{2} && \text{Arithmetic}\\
&= \frac{(n+1)((n+1)+1))}{2} && \text{Associativity}
\end{aligned}
$$

□