COMP 212 Spring 2015
Homework 02 Programming

1 Introduction

In this programming part of the assignment, you will work on defining recursive functions. We expect you to follow the five-step methodology for defining a function, as shown in class.

To submit your programming solutions, place your modified `hw02.sml` file in your handin directory on WesFiles.

2 Recursive Functions

2.1 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 2.1 (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 in the form `x-k` where `k` is a numeral), but it may not use `+` or `*`.

2.2 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:

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

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

Task 2.2 (3%). What is H_0? In comments in your SML file, give a few more examples of elements in the harmonic series.

Task 2.3 (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 on testing: 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 amount 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.

2.3 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 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 2.4 (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 2.5 (10%). Write the function

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

in `hw02.sml`.