COMP 212 Fall 2024
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.

Submit your updated `hw02.sml` file by uploading it to your Google Drive handin folder.

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 pts). 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 Halves

Task 2.2 (5 pts). Write a function

```
halves : int -> int * int
```

such that for any natural number `n`, `halves n` computes a pair `(x,y)` such that `x + y = n` and `x = y` or `x = y + 1`. That is, `x` should be $n/2$ rounded up and `y` should be $n/2$ rounded down. Since the goal is to practice functions that compute pairs by recursion, you can only use addition and subtraction of numerals for this problem. Hint: use one of the alternative recursion schemes that we talked about in lab.

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.

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 2.3 (10 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.

Integer division and modular arithmetic are built in to ML (`div` and `mod`), but **you may not use them for this problem**.

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.

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 2.4 (10 pts). Write the function

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

in `hw02.sml`.

(If the base b part is confusing, assume b is 10 first.)