Homework 03

15-150 Fall 2024

Due: Wed 18th Sept, 2024 at 8:00pm

LaTeX

The written part of your assignments must be written in LaTeX. To make your job easier, a LaTeX template is provided for each assignment where you simply need to fill in your solutions.

If you want to know what is the LaTeX name for a symbol, you can use Detexify. Also, Mathcha can help you type math. For constructs particular to this course (inductive definitions, proofs, etc), we provide a LaTeX guide with templates.

Finally, if you are just lost on how to type something or why your file is not compiling, reach out to the course staff.

Your submission will be penalized if it is not written using LaTeX. The applicable penalties are described in the style guide.

Code Structure

Check out the *basic requirements* for coding tasks in the style guide.

Proof Structure

Check out the *basic structure* for proofs in the style guide.

1 Straight Recursion versus Tail Recursion

In this section, we will practice going back and forth between regular (or straight) recursion and tail recursion when defining functions. Along the way, we will examine the distinction between mathematical proofs and proofs of programs.

1.1 Multiplying a list

Here is the regular definition of the function *mult* that multiplies all the numbers in an integer list $l \in \mathbb{L}_{\mathbb{Z}}$ it takes as input:

$$\begin{cases} mult(nil) & = & 1 \\ mult(n :: l) & = & n \times mult\ l \end{cases}$$

As you may notice, it uses straight recursion.

Task 1.1 (2 points) Give the corresponding tail recursive definition, call it *multTail*.

Coding Task 1.2 (2 points) Write the SML functions

```
mult     : int list -> int
multTail : int list -> int
```

that implement *mult* and *multTail*, respectively, as well as any auxiliary functions they may use.

Task 1.3 (3 points) Write the recurrence relation for the work of `mult`, deduce an upper-bound approximation in closed form, and determine a tight big-O class for it. Do the same for `multTail`. Which one is more efficient in practice? Feel free to refer to calculations seen in class, but cite each occurrence.

Task 1.4 (6 points) Show that the inductive function definitions for `mult` and `multTail` compute the same mathematical function by proving the following property:

Property 1. *For all* `l : int list`, `mult l` $\cong$ `multTail l`

If this proof makes use of an auxiliary lemma, state it and prove it.

1.2 Minimum of a Tree

Consider the following inductive definition for the set $\mathbb{T}$ of trees with integer data their leaves:

$$\begin{cases} \mathit{Leaf} : \mathbb{Z} \to \mathbb{T} \\ \mathit{Node} : \mathbb{T} \times \mathbb{T} \to \mathbb{T} \end{cases}$$

Task 1.5 (2 points) Define the function $\mathit{treeMin} : \mathbb{T} \to \mathbb{Z}$ that returns the smallest number in the tree given as input.

Coding Task 1.6 (2 points) Using the datatype `tree` defined in the handout, write the SML function

```
treeMin : tree -> int
```

that implements *treeMin*.

Task 1.7 (1 points) Is it possible to give a tail-recursive implementation of *treeMin*? If your answer is "yes", give the code for it. If your answer is "no", explain why.

2 Prime factorization

As you know, every number greater than 1 can be decomposed into the product of prime numbers (that's called the *prime factorization theorem*, or *fundamental theorem of arithmetic*). And of course a number is prime if it has exactly two divisors: 1 and itself. In this exercise, we engineer a way to find the prime factors of a number, and then we'll play with it.

2.1 Let's factorize

The basic idea for finding the prime factorization of a number is to divide it in turn by all prime numbers smaller than a certain value and stop when we get to 1. That presupposes that we can tell whether a number is prime, which requires . . . trying to factorize it! This sounds circular. We will do away from knowing the prime numbers in advance and figure out if a potential factor is prime or not as we go. But when do we stop? Clearly, it doesn't make much sense to go beyond the number we are trying to factorize.

Coding Task 2.1 (10 points) Implement the SML function

```
upto: int * int -> int list
```

such that the call `upto (n, N)` returns the list of all the prime factors of `n` that are smaller than or equal to `N`. For example, `upto (60, 4)` ≅ `[2,2,3]` (or a permutation of this list) which is the prime factorization `[2,2,3,5]` of `60` excluding any number that is larger than `4`, which is why we drop `5`. You may assume that both `n` and `N` are greater than 0. The call returns the empty list if `n` is 1.

Coding Task 2.2 (2 points) Implement the SML function

```
factorize: int -> int list
```

such that `factorize n` returns a list of all the prime factors of `n` is some order. For example, `factorize 60` ≅ `[2,2,3,5]` (or a permutation of this list). You may assume that `n` is positive.

2.2 Certificates

As you will see, we will be suspicious of any SML function that claims to return the prime factorization of a number `n`. We will want some evidence. This evidence will be the list of the prime numbers whose product, supposedly, is `n`. We call this a *certificate*.

Coding Task 2.3 (2 points) Implement the SML function

```
mult: int list -> int
```

such that, given any list of numbers `l`, the call `mult l` returns the product of the numbers in `l`.

Coding Task 2.4 (5 points) Implement the SML function

```
valid: int list -> bool
```

such that `valid l` returns `true` if `l` is a list of primes, possibly repeated. It returns `false` otherwise.

2.3 Prime, or not

We will be particularly interested in knowing that a number is *not* prime.

Coding Task 2.5 (3 points) Using `factorize` or any auxiliary function you may have defined, implement the SML function

```
notPrime: int -> bool
```

such that `notPrime n` returns `true` if `n` is not a prime number, and `false` if it is prime.

Assume now that somebody else wrote `notPrime`. How can you know that it has been implemented correctly? Even the most vigorous testing cannot try all inputs. We will address this by modifying `notPrime` so that it returns not only a yes/no answer, but also (in the affirmative case) a certificate consisting of the prime factors of the number we are curious about. This certificate will allow us to check that the response is correct (in the affirmative case — we won't be checking anything if the answer is negative).

Coding Task 2.6 (5 points) Implement the function

```
notPrime_cert: int -> bool * int list
```

such that `notPrime_cert n` returns `(true,l)` whenever `n` is not a prime number, and in this case `l` is its prime factorization; it returns `(false, [])` otherwise.

Coding Task 2.7 (4 points) Your code for `notPrime_cert` is certainly right, isn't it? Now pretend somebody else wrote it — somebody you don't fully trust as a programmer. You can still use this person's function in your code as long as you check the certificate, at least when it claims that the number is not prime. Implement the function

```
notPrime_check: int -> bool
```

so that `notPrime_check n` calls `notPrime_cert n`, and returns `true` only if this call returns `(true,l)` and the certificate `l` is a valid prime factorization of `n`.

3 Conway's Lost Cosmological Theorem

This exercise will get you to become even more familiar with the manipulation of SML lists.

3.1 Look and Say

Given a list of positive integers `l`, the *look-and-say list* of `l` is obtained by reading off out loud adjacent groups of identical elements in `l`. For example, the look-and-say list of

$$l = \texttt{[2,2,2]} \qquad \text{is} \qquad \texttt{[3,2]}$$

because l is exactly "three twos". Similarly, the look-and-say sequence of

$$l = \texttt{[1,2,2]} \qquad \text{is} \qquad \texttt{[1,1,2,2]}$$

because l is exactly "one ones, then two twos".

We will use the term *run* to mean a maximal length sublist of a list with all equal elements. For example,

$$\texttt{[1,1,1]} \qquad \text{and} \qquad \texttt{[5]}$$

are both runs of the list `[1,1,1,5,2]` but

$$\texttt{[1,1]} \qquad \text{and} \qquad \texttt{[5,2]} \qquad \text{and} \qquad \texttt{[1,2]}$$

are not: `[1,1]` is not maximal, `[5,2]` has unequal elements, and `[1,2]` is not a sublist.

3.2 ... and Code too

You will now define a function `lookAndSay` that computes the look-and-say sequence of its argument using a helper function and a new pattern of recursion.

Coding Task 3.1 (5 points) Implement the SML function

```
lasHelp: int list * int * int -> int list * int
```

such that, given a list `l`, a number `x` and a counter `c`, the call `lasHelp (l, x, c)` returns a pair `(rest, c')` such that the new counter `c'` is `c` plus the number of consecutive instances of `x` found at the head of `l`, and `rest` is what is left of `l` after removing all those consecutive instances of `x` at its head. For example,

- `lasHelp([1,2,3,4,4],4,1)` $\cong$ `([1,2,3,4,4],1)` because the number 4 does not appear at the head of the list.
- `lasHelp([2,2,6,2,3],2,3)` $\cong$ `([6,2,3],5)` because there are two consecutive occurrences of 2 at the head of the list which brings the counter to 5.

Coding Task 3.2 (5 points) Using this helper function, implement the SML function

```
lookAndSay: int list -> int list
```

such that the call `lookAndSay l` returns the look-and-say list of `l`.

Coding Task 3.3 (5 points) Implement the SML function

```
lookAndSay': int list -> int list
```

such that the call `lookAndSay' l` returns the look-and-say list of `l` *without using any helper function.*

3.3 Cultural Aside

The title of this problem comes from a theorem about the sequence generated by repeated applications of the "look and say" operation. As `lookAndSay` has type `int list -> int list`, the function can be applied to its own result. For example, if we start with the list of length one consisting of just the number `1`, we get the following first 6 elements of the sequence:

```
[1]
[1,1]
[2,1]
[1,2,1,1]
[1,1,1,2,2,1]
[3,1,2,2,1,1]
```

Conway's theorem states that any element of this sequence will "decay" (by repeated applications of `lookAndSay`) into a "compound" made up of combinations of "primitive elements" (there are 92 of them, plus 2 infinite families) in 24 steps. If you are interested in this sequence, you may wish to consult

> J. Conway. The weird and wonderful chemistry of audioactive decay. In T. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, pages 173–188. Springer-Verlag, 1987.

or other papers about the "look and say" operation.

Because of this, we are not asking you to analyze the work and span of `lookAndSay` :-)