Programming Language, 2022 Spring, Assignment 2
Due: April 6 Wed, 11:59 pm

Submission

Write the functions in problem 1 – 3 in a single file, named "*sol2.sml*".
Then upload the file to the course homepage (assignment 2). The function names (and their types) should be the same as is described for each problem. There should be no error when using the file in repl with the following command: use "sol2.sml"; Because we are going to test your solution with an automated script, if our script cannot import your file with [use "sol2.sml"] then your score for this assignment will be zero. So, make sure you test your code.

Problems

1. Simple Eval (10 pts)

We define a simple Propositional Logic as following:

```
datatype expr = NUM of int
              | PLUS of expr * expr
              | MINUS of expr * expr
datatype formula = TRUE
                 | FALSE
                 | NOT of formula
                 | ANDALSO of formula * formula
                 | ORELSE of formula * formula
                 | IMPLY of formula * formula
                 | LESS of expr * expr
                    (* LESS(a, b) is true if a < b *)
```

Write *eval* function that takes a formula value and returns the Boolean value of the formula; i.e. the function has the following type:

```
eval: formula -> bool
```

`IMPLY` is defined as in this link: http://mathworld.wolfram.com/Implies.html

2. Check MetroMap (10 pts)

We define a data type representing metropolitan area as following:

```
type name = string

datatype metro = STATION of name
               | AREA of name * metro
               | CONNECT of metro * metro
```

Write *checkMetro* function of the following type:

```
checkMetro: metro -> bool
```

The function computes if the given metro is correctly defined. A metro is correctly defined if and only if metro STATION names appear inside the AREA of the same name. For example, the following metros are correctly defined:

```
AREA(“a”, STATION “a”)
AREA("a", AREA("a", STATION "a"))
AREA("a", AREA("b", CONNECT(STATION "a", STATION "b")))
AREA("a", CONNECT(STATION "a", AREA("b", STATION "a")))
```

Notice that for the correct metro definition, the STATION name does not necessarily be the same as the AREA name that immediately contains the STATION. The 4th example above is correct because STATION “a” (the last, or innermost one) is within AREA “a” (the outmost one), even though it is also in AREA “b”.
On the contrary, the following metros are incorrectly defined:

```
AREA(“a”, STATION “b”) ;station “b” not in area “b”
AREA("a", AREA("a", STATION "b")) ;station “b” not in area “b”
AREA("a", AREA("b", CONNECT(STATION "a", STATION "c")))
  ; station “c” is not in area “c”
AREA("a", CONNECT(STATION "b", AREA("b", STATION "a")))
  ; station “b” is not in area “b”
AREA("a", CONNECT(STATION "a", AREA("b", STATION "c")))
  ; station “c” is not in area “c”
```

3. Lazy List (40 pts – (i) 20 pts, (ii) 20 pts)

(i) A lazy list is a data structure for representing a long or even infinite list. In SML a lazy list can be defined as

```
datatype 'a lazyList = nullList
                     | cons of 'a * (unit -> 'a lazyList)
```

This definition says that lazy lists are polymorphic, having a type of 'a. A value of a lazy list is either nullList or a cons value consisting of the head of the list and a function of zero arguments that, when called, will return a lazy list representing the rest of the list. Write the following functions that create and manipulate lazy lists:

- seq(first, last) (4 pts)
  This function takes two integers and returns an integer lazy list containing the sequence of values `first, first+1, ... , last`

- infSeq(first) (4 pts)
  This function takes an integer and returns an integer lazy list containing the infinite sequence of values `first,first+1, ....`
- `firstN(lazyListVal,n)` (4 pts)
  This function takes a lazyList and an integer and returns an ordinary SML list containing the first n values in the lazyList. If the lazyList contains fewer than n values, then all the values in the lazyList are returned. (n is not negative)
- `Nth(lazyListVal,n)` (4 pts)
  This function takes a lazyList and an integer and returns an option representing the n-th value in the lazyList (counting from 1). If the lazyList contains fewer than n values, then `NONE` is returned. (Recall that we defined `'a option = SOME of 'a | NONE`). (n is not negative)
- `filterMultiples(lazyListVal,n)` (4 pts)
  This function returns a new lazy list that has n and all integer multiples of n removed from a lazyList. For example, a non-lazy list version of filterMultiples would behave as follows:

```
filterMultiples([2,3,4,5,6],2) = [3,5]
filterMultiples([3,4,5,6,7,8],3) = [4,5,7,8]
```

  (n is not negative)

(ii) One of the algorithms to compute prime numbers is the "Sieve of Eratosthenes." The algorithm is simple as following.

You start with the infinite list L = 2, 3, 4, 5, ... The head of this list (2) is a prime. If you filter out all values that are a multiple of 2, you get the list 3, 5, 7, 9, ... The head of this list (3) is a prime. Now you filter out all values that are a multiple of 3, you get the list 5, 7, 11, 13, 17, ... You repeatedly take the head of the resulting list as the next prime, and then filter from this list all multiples of the head value.

Write *primes()* function that computes a lazyList containing all prime numbers starting from 2, using the "Sieve of Eratosthenes" technique. To test your function, evaluate `firstN(primes(),10)`; you should get `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]`. Try `Nth(primes(),20)`; you should get `some 71`. (This may take a few seconds to compute.)

Hint: Create a recursive function `sieve(lazyListVal)` that returns a new lazyList. The first element of the cons in this lazyList should indicate the current value, as usual. The second value should be a function that calls `sieve` again appropriately.