CSE 1729 - Introduction to Principles of Programming — Due October 15, 2018

Problem Set 5

1. One list is a *prefix* of another if the other list has the same values starting from the beginning, and potentially more. For example, '(1 2) is a prefix of '(1 2 3 4), and also a prefix of '(1 2); '() is a prefix of any list; and the empty list has only one prefix, itself.
   (a) Define a Scheme function, `is-prefix?`, which takes two lists as arguments; it returns `#t` if the first list is a prefix of the second list, `#f` otherwise. You can depend on the parameters being lists of numbers, so you can use `=` to see if values are equal.
   (b) We can define a *common prefix* of two lists l_1 and l_2 as any list that is a prefix of both lists. For example, '(1 2 3) and '(1 2) have common prefixes '(),'(1),and '(1 2).
   Define a Scheme function, `longest-common-prefix`, which takes two lists as arguments, and returns the longest common prefix of the two lists. In the above example, that would be '(1 2).

2. Generalized consecutives. In lab you wrote a function, `consecutive-squares` based on `consecutive-ints`. Here you will generalize these in two ways.
   (a) First, write a Scheme function `(gen-consecutive f a b)`, where `f` is a function of one argument, and (as before) `a` and `b` are integers. `(gen-consecutive f a b)` evaluates to a list where each element is the result of applying `f` to an integer in the range `a` to `b`; that is, the values in the list are `(f a)(f (+ a 1))...(f b)`. As an example, `(gen-consecutive (lambda(x)(* x x)) 1 10)` will produce the same result as `(consecutive-squares 1 10)`.
   (b) Next, write the even more general Scheme function `(gen-sequence f a b next stop)`, where `f`, `a`, and `b` are as above, `next` is a function of one integer argument, and `stop` is a function of two integer arguments. This function evaluates to the list of values obtained by applying `f` to `a`, then `(next a)`, then `(next (next a))` and so forth, as long as `(stop a b)` is false. As an example, I could produce the list of the odd factorials from 1 to 5 by evaluating `(gen-sequence factorial 1 5 (lambda(x)(+ x 2)) >)`, assuming I had defined the factorial function. But I could also go in reverse, like `(gen-sequence factorial 5 0 (lambda(x)(- x 1)) =)`, which would give me the list `(120 24 6 2 1)`, the list from 5! to 1!. Notice that when `a` and `b` are both zero, the sequence stops, and 0! is not in the generated sequence.

3. A useful function to have for lists is `filter`: a function that finds the values of a list that pass a certain test. Specifically, we want `filter` to take a function and a list as arguments, and return a list whose members are the elements of the original list for which the function returns true. For example,
   `(filter even? '(1 2 3 4 5 6 7)) => (2 4 6).`
   Or
   `(filter (lambda(x)(< (length x) 3)) '((1) (2 3) (4 5 6 7))) => ((1)(2 3)).`
   Write a Scheme function `(filter f lst)` that implements this function. `f` should be a function that takes one argument and returns `#t` or `#f`.

4. Sometimes I have a boolean function that gives me almost what I want, but logically reversed. As an example, suppose I want to use filter (above) to give me all non-prime numbers, but I only have `(prime? k)` as a function, which is `#t` when `k` is prime.

   Write a Scheme function `(fun-not f)` that evaluates to a function of one argument that provides the logical negation of the function `f`, which is a boolean function of one argument. Specifically, if `(f a)` evaluates to `#t`, `((fun-not f) a)` should evaluate to `#f`, and vice-versa. For example, given you have the function `(prime? k)`, `(filter (fun-not prime?) '(2 3 4 5 6 7 8 9 10))` should evaluate to `(4 6 8 9 10)`.

5. Suppose I want to test whether some property holds for every member of a list, that is, for some function `f`, `(f x)` is `#t` for every element `x` of some list. Alternatively, suppose I want to test whether some property holds for some member of a list, that is, for some function `f`, `(f x)` is `#t` for at least element `x` of some list.

   (a) Write a Scheme function `(every? f lst)` that takes a function `f` a list `lst`, and evaluates to `#t` if `(f x)` is `#t` for every element `x` of `lst`, `#f` otherwise. If `lst` is the empty list, `(every? f lst)` should be `#t`.

   This function should only check as many elements as are needed to make a decision; for example, if I evaluate `(every? positive? '(-1 -10 -100 -2))` I should return `#f` as soon as I check one number in the list.

   (b) Write a Scheme function `(some? f lst)` that takes a function `f` a list `lst`, and evaluates to `#t` if `(f x)` is `#t` for any element `x` of `lst`, `#f` otherwise. If `lst` is the empty list, `(some? f lst)` should be `#f`.

   Similar to `every?`, this function should only check as many elements as are needed to make a decision.

6. (a) Write a Scheme function `(value-at-position lst k)` that takes a list *lst* and a positive integer k, and evaluates to the value at the k^{th} position of the list. Position 1 should be the first element of the list. If the position is too large for the list, the function should result in an error.

   (b) Demonstrate how you can use `gen-consecutive`, `filter` and `value-at-position` to write a function `(nth-prime-between a b n)` that will evaluate to the `n`th prime number between `a` and `b`. Feel free to use the `prime?` function below (or from the lecture slides) if that helps.

```
(define (prime? n)
  (define (divisor a) (= (modulo n a) 0))
  (define (smooth k)
    (and (>= k 2)
         (or (divisor k)
             (smooth (- k 1)))))
  (and (> n 1)
       (not (smooth (floor (sqrt n))))))
```

7. **Note:** This question may be changed or made optional. It should be stable by October 8.

   One of the points of complex numbers is that they allow us to find the square roots of a negative number, e.g. $\sqrt{-9} = 3i$.

We can define the principal square root of a complex number $a + bi$ (with $b \neq 0$) as $\gamma + \delta i$, where

$$\gamma = \sqrt{\frac{a + \sqrt{a^2 + b^2}}{2}}$$

and

$$\delta = \text{sgn}(b)\sqrt{\frac{-a + \sqrt{a^2 + b^2}}{2}}$$

sgn(b) is the *signum* of b: -1 if $b < 0$, 0 if $b = 0$, and 1 if $b > 0$. Note that the square roots used in the definitions of γ and δ all have real arguments – use `sqrt` function for these.

(a) Using the complex number implementation from the lecture slides (or using code below), define a Scheme function `(complex-sqrt x)` that calculates the principal square root of a complex number. Given the repeated use of some things, using helper function(s) and/or `let` is probably a good idea.

Test your function on a few complex numbers where $b \neq 0$ (that is, multiply the square root times itself) to show that it works.

(b) `complex-sqrt` has a bad feature, in that it doesn't work correctly for some complex numbers $a + bi$ where $b = 0$. Use complex-sqrt on a number with zero as the imaginary part and verify that it is wrong. Remember that it is not wrong for all of these, but you should be able to find an example where it is.

Modify the definition of `complex-sqrt` to fix this. (HINT: look at what happens when $b = 0$ to sgn(b) – perhaps the fix could be made there.) This function should be called `complex-sqrt-better` so Mimir will get the correct one.

```
;; complex datatype implementation

(define (make-complex a b)
  ; constructs a complex number with real part a
  ; and imaginary part b
  (cons a b))

(define (real x)
  ; returns the real part of a complex number
  (car x))

(define (imag x)
  ; returns the imaginary part of a complex number
  (cdr x))

;; utilities built from complex datatype

(define (complex-add x y)
  (make-complex
```

```
   (+ (real x)(real y))
   (+ (imag x)(imag y))))

(define (complex-sub x y)
  (make-complex
   (- (real x)(real y))
   (- (imag x)(imag y))))

(define (complex-mult x y)
  (make-complex
   (- (* (real x)(real y))
      (* (imag x)(imag y)))
   (+ (* (real x)(imag y))
      (* (imag x)(real y)))))

(define (complex-conj x)
  (make-complex
   (real x)
   (- (imag x))))
```