Programming Language Concepts

CSCI-344 **Programming 3**
Term 20161 **September 16, 2016**

Scheme Programming

Due: September 30, 2016

1 Introduction

In this programming assignment, you will implement a number of functions in μScheme in order to gain familiarity with the language and to practice functional programming.

Download `prog03.scm` and `prog03_tests.scm`. The former is a template for your submission and also includes a number of supporting functions. The latter is a test suite for the assignment.

2 Description

Complete the following exercises. Each one is to define one or more μScheme functions. See Requirements and Submissions for important restrictions.

A. (30pts) Complete all parts of Exercise 2 of Chapter 3 from *Programming Languages: Build, Prove, and Compare* (pp. 194–196, except that the function for part (e) should be named `sublist?` (rather than `contig-sublist?`) and the function for part (f) should be named `subseq?` (rather than `sublist?`).

B. (10pts) Write functions `take` and `drop` that each take two arguments (a non-negative integer and a list) and returns one result (a list). The call `take xs n` should return the longest prefix of `xs` containing at most `n` elements. The call `drop n xs` should remove the longest prefix of `xs` containing at most `n` elements and return whatever remains. You may assume that the argument is a proper list. The exact semantics of `take` and `drop` are given by the following algebraic law:

   ⋆ `take-drop` law: `(append (take n xs) (drop n xs))` = `xs`

C. (5pts) Write a function `interleave` that takes two arguments (both lists) and returns one result (a list). The function `interleave` should return a list, with the first element of the first input list as the first element of the result, the first element of the second input list as the second element of the result, the second element of the first input list as the third element of the result, the second element of the second input list as the fourth element of the result, and so on. For example, `(interleave '(a b c) '(d e f))` should return `(a d b e c f)`. If the lists are of unequal lengths, `interleave` retains the excess elements from the tail of the longer one. For example, `(interleave '(a b c d) '(e f))` should return `(a e b f c d)`. You may assume that the two arguments are proper lists.

D. (bonus 5pts) Write a function `permutation?` that takes two arguments (both lists) and returns one result (a boolean). The function `permutation?` should return `#t` if the first argument list is a permutation of the second argument list (use `equal?` to compare elements) and returns `#f` otherwise. For example, `(permutation? '(a b c d e) '(b d c a e))` should return `#t` and `(permutation? '(a b c d e) '(b c a d))` should return `#f`. Note that both `(permutation? '(a b) '(a b a b))` and `(permutation? '(a b a b) '(a b))` should return `#f`. You may assume that the two arguments are proper lists.

E. (10pts) Complete Exercise 10 of Chapter 3 from *Programming Languages: Build, Prove, and Compare* (p. 199).

F. (5pts) Write a function `arg-max` that takes two arguments (a function (that takes one argument (an element) and returns one result (an integer)) and a *non-empty* list) and returns one result (an element). The function `arg-max` should return the element of input list for which the input function returns the maximum value. If there are multiple elements of the input list for which the input function returns the maximum value, then `arg-max` should return the element that occurs earliest in the list. For example, `(arg-max (lambda (x) (* (+ x 3) (+ x 3))) '(-5 -4 -3 -2 -1 0 1 2 3 4 5))` should return `5` and

`(arg-max (lambda (x) (* x x)) '(-5 -4 -3 -2 -1 0 1 2 3 4 5))` should return `-5`. You may assume that the first argument is a procedure, that the second argument is a non-empty proper list, and that the procedure will not error when applied to an element of the list.

G. (25pts) Complete parts (b), (c), (h), (j), and (k) of Exercise 14 of Chapter 3 from *Programming Languages: Build, Prove, and Compare* (pp. 200–201), except that the function for part (h) should be named `append-via-fold` (rather than `append`) and the function for part (j) should be named `reverse-via-fold` (rather than `reverse`). All of these functions must be defined in terms of `foldl` or `foldr`; parts using explicit recursion will result in zero credit for that part. (Note: This restriction applies only to new code. For example, `gcd`, which is in the initial basis, or `insert`, which is given, may use recursion.)

H. (20pts) Complete part (c) of Exercise 19 of Chapter 3 from *Programming Languages: Build, Prove, and Compare* (p. 203).

I. (5pts) Write a function `clamp` that takes three arguments (a function (call it `f`, that takes one argument and returns one result (a number)) and two numbers (call them `low` and `high`)) and returns one result (a function (that takes one argument and returns one result (a number))). The function `clamp` should return a function that, when applied to an argument (call it `x`), returns `low` if `(f x)` is less than `low`, returns `high` if `(f x)` is greater than `high`, and returns `(f x)` if `(f x)` is greater than or equal to `low` and less than or equal to `high`. Thus, the function `clamp` returns a function that is just like `f`, except that it is "clamped" to return results between `low` and `high`. For example, `((clamp (lambda (x) (* 2 x)) -5 5) 2)` should return `4` and `((clamp (lambda (x) (* 2 x)) -5 5) 4)` should return `5`.

3 Requirements and Submission

In addition to the specifications given in the exercises, your functions must not use any imperative features of μScheme; the use of `begin`, `print`, `set`, or `while` in any exercise will result in zero credit for that exercise. (You may find it useful to use `begin` and `print` while debugging, but they must not appear in your submission.) As a substitute for assignment, use `let` or `let*`.

Helper functions may be defined at the top level. You may also define local functions using `lambda` along with `let`, `let*`, or `letrec`; if you do define local functions, avoid passing redundant parameters.

Your submission must be a valid μScheme program. In particular, it must pass the following test:

```
$ cat prog03.scm | /usr/local/pub/mtf/plc/bin/uscheme -q > /dev/null
```

without any error messages. If your submission produces error messages (e.g., syntax errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit `prog03.scm` to the `Programming 03` Dropbox on MyCourses by the due date.

Document History

September 16, 2016
Original version

A Interpreter

A reference μScheme interpreter is available on the CS Department Linux systems (e.g., `queeg.cs.rit.edu` and ICLs 1, 2, and 3) at:

`/usr/local/pub/mtf/plc/bin/uscheme`

Use the reference interpreter to check your code.

Source code for the interpreter is available on the CS Department file system at:

`/usr/local/pub/mtf/plc/src/bare/uscheme`

and

`/usr/local/pub/mtf/plc/src/commented/uscheme`

A.1 Interactive mode

Simply executing

```
$ /usr/local/pub/mtf/plc/bin/uscheme
```

will run the interpreter interactively, but without line editing.

Executing

```
$ rlwrap /usr/local/pub/mtf/plc/bin/uscheme
```

or

```
$ ledit /usr/local/pub/mtf/plc/bin/uscheme
```

will run the interpreter interactively with line editing. (See the manual pages for `rlwrap` and `ledit` for more details.)

A.2 Batch mode

Executing

```
$ cat prog03.scm | /usr/local/pub/mtf/plc/bin/uscheme
```

will run the interpreter on the contents of the file `prog03.scm`, but with prompts printed.

Executing

```
$ cat prog03.scm | /usr/local/pub/mtf/plc/bin/uscheme -q
```

will run the interpreter on the contents of the file `prog03.scm` without prompts printed.

Executing

```
$ cat prog03.scm prog03_tests.scm | /usr/local/pub/mtf/plc/bin/uscheme -q
```

will run the interpreter on the contents of the files `prog03.scm` and `prog03_tests.scm` without prompts printed.