Project 2: Scheme Programming

Due Date: 10/20 by 11:59p

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Aims of This Project

The aims of this project are as follows:

- To become adept with recursive programming.
- To expose you to functional programming.
- To familiarize you with imperative programming without destructive assignment.

Project Specification

You are required to submit a `prj2.tar.gz` archive such that unpacking that archive into an empty directory and changing into that directory will allow you to run implementations of the following using the racket dialect of Scheme installed on `remote.cs`.

The `^` operator is used below to indicate exponentiation.

1. Repeat prj1 in Scheme. Specifically, complete the code provided in the ugly-regexp parser such that running the parser as:

   ```
   $ ./ugly-regexp/parser.rkt FILE
   ```

   translates *ugly regular expressions* to standard regex syntax exactly as in Project 1. The output (including errors) should not be modified from that which is generated by the provided code.

2. Write a function `(quadratic-roots a b c)` which returns a 2-element list containing the 2 roots of the quadratic equation `a*x^2 + b*x + c = 0` using the classical formula for the roots of a quadratic equation. The first element in the returned list should use the positive square-root of the discriminant, the second element should use the negative square-root of the discriminant. During the course of the computation, no expression should be evaluated more than once.

3. Write a function `(mul-list list x)` which when given a proper list `list` of numbers returns the list formed by multiplying each element of `list` by `x`.

4. Write a function `(sum-lengths list)` which when given a proper list `list` of proper lists returns the sum of the lengths of all the lists contained directly in `list`.

5. Write a function `(poly-eval coeffs x)` which evaluates the polyomial specified by list `coeffs` at `x`. Specifically, given a list of `n + 1` numeric polynomial coefficients `coeffs (c[n] c[n-1] ... c[0])` and a number `x` return the value of `c[n]*x^n + c[n-1]*x^(n-1) + ... + c[1]*x + c[0]`. Your computation should evaluate each term as written above; i.e. each `c[i]*x^i` term should be explicitly evaluated and added together.

6. Write a function `(poly-eval-horner coeffs x)` which uses Horner's method to evaluate the polynomial given by list `coeffs` at number `x`. The list `coeffs` is as specified in (5).

7. Write a function `(count-occurrences s-exp x)` which given a Scheme s-expression `s-exp` returns the number of sub-expressions which are `equal?` to `x`.

8. Write a function `(eval-arith exp)` which returns the result of evaluating arithmetic expression `exp`, where `exp` is either a Scheme number or one of `(`*op* `exp_1 exp_2)` where *op* is one of `'add`, `'sub`, `'mul` or `'div` specifying respectively the binary arithmetic operators `+`, `-`, `*`, `/`.

9. Write a function `(sum-lengths-tr list)` with the same specification as (4) with the additional requirement that all recursive calls must be tail-recursive.

10. Write a function `(poly-eval-tr coeffs x)` with the same specification as (5) with the additional requirement that all recursive calls must be tail-recursive.

11. Write a function `(mul-list-2 list x)` with the same specification as (3) but which replaces all recursion with the use of one or more of `map`, `foldl` or `foldr`.

12. Write a function `(sum-lengths-2 list)` with the same specification as (4) but which replaces all recursion with the use of one or more of `map`, `foldl` or `foldr`.

The project is subject to the following additional restrictions:

- The solution for (1) must be in the file which unpacks into the file `ugly-regexp/parser.rkt`. The archive should also contain any auxiliary files required by that file.

- The solutions for (2-12) must be in the file `fns.rkt`.

- The code created by you may not contain any use of Scheme's mutation operators; i.e. no Scheme function with name ending in `!` may be used.

- Solutions 2-12 should not define any top-level auxiliary functions; i.e. any auxiliary functions needed for the operation of a required function should be defined within that function.

- Some of the function specifications give implementation restrictions. Those must be followed.

Example Log

The following provides a log of interaction with the code submitted with this project:

```
$ ./ugly-regexp/parser.rkt -
(chars(a) . chars(b)) + *chars(c)
((([a][b]))|[c]*)
(chars(a) . chars(b)) + chars(c)*
<stdin>:2:32: syntax error at '*', expecting '<NL>'
chars(a) . chars(b) + *chars(c)
([a]([b]|[c]*))
$ racket
Welcome to Racket v6.1.
> (load "fns.rkt")
> (quadratic-roots 3 -6 3)
'(1 1)
> (quadratic-roots 1 -10 34)
'(5+3i 5-3i)
> (mul-list '() 22)
'()
> (mul-list '(1 2 5) 22)
'(22 44 110)
> (sum-lengths '((1 2 (3 4)) ((5 6)) ()))
4
> (sum-lengths '((1) (2 (3 . 4))))
3
> (poly-eval '(1 2 3 4) 3)
58
> (poly-eval-horner '(2 3 4) 4)
48
> (count-occurrences '((a b c) a (c (d a) b)) 'a)
3
> (count-occurrences '((a b c) a (c (d a) b)) '(d a))
1
> (count-occurrences '() 'a)
0
> (eval-arith '(add (mul 8 5) 2))
42
> (eval-arith '(sub (mul 11 5) 13))
42
> (sum-lengths-tr '(((b c) a) () (c d)))
4
> (poly-eval-tr '() 5)
0
> (poly-eval-tr '(4 3 2 1) 5)
586
> (poly-eval-tr '(1 1 2 0) 3)
42
> (mul-list-2 '() 5)
'()
> (mul-list-2 '(9 3 1) 5)
'(45 15 5)
> (sum-lengths-2 '(((((1)))) (2)))
2
>
$
```

Provided Files

The ./files directory contains the following:

Makefile
: This file contains a `submit` target such that typing `make submit` will create a `prj2.tar.gz` archive containing the files to be submitted. The `clean` target will remove the archive.

  You may edit this file if you choose to use a different organization for your project. When editing, watch out for tabs (the first character of any command-line **must be a tab character**).

README
: A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission. In particular, you should document the data-structure used for your word-store.

UglyRegexp Files
: Code for a ugly-regexp parser for exercise 1. It consists of the following files:

  Parser
  : A skeleton file which will need to be completed by you. It contains the necessary main function to start up the program well as all error handling and utility functions for parsing and translation.

  Scanner
  : A very crude scanner which streams delivers tokens to the parser while ignoring linear whitespace.

  Errors
  : Trivial error reporting utility functions.

fns.rkt
: A file containing skeletons functions for exercises 2-12.

Hints

You may choose to follow the following hints (they are not by any means required). They assume that you are using the project structure supported by the provided Makefile,.

You may choose to work within the `drracket` GUI tool or simply use the `racket` CLI. Documentation is available from within `drracket` or from the web site.

Exercise 1 simply requires reimplementing project 1 in a different language and serves as an introduction to the syntax of Scheme. The provided skeleton parser is derived from the Java solution. The following points are worth noting:

- The code for `term` and `term-rest` is isomorphic to the code for `ugly-regexp` and `regexp-rest` with the obvious changes.

- The code for `factor` will require a decision based on the kind of the lookahead token. Use a `cond` with calls to `check?`. For `chars`, the `factor` function should take care of matching the `CHARS` token and it's parentheses, delegating the task of matching one-or-more comma-separated `CHAR`'s to the `chars` function.

- You will need to use `string-append` as illustrated in the provided code for `regexp-rest` to build up the return'd translations.

- When you need to name values for use in subsequent expressions, use `let` to set up the bindings.

- It is worth looking at the correspondence between the provided code for `parse`, `ugly-regexp` and `regexp-rest` and the corresponding Java functions in the solution for Project 1. This will give you some idea how Java constructs map into Scheme.

- This exercise deals with I/O which is basically stateful. Hence the code is imperative and it makes sense to have a sequence of Scheme expressions as can be seen in the exception-handling function used within `parse`. This would not be true in the absence of side-effects. (Note that the code you add is still not allowed to use any destructive Scheme functions).

- Note that in Scheme, the then-part and else-part of an `if` are syntactically restricted to be a single expression; if you need a sequence of expressions, then use a `begin` to wrap them into a single expression.

  Note that the bodies of `when`, `unless`, `lambda` and the clauses of a `cond` do not have such syntactic restrictions. Hence it is possible to have such bodies contain multiple expressions without needing to wrap them within a `begin`.

The following points are worth noting for the remaining exercises:

- It may be a good idea to initially ignore the requirement of not creating any new top-level functions. Once you have the code for a particular exercise working, then you can squirrel the definitions of any auxiliary functions into the body of the top-level function using `let`, `let*` or `letrec` as appropriate.

- Almost all the exercises require recursive solutions. Hence you need to clearly identify your base case's and recursive case's. For the former, you will need to provide a basic solution not involving any recursive calls; for the latter you will need to figure out how to combine solutions from one or more recursive calls into a solution to the current call.

  In many cases, the recursive solutions will be based on the structure of the data: this is referred to as *structural recursion*.

  - A list is either `'()` or a pair.

- For the `eval-arith` function, an arith expression is either a Scheme number or `(add exp-1 exp-2)`, ... `(div exp-1 exp-2)` where `exp-1` and `exp-2` are themselves arith expressions. So for the base case, your evaluator function needs to return the evaluation of a Scheme number. For the recursive cases, all you need to do is combine the results of the recursive calls to the evaluator on the subexpressions appropriately.

  The `count-occurrences` function is one which requires you to process a Scheme expression to an arbitrary depth. Make sure you clearly identify the base case (one of the examples provided in the log should help).

- Except for the use of `expt` and `sqrt`, your code should not need the use of any Scheme library functions beyond those discussed in class.

Submission

You will need to submit a compressed archive file `prj2.tar.gz` which contains all the files necessary to build your jar file. Additionally, this archive **must** contain a `README` file which should minimally contain your name, B-number, email, the status of your project and any other information you believe is relevant.

If you are using the suggested project structure, then the provided Makefile provides a `submit` target which will build the compressed archive for you; simply type `make submit`.

Note that it is your responsibility to ensure that your submission is complete. To test whether your archive is complete, simply unpack it into a empty directory and see if it runs correctly.

To submit the above archive, please use blackboard by following the `Content->Projects` link.