1 Project 2

Due: Mar 22, before midnight.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes how it can be submitted.

1.1 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 programming without using destructive assignment.

1.2 Project Specification

Update your github repository with a directory `submit/prj2-sol` containing two files `prj2-sol.scm` and `arith-parser.scm`.

The `prj2-sol.scm` file should contain definitions for the following functions:

1. `(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`. *5-points*

2. `(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`. This function should not be tail-recursive. *5-points*

3. `(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 $c[n] \times x^n + c[n-1] \times x^{n-1} + \ldots + c[1] \times x + c[0]$. Your computation should evaluate each term as written above; i.e. each $c[i] \times x^i$ term should be explicitly evaluated and added together.. This function should not be tail-recursive. *10-points*

4. `(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 the previous exercise. *10-points*

5. `(count-occurrences s-exp x)` which given a Scheme s-expression `s-exp` returns the number of sub-expressions which are `equal?` to `x`. *10-points*

6. `(arith-eval 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 `+`, `-`, `*`, `/` and `exp-1` and `exp-2` are arithmetic expressions. *10-points*

7. `(sum-lengths-tr list)` with the same specification as `sum-lengths` but with the requirement that all recursive calls must be tail-recursive. *10-points*

8. `(poly-eval-horner-tr coeffs x)` with the same specification as `poly¬-eval-horner` but with the requirement that all recursive calls must be tail-recursive. *10-points*

9. `(mul-list-2 list x)` with the same specification as `mul-list` but which replaces all recursion with the use of one or more of `map`, `foldl` or `foldr`. *5-points*

10. Write a function `(sum-lengths-2 list)` with the same specification as `sum-lengths but which replaces all recursion with the use of one or more of` map‘, `foldl` or `foldr`. *5-points*

The `arith-parse.scm` file should contain a definition of a function `(arith-¬parse tokens)` which should return an AST representing the structure of the proper list `tokens` as specified by the following EBNF grammar:

```
expr
  : term ( '+ term )*
  ;
term
  : factor ( '* factor )*
  ;
factor
  : NUMBER
  | '< expr '>
  ;
```

Both the `'+` and `'*` operators should be left-associative.

The proper list `tokens` should contain Scheme numbers, `'+`, `'*` and `'<` and `'>` (the latter two symbols are used as parentheses). The AST for `'+` and `'*` should use tags `'add` and `'mul` repectively; i.e. the output of `arith-parse` should be suitable as input to `arith-eval`.

If `tokens` cannot be parsed according to the above grammar, then `(arith-¬parse tokens)` should return `#f`. *20-points*

The project is subject to the following additional restrictions:

- You should not use any of Scheme's mutation operators; i.e. no Scheme function with name ending in `!` may be used.
- The functions defined in `prj2-sol.scm` 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. This restriction does not apply to `arith-parse.scm`.
- None of the functions may use `map`, `foldl`, or `foldr` unless explicitly specified.
- Some of the function specifications give implementation restrictions. Those must be followed.

1.3 Example Log

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

```
$ racket
Welcome to Racket v7.2.
> (load "prj2-sol.scm")

> (mul-list '(3 4 5) 8)
'(24 32 40)
> (mul-list '() 8)
'()

> (sum-lengths '( (1 2 3) (()) (() 2 (3 4 5))))
7
> (sum-lengths '( ))
0

> (poly-eval '(5 4 3 2 1) 1)
15
> (poly-eval '(5 4 3 2 1) 2)
129
> (poly-eval '() 1)
0

> (poly-eval-horner '(5 4 3 2 1) 1)
15
> (poly-eval-horner '(5 4 3 2 1) 2)
129
> (poly-eval-horner '() 1)
0
```

```
> (count-occurrences '( (+ 1 2) (a (+ 1 2) 3) ) 1)
2
> (count-occurrences '( (+ 1 2) (a (+ 1 2) 3) ) 3)
1
> (count-occurrences '( (+ 1 2) (a (+ 1 2) 3) ) '(+ 1 2))
2
> (count-occurrences '( (+ 1 2) (a (+ 1 2) 3) ) '(+ 1 3))
0

> (arith-eval '(mul 3 (add 4 (mul 4 3))))
48
> (arith-eval 45)
45

> (sum-lengths-tr '( (1 2 3) (()) (() 2 (3 4 5))))
7
> (sum-lengths-tr '())
0

> (poly-eval-horner-tr '(5 4 3 2 1) 1)
15
> (poly-eval-horner-tr '(5 4 3 2 1) 2)
129
> (poly-eval-horner-tr '() 2)
0

> (mul-list-2 '(4 5 9) 4)
'(16 20 36)
> (mul-list-2 '() 4)
'()

> (sum-lengths-2 '( (1 2 3) (()) (() 2 (3 4 5))))
7
> (sum-lengths-2 '( ))
0

> (load "arith-parse.scm")
> (arith-parse '( 1 + 2 +  4 * 3 ))
'(add (add 1 2) (mul 4 3))
> (arith-parse '( 1 + < 2 +  4 * 3 > ))
'(add 1 (add 2 (mul 4 3)))
> (arith-parse '( 1 + < 2 + + 4 * 3 > ))
#f
> (arith-parse '( 1 + 2 +  4 * 3 >))
#f
> (arith-parse '( 1 + < 2 +  4 > * 3 ))
```

```
'(add 1 (mul (add 2 4) 3))
> (arith-parse '( @ 1 + < 2 +  4 > * 3 ))
#f

> (arith-eval (arith-parse '( 1 + < 2 +  4 > * 3 )))
19
>
$
```

1.4 Provided Files

The prj2-sol directory contains the following:

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.

prj2-sol.scm A file containing skeletons functions for all except the last exercise.

arith-parse.scm A file containing partial code for the last exercise.

1.5 Hints

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 racket-lang.org.

Note that you can use trace to debug your code.

The following points are worth noting for the initial 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 `arith-eval` 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).

The `arith-parser` illustrates the techniques necessary when programming using pure functions without destructive assignment. If you understand the provided code in conjunction with the following comments, the code needed to complete the functionality should be straight-forward:

- Since we cannot use destructive assignment, we pass around the list of tokens which still need to be consumed by the parser. The lookahead token will be the head of the list.

- We represent the state of the parser as a `parse-state` which is either a non-empty list of remaining tokens, `'()` representing end-of-file, or `#f` representing a syntax error on match failure.

- If a function returns multiple results it is necessary to package up those results into a single return value and then have the caller unpack the return value. An idiom which is used in the provided code:

```
(let* ([fn-result (fn ...)]
       [val1 (access-val1 fn-result)]
       [val2 (access-val2 fn-result)]
       ...
       [valn (access-valn fn-result)])
       ...)
```

  for accessor functions `access-val1`, `access-val2`, ..., `access-valn`.

  [This can be done more conveniently in Scheme using `(match ...)` or `(values ...)` in conjunction with `(call-with-values ...)` but those have not been covered in class.]

- The `(check? tok parse-state)` function returns `#t` iff the lookahead token in `parse-state` matches `tok` (which should be either `'NUMBER` or a Scheme symbol).

- The `(match tok parse-state)` returns the new `parse-state` which returns from matching `tok` against the lookahead from `parse-state` and advancing to the next token if the match succeeds.

- We set up our parsing functions to take a single `parse-state` argument and return a compound `parse-result` which contains the constructed ast and the next `parse-state`. Note that the constructed ast is set to `#f` on error.

- Note that our recursive-descent parser recipe would create the parsing function for `expr` as:

```
    expr() {
      Ast t = term();
      while (lookahead == '+') {
        match('+);
Ast t1 = term();
t = new Ast('add, t, t1);
      }
      return t;
    }
```

  In the absence of destructive assignments, we would need to implement the `while` loop using a recursive auxiliary function like `expr-loop`.

- The code for `(term...)` would be structured identically to that for `(expr...)`.

- Note that Scheme provides `and` and `or` functions with the usual short-circuit semantics but the value returned is that of the last expression which is evaluated:

```
> (and 1 '(a b))
'(a b)
> (or 1 '(a b))
1
> (or #f '(a b))
'(a b)
```