**`pQuadratic` — Using REPL**

## Learning Outcomes

After completing this program, students will be able to

- Use `define` to define named values.
- Use `quote` and its tick mark shorthand to enter literal values.
- Use `cons`, `car`, `cdr`, to build and manipulate Scheme lists.

## Racket

I will **only** test code using `racket` with the following command:

```
$ racket -I simply-scheme -f <your-file.scm> -i
```

**If** your code
**Loads with errors** I stop grading and give a 0.0.
**Fails to run due to interpreter errors** I stop grading and give 0.0.
**Produces incorrect results** I grade with the generated results against the requirements in the assignment.

**You** are obligated to test your code on an appropriate interpreter. Racket 8.15 is installed in the lab and `drracket` is available on most major platforms.

## Procedure

Read the **whole** assignment. This is important for *every* assignment: it puts the task into your brain so that it can begin working on answering the questions. Of particular interest when you read are the SLO (first section above); gives you the links between the assignment to the big picture (learning computer science).

### Expressions in Scheme

Scheme is a (relatively) functional language. Each *expression* is written in a **prefix** notation: the first element in a list is interpreted as a *function* to be applied to the remaining elements in the list:

```
> 17
17
> (first '(a b c))
'a
> (* (+ 5 3) (- 11 4))
56
```

As shown in the third example, list evaluation continues, *recursively*, if any parameters are themselves lists. Numeric literals evaluate to their own value.

### Setting Symbols in Scheme

**Note** the following bit deviates from the pedagogy of *Simply Scheme* by showing you how to set symbols to have non-function values. It violates the idea of *functional purity*.

Scheme has a *special form* (which you can treat just like a **function** for the moment), `define`, which associate a value with a name. If, in Java, you would write something like `a = 17` or `b = (5 + 3) * (11 - 4)`, you would use the following in Scheme:

```
> (define a 17)
> (define b (* (+ 5 3) (- 11 4)))
> a
17
> (+ b 9)
65
```

**Note** that `define` is also used to define named functions.

It is odd that `define` does not return a value. This is a peculiarity of Scheme: each Scheme implementation can return whatever they want for `define`; Chicken Scheme chose nothing.

## Lists in Scheme

When we give a list to Scheme, it assumes the first element names a *function* and the rest of the list can be treated as *parameters*. This is what happens whenever Scheme **evaluates** a list.

Sometimes we want to give Scheme an expression (variable name, list) *without* it being evaluated. This is done by using the `quote` special form around the expression:

```
> (quote (* (+ 5 3) (- 11 4)))
(* (+ 5 3) (- 11 4))
```

Typing the `quote` around an expression is tedious *and* it adds another set of parentheses, so there is a shorthand for it using the tick mark, `'`.

```
> '(* (+ 5 3) (- 11 4))
(* (+ 5 3) (- 11 4))
```

1. You will be turning in a Scheme source file through the classroom management system. Put code at the top of the file to set symbols that identify you:
   (a) Set the symbol `full-name` to a list of your first and last names. Each of the component names will be *symbols* in the list.
   (b) Set the symbol `age` to your age, in years, at your last birthday. (If, for any reason, you do not want to share your actual age, put in a semi-plausible age and I will never know the difference.)
   (c) Set the symbol `age-in-months` to the equivalent age in months, using Scheme to *calculate* the number of months you were old on that date.
   (d) Set the symbol `expression-`**`for`**`-age-in-months` to the expression that you used to calculate your age in months. It should be a list containing a valid Scheme expression.

### Lists

Everything in Scheme is either an *atom* or a *list*. An atom is a symbol, a number, or a string. A list is either empty or a *head* followed by a list. **Simply Scheme** manipulates both lists and symbol names the same way.

Every expression you have typed in to your file or the REPL is either an atom or a list (these are not being typed into the REPL):

```
(* (+ 5 3) (- 11 4))  ; a list that evaluates to a number
(define kilo (* (+ 5 3) (- 11 4))) ; a list that sets the value of
                                   ; a symbol to a number
kilo ; an atom, a symbol, that evaluates to its defined value
999 ; an atom, a number, that evaluates to itself

(define pets '(knight scooby rex fido willy))
; a list that sets a symbol's value to a list

pets ; evaluates to the list (defined vaule of symbol)
```

How can we manipulate the list `'(knight␣scooby␣rex␣fido␣willy)`? Using `simply-scheme`:

```
> (first '(knight scooby rex fido willy))
'knight
> (butfirst '(knight scooby rex fido willy))
'(scooby rex fido willy)
> (last '(knight scooby rex fido willy))
'willy
> (butlast '(knight scooby rex fido willy))
'(knight scooby rex fido)
> (butlast 'willy)
> 'will
```

The opposite of taking a word (atom) or a sentence (list) apart is to assemble one:

```
> (sentence 'garfield '(knight scooby rex fido willy))
'(garfield knight scooby rex fido willy)
> (word 'te 'le 'phone)
'telephone
```

Notice that `garfield` had to be quoted because all parameters to a function (all elements after the first in an evaluated list) will be evaluated. So, without the quote, `garfield` becomes whatever it is `defined` to be. With the quote, it is just the symbol.

The *empty list* is `()` and it evaluates to itself.

We can build a list, say `(cat in the hat)` with `sentence`:

```
> (sentence 'cat 'in 'the 'hat)
'(cat in the hat)
```

2. Add code to your `.scm`:

- Use the `first` and `butfirst` (and/or `last`/`butlast`) to extract `(+ 5 3)` from the list `(* (+ 5 3) (- 11 4))`.
- Extract the `11` from the list above. You will need to get the third element from the top list and then get the second element from that.
- Build the sentence `(one fish two fish red fish blue fish)` using quotes **only** on individual symbols.
- Define `k` as the list `(oh the places)`. Then evaluate the following two expressions and explain the difference in results:

```
(sentence 'k '(you will go))
(sentence k '(you will go))
```

- Add this to your file.

```
(define better-pets '((cat knight) (dog scooby) (fish rex) (moose fido) (orca willy)))
```

  - Write a Scheme expression that uses `better-pets` and evaluates to `(moose fido)`.
  - Write an expression that evaluates to the type of animal `willy` is. Not a function.
  - Write an expression that evaluates to the name (not a *list* with the name but just the name) of the fish.

## Submit through Classroom Management System

Submit your *commented* (name, answers to questions) Scheme code through the classroom management system.