EECS 311: Data Structures and Data Management	Program 2
Assigned: 11/17/08	Checkpoint: 11/25/08; Due: 12/03/08

1 Project: Scheme Calculator.

In your last programming assignment you wrote a parser that reads in scheme data types. Scheme data types and scheme programs looks the same. This programming assignment will be to begin to build an evaluator for scheme objects. You will not be required to write a full scheme interpreter; though, doing so would only be a modest extension of your project (See Section 1.3).

Well-formed scheme expressions are lists of lists such as `(* (+ x 10) (- y 300))`. How are such these expressions evaluated? *Recursively!*

1. The emptylist evaluates to itself.
2. A *number* evaluates to itself.
3. A *symbol* is considered to be a variable name. To evaluate a symbol we look up its value in a *symbol table* that contains *variable bindings*, i.e., that maps variable names to values. We will call this symbol table the *environment*. It is an programmer error to evaluate a symbol is not bound.
4. A *list* evaluates as a function *application*. For instance the list `(+ 1 2 3)` applies the function `+` to arguments `(1 2 3)`. How does this work? It first recursively evaluates each item in the list. Here `+` is a symbol. As above, a symbol is evaluated by looking it up in the environment. What should be returned is a scheme object that represents "the addition function". The numbers `1`, `2`, and `3`, are numbers, and as above, each evaluates to themself. To complete the evaluation we apply the arguments 1, 2, and 3, to the addition function. The value of this application is the return value of the addition function. In this case, it would be 6. It is a programmer error if the first element of a list does not evaluate to a function.

Essentially, evaluating scheme objects is a *postorder* traversal. See Section 4 for an example.

1.1 Additional Scheme Objects.

For the purpose of creating a scheme calculator, two additional primitive scheme data types must be added, *numbers* and *functions*. A number is self explanitory, a number is a number. One of the elegant simplicities made by the designers of scheme is in making functions *first-class* data objects, meaning, a function can be stored in a variable, or passed as an argument to another function. Notice, in C and C++ functions are not really first-cass data objects.[1] In scheme functions can be both user defined and *primitive*. A primitive function is one that is provided by the interpreter. You will only need to implement primitive functions for this project.

[1] Though, there are ways of achieving the same effect.

1.2 Additional Scheme Objects in C++.

You will need to add to your `SchemeObject` hierarchy. The first subclass you will need to add is (`SchemeNumber` n), where member field n is a number. You are welcome to limit the number n to be in the same range as C++ integers. You will need to implement an *abstract* subclass `SchemeFunction`. Recall scheme functions are objects that know how to be *applied*. You will need to implement `SchemeAddition`, `SchemeSubtraction`, `SchemeMultiplication`, `SchemeLength`, `SchemeList`, and `SchemeMap` as subclasses of `SchemeFunction`.

`SchemeAddition` (resp., `SchemeMultiplication`) should add (resp., multiply) its arguments together returning their sum (resp., product). If there are no arguments `SchemeAddition` (resp., `SchemeMultiplication`) can return the additive identity 0 (resp., the multiplicative identity 1). The functionality of `SchemeSubraction` depends on the number of arguments. On one argument, `SchemeSubtraction` is equivalant to negation. On two or more arguments, `SchemeSubtraction` returns the first argument less the sum of the remaining arguments. `SchemeLength` takes a single parameter, a list, and returns the number of elements in the list. `SchemeList` takes any number of arguments and simply returns them as a list. `SchemeMap` takes two arguments. The first is a one argument function and the second is a list. It returns the list obtained by applying the function (first argument) to each element in the list (second argument).

While it may seem complicated to support functions with varying number of arguments, actually, it is quite simple. Each `SchemeFunction` has an `amply(args,env)` function where `args` is a list of arguments (i.e., a `SchemePair`) and `env` is the environment. The return result of `apply` should be a `SchemeObject`.

1.3 Additional Discussion

There are two main parts of scheme we will not be implementing. The first is the *garbage collector*. Keeping track of memory in a language like scheme is next to impossible. If we were going to write a real scheme interpreter we would have to implement a garbage collector to ensure that there are no memory leaks. For this project only, you are permitted to ignore memory leaks of `SchemeObjects` (i.e., you can assume that you will implement a garbage collector later). You should not have memory leaks of any other data.

The second main part that is missing from your evaluator is support for *macros*. Notice that a command like `(define x 10)`, which is standard in scheme, would not work if `define` was a normal function. Why? First recall that `(define x 10)` is attempting to (re)define varaible `x` in the environment to have value `10`. Our recursive approach to evaluating functions, however, would have us first evaluating `define`, `x`, and `10`. If `x` is not already bound, this would be an error; if `x` is already bound, this would replace `x` with its current value. Thus, it would be impossible for `define` to actually change `x`. To solve this problem, special functions, sometimes referred to as *macros* specify that their arguments not be evaluated before being passed in. In this project, you do not have to implement this functionality. Other important marcros are `if` and `lambda`, the latter is for creating user-defined functions.

2 Tasks.

0. Start from either your solution to Programming Assignment 1 or the TA's solution. The TA's solution has been tested and works on `g++ 4.3.2`. Complile it with the command "`g++`

`main.cpp -o main`". You can download the TA's solution from Blackboard at the convenient URL: `<http://tinyurl.com/prog1solutions>`.

1. Implement the Dictionary ADT. Your dictionary should hold key-value pairs. Looking up a key in the dictionary should return the value or "not found". You will use your Dicitonary ADT to implement your symbol table that contains variable bindings. Your dictionary should have $O(\log n)$ lookup, in either expectation (e.g., via hash-table with universal hashing), worst-case (e.g., via AVL tree), or amortized (e.g., via splay tree).

2. Implement an `eval(env)` method on each class in the `SchemeObject` hierarchy to evaluates the object. Evaluate takes one argument, `env`, which is the environment in which the object is being evaluated, i.e., the symbol table. For this task, it may be helpful to temporarily implement the evaluate method on (`SchemePair` a b) to evaluate a to get a' and b to get b' and return (`SchemePair` a' b').

3. Implement an interactive program that reads in scheme expressions, evaluates them in a global dictionary, and prints them out. To make this interesting you should add some variable definitions to the global environment. For instance:

x	100
y	400
z	200
z	200
hello	hi
hal	dave

   Assuming we have followed the advice of Task 2 and implemented pair evaulation to just return a pair with each element of the pair evaluated, the example below gives a sample exeution of the program (the input is underlined):

```
parse> (x (y z) ((x)))
(100 (400 200) ((100)))
parse> x
100
parse> (hello
hal)
(hi dave)
parse>
```

4. Implement (`SchemeNumber` n) and modify your parser to read in scheme numbers.

5. Implement `SchemeFunction` abstract class with virtual `apply(args,env)` where `args` is a list of arguments (represented by a pair) and `env` is the environment. Implement subclasses `SchemeAddition`, `SchemeSubtraction`, and `SchemeMultiplication` that add, subtract, and multiply the contents of the lists that are passed as arguments. Also implement `SchemeLength`, `SchemeList`, and `SchemeMap`. Notice that `apply` should return a `SchemeObject` (actually a pointer to one). Be sure to also write a reasonable `print` method for `SchemeFunction` subclasses.

6. Modify your `eval` method on scheme pairs so that evaluating a list results in evaluating each element of the list and then calling the `apply` method on the first element of the evaluated list with the remaining elements of the evaluated list as arguments. The return result should be the return result of `apply`.

7. Add your scheme functions to the global environment. The names `+`, `-`, `*`, `length`, `list`, and `map` are standard. Ensure that your interactive program now works as follows:

```
parse> (list 1 2 3)
(1 2 3)
parse> (length (list 1 2 3))
3
parse> (* (- 2 3) (+ 4 5))
-9
parse> (map length (list (list 1 2) (list 3) (list 1 2 3)))
(2 1 3)
parse> (map - (list 1 2 3 4))
(-1 -2 -3 -4)
parse>
```

3 Logistics.

This assignment will be graded out of a total of 20 points.

This program will be completed in two parts. The first part is due on Tuesday, 11/25/08 (at midnight) and will be graded out of 10 points. The second part is due on 12/03/08 (at midnight) and will be graded out of 10 points. If you fix any problems with the first part when you turn in the second part you can regain up to half the points lost. For example, you made two mistakes on the first part and lost two points each. If you fix one of the mistakes for the final submission you will get one point back. If you fix both mistakes for the final submission you will get two points back.

Part 1: Complete Tasks 1-3. You should also write and turn in a test program that thoroughly verifies that all tasks for this part are properly completed.

Part 2: Complete Tasks 4-7.

Submitting your code. The T-Lab (Tech F252) is available for your programming needs and the TA will hold extra office hours (TBA) in the T-Lab to assist you. Your programs should compile under `g++`. To submit your program send email the source code by email to the TA. Use the subject line "SUBMIT PROG2 PART1" and "SUBMIT PROG2 PART2" for each respective part. Do not submit executables. If you use a *makefile*, submit the makefile. If you compile your program directly on the command line, specify the command line in your email.

Grading guidelines. You will be graded on your ability to write efficient code. You will be graded on your ability to write reasonable C++ code. You will be graded on your ability to implement the required tasks. You will be graded on your ability to manage memory (i.e., be careful of *memory leaks* and other bugs with memory usage).

Resources. You may consult your text book or other books on C++ and data structures. You may not use the Standard Template Library. You must not copy code from anywhere. You may talk with your classmates about the project at a high level, but your implementations must be 100% original. You may consult with the instructor and TA on any aspect of the project.

4 Example: Evaluation

Suppose we have the following variable bindings in the environment

`x`	100
`y`	400
`z`	200
`*`	"the multiply function"
`+`	"the addition function"
`-`	"the subtraction function"

and we evaluate `(* (+ x 10) (- y 300))`. The following results:

1. `(* (+ x 10) (- y 300))` evaluates by recursively evaluate `*`, `(+ x 10)`, and `(- y 300)`.
   (a) `*` evaluates by looking up the symbol `*` in the environment and getting "the multiply function".
   (b) `(+ x 10)` evaluates by recursively evaluating `+`, `x`, and `10`.
      i. `+` evaluates by looking up the symbol `+` in the environment and getting "the addition function".
      ii. `x` evaluates by looking up the symbol `x` in the environment and getting the number 100.
      iii. `10` evaluates to itself. The result is the number 10.

      Then apply the first to the latter two. I.e., apply the addition function to the numbers 100 and 10 to get the number 110.
   (c) `(- y 300)` evaluates by recursively evaluating `-`, `y`, and `300`.
      i. `-` evaluates by looking up the symbol `-` in the environment and getting "the subtraction function".
      ii. `y` evaluates by looking up the symbol `y` in the environment and getting the number 400.
      iii. `300` evaluates to itself. The result is the number 300.

      Then apply the first to the latter two. I.e., apply the subtraction function to the numbers 400 and 300 to get the number 100.

   Then apply the first to the latter two. I.e., apply the multiplication function to the numbers 110 and 100 to get the number 1100.