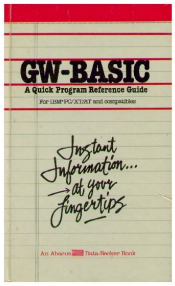
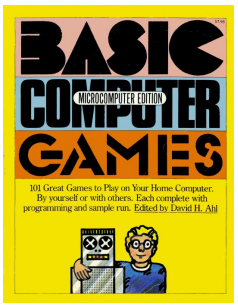
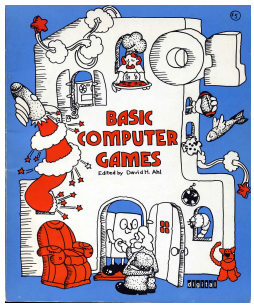
CSE 341, Autumn 2010
Assignment #8 - Scheme: BASIC Interpreter
"Part A" due: Friday, November 26, 2010, 11:30 PM
"Part B" due: Wednesday, December 1, 2010, 11:30 PM

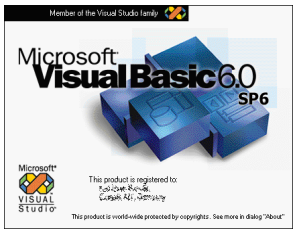
In this assignment you will implement a significant subset of the BASIC programming language. Include your answers in a file named `interpreter.scm`. Supporting code is provided on the course web site.

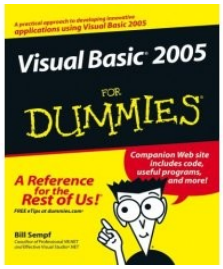
Background Information about the BASIC Language:












```
10 LET X = 15.4
20 LET Y = 78.3
30 GOSUB 90
40 LET X = 19.8
50 LET Y = 82.3
60 GOSUB 90
70 PRINT "ALL DONE NOW"
80 GOTO 140
90 PRINT "X =", X
100 PRINT "Y =", Y
110 LET D = SQR(X*X + Y*Y)
120 PRINT "DISTANCE FROM ORIGIN =", D
130 RETURN
140 END
```

BASIC (Beginner's All-purpose Symbolic Instruction Code) is a programming language created in 1964 by two faculty of Dartmouth University. At the time of BASIC's creation, most commonly used programming languages were considered difficult to learn and had complex and confusing syntax. Unlike other popular languages of the time, BASIC was specifically designed to be easy to read, simple (yet powerful enough to create complex programs), heavily interactive, provide (relatively) friendly error messages, shield the users from the details of the hardware and operating system, and able to run quickly enough that small programs would see a fast response time.

BASIC became extremely popular during the "personal computer" era in the late 1970s and 1980s as people started purchasing computers for their own homes. Many well-known machines such as the original IBM PC, the Apple II, the TRS-80, and the Commodore 64 shipped with a version of BASIC pre-installed and several BASIC programs to run.

Dialects and children of BASIC are still in wide use today. Perhaps the most common offshoot of BASIC is Microsoft's Visual Basic, created by that company for use in developing simple graphical applications for the Windows operating system. The most modern version, Visual Basic .NET, sees widespread usage as part of Microsoft's successful .NET Framework. Dialects of BASIC are also still used today as macro languages embedded within larger applications; for example, the Microsoft Office and OpenOffice applications can be automated using "VBScript" and other variations.

BASIC has been an important and influential programming language, but not everyone lauds it as a positive achievement in the history of programming. Famed computer scientist Edsger Dijkstra was well-known to despise BASIC, once writing an influential 1968 ACM SIGCSE conference paper titled, *"GOTO Considered Harmful."* Among other quotes, Dijkstra has stated of the language, "the teaching of BASIC should be rated as a criminal offense: it mutilates the mind beyond recovery," and, "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

If you are curious, the original 1964 Dartmouth report on BASIC is still available here:

- http://www.bitsavers.org/pdf/dartmouth/BASIC_Oct64.pdf

Provided Code:

Your BASIC interpreter will build on the parsing procedures of Homework 7. You will be provided with a set of utilities in `hw8-support.scm` with code for handling the top-level user interaction. The support file also contains a procedure called `tokenize` that will convert user input into lists of tokens. Your program should begin with the following calls:

```
(include "parser.scm")
(include "hw8-support.scm")
```

A working solution to Homework 7 will be provided to all students once the final HW7 turnin deadline has passed.

The support code includes the following helper procedures *(you do not implement these; they are provided to you)*:

- `(variable?` ***expr***`)`: Returns `#t` if the given expression is a symbol that could be a valid BASIC variable name.
- `(try-parse-expression` ***list n***`)`: Attempts to call your HW7 parser's `parse-expression` procedure and return its result. If there is any error thrown by `parse-expression`, this procedure will throw a new error with the given line number ***n*** in front of it.
- `(try-parse-test` ***list n***`)`: Much like `try-parse-expression`, but instead calls your HW7 parser's `parse-test` procedure to parse a logical test and return its result.
- `debug`: A global variable that can be used for debugging purposes. Initially it is set to `null`, but it can be modified by your code or other code to store another value. In particular, every time your `run-program` procedure is called, the `debug` variable is set to store the parameter that was passed to it.
- `(main-loop)`: Executes the main interpreter shell. From here you can execute the following provided commands (most of which come from the original BASIC specification):
  - `LIST`: Prints the contents of the given program's source code to the console.
  - `NEW`: Erases the current program and prompts the user to enter a filename into which the new current program's source code can later be saved.
  - `OLD` ***filename***: Loads the source code from the given filename into the interpreter.
  - `RENAME`: Prompts to enter a new file name to save to (only takes effect for future calls on `SAVE` or `UNSAVE`).
  - `RENUMBER`: Re-numbers the lines of the current program to be ascending multiples of 10. Useful if the programmer has been forced to insert new code between existing lines of code.
  - `RUN`: Runs the currently loaded BASIC program using your interpreter, by calling the `run-program` procedure that you'll write, on the interpreter's current program.
  - `SAVE`: Saves the current program to `program.bas`, or the filename most recently passed to `NEW` or `RENAME`.
  - `STOP` : Halts the interpreter and exits the program.
  - `UNSAVE`: Re-reads the current filename's source code into the interpreter.

For the commands that involve specifying a file name (`NEW`, `OLD`, `RENAME`), you can either type the command and hit enter or you can type the command followed by the file name.

A quick way to test your interpreter is to run your code in DrScheme, then from the Interactions window, call `(main-loop)`. From there, assuming you've saved the supporting files, type something like `OLD program1.bas`, then `RUN`.

In addition, the user can enter lines of the program, which will all begin with a line number. They can be entered in any order. The user can also replace a line. To delete a line, the user should type the line number followed by enter.

The supporting code guarantees certain invariants for the program variable passed as a parameter to `run-program`:

- There will be exactly one legal `END` statement (described later), and it will be the last line in the program.
- No line will be empty.
- Line numbers do not necessarily increase in increments of 10, but will be positive integers in increasing order.

Implementation:

`run-program`

The primary task for this assignment is to write a procedure called **`run-program`**. This procedure gets called by the interpreter support code whenever a program has been typed or loaded and then the `RUN` command is issued. Your `run-program` procedure will be passed a **list of BASIC instructions** to execute, with one of the list for each line in the original source code file. Each element of the overall program list is itself a list that contains two parts: a line number and a list of tokens for the input line. For example, the BASIC program below at left:

```
10 LET X = 203.4 * 37.7
20 PRINT X
30 LET X = X/2
40 IF X>200 THEN 20
50 END
```

Would be represented in list form as follows:

```
((10 (LET X = 203.4 * 37.7))
 (20 (PRINT X))
 (30 (LET X = X / 2))
 (40 (IF X > 200 THEN 20))
 (50 (END)))
```

In other words, if a variable `lst` stores the program at left, then:

- `(car lst)` is the list `(10 (LET X = 203.4 * 37.7))`
- The `car` of the above is the integer `10`
- The `cdr` of the above is a list `((LET X = 203.4 * 37.7))`
- The `car` of that is a list `(LET X = 203.4 * 37.7)`
- The `car` of that is the symbol `LET`
- The `cdr` of that is a list `(X = 203.4 * 37.7)`
- etc...

You will want to write many **helper procedures** that are called by `run-program`. (For example, consider a procedure to process each kind of statement in the following section.) You are allowed to define these in the top-level environment.

Your `run-program` procedure (and any other helpers it invokes) should properly implement the BASIC commands described on the next several pages. They are listed in the order that we suggest you implement them.

The descriptions of each command will indicate that you should generate various specific **error** messages. That means that you should call the `error` procedure with the given text. In addition, unless otherwise specified, your error string must include the line number on which the error occurred, following this exact format:

```
LINE 30: ILLEGAL FOR
```

You are not required to handle every potential error; this spec lists all of the errors you are required to handle. You can include extra error cases if you like, but that is optional.

Please note that your run-program procedure could be called more than once during a given execution of the interpreter. Your interpreter should behave properly for all executed programs, not just the first one. This means that if you store any global state while the interpreter is running, you'll need to clear out that state if `run-program` is ever called again later.

BASIC Statements to Implement:

1. `REM`

The `REM` command is for comments (short for "remarks"). The general form is:

```
REM text
```

A `REM` command does nothing when executed. (Implementing `REM` is not the hard part of the assignment.)

2. `END`

The `END` command terminates a program. The general form is:

```
END
```

Program execution should proceed sequentially from the first line of the program until it encounters the `END` command. Your interpreter should produce this message when it encounters the `END` command and finds no errors to report:

```
PROGRAM TERMINATED
```

The `hw8-support` guarantees that there will be exactly one `END` statement, and it will be the *last* line of the program.

3. `LET`

The `LET` command assigns a value to a variable. The general form is:

`LET` ***variable*** `=` ***expression***

The expression will always have a **numeric** result. The `LET` command can be used either to define a new variable or to assign a new value to a variable that was previously defined.

The expression might include references to other variables, such as `LET Y = X + 3`. It is your interpreter's job to look over the entire expression and **substitute** every referenced variable's current value in place of its name. So, for example, in the preceding code, if `X` stores the value `7`, you should change the expression from `(X + 3)` to `(10 + 3)` before trying to call `parse-expression` on it.

You should make sure that `LET` is followed by a **legal variable name** which is followed by an equals sign. Variable names in BASIC are very limited. They can be one letter long or one letter followed by one digit. The `hw8-support` file includes a predicate called `variable?` that can be used to test whether a variable name is legal in BASIC.

If the line begins with these three elements (variable, equals sign, expression), you should use the code from your HW7 parser to parse the expression. You will want to call the `parse-expression` procedure of HW7, but see below regarding errors first. If the expression is legal, you will need to remember its resulting value for the given variable.

Keeping track of variables:

Most compilers and interpreters contain a data structure called a *symbol table* that stores a mapping from all currently defined variables to information about those variables. For example, a compiler might map from each variable's name to its type, and an interpreter might map from each variable's name to its value. Symbol tables are often implemented as maps or association lists that are mutated as the interpreter runs. See the lecture notes about parsing and memoization for more information about Scheme's syntax for association lists.

As new variables are declared, they can be added to the symbol table. When a variable is given a new value, you do not need to remove the existing mapping from the symbol table; since Scheme's `assoc` procedure grabs the first mapping for a given key, you can simply re-add a second pair for the same variable to the list. As long as it comes earlier in the association list than the previous mapping, it will be used for future `assoc` calls; the old mapping is effectively unused.

(In this section and the following sections, there are various error(s) that can occur in the statement. We will list the errors that you must handle for each statement, along with the error message your interpreter should emit from the `error` procedure. In all cases, it is implied that the error message should be preceded by the current line number. For example, if we say that a particular error message should be `ILLEGAL FOO`, and such an error occurs on line 42, the exact actual error message text to emit is: `LINE 42: ILLEGAL FOO` .)

`LET` errors:

- If the `LET` is not followed by a legal variable name and equals sign, generate the error message: `ILLEGAL LET`
- If the expression is followed by extraneous text, generate: `EXTRANEOUS TEXT AFTER LET EXPRESSION`
- If the expression itself is illegal by the rules of BASIC syntax, generate: `ILLEGAL EXPRESSION`
    - To parse expressions, you will want to call the `parse-expression` procedure of Homework 7, but to do so, you should instead call the `try-parse-expression` procedure provided in the `hw8-support` file. This version takes an extra parameter that specifies the line number, so that if an error occurs, it will be reported with the line number, such as: `LINE 55: ILLEGAL EXPRESSION`
- If the expression refers to a variable that has not been defined, generate: `ILLEGAL EXPRESSION`

4. INPUT

The INPUT command **reads a number from the console** and stores it in a variable. The general form is:

```
INPUT variable
```

This command wasn't part of the original BASIC specification, but it is obviously very useful. If the variable's name is legal, you should call Scheme's `read` procedure to read the number (it takes no arguments, as in `(read)`). You should make sure that the input you get is either an integer or a real number. If so, assign the given variable that value.

INPUT errors:

- If the variable is missing or if the variable name is not legal (recall the predicate `variable?`) or if there is extraneous text after the variable name, generate the error message: `ILLEGAL INPUT COMMAND`
- If the input is not an integer nor a real number, generate the error message: `INPUT MUST BE A NUMBER`

5. PRINT

The PRINT command prints a sequence of expressions and strings separated by commas. The general form is:

```
PRINT (expression | string) {"," (expression | string)}
```

Commas will be represented with the token `comma`, as in: `PRINT "x =" comma 7.8` . Each string should be displayed and each expression should be evaluated and displayed (the `display` procedure works for both strings and numbers). Values should be separated by a space. For this general form of the PRINT command, there should be no extraneous space on the end of the line and each PRINT command should produce a complete line of output (use the Scheme `newline` procedure). The expressions are guaranteed to be numeric, although they might refer to variables. As with LET, you must substitute in the values of any such variables. Call `try-parse-expression` to parse the expressions.

There are **special cases** of PRINT to handle. A PRINT call might appear on a line by itself; if so, produce a blank line. The final expression might be followed by a comma that has nothing after it. That's BASIC's way to request the equivalent of a `System.out.print` versus `println`. In other words, do not call `newline` to complete that line of output; instead, terminate the line with a space. For example, if you execute these lines of code:

```
10 PRINT 2 + 2,
20 PRINT 3,
30 PRINT 42, 15
```

The result would be a single line of output with a single space separating the items and no trailing space at the end:

```
4 3 42 15
```

PRINT errors:

- If a legal expression is followed by something other than `comma`, generate the error message: `ILLEGAL PRINT`
- If an expression is missing or otherwise invalid (two commas in a row, etc.), generate: `ILLEGAL EXPRESSION`

6. GOTO

The GOTO command transfers control to another part of the program. The general form is:

```
GOTO lineNumber
```

A GOTO command transfers control to the given line number. The line number will be a simple integer (not an expression).

This is the first of several "jumping" statements that break the sequential flow of the program by advancing the interpreter to a different line. You will need to think about what state you must keep track of, in order to be able to access and jump to any other line number on command. This state will also be useful for other statements to be described in a moment.

GOTO errors:

- If the line number does not exist, generate the error message: `NO SUCH LINE NUMBER`

- If the `GOTO` is followed by a non-integer or has extraneous text after the integer, generate: `ILLEGAL GOTO`

7. `IF`

The `IF` command is for conditionally transferring control. The general form is:

`IF` ***test*** `THEN` ***lineNumber***

The test will be of the form described in the Homework 7 grammar's <*test*> rule. The line number will be a simple integer. If the test evaluates to `#t`, control should be transferred to the given line number. Otherwise it should proceed to the next statement, as usual.

To parse the test, you will want to call the `parse-test` procedure of Homework 7, but to do so, you should instead call the `try-parse-test` procedure provided in the `hw8-support` file. This version takes an extra parameter that specifies the line number, so that if an error occurs, it will be reported with the line number, such as: `LINE 55: ILLEGAL TEST.`

`IF` errors:

- If the word `THEN` is missing or if it is not followed by an integer or if the integer has extraneous text after it, you should generate the error message: `ILLEGAL IF`
- If the program attempts to transfer to a line number that does not exist, you should generate the error message: `NO SUCH LINE NUMBER`

8. `GOSUB/RETURN`

The `GOSUB` command transfers control to a subroutine (which is like a procedure; a section of code that is terminated with a `RETURN` command). The general form is:

`GOSUB` ***lineNumber***

The `GOSUB` command is similar to `GOTO` in that it transfers control to another part of the program. But in the case of `GOSUB`, a subsequent call on `RETURN` will return the program back to the statement after the call on `GOSUB`. The basic form of the `RETURN` command is:

`RETURN`

BASIC supports only a single subroutine execution at any given time. It is not legal for one subroutine to call another.

You will need a way to keep track of places in the code to "jump" back to at a later point in time. This may require you to keep track of various important information in some global variables. Many real interpreters maintain a "call stack" of data about all the functions that are currently being executed. But since our version of BASIC does not allow more than one subroutine to be invoked at any given time, the state you keep track of can be much simpler.

`GOSUB` / `RETURN` errors:

- If the `GOSUB` command is not followed by an integer or if it has extraneous text after the integer, you should generate the error message: `ILLEGAL GOSUB`
- If the `RETURN` has extraneous text after it, you should generate the error message: `ILLEGAL RETURN`
- If the line number mentioned in `GOSUB` does not exist, generate the error message: `NO SUCH LINE NUMBER`
- If the program reaches an `END` while a subroutine execution is active (i.e., without having encountered a `RETURN` statement), generate the error message: `MISSING RETURN` , with the line number of the `END` command.
- If the program attempts to execute two `GOSUB` commands in a row without a call on `RETURN` in between, your program should generate the error message: `ALREADY IN SUBROUTINE`
- If the program attempts to execute a `RETURN` command when there is no active subroutine execution, your program should also generate the error message: `NOT IN SUBROUTINE`

9. FOR/NEXT

The FOR command is used for looping. *(This is the most difficult statement to implement!)* The general form is:

FOR ***variable*** = ***expression*** TO ***expression***

The expressions represent the starting and ending values for the loop. The expressions must be numerical, but they need not be integers. Each FOR command must be paired with a corresponding NEXT command using the same variable name:

NEXT ***variable***

When the NEXT command is encountered, you examine the value obtained by incrementing the loop variable (adding 1 to it). If that value is less than or equal to the final value for the loop, then you set the variable to that value and go back to the top of the loop. If the value after incrementing would be larger than the final value, then you leave the variable unchanged and transfer control to the statement that comes after the NEXT command.

For example, the loop below produces the output at right:

```
100 FOR X = 0.5 TO 5
110 PRINT X
120 NEXT X
130 PRINT "AFTER LOOP, X =", X
```

```
 0.5
 1.5
 2.5
 3.5
 4.5
AFTER LOOP, X =  4.5
```

Notice that the loop stops when X becomes 4.5 because incrementing it again would cause it to be greater than the final value of 5. Also notice that X has the value 4.5 after the loop.

The BASIC standard says that if the starting value for a loop is greater than the ending value, then the body of the loop should be skipped and afterwards the variable should be one less than the starting value. For example:

```
10 FOR X = 5 to 3
20 PRINT X
30 NEXT X
40 PRINT X
```

The body of the loop at left is not executed (the PRINT in line 20) and after the loop the value of X is reported to be 4 (one less than the starting value).

Loops can be **nested**. For example:

```
100 FOR X = 1 TO 3
110 FOR Y = 1 TO X
120 PRINT X, Y, X + Y
130 NEXT Y
140 NEXT X
```

Loops are required to be properly nested. For example, in the code above you would not be allowed to switch the order of lines 130 and 140 because then we would encounter the NEXT command for the outer loop before encountering the NEXT command for the inner loop.

The BASIC specification describes a loop in terms of the FOR command that begins the loop, the NEXT command that ends the loop and the body of commands that appear in between. The **nesting** requirement says that if one FOR/NEXT loop is going to appear in the body of another, then the nested loop must appear in its entirety in the body of the outer loop (i.e., the FOR command, the body of the inner loop and the NEXT command).

You will need to keep track of some state about any current for loop(s) that are active, their associated variables, the relevant line number(s), etc. Remember that for loops can be nested, so you may need to keep track of multiple states.

There are some unusual cases that can arise if a programmer calls GOTO or GOSUB inside of a loop. In general, a programmer should not try to escape a loop with a GOTO command. A call on GOSUB can be okay, although there are some odd cases (for example, what if the GOSUB code includes a NEXT command for the original loop?). In general, we aren't going to worry about many of these subtle errors. Your interpreter's behavior under such cases is unspecified.

Call `try-parse-expression` to parse the two expressions, substituting any variables' names for their values.

FOR errors:

- If FOR is not followed by a legal variable name or if the variable is not followed by =, generate: ILLEGAL FOR
- If the first expression is not followed by TO, generate the error message: MISSING TO
- If there is extra text after the second expression, generate: EXTRANEOUS TEXT AFTER ENDING EXPRESSION

FOR errors (continued):

- The variable mentioned in a NEXT command should match the variable from the most recently encountered FOR command. If it doesn't, you should generate the error message: VARIABLE IN NEXT DOESN'T MATCH
- If a program uses the same variable in both an inner and outer loop, generate: ILLEGAL NESTED LOOP
- If a NEXT command is not followed by a legal variable name and nothing else, generate: ILLEGAL NEXT
- If a NEXT command does not have a corresponding FOR, generate the error message: NEXT WITHOUT FOR
- If you reach the END command without completing all loops (this will include catching the case where there is a FOR without a corresponding NEXT), generate the error message: MISSING NEXT

Miscellaneous

There is one last kind of error you must handle that is not related to any one particular procedure:

- If your program encounters an illegal statement (a line whose first token is not any of the statements mentioned previously), you should generate the error message: ILLEGAL COMMAND

Other than the specific error handling described in this spec, you may assume that the program you are provided is legal.

Milestones:

For the first milestone of this project, **"Part A,"** you must complete only the LET statement and the single-expression PRINT statement (without commas). In other words, a program such as the following should run and output 42 and 3:

```
10 LET X = 6 * 7
20 PRINT X
30 PRINT 1 + 2
40 END
```

(You may want to handle REM and END while you're at it.) You don't need to do any error checking for Part A. Part A will be worth fewer points and will not be graded on internal correctness, only on whether it passes a few short test cases.

That does not sound like very much work, but setting up your overall assignment infrastructure, helper functions, parsing expressions, etc., is still a sizable challenge. Be careful not to put this off until the last minute!

Part B is due later and must include *all* of the previously specified BASIC syntax, including error checking. Part B is worth the bulk of the points and will be graded on both external and internal correctness as specified below.

It is fine if your Part A code implements *more* than LET and single-expression PRINT. Any other features besides those two won't be tested. If you want to get a more aggressive head start on the assignment, try to implement all of the statements other than the FOR loop and some error checking for your first milestone.

Each of the two parts of the assignment are considered indpendently with regards to lateness. If you turn in Part A late, you use late days on it; if you turn in Part B late, you use late days there. Or if you're out, deductions apply. And so on.

Suggested Development Strategy:

THIS PROGRAM IS HARD. It is by a wide margin the most difficult program you will write in this course. Do not underestimate the challenge; start early, debug often, ask for help, get stuck, get unstuck, and so on.

The most difficult part of the assignment is the `for` loop. The "jumping" commands such as GOTO and GOSUB / RETURN can also be tricky. If you create global variables to keep track of important state for these commands, frequently `display` the values of those global variables to make sure that their contents are what you expect them to be.

Recall that you can use the **DrScheme debugger** to step through the execution of your various procedures to find bugs. You can also insert calls to the `display` procedure to view printed output on the console while your code is running. **We strongly recommend lots and lots and lots of calls to `display` while you are developing your procedures.** For example, many procedures accept a complex list of various symbols and tokens. If you're pulling apart that list, try to display the various pieces before you actually process them, to make sure the pieces you've pulled apart are what you

think they are. In our own solution, we had to do this liberally to find bugs and get the interpreter working. Remember that `display` can print almost any kind of value, including an entire list.

The supporting code contains a global variable called `debug` that can be helpful for debugging purposes. After you give a `STOP` command to exit the interpreter, you can use the variable `debug` to access the last program you were running (i.e., the last value passed to your `run-program` procedure).

If you produce any **testing code** for this assignment, you are welcome to share it with your classmates using our course discussion forum. Shared testing code may test external behavior but may not test any aspects of the internal correctness of the Scheme source code. The instructor and TAs reserve the right to remove testing code that we feel is inappropriate.

Grading and Submission:

Submit your finished `interpreter.scm` file electronically using the link on the class web page. For reference, our solution is roughly 176 "substantive" lines long according to the course Indenter page. You don't need to match this number or even come close to it; it is just a rough guideline.

Your program should not produce syntax or runtime **errors** when executed. You may lose points if you name your procedures or top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own **testing**, and remember to test edge cases.

On this assignment you are allowed to use Scheme's language features that involve **mutation**, such as the `set!`, `set-car!`, or `mcons` procedures, though you should do so sparingly. Your code should compile and execute properly using the **"Pretty Big"** language level of PLT Scheme. Otherwise you may use any Scheme library constructs you like to help you solve a problem unless those constructs are explicitly forbidden by this spec or the problem description.

As always, if the solution to one procedure is useful in helping you solve a later procedure, you should call the earlier procedure from the later procedure. You can include any testing code you develop (although it should be labeled as such).

You may define other **global variables or procedures**, as long as they are non-trivial and are used by more than one top-level procedure from this document. It is helpful to declare a small number of global variables, although you don't want to overuse this. You are allowed to declare up to **7 global variables** in your solution (our solution code has 5 globals).

If you write inner helper procedures, choose a suitable set of **parameters** for each one. Don't pass unnecessary parameters or parameters that are unmodified duplicates of existing bound parameters from the outer procedure.

The lists of tokens passed to your various procedures often contain various **symbols**. You should process and handle these symbols without converting them into strings first. (Treating symbols as strings everywhere is bad Scheme Zen.)

You are expected to use good programming **style**, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines. Recall that DrScheme is able to auto-indent your entire program for you simply by selecting code and pressing the Tab key.

Place a descriptive **comment** header at the top of your program along with a comment header on each procedure, including a description of the meaning of its parameters, its behavior/result, and any preconditions it assumes. If you declare a non-trivial or non-obvious inner helper procedure, also briefly comment the purpose of that helper in a similar fashion. Since the procedures in this assignment are larger and more elaborate, you should put comments within the procedures explaining the complex code, such as which BASIC syntax a particular part of the code is processing, etc. All comments in your program must be written in your own words, not copied directly from this document or elsewhere.

Efficiency on a fine level of detail is not crucial on this assignment. But code that is unnecessarily computationally inefficient (such as code that performs an algorithm that should be $O(n)$ in $O(n^2 \log n)$ time) might receive a deduction.

Redundancy, such as recomputing a value unnecessarily or unneeded recursive cases, should be avoided. This assignment involves a lot of similar but not identical code, such as checking whether a given expression matches a particular pattern, then breaking apart the list to process it based on that pattern. Intelligently utilize helper procedures to capture common code to avoid redundancy. Avoid repeating large common subexpressions as much as possible; for example, the pseudo-code *"if (test) then (1 + really long expression) else (2 + really long expression)"* is better written as *"((if test then 1 else 2) + really long expression)"*.