

1 Preliminaries

Before you start the Scheme interpreter, an environment variable needs to be set so that the Scheme on the Suns will be compatible with the version used in the textbook. You can set this environment variable by issuing the command

```
% setenv MITScheme_BAND sicp.bin
```

before you start Scheme.

In order to have this variable set automatically each time you login, use your favorite editor and append the following line to the end of your `.cshrc` file located in your home directory.

```
setenv MITScheme_BAND sicp.bin
```

The variable `MITScheme_BAND` will now be set properly every time you log in. (If this command was not in your `.cshrc` file when you started your current log-in session, you will need to issue the command by hand as shown above.)

Scheme can be invoked simply by issuing the command

```
% scheme
```

However, we recommend that you use GNU Emacs as an interface to the Scheme interpreter (as explained later in this problem set). To start the system, type `M-x run-scheme`. Use the Emacs help system (`C-h m`) or consult the *MIT Scheme User's Manual* to find more details on using the Scheme interpreter.

There are two different Emacs modes for interacting with Scheme. To make sure that you are using the correct mode, you should have the following lines in your `.emacs` file:

```
(autoload 'scheme-mode "xscheme"  
  "Major mode for Scheme." t)  
(autoload 'run-scheme "xscheme"  
  "Switch to interactive Scheme buffer." t)
```

If you do not have a `.emacs` file yet, there is a rather comprehensive one that you can copy to your home directory using

```
% cp /usr/local/courses/cse/cse233.01/etc/DotEmacs ~/.emacs
```

Note that this file has customizations for Scheme, VM mail, Gnus, and W3 (the mail, news, and web browsing packages in Emacs). If you have questions about what any of it does, ask the instructor or one of the TAs.

Though the examples shown are often indented and printed on several lines for readability, an expression may be typed on a single line or on several, and redundant spaces and carriage returns are ignored. It is to your advantage to format your work so that you (and others) can easily read it.

When you make your subdirectories for the homework assignments and do your scheme programs, it would be best to stay with the specified names for the files and procedures. This will make it easier on the TAs when you have questions in lab, as well as, help in the grading of your assignments.

2 Homework exercises

Reading: From text, Chapter 1, section 1.1.

The purpose of the exercises below is to familiarize you with the basics of the Scheme language and the Scheme interpreter. Spending a little time on simple mechanics now will save you a *great* deal of time over the rest of the semester.

Write up and turn in the following exercises:

1. Evaluation of Expressions

Below is a sequence of expressions. What is the result you would expect from a scheme interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented. Do this *without* using the Scheme interpreter!

```
==> (- 8 9)
```

```
==> (> 3.7 4.4)
```

```
==> (- (if (> 3 4) 7 10) (/ 16 10))
```

```
==> (define b 13)
```

```
==> 13
```

```
==> b
```

```
==> (define square (lambda (x) (* x x)))
```

Equivalent to (define (square x) (* x x))

```
==> square
```

```
==> (square 13)
```

```
==> (square b)
```

```
==> (square (square (/ b 1.3)))
```

```
==> (define multiply-by-itself square)
```

```
==> (multiply-by-itself b)
```

```
==> (define quad (lambda (x)
  (square (square x))))
```

```
==> quad
```

```
==> (define a b)
```

```
==> (= a b)
```

```
==> (quad (/ a 1.3))
```

```
==> (if (= (* b a) (square 13))
  (< a b)
  (- a b))
```

```
==> (cond ((>= a 2) b)
  ((< (square b) (multiply-by-itself a)) (/ 1 0))
  (else (abs (- (square a) b))))
```

2. Exercise 1.4 (from the text)

3 Getting started with the Scheme Interpreter

For this course, it is *highly* recommended that you interface to Scheme through GNU Emacs. Even if you are already familiar with Emacs, you should take some time *now* to run the Emacs tutorial. This can be invoked by typing `C-h` followed by `t`. You will probably get bored before you finish the tutorial. (It's too long, anyway.) But at least skim all the topics so you know what's there. You'll need to gain reasonable facility with the editor in order to complete the exercises below. Being comfortable with the editor will greatly improve your productivity in this and future problem sets (i.e., it will greatly cut down the amount of time you spend working on problem sets) — invest a little time now to learn it.

3.1 Evaluating expressions

After you have learned something about Emacs, go to the Scheme buffer (i.e., the buffer named `*scheme*`).¹ As with any buffer, you can type Scheme expressions, or any other text into this buffer. What distinguishes the Scheme buffer from other buffers is the fact that underlying this buffer is a Scheme evaluator, which you can ask to evaluate expressions, as explained in the section of the tutorial entitled “Evaluating Scheme expressions.”

Type in and evaluate (one by one) the expressions from Section 2 of this assignment to see how well you predicted what the system would print. If the system gives a response that you do not understand, ask for help from a TA or from another student in the course. *Read the rest of this problem set before attempting to type expressions into Scheme via Emacs so that you understand how to get Emacs to interpret your expressions.*

Observe that some of the examples printed above in Section 2 are indented and displayed over several lines for readability. An expression may be typed on a single line or on several lines; the Scheme interpreter ignores redundant spaces and carriage returns. It is to your advantage to format your work so that you (and others) can read it easily. It is also helpful in detecting errors introduced by incorrectly placed parentheses. For example the two expressions

```
( * 5 ( - 2 ( / 4 2 ) ( / 8 3 ) ) )
```

```
( * 5 ( - 2 ( / 4 2 ) ) ( / 8 3 ) )
```

look deceptively similar but have different values. Properly indented, however, the difference is obvious.

```
( * 5
  ( - 2
    ( / 4 2 )
    ( / 8 3 ) ) )
```

```
( * 5
  ( - 2
    ( / 4 2 ) )
  ( / 8 3 ) )
```

Emacs provides several commands that help you “pretty-print” your code, e.g., indenting lines to reflect the inherent structure of the Scheme expressions.²

3.2 Creating a file

Since the Scheme buffer will chronologically list all the expressions you evaluate, and since you will generally have to try more than one version of the same procedure as part of the coding and debugging process, it is usually better to keep your procedure definitions in a separate buffer, rather than to work only in the Scheme buffer. You can save this other buffer in a file on your disk so you can split your lab work over more than one session.

The basic idea is to create another buffer, into which you can type your code, and later save that buffer in a disk file. You do this by typing `C-x C-f filename`. If you already have a buffer open for that file Emacs simply switches you into this buffer. Otherwise, Emacs will create a new buffer for the file. If you give the system a name that has

¹If you don't know how to do this, go back and learn more about Emacs.

²Make a habit of typing `C-j` at the end of a line, instead of `enter`, when you enter Scheme expressions, so that the cursor will automatically indent to the right place.

not yet been used, with an extension of `.scm`, Emacs will automatically create a new buffer with that file name, in scheme mode. In a Scheme-mode buffer some editing commands treat text as code, for example, typing `C-j` at the end of a line will move you to the next line with appropriate indentation.

Once you are ready to transfer your procedures to Scheme, after you have `M-x run-scheme`, you can use any of several commands: `M-z` to evaluate the current definition, `C-x C-e` to evaluate the expression preceding the cursor, `M-o` to evaluate the entire buffer, or using `M-x eval-region` to evaluate a “marked” region in the buffer. Each of these commands will cause the Scheme evaluator to evaluate the appropriate set of expressions (usually definitions). By returning to the `*scheme*` buffer, you can now use these expressions.

To practice these ideas, create a new buffer, called `ps1-ans.scm`. In this buffer, create a definition for a simple procedure, by typing in the following (verbatim):

```
(define square (lambda (x) (*x x)))
```

Now evaluate this definition by using either `M-z` or `M-o`. Go back to the Scheme buffer, and try evaluating `(square 4)`.

If you actually typed in the definition *exactly* as printed above, you should have hit an error at this point. The system will now be offering you a chance to enter the debugger. For now, type `C-c C-c` to quit the evaluation (we’ll see how to use the debugger below). Go back to the `ps1-ans.scm` buffer and edit the definition to insert a space between `*` and `x`. Re-evaluate the definition, and return to Scheme and try evaluating `(square 4)` again.

As a second method of evaluating expressions, try the following. Go to the `*scheme*` buffer, and again type in:

```
(define square (lambda (x) (*x x)))
```

Place the cursor at the end of the line, and type `C-x C-e` to evaluate this expression. Again try `(square 4)`. When it fails, again use `C-c C-c` to quit out of the evaluation. This time, you should type `M-p` several times, until the definition of `square` that you typed in appears on the screen. Edit this definition to insert a space between `*` and `x`, evaluate the new expression, and use `M-p` to get back the expression `(square 4)`. Make a habit of using `M-p`, rather than going back and editing previous expressions in the Scheme buffer in place. That way, the buffer will contain an intact record of your work.

4 Adventures in Debugging

During the semester, you will often need to debug programs that you write. This section contains an exercise that you should work through *at the terminal* to acquaint you with some of the features of Scheme to aid in debugging. There is nothing you need turn in for this part of the problem set, but we strongly suggest that you do this exercise. Learning to use the debugging features will save you much grief on later problem sets.

The file `ps1-debug.scm` is a short file that has been provided for you in the directory

```
/usr/local/courses/cse/cse233.01/homework/ps1
```

Copy this file to your own directory preferably a `cse233/homework/ps1` subdirectory. The file contains three tiny procedures called `p1`, `p2` and `p3`. You can examine them using a text editor, but there is no need to do that now – you will see them later on, during debugging. Start Scheme and load the file. Now evaluate the expression `(p1 1 2)`. This should signal an error, with the message

```
;The procedure #[compound-procedure 4 p2] has been called with 1 argument
; it requires exactly 2 arguments.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

Don’t panic. Beginners have a tendency when they hit an error to quickly type `C-g` and return to the editor. Then they stare at their code until they see what the bug is. Indeed, the example here is simple enough so that you probably

can see the bug just by reading the code. But don't do this. Let's instead see how Scheme can be coaxed into producing some helpful information about the error.

First of all, there is the error message itself. It tells you that the error was caused by a procedure being called with 1 argument, which is the incorrect number. The next line tells you that the error occurred within the procedure *P2*. The third line tells you that *P2* requires a minimum of 2 arguments.

Notice, that the prompt now reads

```
2 error>
```

which is Scheme's way of indicating that you can now evaluate expressions within the context of the error, and that this context is "at level 2," namely, one level down from the "top level," which is the initial Scheme command level.

Unfortunately, the error message alone doesn't say where in the code the error occurred. In order to find out more, you need to use the debugger. To do this execute the expression (*debug*). The debugger allows you to grovel around examining pieces of the execution in progress in order to learn more about what may have caused the error. When you typed (*debug*) just above, Scheme should have responded:

There are 6 subproblems on the stack.

```
Subproblem level: 0 (this is the lowest subproblem level)
```

```
Expression (from stack):
```

```
(#[compound-procedure 4 p2] 2)
```

```
There is no current environment.
```

```
The execution history for this subproblem contains 1 reduction.
```

```
You are now in the debugger. Type q to quit, ? for commands.
```

```
3 debug>
```

This says that the expression that caused the error was the evaluation of the compound procedure *p2* called with the single argument 2. That's not much more information than what was given by the error message. Let's use the debugger to get some more information.

The debugger differs from an ordinary Scheme command level in that you use single-keystroke commands rather than typing expressions. One thing you can do is move "backwards" in the evaluation sequence to see how we got to the point that signaled the error. To do this, type the character *b*. Scheme should respond:

```
Now using information from the execution history.
```

```
Subproblem level: 0 Reduction number: 0
```

```
Expression (from execution history):
```

```
(p2 b)
```

```
Environment created by the procedure: P3
```

```
applied to: (1 2)
```

```
3 debug>
```

This says that the expression that caused the error was (*p2 b*), evaluated within the procedure *p3*, which was called with the arguments 1 and 2. That's probably enough information to let you find and fix the error, but let's move back one more step in the evaluation.

Press *b* again, and you should see

```
No more reductions; going to the next (less recent) subproblem.
```

```
Subproblem level: 1 Reduction number: 0
```

```
Expression (from execution history):
```

```
(+ (p2 a) (p2 b))
```

```
Environment created by the procedure: P3
```

```
applied to: (1 2)
```

```
3 debug>
```

Remember that the expression being evaluated at the time of the error was $(p2\ b)$. Now that we have moved “back,” we learn that this expression was being evaluated as a subproblem of evaluating the expression

```
(+ (p2 a) (p2 b))
```

which is still within procedure $P3$ applied to the arguments 1 and 2. Press b again and you should see

```
Subproblem level: 1 Reduction number: 1
Expression (from execution history):
  (p3 x y)
Environment created by the procedure: P1
  applied to: (1 2)
```

```
3 debug>
```

which tells us that it got to the place we just saw above as a result of trying to evaluate the expression $(p3\ x\ y)$ in the $p1$ procedure definition. Press b again and you should see

```
No more reductions; going to the next (less recent) subproblem.
Subproblem level: 2 Reduction number: 0
Expression (from execution history):
  (+ (p2 x y) (p3 x y))
Environment created by the procedure: P1
  applied to: (1 2)
```

```
3 debug>
```

This expressions corresponds to the entire definition of the $p1$ procedure. Pressing b again gives us

```
Subproblem level: 2 Reduction number: 1
Expression (from execution history):
  (p1 1 2)
Environment created by a MAKE-ENVIRONMENT special form
  applied to: ()
```

```
3 debug>
```

The expression $(p1\ 1\ 2)$ was the original line that you typed in at the Scheme prompt. But what happens if you press b a few more times? Go ahead and try it. You should see a horrible mess. What you are looking at is some of the guts of the Scheme system – the part shown here is a piece of the interpreter’s read-eval-print program. In general, backing up from any error will eventually land you in the guts of the system. At this point you should stop backing up unless, of course, you want to explore what the system looks like. (Yes: almost all of the system is itself a Scheme program.)

In the debugger, the opposite of b is f , which moves you “forward.” Go forward until the point of the actual error, which is as far as you can go.

Besides b and f , there are about a dozen debugger single-character commands that perform operations at various levels of obscurity. You can see a list of them by typing $?$ at the debugger.

Hopefully, the debugger session has revealed that the bug was in $p3$, where the expression $(+ (p2\ a) (p2\ b))$ calls $p2$ with the wrong number of arguments. Notice that the call that produced the error was $(p2\ b)$, and that $(p2\ a)$ would have also given an error. This indicates that in this case Scheme was evaluating the arguments to $+$ in right-to-left order, which is something you may not have expected. You should never write code that depends for its correct execution on the order of evaluation of the arguments in a combination. The “official definition” of Scheme (whatever that means) does not guarantee that any particular order will be followed, nor even that this order will be the same each time a combination is evaluated. You can type $C-g$ to return to the Scheme top level interpreter.

You may also be interested in the *trace* procedure, which will trace the execution of a specified function. For instance, to trace the execution of $p3$, you could type in

```
(trace p3)
```

What happens now when you enter $(p1\ 1\ 2)$?

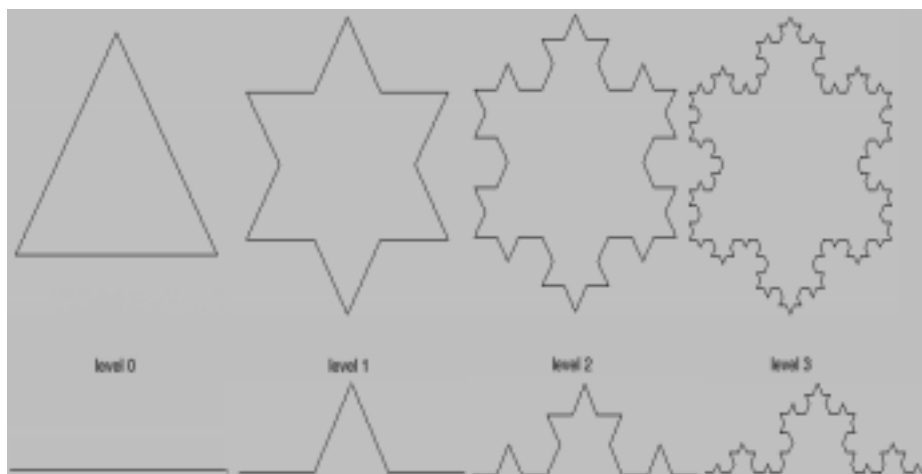


Figure 1: Snowflake curves at levels 0 through 3, and a recursive scheme for drawing a side of the snowflake.

5 Programming Assignment: The Snowflake Curve

In the programming assignment for this week, you will be experimenting with simple procedures that draw variants of a curve called the *snowflake curve* or *Koch curve*, after the mathematician H. von Koch who first described such curves in 1904. You will need to retrieve the file:

```
/usr/local/courses/cse/cse233.01/www/ps1/ps1-pen.scm
```

in order to do this part of the assignment. Load it into your scheme interpreter by executing the command:

```
(load "ps1-pen")
```

To generate the snowflake curve, you begin with a triangle and apply, over and over again, the process of replacing each side by a curve that consists of four smaller sides; each of these smaller sides is in turn replaced by four smaller sides, and so on. If you do this an infinite number of times, the result is a strange topological beast called a *fractal*.³ Fractals have received a lot of attention over the past few years, partly because they tend to arise in the theory of nonlinear differential equations, but also because they are pretty, and their finite approximations can be easily generated with recursive computer programs.

Figure 1 shows some approximations to the snowflake curve, where we stop replacing sides after a certain number of levels: a level-0 side is simply a straight line, a level-1 side consists of four level-0 sides, a level-2 side four level-1 sides, and so on. The figure also illustrates a recursive strategy for drawing a side of the snowflake: a level- n side is a made up of four level- $(n - 1)$ sides, each $1/3$ the length of the original side. Two of the four sides are parallel to the original, one is rotated by $\pi/3$ (60 degrees) and one is rotated by $-\pi/3$.

We can translate this strategy into a pair of procedures that draw side of level n and length L , oriented at a given angle: To draw a side, draw either a single line (at level 0) or else draw four sides of level $n - 1$ and length $L/3$:

```
(define (side angle length level)
  (if (= level 0)
      (plot-line angle length)
      (4sides angle (/ length 3) (- level 1))))
```

```
(define (4sides angle length level)
  (side angle length level)
  (side (+ angle (/ pi 3)) length level)
  (side (- angle (/ pi 3)) length level)
  (side angle length level))
```

³A fractal curve is a “curve” which, if you expand any small piece of it, you get something similar to the original. Fractals have striking mathematical properties. The Koch curve, for example, is neither a true 1-dimensional curve, nor a 2-dimensional region in the plane, but rather something in between. In the mathematics of fractals, the dimension of the Koch curve turns out to be $\log 4 / \log 3 \approx 1.26$.

The procedure `plot-line` is a primitive graphics command. It draw a line by moving a graphics “pen” through a specified distance, in the direction of a specified angle. (Angle 0 is towards the left; $\pi/2$ is straight up.) Other graphics primitives are `clearscreen`, a procedure of no arguments that clears the graphics screen and returns the pen to the center of the screen; and `set-pen-at`, a procedure of two arguments, x and y , that sets the pen at the point (x, y) without leaving any trace.⁴

Finally, we can draw the snowflake curve by drawing three sides, oriented at 60 degrees, -60 degrees, and 120 degrees:

```
(define (snowflake length level)
  (side (/ pi 3) length level)
  (side (- (/ pi 3)) length level)
  (side pi length level))
```

5.1 Trying the program

Create a new file named `ps1.scm` to hold the code for your program. Type in the definitions of the procedures `snowflake`, `side`, and `4sides`. It’s a good idea to leave a blank line between procedure definitions.

You will also need to define a value for the symbol `pi`:

```
(define pi (* 4 (atan 1 1)))
```

This expression defines `pi` to be 4 times the arctangent of 1, which is $\pi/4$.

Evaluate your procedures from the editor in the Scheme interpreter. Test your program by evaluating

```
==> (clearscreen)
```

```
==> (snowflake 200 4)
```

to draw a snowflake curve of side 200 and level 4.

Try drawing some snowflake curves at various sizes and levels. Use `clearscreen` to clear the screen and `set-pen-at` to change the initial position from which the curve is drawn.

5.2 Varying the parity

The recursive scheme at the bottom of figure 1 suggests some simple variations. For instance, in drawing the 4 small sides rather than turning $\pi/3$ (“outward”) and then turning $-\pi/3$ (“inward”) you could first turn inward and then outward. This would orient the bulge formed by the middle two small sides towards the inside of the snowflake rather than towards the outside. More generally, the decision to turn first $\pi/3$ and then $-\pi/3$, versus turning first $-\pi/3$ and then $\pi/3$, could be made on a level by level basis. One way to implement such a choice is by a procedure `f` that takes the level number as argument and returns either 1 or -1 . Then, in lines 2 and 3 of the procedure `4sides`, the calls to `side`

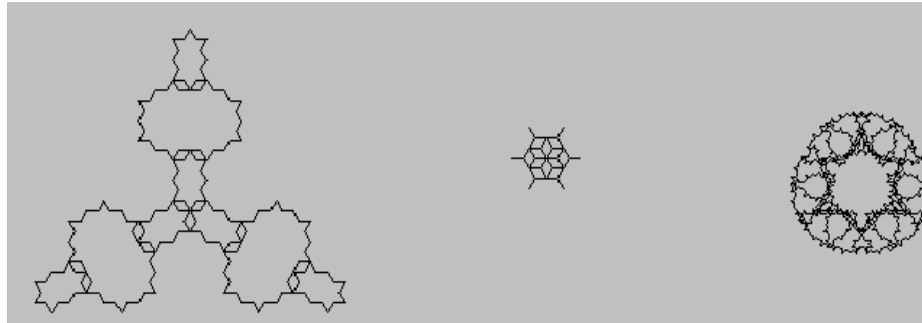
```
(side (+ angle (/ pi 3)) ...)
(side (- angle (/ pi 3)) ...)
```

should be changed to

```
(side (+ angle (* (f level) (/ pi 3))) ...)
(side (- angle (* (f level) (/ pi 3))) ...)
```

If `f` returns 1 for that level, the bulge will be oriented outward, while if `f` returns -1 the bulge will be oriented inward.

⁴The code for the graphics primitives `plot-line`, `clearscreen`, and `set-pen-at` are implemented as procedures that were loaded into Scheme when you loaded the code for problem set 1. They illustrate programming techniques that we will study near the middle of the semester. You might dispute whether these are actually primitives, since they are implemented as ordinary Scheme procedures. Start getting used to the notion that often a “primitive” is simply something that we choose not to look inside of. This illustrates a major idea that we will see throughout the course: One does not create complex systems by focusing on how to construct them out of ultimate primitive elements. Rather, one proceeds in layers, erecting a series of levels, each of which is the “primitives” upon which the next level is constructed.



```

level: 3
f: (lambda (side)
    (if (= side 0)
        1
        -1))

level: 3
f: (lambda (x)
    (+ (* x x)
        (* -3 x)
        5))

level: 4
f: (lambda (x)
    (+ (* x x)
        (* -.4 x)
        5))

```

Figure 2: Variations on the snowflake curve.

Problem 1 Make the change to `4sides` indicated above. Also change `snowflake`, `side`, and `4sides` so that all three procedures take `f` as an extra argument. To test your code, call your new `snowflake` procedure with an `f` that always returns 1

```
(snowflake 200 4 (lambda (side) 1))
```

and verify that you obtain the same snowflake curve as before. What happens if `f` always returns `-1`? Try an `f` that returns 1 at level 0 and `-1` otherwise, and see if you get the first shape shown in figure 2. Explore other choices for `f`, and look for interesting shapes. For this problem you should turn in listings of your modified procedures. Do not turn in any pictures (but see the optional contest at the end of this assignment).

5.3 Varying the angle

There is no reason to restrict the choices for `f` to procedures that return 1 or `-1`. Pretty much *any* function will lead to symmetric designs. Figure 2 shows some examples. Try some choices for `f` and see what interesting designs you can come up with.

5.4 Varying the side length

Going further, you can scale the lengths of the sides as a function of the level by using a procedure `g` and changing the final line of `side` to

```
(4sides angle (* (g level) (/ length 3)) (- level 1) ...
```

Problem 2 Make this change, adding `g` as a new argument to each of your procedures and changing the calls to the procedures accordingly. To test your code, call `snowflake` with a `g` that always returns 1, and verify that you get the same pictures as before. Now try other choices for `g`. You may need to change the initial drawing point (using `set-pen-at`) to get nicely scaled pictures that fit on the screen. Turn in a listing of your modified procedures.

5.5 Total length of the snowflake curve

Problem 3 Write three new procedures `snowflake-length`, `side-length`, and `4sides-length` that take the same arguments as your `snowflake` program procedures (as modified in problem 2). However, instead of drawing anything, calling `snowflake-length` should return the *total length* (i.e., the total distance moved by the pen) of the curve that `snowflake` draws, given the same arguments. (Hint: The three new procedures should call each other in the same way as the original three and each should return the length of the part of the curve that would be drawn.

Then `snowflake-length` should return the sum of the results generated by the three calls to `side-length`, which should return the sum of the results generated by the four calls to `4side-length` and so on.) Turn in a listing of these procedures.

Problem 4 Give a formula for the length of the (original, ordinary) snowflake curve of side L and level n , and give a short proof that formula is true. (Hint: How many short line segments are there? How long is each one?) Test your formula against your answer for problem 3 by computing the length of some snowflake curves (taking `f` and `g` to be procedures that always return 1).

Problem 5 Use your procedures to compute the total length of curves with side 100, levels 0 through 4, and the following choices for `g` (note that the choice of `f` doesn't matter):

- `(lambda (x) (+ (* x x) (* 3 x) 2))`
- `sqrt`
- The function of x that is 1 if $x = 0$ and $1/x^2$ otherwise.

For each choice, show the answer, along with the call to Scheme that you typed in order to compute it.

Problem 6 You've undoubtedly noticed that, whenever you draw a snowflake curve, Scheme always prints `PEN-MOVED` after drawing the curve. Explain why.

Problem 7 Tell us how much time you spent in lab on this assignment, and how much time did you spend in total on this assignment.

6 4. Preparing for class discussion

You should be prepared to discuss these problems orally in class:

Class problem 1 Show how to rewrite the `snowflake` procedure so that `side` and `4sides` appear as internally defined procedures, rather than separate procedures as in the written assignment. If you do this, then your answers to problems 1 and 2 can be a bit simpler, because you needn't pass `f` and `g` explicitly as parameters to the internal procedures. Of all the parameters—`angle`, `length`, `level`, `f`, `g`—which ones *must* still be explicitly passed to the internal procedures?

Class problem 2 In the recursive snowflake scheme shown in figure 1, each side gets replaced by a small triangle. More generally, we could replace each side by a small square, and small pentagon, a small hexagon, . . . How would you write a version of `snowflake` that takes an additional argument `pattern` to specify this? That is, a `pattern` of 3 would give the current designs, a `pattern` of 4 would grow small squares, and `pattern` of 5 would grow small hexagons, and so on. Observe that this requires replacing `4sides` by a procedure that contains some sort of iterative loop to draw the correct number of sides at the correct angles.

Class problem 3 In your answers to problems 1 and 2, you wrote procedures that accepted procedures `f` and `g` as arguments. Subsequently in lecture, you've seen how to write procedures that return procedures as values. For example, suppose that you want to be able to call `snowflake` (as modified in problems 1 and 2) with an appropriate procedure `f` that returns 1 at level n and 0 otherwise, where n is some parameter you specify. You could write a procedure like this:

```
(define (snowflake-change-at-n length level n g)
  (snowflake length level (1-at-n n) g))
```

Give a definition of the required procedure `1-at-n`, which takes an argument n and returns a procedure as value. This value is a procedure that returns 1 if its argument is equal to n and 0 otherwise. Give some other examples of higher-order procedures that generate appropriate procedures `f` and `g`, and show sample uses of them with `snowflake`.

7 Extra Credit

Exercise 1.3 in the book.

8 What/Where to Turn in

Answer Homework Problems 1 and 2 and place your answers in text files named `hwk1.txt` and `hwk2.txt` respectively. For Problem 1, be sure to specify the expression you are evaluating and the result you expect. Do these *without* using the Scheme interpreter!

Place your solutions to the Programming Assignment in files named `prog#.scm` where # represents the problem number. Include only the procedures that you have modified or added to solve the specific problem.

After you have completed the assigned work, place a copy of everything in your turnin directory:

`/usr/local/courses/cse/cse233.01/dropbox/$USER/ps1`

replacing `$USER` with your afs id.