MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1989

Problem Set 1

Issued: February 7, 1989
Due:

- for sections 1–7 in Recitation on Wednesday, February 15
- for sections 8–14 in Recitation on Friday, February 17

Problem sets should always be handed in at recitation. Late work will not be accepted. We are staggering the due dates for problem sets to minimize crowding in the lab. The Wednesday/Friday dates will be reversed during the second half of the semester to assure equitable treatment of students in both morning and afternoon sections. Specific due dates will be announced as each problem set is handed out.

Reading:

- "6.001 General Information": Review this if you haven't already done so. Note in particular the hours when the lab is open.
- Text: section 1.1.
- "A 6.001 User's Guide to the Chipmunk System": Chapters 1 and 2. Bring this manual to lab with you to use as a reference.

Before you can begin work on this assignment, you will need to get a copy of the textbook and pick up the manual package. To obtain the manual package, and follow the instructions given in the General Information memo.

The laboratory assignments for 6.001 have been designed with the assumption that you will do the required reading and textbook exercises before you come to the laboratory. You should also read over and think through the entire assignment before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have planned before coming into lab than to try to do the planning "online." Students who have taken 6.001 in previous terms report that failing to prepare ahead for laboratory assignments generally ensures that the assignments will take much longer than necessary.

Part 1: Homework exercises

Do the following exercises before going to the lab. Write up your solutions and include them as part of your homework.

Exercise 1.1 Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented. You should determine the answers to this exercise without the help of a Chipmunk, then check your answers when you come to the lab.

```
==> (+ 7 8 9)

==> (> 10 9.7)

==> (- (* 2 5) (/ 16 10))

==> (define double (lambda (x) (* x 2)))

==> double

==> (define c 4)

==> c

==> (double c)

==> c

==> (double (double (+ c 5)))

==> (define times-2 double)

==> (times-2 c)

==> (define d c)

==> (= c d)

==> (if (= (* c 6) (times-2 d))
        (< c d)
        (+ c d))

==> (cond ((>= c 2) d)
          ((= c (- d 5)) (+ c d))
          (else (abs (- c d))))

==> (define try (lambda (x) (x 3)))

==> (try double)

==> (try (lambda (z) (* z z)))
```

Observe that some of the examples shown above are indented and displayed over 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 read it easily.

Exercise 1.2 For each of the following four expressions, show the successive steps—using the substitution model of evaluation—that reduce the expression to a number. In each expression, assume that exp_1 and exp_2 are subexpressions that evaluate to 5 and 12, respectively. (Of course, you can't show the details of evaluating these two subexpressions.) Also, for each expression, tell whether applicative-order evaluation stipulates that exp_1 is evaluated before exp_2, or vice versa, or whether the order of evaluating these two subexpressions is unspecified.

```
((lambda (x) ((lambda (y) (- y x)) exp1)) exp2)

((lambda (y) ((lambda (x) (- y x)) exp2)) exp1)

((lambda (x y) (- y x)) exp2 exp1)

((lambda (y) (lambda (x) (- y x)) exp1) exp2)

(((lambda (x) (lambda (y) (- y x))) exp1) exp2)
```

Exercise 1.3 Translate the following arithmetic expression into Lisp prefix notation:

$$\frac{3+2+(1-(5-(2+\frac{3}{4})))}{4*(7-3)*(1-6)}$$

Exercise 1.4 Do exercise 1.4 of the text.

Part 2: Getting started in the lab and using the debugger

When you come to the lab, find a free computer and log in and initialize one of your disks, as described at the beginning of chapter 1 of the Chipmunk manual. You should also write your name and address on a label affixed to the jacket of the floppy disk. Floppy disks are notoriously unreliable storage media, and it is a good idea to copy your data onto a second disk (that is used only for this purpose) when you have completed each laboratory assignment. See the description of how to copy disks in the Chipmunk manual, and do not hesitate to ask the lab assistants for help.

After you have successfully logged in, load the material for problem set 1, using the method described in chapter 1 of the Chipmunk manual—namely, press $\boxed{\text{EXTEND}}$ (i.e., the key marked $\boxed{\text{K}_2}$) then type `load problem set` and press $\boxed{\text{ENTER}}$. When the system asks for the problem set number, type `1` and press $\boxed{\text{ENTER}}$. The system will load some files, and leave you connected to the Scheme interaction buffer, with Scheme prompting you for an expression to evaluate.

Evaluating expressions

To get used to typing at the Chipmunks, check your answers to exercise 1.1. After you type each expression, press the $\boxed{\text{EXECUTE}}$ key (at the lower right side of the keyboard) to evaluate the expression and see the result. Notice that when you type a right parenthesis, the matching left parenthesis is briefly highlighted. You can correct typing errors with the $\boxed{\text{BACKSPACE}}$ key.[1]

To type expressions that go over a single line, use the $\boxed{\text{ENTER}}$ key (not the $\boxed{\text{EXECUTE}}$ key) to move to the next line. Notice that the editor automatically “pretty prints” your procedure as you type it in, indenting lines to the position determined by the number of unclosed left parentheses.

[1]Edwin is a version of Emacs, the editor used on Athena workstations. It has many powerful editing features, which you can read about in chapter 2 of the Chipmunk manual. Although simple cursor motion will be your primary editing tool at first, it will be greatly to your advantage to learn to use some of the more sophisticated editing commands as the semester progresses.

Using the debugger

When you program, you will certainly make errors, both simple typing typing mistakes and more significant conceptual bugs. Even expert programmers produce code that has errors; superior programmers make use of debuggers to identify and correct the errors efficiently. At some point over the next month[2] you should read chapter 3 of the Chipmunk manual, which describes Scheme's debugging features in detail. For now, we've provided a simple exercise to acquaint you with the debugger.

Loading the code for problem set 1, which you did above, defined three tiny procedures called `p1,` `p2` and `p3`. We won't shown you the text of these procedures—the only point of running them is to illustrate the debugger.

Evaluate the expression `(p1 1 2)`. This should signal an error. The screen splits into two windows, with the ordinary interaction buffer at the top, and a `*Scheme Error*` window at the bottom. The error window contains the message

```
Wrong Number of Arguments 1
within procedure #[COMPOUND-PROCEDURE P2]
```

At the bottom of the screen is a question asking whether or not you want to debug the error.

Don't panic. Beginners have a tendency, when they encounter an error, to immediately respond "No" to the offer to debug, sometimes without even reading the error message. 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 says that the error was caused by a procedure being called with 1 argument, which is the wrong number of arguments for that procedure. The next line tells you that the error occurred within the procedure `P2`, so the actual error was that `P2` was called with 1 argument, which is the wrong number of arguments for `P2`.

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. 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.

Type `y` in response to the system's question. Scheme should respond:

```
Subproblem Level: 0  Reduction Number: 0
Expression:
(P2 B)
within the procedure P3
applied to (1 2)
```

This says that the expression that caused the error was `(P2 B)`, within the procedure `P3`, which was called with arguments 1 and 2. Ignore the stuff about subproblem and reduction numbers. You can read about them in chapter 4 of the Chipmunk manual.

The debugger differs from an ordinary Scheme command level, in that you use single-keystroke commands, rather than typing expressions and pressing $\boxed{\text{EXECUTE}}$. One thing you can do is move "up" in the evaluation sequence, to see how the program reached the point that signaled the error. To do this, type the character `u`. The debugger should show:

```
Subproblem level: 1 Reduction number: 0
Expression:
(+ (P2 A) (P2 B))
within the procedure P3
applied to (1 2)
```

[2] Don't do it now—you have enough to do now.

Remember that the expression evaluated to cause the error was `(P2 B)`. Now that we have moved "up," we learn that this expression was being evaluated as a subproblem of evaluating the expression `(+ (P2 A) (P2 B))` still within procedure `P3` applied to 1 and 2.

So we've learned from this that the bug is in `P3`, where the expression `(+ (P2 A) (P2 B))` calls `P2` with the wrong number of arguments. At this point, one would normally quit the debugger and edit `P3` to correct the bug.

Before leaving the debugger, let's explore a little more. Press `u` again, and you should see

```
Subproblem Level: 2  Reduction Number: 0
Expression:
(+ (P2 X Y) (P3 X Y))
within the procedure P1
applied to (1 2)
```

which tells us that the program reached the place we just saw above as a result of trying to evaluate an expression in `P1`.

Press `u` again and you should see some mysterious stuff. 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 loop. 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 `u` is `d`, 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 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. For more information, see the Chipmunk manual.

Type `q` to quit the debugger and return to ordinary Scheme.

3. Programming Assignment: The snowflake curve

Now that you've gained some experience with the Chipmunks, you should be ready to work on the programming assignment. When you are finished in the lab, you should write up and hand in the numbered problems below. You may want to include listings and/or pictures in your write-up. Chapter 1 of the Chipmunk manual explains how to use the lab printers.

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.

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

[3] A 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 $\log 4/\log 3 \approx 1.26$.

`level 0`   `level 1`   `level 2`   `level  3`

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

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 **??** 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))
```

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 180 degrees:

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

Trying the program

Start a new file on your disk to hold the code for your program by pressing $\boxed{\text{FIND FILE}}$ ($\boxed{\text{SHIFT}}$-$\boxed{K_0}$) and enter a name for this file: `ps1.scm`. You can use some other name besides `ps1` if you like, but you should end the name in `.scm`. This tells the editor that the file contains Scheme code. Edwin will inform you that this is a new file, and will create an empty edit buffer for it. Subsequent laboratory assignments will include large amounts of code, which will be loaded automatically into an edit buffer when you begin work on the assignment. This time, however, we are asking you to type the code yourself, to give you practice with the editor.

Type in the definitions of the procedures `snowflake`, `side`, and `4sides`. Use the $\boxed{\text{ENTER}}$ key to go from one line to the next, so that Edwin will automatically indent the code. If Edwin's indentation does not match what you expected, you have probably omitted a parenthesis—observing Edwin's indentation is an excellent way to catch parenthesis errors. It's also 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 $\pi/4$.

"Zap" your procedures from the editor into Scheme, using the $\boxed{\text{EVALUATE BUFFER}}$ key ($\boxed{\text{SHIFT}}$-$\boxed{K_7}$).[5]. If you get an error at this point, you probably have a badly formed procedure definition. Ask for help. Otherwise, test your program by evaluating

```
==> (clearscreen)

==> (snowflake 200 4)
```

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

To see the drawing, press the key marked $\boxed{\text{GRAPHICS}}$ at the upper right of the keyboard. The Chipmunk system enables you to view either text or graphics on the screen, or to see both at once.

[4]If you consult the Chipmunk manual, you will not find `plot-line`, `clearscreen`, and `set-pen-at` documented in the list of Scheme primitives. That is because they are implemented as procedures that were loaded into Scheme when you loaded the code for problem set 1. If you want to play, you can look at these procedures. (Figure out how to do this.) 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 6.001: 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.

[5]$\boxed{\text{EVALUATE BUFFER}}$ transmits the entire buffer to Scheme, and is the easiest thing to use when you only have one or two procedures in your buffer. With larger amounts of code, it is better to use other "zap" commands, which transmit only selected parts of the buffer, such as individual procedure definitions. For a description of these commands, see Chapter 3 of the Chipmunk manual, under the heading "Scheme Interaction."

This is controlled by the keys marked $\boxed{\text{GRAPHICS}}$ and $\boxed{\text{ALPHA}}$. Pressing $\boxed{\text{GRAPHICS}}$ shows the graphics screen superimposed on the text. Pressing $\boxed{\text{GRAPHICS}}$ again hides the text screen and shows only graphics. $\boxed{\text{ALPHA}}$ works similarly, showing the text screen and hiding the graphics screen.

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.

To save your work on your disk, press $\boxed{\text{SAVE FILE}}$ ($\boxed{\text{K}_0}$). It's a good idea to save your work periodically, to protect yourself against system crashes.

Varying the parity

The recursive scheme at the bottom of figure **??** 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 means of 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.

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 **??**. 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).

Varying the angle

Actually, 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 **??** shows some examples. Try some choices for `f` and see what interesting designs you can come up with.

```
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.

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.

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

Contest (Optional)

Hopefully, you generated some interesting designs in doing this assignment. If you wish, you can make graphics prints of your best designs and enter them in the 6.001 snowflake design contest. The Chipmunk manual explains how to use the graphics printers; or ask the lab assistants for help. Turn in your design collection together with your homework, but *stapled separately*, and make sure your name is on the work. For each design, show the length, level, `f`, and `g` that produced the result. In order minimize demand on the graphics printers *do not turn in more than four pictures.* Designs will be judged by 6.001 staff and other internationally famous snowflake design experts, and prizes will be awarded in lecture.

Finally

Please indicate the names of any students you cooperated with in doing this assignment.

How much time did you spend in lab on this assignment? How much time did you spend in total on this assignment?