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

Problem Set 1

Issued: 12 September 1989

Due: In recitation, 20 September 1989

Problem sets should always be handed in at recitation. Late work will not be accepted.

Reading:

- "Course Organization": Review this if you haven't already done so.
- 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 to pick up the manual package. To obtain the manual package, follow the instructions given in the Course Organization 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 Exercise 1.1 on page 19 of the text. You may use the computer to check your answers, but first do the evaluation by hand so as to get practice with understanding Scheme. Typing expressions such as these to the interpreter and observing its behavior is one of the best ways of learning about the language. Experiment and try your own problems (you need not turn in your own problems). 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.

Exercise 2 For each of the following four expressions, show a sequence of 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)) `exp_1`)) `exp_2`)`

`((lambda (y) ((lambda (x) (- y x)) `exp_2`)) `exp_1`)`

`((lambda (x y) (- y x)) `exp_2 exp_1`)`

`((lambda (y) (lambda (x) (- y x)) `exp_1`) `exp_2`)`

`(((lambda (x) (lambda (y) (- y x))) `exp_1`) `exp_2`)`

Exercise 3 Translate the following arithmetic expression into Scheme prefix notation:

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

Exercise 4 Do Exercise 1.4 on page 23 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, so 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 EXTEND (i.e., the key marked K_2) then type `load problem set` and press ENTER. When the system asks for the problem set number, type `1` and press ENTER. The system will load some files and will leave you connected to the Scheme interaction buffer, where Scheme prompts 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 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 cursor briefly highlights the matching left parenthesis. You can correct typing errors with the BACKSPACE key.[1]

[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

To type expressions that go over a single line, use the ENTER key (not the 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.

Using the debugger

When you program, you will certainly make errors, both simple typographical typographical 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. This chapter 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 show 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.

---

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.

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

The debugger differs from an ordinary Scheme command level, in that you use single-keystroke commands, rather than typing expressions and pressing 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)
```

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 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. For more information, see the Chipmunk manual.

Type `q` to quit the debugger and return to ordinary Scheme. (If you are left in the *Help* buffer, just press the SCHEME key (labeled K7) to return to the Scheme interaction buffer.)

3. Programming Assignment: Graphing Epicycloids and Hypocycloids

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.

Figure 1: Constructing epicycloids and hypocycloids

In this assignment, you are to experiment with simple procedures that plot curves called epicycloids and hypocycloids. These are defined formally as follows.

Consider a circle of radius `b` and a disk of radius `a` rolling on the circumference of the circle. Consider a point `p` that rotates with the disk at a distance `a1` from the center of the disk. The curve traced by this point as the disk rolls around the circle is called an *epicycloid* if the disk is rolling around the outside of the circle, and a *hypocycloid* if the disk is rolling around the inside of the circle.

If the equation of the circle with center point at $(0, 0)$ is

`x`^{2} `+ y`^{2} `= b`^{2}

then the locus of the point `p=(x,y)` can be described for the epicycloid as follows:

$$\begin{aligned} x(t) &= (a+b)sin(\tfrac{a}{b}t) - a_1 sin(\tfrac{a+b}{b}t) \\ y(t) &= (a+b)cos(\tfrac{a}{b}t) - a_1 cos(\tfrac{a+b}{b}t) \end{aligned}$$

The corresponding equations for the hypocycloid are

$$\begin{aligned} x(t) &= (b-a)sin(\tfrac{a}{b}t) - a_1 sin(\tfrac{b-a}{b}t) \\ y(t) &= (b-a)cos(\tfrac{a}{b}t) - a_1 cos(\tfrac{b-a}{b}t) \end{aligned}$$

We can graph epicycloids on the Chipmunk using the procedure below. The arguments to the procedure are the radii `a` and `b` of the disk and the circle, the distance `a1` from the center of the

disk to the plotted point, and a parameter `dt` that determines the increment at which points are plotted.

```
(define (epicycloid a b a1 dt)
  (define (horiz t)
    (- (* (+ a b) (sin (/ (* a t) b)))
       (* a1 (sin (/ (* (+ a b) t) b)))))
  (define (vert t)
    (- (* (+ a b) (cos (/ (* a t) b)))
       (* a1 (cos (/ (* (+ a b) t) b)))))
  (define (iter t)
    (draw-line-to (horiz t) (vert t))
    (iter (+ t dt)))
  (iter 0))                              ; start drawing
```

The `epicycloid` procedure contains internally defined procedures that compute the horizontal and vertical coordinates corresponding to a given value of `t`, and a procedure `iter` that repeatedly draws lines connecting the points for consecutive values of `t`. The primitive `draw-line-to`, used by `iter`, moves the graphics pen to the indicated `x` and `y` coordinates, drawing a line from the current point.

Trying the program

Start a new file on your disk to hold the code for your program by pressing FIND FILE (SHIFT–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 procedure `epicycloid`. Use the 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.

"Zap" your procedures from the editor into Scheme, using the EVALUATE BUFFER key (SHIFT–K_7).[3]. 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

```
==> (clear-graphics)

==> (epicycloid 40 35 60 0.1)
```

to draw an epicycloid curve.

To see the drawing, press the key marked 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. This is controlled by the keys marked GRAPHICS and ALPHA. Pressing GRAPHICS

[3] 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."

Figure 2: `(epicycloid 40 35 60 0.1)`

shows the graphics screen superimposed on the text. Pressing GRAPHICS again hides the text screen and shows only graphics. ALPHA works similarly, showing the text screen and hiding the graphics screen.

Try drawing some epicycloid curves with various parameters. Use `clear-graphics` to clear the screen and `position-pen` to change the initial position from which the curve is drawn.

To save your work on your disk, press SAVE FILE (K0). It's a good idea to save your work periodically, to protect yourself against system crashes.

If you typed the program incorrectly, you will need to debug the procedure and return to the editor to make corrections.

Observe that the `epicycloid` procedure does not terminate. It will keep running until you stop execution by typing CTRL–G or ABORT. (The CTRL key at the upper left of the keyboard is used like a SHIFT key. To type CTRL–G, hold down the CTRL key and press G.)

Exercise 5

Define a procedure `radius-scale` that takes as argument a number `m`, and calls `epicycloid` with `a` = 30, `b` = 20, `a1` = `m` × 60, and `dt` = 0.1.

Explore the family of figures generated by `radius-scale` as `m` varies smoothly from 0 to 1. Turn in two or three sketches showing how the figures change, together with a listing of the procedure `radius-scale`.

Exercise 6

A hypocycloid is similar in definition to an epicycloid. Using the editor, make a copy of your code for `epicycloid`, and modify it to implement a definition for `hypocycloid`. Turn in a listing of the procedure.

Exercise 7

Investigate the family of hypocycloids generated with `a` = 60, `b` = 30, `dt` = 0.2, and `a1` varying from 0 to 70. All these figures should have 3-fold symmetry. Turn in sketches of some of the figures to indicate how the shape varies as `a1` changes.

Exercise 8

Investigate the hypocycloids that are generated with `a` = 60, `a1` = 60, `dt` = 0.2, and `b` taking on values that are integer multiples of 5 that are greater than 0 and less than 60. Pay particular attention to the symmetry of the figures. (For example, `b` = 30 produces 3-fold symmetry and `b`=40 produces 4-fold symmetry.) Find a value of `b` that produces a figure with 5-fold symmetry; with 11-fold symmetry. What kind of symmetry is produced by `b` = 35? Turn in a sketch of one of the more interesting figures you discover.

Exercise 9

If we restrict `a` and `b` to be integers, then the epicycloid will be completely drawn by the time that `t` reaches $2\pi \times b$. Explain why. (In general, waiting until `t` reaches $2\pi \times b$ may cause the figure to be traced more than once. The number of times that the figure will be retraced can be specified in terms of `a` and `b`. You might think about how to do this.) Modify the `epicycloid` program so that it stops when `t` becomes greater than $2\pi \times b$. (Let the procedure return 0 when it stops). This requires only a small modification to the `iter` procedure. In order to do this conveniently, in Scheme define `pi` to be 3.14159 (or `(* 4 (atan 1 1))` for a more accurate value). Also define `2pi` to be a constant equal to twice `pi`. Turn in a listing of your modified procedure, as well as the expressions you type to Scheme to define `pi` and `2pi`.

Exercise 10

(*Design problem*) The `epicycloid` procedure given above does a lot of unnecessary computation. Each time it plots a point, it computes the sum of `a` and `b` four times. In computing the argument to `sin` and `cos` it multiplies `t` by a constant and then divides the result by `b`, rather than just multiplying by a single constant that could have been computed just once at the beginning of the plot. It takes no advantage of the fact that many of the same quantities, such as `(/ (* (+ a b) t) b)`, are used in computing both the `x` and the `y` coordinates.

Redesign the procedure to eliminate as many of these redundant computations as you can. (You may need to define extra internal variables or procedures.) Implement your design as a new procedure `epicycloid1`. Test your procedure to verify that it draws the same figures as does the original `epicycloid`. Does your modification improve the speed by a noticeable amount? (Use a wristwatch,

or the clock at the lower right of the screen, to determine approximate timings.) Turn in a listing of `epicycloid1`, together with the expressions you ran to make timing tests, and the comparative timings for `epicycloid` and `epicycloid1`.

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?