Structure and Interpretation of Computer Programs
Second Edition
SAMPLE

**Simple Problem Set**

Reading:

- Text (SICP 2nd Edition by Abelson & Sussman): section 1.1

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

## 1. Before Turning on your Computer

Read the course notes through section 1.1. Then, predict what the interpreter will print in response to evaluation of each of the following expressions. Assume that the sequence is evaluated in the order in which it is presented here.

```
(- 8 9)

(> 3.7 4.4)

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

(define b 13)

13

b

>

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

square

(square 13)

(square b)

(square (square (/ b 1.3)))

(define multiply-by-itself square)

(multiply-by-itself b)

(define a b)

(= a b)

(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. Getting started with the Scheme Interpreter

In the Scheme interpreter, you will use a text-editing system called Edwin, which is an editor closely resembling the widely used editor Emacs. Even if you are already familiar with Emacs, you should take some time *now* to run the Emacs/Edwin 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.

### Evaluating expressions

After you have learned something about Edwin, 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 1 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 lab tutor or from the person sitting next to you.

Observe that some of the examples printed above in section 1 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))
```

Edwin provides several commands that "pretty-print" your code, e.g., indents lines to reflect the inherent structure of the Scheme expressions.[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

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

[2] Make a habit of typing `ctrl-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.

that file Edwin simply switches you into this buffer. Otherwise, Edwin will create a new buffer for the file. If you give the system a name that has not yet been used, with an extension of `.scm`, Edwin 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, you can use any of several commands: `M-z` to evaluate the current definition, `ctrl-x ctrl-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 `Q` 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 `Q` to quit out of the debugger. 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.

## 3. The Scheme Debugger

While you work, you will often need to debug programs. This section contains an exercise to acquaint you with some of the features of Scheme to aid in debugging. Learning to use the debugging features will save you much grief on later problem sets. Additional information about the debugger can be found by typing `?` in the debugger.

Load the code for this problem set using the Edwin `C-x C-f` command. Evaluate the code using `M-x eval-buffer`. This will load definitions of the following three procedures `p1`, `p2` and `p3`:

```
(define p1
  (lambda (x y)
    (+ (p2 x y)
       (p3 x y))))

(define p2
  (lambda (z w)
    (* z w)))

(define p3
  (lambda (a b)
    (+ (p2 a)
       (p2 b))))
```

In the Scheme buffer, evaluate the expression `(p1 1 2)`. This should signal an error, with the message:

```
;The procedure #[compound-procedure P2] has been called with 1 argument
;it requires exactly 2 arguments.
;Type D to debug error, Q to quit back to REP loop:
```

Don't panic. Beginners have a tendency, when they hit an error, to quickly type `Q`, often without even reading the error message. Then they stare at their code in the editor trying to see what the bug is. Indeed, the example here is simple enough so that you probably can find the bug by just reading the code. Instead, however, let's 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 one argument, which is the wrong number of arguments for that procedure. 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 type `D` to start the debugger.

## Using 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. When you start the debugger, it will create a new window showing two buffers. The top buffer should look like this.

```
     COMMANDS:   ? - Help   q - Quit Debugger   e - Environment browser
This is a debugger buffer:
Lines identify stack frames, most recent first.
   Sx means frame is in subproblem number x
   Ry means frame is reduction number y
The buffer below describes the current subproblem or reduction.
-----------
The *ERROR* that started the debugger is:
  The procedure #[compound-procedure 119 p2] has been called with 1 argument;
  it requires exactly 2 arguments.
```

```
>S0  (#[compound-procedure 119 p2] 2)
    R0  (p2 b)
 S1  (+ (p2 a) #(p2 b)#)
    R0  (+ (p2 a) (p2 b))
    R1  (p3 x y)
 S2  (+ (p2 x y) #(p3 x y)#)
    R0  (+ (p2 x y) (p3 x y))
    R1  (p1 1 2)
 --more--
```

You can select a "frame" by clicking on its line with the mouse or by using the ordinary cursor line-motion commands to move from line to line. Notice that the bottom, information, buffer changes as the selected line changes.

The frames in the list in the top buffer represent the steps in the evaluation of the expression. There are two kinds of steps—subproblems and reductions. For now, you should think of a reduction step as transforming an expression into "more elementary" form, and think of a subproblem as picking out a piece of a compound expression to work on.

So, starting at the bottom of the list and working upwards, we see `(p1 1 2)`, which is the expression we tried to evaluate. The next line up indicates that `(p1 1 2)` reduces to `(+ (p2 x y) (p3 x y))`. Above that, we see that in order to evaluate this expression the interpreter chose to work on the subproblem `(p3 x y)`, and so on, moving upwards until we reach the error: the call to `(p2 b)` from within the procedure `p3` has only one argument, and `p2` requires two arguments.[3]

Take a moment to examine the other debugger information (which will come in handy as your programs become more complex). Specifically, in the top buffer, select the line

```
>S2  (+ (p2 x y) #(p3 x y)#)
```

The bottom buffer should now look like this:

```
                        SUBPROBLEM LEVEL: 2
Expression (from stack):
 Subproblem being executed highlighted.
    (+ (p2 x y) (p3 x y))
------------------------------------------------------------------
ENVIRONMENT named: (user)
    p1 = #[compound-procedure 31 p1]
    p3 = #[compound-procedure 32 p3]
    p2 = #[compound-procedure 27 p2]
==> ENVIRONMENT created by the procedure: P1
      x = 1
      y = 2
------------------------------------------------------------------
;EVALUATION may occur below in the environment of the selected frame.
```

[3]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 Scheme system 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.

The information here is in three parts. The first shows the expression again, with the subproblem being worked on. The next major part of the display shows information about the *environments*. We'll have a lot more to say about environments later in the course, but, for now, notice the line

```
==> ENVIRONMENT created by the procedure: P1
```

This indicates that the evaluation of the current expression is within procedure `p1`. Also we find the environment has two *bindings* that specify the particular values of `x` and `y` referred to in the expression, namely `x = 1` and `y = 2`. At the bottom of the description buffer is an area where you can evaluate expressions in this environment (which is often useful in debugging).

Before quitting the debugger try one final experiment (you may have already done this). Continue to scroll down through the stack past the line: `R1 (p1 1 2)` (you can also click the mouse on the line `--more--` to show the next subproblem). You will then see additional frames that various complicated expressions. 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. (Yes: almost all of the system is itself a Scheme program.)

You can type `q` to return to the Scheme top level interpreter.

## 4. Exploring the system

The following exercises are meant to help you practice editing and debugging, and using on-line documentation.

**Exercise 1: More debugging** The code you loaded for problem set 1 also defined three other procedures, called `fold`, `spindle`, and `mutilate`. One of these procedures contains an error. Evaluate the expression `(fold 1 2)`. What is the error? How should the procedure be defined? (Notice that you can examine the code for a procedure by using the `pp` ("pretty print") command. For example, evaluating the expression `(pp fold)` will print the definition of `fold`.

**Exercise 2: Still more debugging** The code you loaded also contains a buggy definition of a procedure meant to compute the factorials of positive integers: $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$. Evaluate the expression `(fact 5)` (which is supposed to return 120). Use the debugger to find the bug and correct the definition. Compute the factorial of 243. Copy the corrected definition and the value of `(fact 243)` into your `ps1-ans.scm` buffer.

**Exercise 3: Defining a simple procedure** The number of combinations of $n$ things taken $k$ at a time is given by $n!/k!(n-k)!$. Define a procedure `(comb n k)` that computes the number of combinations, and find the number of combinations of 243 things taken 90 at a time. Include your procedure definition and your answer in `ps1-ans.scm`.

**Exercise 4: Practice with the editor** Find the Free Software Foundation copyright notice at the end of the Edwin tutorial. Copy it into `ps1-ans.scm`.

**Exercise 5: Learning to use `Info`** Start up the `info` program with the Edwin command `M-x info`. `Info` is a directory of useful information. You select a topic from the menu, by typing `m` followed by the name of the topic. (You can type a few characters that begin the topic name, and then type a space, and Edwin will complete the name. One of the `info` options (type `h`) gives you a brief tutorial in how to use the program. Use `info` to find the cost of a 12-inch cheese pizza from Pizza Ring, and copy this to the answer buffer.

**Exercise 6: Hacker Jargon** The info `Jargon` entry is a collection of hacker terms that was eventually published in 1983 as the book *The Hacker's Dictionary.* The `New Jargon` info entry is an expanded version that was published in 1991 by MIT Press as *The New Hacker's Dictionary.* The old Jargon file is structured as an `info` file, with submenus; the new version is a single text file, which you can search. Find the definition of "Unix conspiracy" from the new jargon file, and find the definition of "Phase of the Moon" from either jargon file. Include these in the answer buffer.

**Exercise 7: Scheme documentation** The `Scheme` info entry is an on-line copy of the Scheme reference manual that was distributed with the notes. Find the description of "identifiers" in the documentation. What is the longest identifier in the list of example identifiers?

**Exercise 8: More documentation** There is an Edwin command that you can use to automatically re-indent expressions. For example, if you have a procedure typed as

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

you can move the cursor to the beginning of the line with the `define`, type this Edwin command, and the expression will automatically be re-indented as

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

Find the description of this command. What is the command? Where did you find it?

**Exercise 9: Running Shell Commands** If you run the Scheme interpreter within a *nix environment, the Edwin command `M-x shell` will create a shell buffer, which you can use to run various *nix programs. Some of the more interesting ones are `ls`, `date`, and `echo`. For example, typing

```
date
```

will show you the current date and time (e.g. `Tue Jul 2 15:39:45 EDT 1996`). See if you can figure out what the other commands do, and copy your results into your answer buffer.

**Exercise 10: Getting information from around the world** You can also use the finger program to query computers on the Internet, all over the world. You can finger a particular name at some location, or just finger the location (e.g., `finger @mit.edu`) to get general information. Try fingering some of the following places:

- `cs.berkeley.edu`—a computer run by the computer science department at UC Berkeley
- `idt.unit.no`—a computer at the Norwegian Institute of Technology, a part of the The University of Trondheim, Norway.
- `whitehouse.gov`—the White House

See what else you can find. Include some piece of this in your answer file.

**Exercise 11:** An *application* of an expression $E$ is an expression of the form $(E\,E_1 \ldots E_n)$. This includes the case $n = 0$, corresponding to an expression $(E)$. A *Curried application* of $E$ is either an application of $E$ or an application of a Curried application of E. For each of the procedures defined below, give a Curried application of the procedure which evaluates to 3.

```
(define foo1
  (lambda (x)
    (* x x)))
(define foo2
  (lambda (x y)
      (/ x y)))
(define foo3
  (lambda (x)
    (lambda (y)
      (/ x y))))
(define foo4
  (lambda (x)
    (x 3)))
(define foo5
  (lambda (x)
    (cond ((= x 2)
           (lambda () x))
          (else
           (lambda () (* x 3))))))
(define foo6
  (lambda (x)
    (x (lambda (y) (y y)))))
```

Type in these definitions, and include a demonstration of your answers in the answer buffer.