

University of Massachusetts Lowell

91.301: Organization of Programming Languages
Spring 2007

Problem Set 1 **Getting to Know Scheme**

Issued: Tuesday, 23 January 2007

Due: Tuesday, 30 January 2007

Reading: Text (SICP 2nd Edition by Abelson & Sussman): Section 1.1

Overview

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 course.

Any place that we ask for a piece of code or a written answer to be turned in you should place the code or answer in `ps1-ans.ss`. Written answers that are not code should be in comments in `ps1-ans.ss`.

Print the answers buffer and turn it in at the beginning of class on the due date. Staple any handwritten portions of the assignment to your code file that you turn in. Also submit your answer buffer using the submit command on mercury: `submit holly 301-ps1 ps1-ans.ss`

1. Before Turning on your Computer

Read the text 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 development environment called DrScheme. The DrScheme main window has two panels. In the upper panel you edit your answer file (or whatever Scheme code you happen to be working on). The DrScheme documentation calls this panel the *definitions window*. The lower panel is for interaction with the Scheme interpreter, and is called the *interactions window* in the DrScheme documentation.¹ Use the “Execute” button or the F5 key to load the contents of the definitions window into the Scheme interpreter. Executing will wipe out any previous contents in the interactions window. Use the interactions window for typing small tests, for finding the values of expressions, and for any text I/O (resulting from evaluating read, write, or display expressions).

DrScheme uses key bindings for editing commands similar to those used by the Emacs editor (just different enough to be annoying). You can also perform many editing tasks using the mouse and the menus.

To start Scheme in the lab on a Linux machine, type `drscheme` at the command line of a terminal window. You can download DrScheme for your own computer (Windows, Mac, or Unix) from the web site linked on the course web page.

¹ <http://download.plt-scheme.org/doc/350/pdf/drscheme.pdf>

Evaluating expressions

After you have learned something about editing in DrScheme, go to the interactions window (the lower panel in the DrScheme window) You can type Scheme expressions, and they should be evaluated and the result printed out.

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 staff member 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))
```

When you type Enter at the end of a line, DrScheme will automatically indent the next line as needed.

Creating a file

Do any lab work that you want to submit (or any other work that needs to be preserved) in the definitions window. If modifying an existing file, use the “Open” dialog from the File menu. If starting a new file, just type a few characters and then press the Save button. A dialog should pop up requesting the file name to save into.

To practice these ideas, create a new file, called `ps1-ans.ss`. In this file, 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 F5 or the “Evaluate” button. Go back to the interactions window, 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 dtrace back the error (by clicking on the bug icon in the interactions window). Go back to the definitions window (open file `ps1-ans.ss`) and edit the definition to insert a space between `*` and `x`. Re-evaluate the definition, and return to the interactions window and try evaluating `(square 4)` again.

As a second method of evaluating expressions, try the following. Go to the interactions window, and again type in:

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

Place the cursor at the end of the line, and press Enter to evaluate the definition. Again try (`square 4`). 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 interactions window in place. That way, the buffer will contain an intact record of your work (since the last “Execute”).

3. The Scheme Debugger

While you work, you will often need to debug programs. Unfortunately, DrScheme does not have as much debugging capability as some other Scheme and Lisp systems. It *does* have the ability to show you the sequence of evaluations that were performed before the bug occurred.

For your experiments with the debugger, make sure that the second line in the interactions window says “Language: Textual(MzScheme, includes R5RS)”. DrScheme includes a group of “teaching languages”, some of which give different debugging information than the languages in the group of “professional languages”. To change the language, click the “Language” entry on the DrScheme menu bar, click “Choose Language...” from the drop-down menu, expand the “PLT” tab in the pop-up window, click on “Textual(MzScheme, includes R5RS)”, click on “OK”, finally, click on the “Execute” button in the DrScheme window.

You can find the code for this problem set at

<http://www.cs.uml.edu/~holly/91.301/ps1.ss>

Save the code to your directory. Once you’ve saved the file, load the code for this problem set. Evaluate the code. 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 interactions window, evaluate the expression `(p1 1 2)`. This should signal an error.

Don’t panic. Beginners have a tendency, when they hit an error, to immediately try to change their code, 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 look at the backtrace. Click on the bug icon in the interactions window to bring up a backtrace window.

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? What simple change to the procedure `fold` will fix the error?

Exercise 2: Computing factorials Type the following code into your buffer and evaluate it. You'll need this for Exercise 3.

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

Exercise 3: Defining a simple procedure The number of combinations of n things taken k at a time, $\binom{n}{k}$, 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 118 things taken 52 at a time. Include your procedure definition and your answer in `ps1-ans.ss`.

Exercise 4: Write a procedure that takes one argument and triples it. Call this procedure `triple` (you'll use it below).

Exercise 5: Write a procedure that takes two arguments and returns the difference of their tripled values. You should use the `triple` procedure that you wrote for Exercise 4.

Exercise 6: Write a procedure that takes two arguments and returns the smallest number. Call it `smallest-of-two`.

Exercise 7: Write a procedure that takes two arguments and returns the tripled value of the smallest number. You should use the procedures that you wrote for Exercises 4 and 6.

Exercise 8: Look through the DrScheme menus to find out how to re-indent in the region selected by the mouse (or to reindent the current line if no region is selected by the mouse). Explain how to do this in a comment in `ps1-ans.ss` (which should be in the definitions window).

Also find out how to re-indent an entire buffer and explain how to do this in a comment in `ps1-ans.ss`.

What is the key shortcut to re-indent the contents of the buffer? (Look for the "Keybindings" entry in one of the menus.) Put the answer into a comment in `ps1-ans.ss`. (Figuring out the pattern here? You can type answers to questions into comments in the definitions window – this will allow you to evaluate the whole buffer to update your environment or to reload the code when you start working on your problem set again after a break.)

Exercise 9: Starting from the Help menu in DrScheme, open the Help Desk and use it to find the subsection on “Whitespace and Comments” in the “Revised(5) Report on the Algorithmic Language Scheme” (often called the “R5RS”).

Copy that section into your definitions window.

Look up the keybinding in DrScheme that comments out code selected by the mouse.

Use that combination of keys to comment out the text you just inserted.

Use F5 or the execute button to load your definitions into your Scheme session. (If part of the description didn’t get commented out, you will either get a bunch of error messages in the interactions window, or you will define the fact procedure, or both.)

Use the index of the R5RS to look up the “zero?” predicate. What other library procedures are defined immediately after the “zero?” predicate?

Exercise 10: Type the following code into your definitions window:

```
(define mystery
  (lambda (a b)
    (cond ((zero? b) 0)
          ((odd? b)
           (+ a (mystery (+ a a)
                         (quotient b 2))))
          (else
           (mystery (+ a a)
                     (quotient b 2))))))
(mystery 7 2)
```

Change the DrScheme language to “Intermediate Student” to use the stepper.

With the cursor on the

```
(mystery 7 2)
```

button.

How many steps does it take to get the value of

```
(mystery 7 2)
```

Now we can try the substitution model as shown in class. You can use the stepper to check your intermediate results.

Put any evaluated expression into curly braces.

Put an expression that has all evaluated, but is not yet applied into square braces.

For example:

<code>((lambda (x) (* x x)) 5)</code>	evaluate the lambda
<code>{(proc (x) (* x x)) 5}</code>	evaluate 5
<code>[[proc (x) (* x x)] {5}]</code>	substitute into the body
<code>(* {5} {5})</code>	evaluate *
<code>[[mult] {5} {5}]</code>	apply primitive {mult}
<code>{25}</code>	

Assume we have definitions:

```
(define abs (lambda (x)
              (if (> 0 x)
                  (- 0 x)
                  x)))
(define square (lambda (x) (* x x)))
(define a -2.0)
```

Your job is to use the substitution model to evaluate the combination

```
(square (abs a))
```