CS251 Programming Languages
Prof. Lyn Turbak
Wellesley College

Handout # 39
Friday, April 19

Problem Set 8

Due: Friday, April 26, 2002

Reading:

- Handouts #37 (Imperative Programming) and #38 (Parameter Passing)
- *SICP* 3.1-3.3 (side effects, environment diagrams) and 5.3 (garbage collection)
- *MLWP* Chapter 8 (Imperative Programming)

Submission:

- Problems 1, 2, 3, and 5 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- Your hardcopy submission for Problem 4 should be your file `param.hic`; your softcopy submission should be your entire `ps8` directory.

Problem 1 [20]: Counters

Consider the following procedures in imperative call-by-value statically-scoped Scheme:

```
(define make-counter1
  (let ((count 0))
    (lambda ()
      (lambda ()
        (begin (set! count (+ count 1))
               count)))))

(define make-counter2
  (lambda ()
    (let ((count 0))
      (lambda ()
        (begin (set! count (+ count 1))
               count)))))

(define make-counter3
  (lambda ()
    (lambda ()
      (let ((count 0))
        (begin (set! count (+ count 1))
               count)))))

(define test-counters
  (lambda (make-counter)
    (let ((a (make-counter))
          (b (make-counter)))
      (list (a) (b) (a)))))
```

For each of the following expressions, (1) give the value of the expression and (2) draw an environment diagram that justifies why the expression has that value. You should assume that all operands are evaluated in left-to-right order.

a. `(test-counters make-counter1)`

b. `(test-counters make-counter2)`

c. `(test-counters make-counter3)`

Problem 2 [20]: Environment Diagrams in the Presence of State

Figure 1 shows an environment diagram depicting the state of a program in Scheme. Recall that in Scheme, all variables are implicitly bound to cells, which are implicitly dereferenced when variables are looked up. The contents of a cell can be changed by Scheme's set! special form.

a. Given that Scheme is in the state shown by Fig. 1, describe what results are printed (via print) by the following sequence of Scheme definitions and expressions:

```
(define print
  (lambda (x)
    (begin (display x) (newline))))

(define test
  (lambda () (print (list (f 1) (g 1)))))

(begin (test) (g 'b)
       (test) (f 'a)
       (test) (f 'b)
       (test) (g 'f)
       (test))
```

b. Give a sequence of Scheme definitions and expressions that would leave Scheme in the state depicted by Fig. 1. (Your sequence is allowed to create structures not shown in Fig. 1, but all the structures in Fig. 1 must be created by your sequence.)

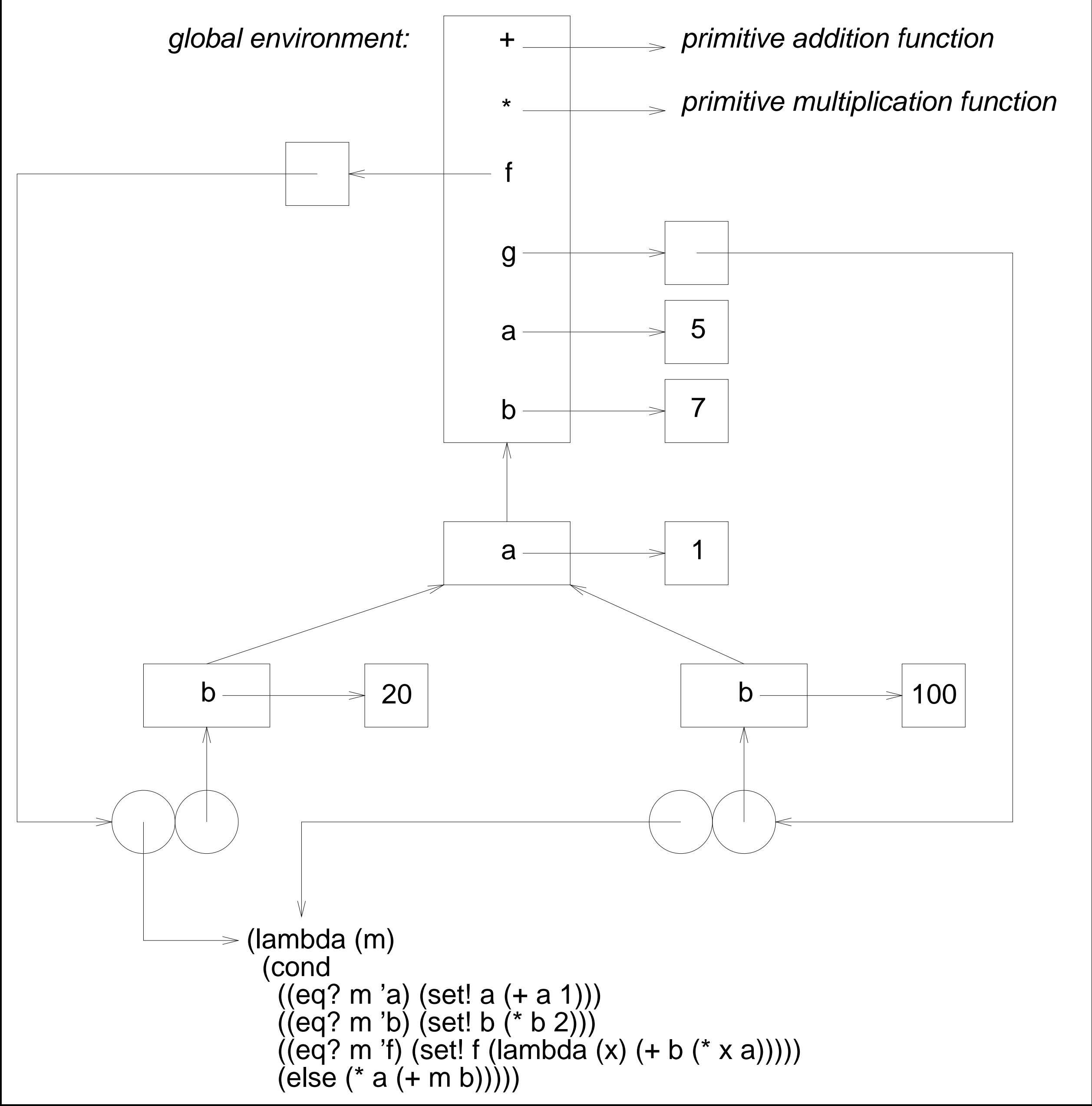

Figure 1: An environment diagram.

Problem 3 [20]: Safe Transformations

A transformation that rewrites one expression to another is said to be safe if performing the transformation anywhere in a program will not change the behavior of the program . For each of the following transformations, indicate whether it is safe in (i) a purely functional version of call-by-value Scheme and (ii) a version of call-by-value Scheme that supports side effects. For each transformation you specify as unsafe, give an example whose behavior is changed by the transformation. Changes in behavior include:

- the program returns different values before and after the transformation.
- the program loops infinitely before the transformation, but returns a value after the transformation.
- the program returns a value before the transformation, but loops infinitely after the transformtation.

In each expression, I stands for a variable reference and E stands for an expression. You may assume that all subexpressions of an application are evaluated in left-to-right order.

a. `(+` I I`)` $\Longrightarrow$ `(* 2` I`)`

b. `(+` E E`)` $\Longrightarrow$ `(* 2` E`)`

c. `(+` E_1 E_2`)` $\Longrightarrow$ `(+` E_2 E_1`)`

d. `(+` E_1 E_2`)` $\Longrightarrow$ `(let ((x` E_1`)) (+ x` E_2`))`

e. `(+` E_1 E_2`)` $\Longrightarrow$

```
(let ((x (lambda () E1)))
      (y (lambda () E2))
  (+ (x) (y)))
```

f. `(if #t` E_1 E_2`)` $\Longrightarrow$ E_1

g. `(if` E_1 E_2 E_2`)` $\Longrightarrow$ E_2

h. `(if (if` E_1 E_2 E_3`)` E_4 E_5`)` $\Longrightarrow$ `(if` E_1 `(if` E_2 E_4 E_5`) (if` E_3 E_4 E_5`))`

Problem 4 [20]: Parameter-Passing Mechanisms

In the file `~/cs251/ps8/param.hic`, write a single nullary (zero-argument) program P_{param} in the HOILIC language such that:

- P_{param} would evaluate to the string `"value"` in call-by-value HOILIC;
- P_{param} would evaluate to the string `"name"` in call-by-name HOILIC;
- P_{param} would evaluate to the string `"need"` in call-by-need HOILIC;
- P_{param} would evaluate to the string `"reference"` in call-by-reference HOILIC.

You can use any HOILIC expressions, but the only types of values that your expression should manipulate are strings, functions, and the implicit mutable variables of HOILIC. That is, your example should not involve any numbers, booleans, lists, etc. Strive to make your expression as simple and understandable as possible.

Notes:

- Recall that HOILIC is an extension to HOFL in which all variables denote implicit cells (as in Scheme). These implicit cells are automatically dereferenced on every variable lookup, and can be assigned to via the `<-` construct (which is analogous to `set!` in Scheme). For example, here is an iterative factorial program written in HOILIC:

```
(program (n)
  (bind ans 1
    (bindrec ((loop ()
                (if (<= n 0)
                    ans
                    (seq (<- ans (* ans n))
                         (<- n (- n 1))
                         (loop)))))
      (loop))))
```

- You may assume that all operand expressions in a function application are evaluated in a left-to-right order.
- If you cannot solve this problem with just strings, functions, and implicit mutable variables, you can get partial credit by solving the problem using other types of values.
- You can get partial credit if your expression distinguishes some, but not all, of the parameter-passing mechanisms.
- As of this writing, interpreters for the different parameter-passing versions of HOILIC do not exist, and it is unclear if they will come into existence this semester. This means that, for the time being at least, you will have to simulate the different parameter-passing mechansims in your head and cannot check them mechanically. If this situation changes, I will let you know.

Problem 5 [20]: Garbage Collection

Consider the memory shown below, where entities beginning with n are integers and entities beginning with p are pointers. (Recall that p0 is the distinguished **null pointer**.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p13	p5	n1	p5	n2	p13	p11	p9	n3	p0	n4	p7	p5	p15	n5	p0

a. Suppose that the above memory represents a collection a list nodes, each of which is allocated in two contiguous cells. Draw a box-and-pointer diagram showing all the list nodes.

b. Suppose that the list memory shown above is the "from space" in a stop-and-copy garbage collector, and that the list node at address p1 is the root of the accessible list nodes. Show the "to space" that results from performing a stop-and-copy garbage collection. Assume that the addresses of to space are 17 through 32.

c. Answer the following questions:

- What is the main problem with reference counting as a form of garbage collection?
- What is the key advantage of stop-and-copy garbage collection in comparison with mark-sweep garbage collection?
- What is an advantage of mark-sweep garbage collection over stop-and-copy garbage collection?

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 8
Due Friday April 26

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Problem 2 [20]		
Problem 3 [20]		
Problem 4 [20]		
Problem 5 [20]		
Total		