CS251 Programming Languages **Handout #39**
Prof. Lyn Turbak **May 4, 2001**
Wellesley College

OPTIONAL PROBLEM SET 8
Due Friday, May 11, 2001

This is an *optional* problem set. If you turn in this problem set (or any part of it) it can only help your grade. Even though the problem set is optional, you are required to understand all of the material covered by this problem set for the final exam. Solutions to this problem set will be posted over the May 12-13 weekend so you can study the solutions for the final exam if you desire. Of course, once you study the solutions to a problem, you are on your honor not to turn in a solution to that problem.

Reading:

- Handouts 35 (Imperative Programming), 36 (Parameter Passing), 37 (Laziness), and 38 (Advanced Control Constructs)
- *SICP*, 3.1-3.3; 3.5 (Covers side effects, environment diagrams, streams)
- *MLWP:* 5.12—5.20 (Lazy data); Chapter 8 (Imperative Programming)
- Hughes's *Why Functional Programming Matters*.

Submission:

- Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- Your hardcopy submission for Problem 3 should be your file `param.hic`.
- Your hardcopy submission for Problem 4 should include your paragraph for 4a, as well as your files `sqrt.scm` and `hamming.scm`.
- Your hardcopy submission for Problem 5 should be your file `trees.scm`.
- Your softcopy submission for this assignment should be your entire `ps8` directory.

Problem 1 [15]: 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.

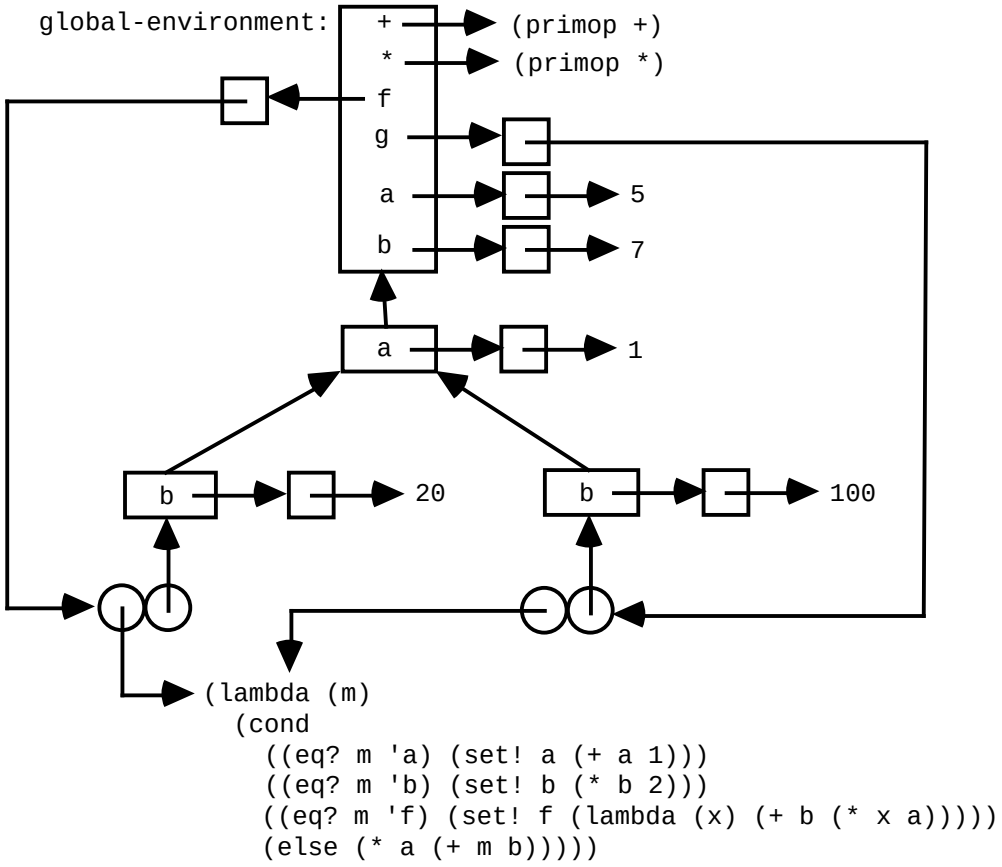

Figure 1: An environment diagram

Part a. Given that Scheme is in the state shown by Figure 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))
```

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

Problem 2 [16]: 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) => (* 2 I)`
b. `(+ E E) => (* 2 E)`
c. `(+ E1 E2) => (+ E2 E1)`
d. `(+ E1 E2) => (let ((x E1)) (+ x E2)`
e.
```
(+ E1 E2) => (let ((x (lambda () E1))
                  (y (lambda () E2)))
              (+ (x) (y)))
```
f. `(if #t E1 E2) => E1`
g. `(if E1 E2 E2) => E2`
h. `(if (if E1 E2 E3) E4 E5) => (if E1 (if E2 E4 E5) (if E3 E4 E5))`

PROBLEM 3 [14]: 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 4 [30]: Lazy Data

a [5] In his paper, "Why Functional Programming Matters", John Hughes argues that lazy evaluation is an essential feature of the functional programming paradigm. Briefly summarize his argument in one paragraph.

b [10] In the file `~/cs251/ps8/sqrt.scm`, translate the Newton-Rhapson square-root example from pp. 27--29 of Hughes's paper into Scheme using streams. Use your procedure to compute the square root of 2 with tolerances of 1, 0.1, and 0.01.

c [15] In the file `~/cs251/ps8/hamming.scm`, write the following procedures using Scheme streams:

- The `scale` procedure takes a scaling factor and a stream of integers and returns a new stream each of whose elements is a scaled version of the corresponding element of the original list.

- The `merge` procedure takes two streams of integers, each in sorted order, and returns a new stream, also in sorted order, that has all the elements of both input streams. The resulting list should not contain duplicates.

- The *Hamming numbers* are the set of positive integers whose prime factors only include the numbers 2, 3, and 5. For example, the first 15 Hamming numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, and 24. Define an infinite stream named `hamming` that contains all of the Hamming numbers, in order. (*Hint:* use `scale` and `merge` from above.) Using the `take` procedure discussed in class, return a list of the first 52 Hamming numbers.

Problem 5 [25]: Non-Local Exits and Exceptions

Here we consider non-local exits and exceptions in the context of some Scheme procedures for manipulating binary trees. Below is a simple Scheme implementation of a binary tree ADT, along with some sample trees.

```
(define leaf (lambda () '()))
(define leaf? (lambda (thing) (null? thing)))
(define node (lambda (value left right) (list value left right)))
(define value (lambda (node) (first node)))
(define left (lambda (node) (second node)))
(define right (lambda (node) (third node)))

(define tree1 (node 3
                    (node 4
                          (node 2 (leaf) (leaf))
                          (leaf))
                    (node 5 (leaf) (leaf))))

(define tree2 (node 3
                    (node 4
                          (node 0 (leaf) (leaf))
                          (leaf))
                    (node 5 (leaf) (leaf))))

(define tree3 (node 3
                    (node 4
                          (node 'x (leaf) (leaf))
                          (leaf))
                    (node 5 (leaf) (leaf))))

(define tree4 (node 3
                    (node 0 (leaf) (leaf))
                    (node 'x (leaf) (leaf))))

(define tree5 (node 3
                    (node 'x (leaf) (leaf))
                    (node 0 (leaf) (leaf))))
```

The trees `tree1` and `tree2` contain only numeric values, but `tree3`, `tree4`, and `tree5` all contain a non-numeric value (the symbol `x`).

For testing various tree-manipulating functions, we introduce the following function, which applies a given function to each of the five trees defined above:

```
(define test-trees
  (lambda (f)
    (map f (list tree1 tree2 tree3 tree4 tree5))))
```

The following `product` procedure calculates the product of a tree of numbers, but it handles several cases specially:

- If a node with a non-numeric value is encountered, the symbol `non-number` is returned as the product of that node without examining its left and right subtrees. This symbol is propagated as the result of product on the entire tree.

- If a node with a zero is encountered, a zero is returned as the product of that node without examining its left and right subtrees.

- If a node has a left subtree whose product is a zero, a zero is returned as the product of that node without examining its right subtree.

```
(define product
  (lambda (tree)
    (if (leaf? tree)
      1
      (let ((v (value tree)))
        (if (not (number? v))
          'non-number
          (if (= v 0)
            0
            (let ((left-result (product (left tree))))
              (if (eq? left-result 'non-number)
                'non-number
                (if (= left-result 0)
                  0
                  (let ((right-result (product (right tree))))
                    (if (eq? right-result 'non-number)
                      'non-number
                      (* v (* left-result right-result)))))))))))))
```

For example, evaluating `(test-trees product)` yields `(120 0 non-number 0 non-number)`.

You should begin this problem by evaluating `(load "~/cs251/ps8/trees.scm")`. In addition to loading the tree code discussed above, it also loads the file `~/cs251/util/control.scm`, which extends Scheme with the `label`, `jump`, `handle`, `trap`, and `raise` constructs presented in class. Once this file is loaded, you can program in regular Scheme (not a toy language!) using these constructs.

Part a [10]: Non-local Exits

It is clumsy for `product` to perform checks that propagate `non-number` and zero. In a language that supports the `label` and `jump` constructs, such behavior can be expressed more elegantly by using `label` and `jump` to immediately return 0 when a 0 is encountered, or the symbol `non-number` when a non-number is encountered.

In this problem, you should flesh out in `trees.scm` the skeleton of the procedure `product-nonlocal` in that behaves like `product` except that it performs non-local exits for the 0 and non-number cases via `label` and `jump`. As with the list product example discussed in class, `product-nonlocal` should be defined in terms of a local recursive procedure `inner`. Your resulting procedure should be able to pass the following test: evaluating `(test-trees product-nonlocal)` should yield `(120 0 non-number 0 non-number)`.

Warning: MIT-Scheme evaluates the arguments to a function from right to left rather than from left to right. Take this into account when writing `product-nonlocal`, as this fact can affect the results you observed.

Part b [15]: Exception Handling

The `product` and `product-nonlocal` procedures defined above contain hardwired assumptions about how to handle zeroes and non-numeric values. Suppose we want a version of `product` that always returns `non-number` if the tree argument contains a non-numeric values, regardless of whether it contains any zeroes. Then we must delve into the code for `product` and change the way it handles zeroes.

In a language that supports exception handling, a more flexible approach is to use exceptions to handle special cases like zero and non-numeric values. Consider the following `product-exception` procedure, which is defined in `~/cs251/ps8/trees.scm`.

```
(define product-exception
  (lambda (tree)
    (if (leaf? tree)
      1
      (let ((v (value tree)))
        (if (not (number? v))
          (raise non-number tree)
          (if (= v 0)
            (raise zero tree)
            ;; Use LET to explicitly process left subtree before right
            (let ((left-prod (product-exception (left tree))))
              (let ((right-prod (product-exception (right tree))))
                (* v (* left-prod right-prod)))))))))
```

This procedure raises exceptions for the special cases where the node value is zero or not a number. In both cases, the current node is passed as the argument to the exception handler. This allows the caller of product-exception to determine how the special cases should be handled. In particular, different callers can deal with the special cases in different ways.

In the following parts, you will write procedures `product1`, `product2`, and `product3` that handle these cases in three different ways. Each of your `product`*`i`* procedures should have the following form:

```
(define producti
  (lambda (tree)
    (<handler1> non-number
             (lambda (tree) <exp1>)
             (<handler2> zero
                     (lambda (tree) <exp2>)
                     (product-exception tree)))))
```

where <*`handler1`*> and <*`handler2`*> are either `handle` or `trap`, whichever is appropriate.

i. [5]. `product1` has the same behavior as the `product` and `product-nonlocal` procedures above:

```
(test-trees product1)
; Value: (120 0 non-number 0 non-number)
```

ii. [5]. `product2` is like `product` exept that for trees containing a non-numeric values, it returns the symbol `non-number` regardless of whether there are any zeroes in the tree.

```
(test-trees product2)
; Value: (120 0 non-number non-number non-number)
```

iii. [5]. `product3` is like `product` exept that it treats every non-numeric value as 1 for the purposes of calculating the product.

```
(test-trees product3)
; Value: (120 0 60 0 0)
```

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

CS251 *Optional* Problem Set 8

Due Friday, May 11, 2001

Name:

Date & Time Submitted *(only if late)*:

Collaborators (*anyone you collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [15]		
Problem 2 [16]		
Problem 3 [14]		
Problem 4 [30]		
Problem 5 [25]		
Total [100]		