Problem Set 8
Due: Saturday, May 14 (Individual)
Monday, May 23 (Group)

Revisions: May 7: (1) Group Problems have been added and (2) “need” has been changed to “lazy” in Individual Problem 3.

Overview:
The individual problems on this assignment test your understanding of imperative programming, parameter passing, and closure conversion. All individual problems are required. The group problems on this assignment memory management, laziness, control, and simple programming in Scheme, Haskell, and C. All Group Problems are completely optional; you should undertake these only if you have both the time and the interest.

You should complete all Individual Problems before starting any Group Problems. The scores for parts of Group Problems will only be included in your Group Problem average if they increase your average. You may turn in whatever Group Problems you have completed at any time before the end of finals (4:30 Monday May 23). When you submit your Group Problems, you will receive the Group Problems solutions.

It is strongly recommended that you submit your Group Problems before you take your CS251 final so that you may study the PS8 Group Problem solutions as part of preparing for the exam. In terms of preparing for the final exam, it is recommended that you do the Group Problems in their given order. I.e., if you only have time for one problem, do Problem 1; if you have more time, do Problem 2, and so on.

Reading:
- Handout #41: Compound Data and Memory Management
- Handout #43: You Can Do More If You’re Lazy
- Handout #44: John Hughes’s article Why Functional Programming Matters
- Handout #46: Haskell and HUGS
- Handout #47: Scheme
- Handout #48: Control

Individual Problem Submission:
Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn’s office door any time on Sat. May 14. The packet should include:
1. an individual problem header sheet;
2. your environment diagram from Problem 1a;
3. your HOILIC program diagram.hic from Problem 1b;
4. your HOILIC definition file cell.def from Problem 2;
5. your HOILIC definition file param.def from Problem 3;
6. your FOLP+ program closures.ffp from Problem 4.

Each student should also submit a softcopy (consisting of your final ps8-individual directory) to the drop directory by executing:

    cd /students/username/cs251
    cp -R ps8-individual ~cs251/drop/ps8/username/
Working Together:

If you want to work with a partner on this assignment, you should try to find a different partner than you worked with on a previous assignment. If this proves difficult, please email Lyn describing your situation.

Group Problem Submission:

Each team should turn in a single hardcopy submission packet for all Group problems by slipping it under Lyn's office door any time before 4:30pm on Mon. May 23.

1. a team header sheet indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. the final version of trees.scm for Problem 1.
3. the paragraph for Problem 2a; the final version of sqrt.scm for Problem 2b; the final version of Hamming.hs for Problem 2c; and (i) the final version of Hamming.java and (ii) the answer to the efficiency question for Problem 2d.
4. your pencil-and-paper answers to Problem 3.
5. the final version of sortlines.c for Problem 4.

Each team should also submit a single softcopy (consisting of your final ps8-group directory) to the drop directory ~cs251/drop/p8/username, where username is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute:

```bash
cd /students/username/cs251
cp -R ps8-group ~cs251/drop/ps8/username/
```
Individual Problems

These are individual problems. Each student must solve this problem on her own without consulting any other person (except Lyn).

Individual Problem 1 [20]: Stateful Environment Diagrams

Fig. 1, shows an environment diagram depicting the state of a HOILIC program. Recall that in HOILIC, 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 the assignment construct, $<-$. 

a. [15] Suppose that a HOILIC program is in the state shown in Fig. 1, and the following expression $E_{test}$ is evaluated in environment frame $F_1$.

\[
E_{test} \equiv (\text{seq } (\text{println } (\text{list } a \ b)) \\
(\text{println } (\text{list } (g \ 1) \ (h \ 1))) \\
(h \ "b") (\text{println } (\text{list } (g \ 1) \ (h \ 1))) \\
(g \ "a") (\text{println } (\text{list } (g \ 1) \ (h \ 1))) \\
(g \ "b") (\text{println } (\text{list } (g \ 1) \ (h \ 1))) \\
(h \ "g") (\text{println } (\text{list } (g \ 1) \ (h \ 1))) \\
(\text{println } (\text{list } a \ b)))
\]

- Make a copy of Fig. 1 and draw all new environment frames that are created during the evaluation of $E_{test}$.
- Show how the contents of cells in your diagram change over time by crossing out old values and writing the new values to their right.
- Write down the values that are displayed when $E_{test}$ is evaluated.

b. [5] Write a HOILIC program containing $E_{test}$ that, when executed on the two arguments 5 and 7, would create the environments depicted in Fig. 1 and would evaluate $E_{test}$ in frame $F_1$.

Notes:

- *Hint:* What must the abstraction (fun ...) of the closure named f be?
- In HOILIC, bindrec is not a kernel form, but is defined by the following syntactic sugar:

\[
\begin{align*}
\text{bindrec } & ((I_1 \ E_1) \ ... \ (I_n \ E_n)) \ E_{body} \\
\to & \ \text{bindpar } ((I_1 \ \#f) \ ... \ (I_n \ \#f)) \\
& (\text{seq } (<- \ I_1 \ E_1) \\
& \quad ; \\
& \quad (<- \ I_1 \ E_n) \\
& E_{body}))
\end{align*}
\]

Not only does this guarantee that the identifiers $I_1 \ ... \ I_n$ are defined in a single mutual recursive scope, but it also allows the expression $E_i$ to directly reference the identifiers $I_1 \ ... \ I_{i-1}$. (In HOFL and HOILEC, any such references would denote “black holes”.) For example, the expression

\[
\begin{align*}
\text{bindrec } & ((a \ 1) \\
& (f \ (\text{fun } () \ (\text{seq } (<- \ a \ (* \ a \ 10)) \ a))))) \\
& (b \ (* \ 2 \ a)) \\
& (c \ (f)) \\
& (d \ (* \ 3 \ a) \ (* \ 4 \ (f)) \ (* \ 5 \ a))) \\
& (\text{list } a \ b \ c \ d))
\end{align*}
\]

evaluated to the value (list 100 2 10 930).
You are not required to test your program, but if you wish to do so, you can write it in the file ~/cs251/ps8-individual/diagram.hic and can test it by executing the following in the OCAML interpreter:

```ocaml
#cd "/students/your-username/cs251/ps8-individual";;
#use "load-diagram.ml";;
testDiagram();;
```

The first two lines load the HOILIC interpreter and testing code. These only need to be evaluated once. You can re-evaluate `testDiagram` every time you change `diagram.hic`.

![Figure 1: A HOILIC environment diagram.](image-url)
Individual Problem 2 [10]: Explicit mutable cells

HOILIC does not support the explicit mutable cells of HOILEC. However, it is possible for a HOILIC user (not just the language implementer) to add these to HOILIC by fleshing out the following skeleton HOILIC definitions in file ~/ps8-individual/cell.def:

(def (cell contents) E_{cell-body})
(def (^ c) (c #t))
(def (:= c v) E_{set-body})

Note that ^ has already been defined for you.

Notes:

• **Hint:** Use the message-passing approach to implementing stateful objects covered in Handout #38. However, your definitions should be considerably simpler than those in the OOP example from Handout #38. Each expression should be a one-liner.

• You can use any HOILIC expressions you want, but the only types of literal values that your expressions should use are booleans and functions. Your example should not use any integers, characters, symbols, strings, or lists. (You may submit solutions with values of these other types, but you will only receive partial credit if you do so.)

• Unlike the HOILEC := primitive operator, your HOILIC := function will be curried. I.e., (:= a 5) is equivalent to ((:= a) 5).

• You can test your definitions by executing the following in the OCAML interpreter:

```shell
#cd "/students/your-username/cs251/ps8-individual";
#use "load-cell.ml";
@testCell();
```

The first two lines load the HOILIC interpreter and testing code. These only need to be evaluated once. You can re-evaluate testCell every time you change cell.def.
Individual Problem 3 [15]: Parameter-Passing Mechanisms

In the file ~/cs251/ps8-individual/param.def, define a single nullary (zero-argument) function \texttt{param} in the HOILIC language such that:

- \((\texttt{param})\) evaluates to the string "value" in call-by-value HOILIC;
- \((\texttt{param})\) evaluates to the string "reference" in call-by-reference HOILIC;
- \((\texttt{param})\) evaluates to the string "name" in call-by-name HOILIC;
- \((\texttt{param})\) evaluates to the string "lazy" in call-by-lazy HOILIC.

Notes:

- Your file \texttt{param.def} should consist of a single definition of the form \((\texttt{def (param) E}_{body})\).
- You may use any HOILIC expressions you want, 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 use any integers, booleans, characters, symbols, or lists. Strive to make your expression as simple and understandable as possible.
- 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 will get partial credit if your expression distinguishes some, but not all, of the parameter-passing mechanisms.
- You can test your definition by executing the following in the OCAML interpreter:

\begin{verbatim}
#cd "/students/your-username/cs251/ps8-individual";;
#use "load-param.ml";;
testParam();;
\end{verbatim}

The first two lines load the HOILIC interpreter and testing code. These only need to be evaluated once. You can re-evaluate \texttt{testParam} every time you change \texttt{param.def}.  


Individual Problem 4 [15]: Closure Conversion

Consider the following HOFL program $P_{Hofl}$:

$$(\text{hofl } (i \ j) \ (\text{bind } q \ (f \ p) \ (\text{bind } r \ (q \ i \ j) \ \ (\text{bind } \text{answer} \ (p \ (r \ m) \ (r \ (f \ m))) \ \ \ \text{answer } s)))$$

$$(\text{def } (f \ g) \ (\text{fun } (a \ b) \ (g \ b \ a)))$$

$$(\text{def } (p \ d \ e) \ (\text{fun } (h) \ (h \ d \ e)))$$

$$(\text{def } (m \ x \ y) \ (\% \ x \ y))$$

$$(\text{def } (s \ t \ u) \ (+ \ (* \ 10 \ t) \ u)))$$

This is the same as the HOFL program you studied in Individual Problem 1 of Problem Set 7 except that rather than returning the Church pair answer, it returns the result of applying answer to the numerical function s.

In this problem you will help to translate $P_{Hofl}$ into FOFL+. Recall that FOFL+ (i.e., FOFL-PLUS from Handout #37) is a version of FOFL extended with the following two constructs:

- $(\text{fref } F)$ returns the function value denoted by $F$

- $(\text{fapp } E_{\text{rator}} \ E_{\text{rand}_1} \ldots \ E_{\text{rand}_n})$ invokes the function denoted by $E_{\text{rator}}$ to the values denoted by the operands $E_{\text{rand}_1} \ldots E_{\text{rand}_n}$.

Here is a skeleton of the FOFL+ program $P_{Fofl+}$ that is the result of manually closure-converting the higher-order program $P_{Hofl}$ to first-order FOFL+:

$$(\text{fofl+ } (i \ j) \ (\text{bind } q \ (f \ (\text{list } (\text{fref } p))) \ (\text{bind } r \ (\text{applyClosure2 } q \ i \ j) \ \ (\text{bind } \text{answer} \ (p \ (\text{applyClosure1 } r \ (\text{list } (\text{fref } m))) \ E_1) \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \text{applyClosure1 } \text{answer } E_2)))$$

$$(\text{def } (f \ g) \ (\text{list } (\text{fref } f\text{Helper} \ g)))$$

$$(\text{def } (p \ d \ e) \ E_3)$$

$$(\text{def } (m \ x \ y) \ (\% \ x \ y))$$

$$(\text{def } (s \ t \ u) \ (+ \ (* \ 10 \ t) \ u))$$

$$(\text{def } (\text{applyClosure1 } \text{clo } \text{arg}) \ (\text{fapp } (\text{nth } 1 \ \text{clo}) \ \text{arg } \text{clo}))$$

$$(\text{def } (\text{applyClosure2 } \text{clo } \text{arg1 } \text{arg2}) \ E_4)$$

$$(\text{def } (f\text{Helper } a \ b \ \text{clo}) \ (\text{applyClosure2 } (\text{nth } 2 \ \text{clo}) \ b \ a))$$

$$(\text{def } (p\text{Helper } h \ \text{clo}) \ E_5)$$

Your job is to flesh out the missing expressions above in the file ~/cs251/ps8-individual/closures.ffp so that $P_{Fofl+}$ has the same behavior as $P_{Hofl}$ when executed on two integers.

Notes:

- The primitive application $(\text{nth } i \ vs)$ returns the (1-based) $i$th element of the list vs.

- You can test your definition by executing the following in the OCAML interpreter:

  ```ocaml
  #cd "/students/your-username/cs251/ps8-individual";;
  #use "load-closures.ml";;
  testClosures();;
  ```

  The first two lines load the HOLIC interpreter and testing code.  These only need to be evaluated once. You can re-evaluate testParam every time you change closures.ffp.
Group Problems

Group Problem 1 [45]: Non-Local Exits and Exceptions

Here we consider non-local exits and exceptions in the context of some SCHEME procedures for manipulating binary trees. Fig. 2 presents a simple SCHEME implementation of a binary tree ADT, along with some sample trees. The trees tree1 and tree2 contain only numeric values, but tree3, tree4, and tree5 all contain a non-numeric value (the symbol x).

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

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

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

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

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

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

Figure 2: Binary trees in SCHEME.

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))))
```

Fig. 3 presents a product procedure that calculates the product of a binary tree of numbers by multiplying the numbers it encounters at nodes in a left-to-right pre-order depth-first walk of the tree. 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.

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

```
(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))))))))))))
```

Figure 3: A Scheme procedure for computing the product of binary tree of numbers.

You should begin this problem by evaluating (load "~/cs251/ps8-group/trees.scm") within a Scheme interpreter. (See Handout #47 for how to launch a Scheme interpreter and how to write and test Scheme procedures.) In addition to loading the tree code discussed above, it also loads the file "~/cs251/utils/control.scm", which extends Scheme with the label, jump, handle, trap, and raise constructs described in Handout #48. Once this file is loaded, you can program in regular Scheme (not a toy language!) using these constructs.

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 part, you should flesh out the skeleton of the procedure product-nonlocal-jump (in trees.scm) 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 Handout #48, product-nonlocal-jump 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-jump)

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-jump`, as this fact can affect the results you observe.

**b. [10]: Call-with-current-continuation**

An alternative to using `label` and `jump` to perform a non-local exit is to use Scheme’s built-in `call-with-current-continuation` procedure. In the file `trees.scm`, flesh out the skeleton of the procedure `product-nonlocal-cwcc` so that it behaves like `product-nonlocal-jump` but is implemented in terms of `call-with-current-continuation` rather than in terms of `label` and `jump`. Your resulting procedure should be able to pass the following test: evaluating `(test-trees product-nonlocal-cwcc)` should yield (120 0 non-number 0 non-number).

**c. [10]: Continuation-Passing Style**

An alternative to using Scheme’s implicit continuations to perform a non-local exit is to use explicit continuations via continuation-passing style. Suppose that `product-nonlocal-cps` is defined as follows:

```scheme
(define product-nonlocal-cps
  (lambda (tree)
    (product-cps tree (lambda (v) v))))
```

Here, `product-cps` is a function that finds the product of the numbers in its first argument (a tree) and ”returns” the product by invoking its second argument (an explicit continuation) on the product. Your goal in this problem is to flesh out the definition of `product-cps`. You should not use `label`, `jump`, `call-with-current-continuation`, `raise`, `handle`, or `trap` in your definition. Nevertheless, your procedure should return immediately upon encountering a 0 or a non-number. After you have defined `product-cps`, the enclosing `product-nonlocal-cps` procedure should be able to pass the following test: evaluating `(test-trees product-nonlocal-cps)` should yield (120 0 non-number 0 non-number).

**d. [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-group/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 producti procedures should have the following form:

(define producti
  (lambda (tree)
    (handler1 non-number
      (lambda (tree) E1)
    (handler2 zero
      (lambda (tree) E2)
    (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 except 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 except 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)
Group Problem 2 [35]: Lazy Data

a. [5]: Why Laziness Matters   In his paper, “Why Functional Programming Matters” (Handout #44), John Hughes argues that lazy evaluation is an essential feature of the functional programming paradigm. Briefly summarize his argument in one paragraph.

d. [10]: Square roots   Create a file ~/cs251/ps8-group/sqrt.scm in which you 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. (See Handout #47 for how to launch an MIT-Scheme interpreter and how to write and test SCHEME procedures. See App. A for more on Scheme streams.)

b. [10]: Hamming Numbers in HASKELL   Create a file ~/cs251/ps8-group/Hamming.hs in which you define the following HASKELL functions. (See Handout #46 for how to write and test HASKELL functions using HUGS.)

   - The scale function takes a scaling factor and an infinite list of integers and returns a new list each of whose elements is a scaled version of the corresponding element of the original list.
   - The merge function two infinite lists of integers, each in sorted order, and returns a new list, also in sorted order, that has all the elements of both input streams. The resulting list should not contain duplicates (use == to test for equality).
   - 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 list named hamming that contains all of the Hamming numbers, in order. (Hint: use scale and merge from above.) Using the HASKELL take function, give a list of the first 52 Hamming numbers.

c. [10]: Hamming Numbers in JAVA

   - In the file ~/cs251/ps8-group/Hamming.java, flesh out the skeleton of the Hamming class (Fig. 4) that implements the Enumeration interface and enumerates the Hamming numbers. Study the FibEnumeration class at the end of Handout #43 as an example of a JAVA class that enumerates an infinite sequences of integers. As in FibEnumeration, you will have to enumerate integers wrapped in the Integer class to satisfy the constraint that nextElement must return an Object.

   Choose the simplest strategy you can think of for generating the Hamming numbers one at a time. Compile your file using javac Hamming.java, and test it via java Hamming, which will display the first 52 elements of your enumeration. (If n is a non-negative integer, then java Hamming n will enumerate the first n elements.)

   - Which approach to generating Hamming numbers is more efficient: the approach you use in your HASKELL program or the approach you use in your JAVA program? Explain.
import java.util.*; // imports Enumeration interface

public class Hamming implements Enumeration {

    // Put instance variable(s) here.

    public Hamming () {
        // Flesh out this constructor method
    }

    public boolean hasMoreElements () {return true;}

    public Object nextElement () {
        // Replace this stub.
        return new Integer(1);
    }

    // Add any auxiliary methods here.

    // Testing method
    public static void main (String[] args) {
        int i = 52; // default number
        if (args.length == 1) {
            i = Integer.parseInt(args[0]);
        }
        Enumeration h = new Hamming();
        while (i > 0) {
            System.out.print(h.nextElement());
            System.out.print(" ");
            i--;
        }
    }
}

Figure 4: Skeleton of the Hamming class for enumerating Hamming numbers.
Group Problem 3 [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**.)

\[\begin{array}{cccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
p13 & p5 & n1 & p5 & n2 & p13 & p11 & p9 & n3 & p0 & n4 & p7 & p5 & p15 & n5 & p0
\end{array}\]

a. [4] 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. [10] 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, and that the garbage collection begins with by copying the root pointer \( p1 \) to slot 17.

c. [6] 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?
Group Problem 4 [50]: C Programming
The purpose of this problem is to give you some experience writing C code. In particular, you will get some experience dealing with explicit pointers and explicit storage management.

In this problem, your task is to write a C program in the file ~/cs251/ps8-group/sortlines.c that sorts the lines of text from an input file. The input to the program is a text file, whose name is specified as the first command-line argument to the program. The output of the program is the lines of the file, sorted in lexicographic order, printed to standard output. For example, suppose the file tiny.txt contains the following 16 lines:

kanji
mace
each
aback
dad
lab
ibex
ha
gab
fable
oaf
cab
jab
babe
nab
pace

Then your program should behave as shown below:

[gdom@cardinal ps8-group] gcc -o sortlines sortlines.c
[gdom@cardinal ps8-group] ./sortlines tiny.txt
aback
babe
cab
dad
each
fable
gab
ha
ibex
jab
kanji
lab
mace
nab
oaf
pace

Your program should sort the lines using the following steps:

1. Open the file and read each line of the file into a list of strings. Study the readline and sumlist examples from Handout #29 to see how to do this. The string for each line should not include the terminating newline character from the line but should include the terminating NULL character for the string. You will need to define a stringlist type similar to the
intlist type in the sumlist program. You will need to malloc both the nodes of the string list and the strings in the list. The order of the strings in this list is immaterial, though you should not sort the strings yet.

2. Once you have a string list of all the lines in the file, create an array of all the strings in the list and deallocate (using free) any space associated with the list nodes. Again, the order of the strings in the array does not matter, but you should not sort the strings yet.

3. Use an in-place quicksort algorithm to sort the strings in the array. (A array sorting algorithm is in-place if it uses only constant memory in addition to the array being sorted.) You may look at an algorithms book to remind yourself how to do quicksort; the Lomuto partitioning method is a particularly good approach. There are many algorithms texts in Sci 173 that you may consult, some of which have C programs for quicksort (which you may adapt to suit your purposes).

4. Use printf to print to standard output the strings of the sorted array in lexicographic order, one per line.

Notes:

- In addition to studying the C examples on Handout #41, you should also study the examples in App. B.
- Include the following declarations at the top of your file:

```c
#include <stdio.h>
#include <stddef.h>
```

- Use any auxiliary functions you find helpful to simplify the structure of your program.
- You can test your program on the files tiny.txt (16 lines), small.txt (476 lines), medium.txt (5525 lines), and large.txt (45425 lines). All of these are files containing randomly permuted words (one word per line) from the Linux dictionary. The files are all in your ps7 directory. For example, here is a simple test of your program:

```
./sortlines tiny.txt
```

You can “pipe” the output of your sortlines program to a text file using the > redirection operator. E.g.:

```
./sortlines tiny.txt > tiny-sorted.txt
```
A Scheme Streams

We have seen that lazy lists are supported by lazy languages like HASKELL and call-by-lazy HOILIC. However, it is not difficult to implement lists with lazy features in strict functional languages like SCHEME and and OCAML. Indeed, MIT-Scheme supports lazy lists known as streams. In a stream, the head elements are computed eagerly but the tails are computed lazily. Here are the contracts for MIT-Scheme's stream operations:

\[(\text{cons-stream } E_{\text{head}} E_{\text{tail}})\]

Creates a stream node. The head expression, \(E_{\text{head}}\), which denotes the first element of the stream, is evaluated strictly. The tail expression, \(E_{\text{tail}}\), which denotes the remaining elements of the stream, is evaluated lazily. \((\text{cons-stream } E_{\text{head}} E_{\text{tail}})\) is equivalent to \((\text{cons } E_{\text{head}} (\text{delay } E_{\text{tail}}))\).

\[(\text{head stream})\]

Returns the head component of a stream. Equivalent to \((\text{car stream})\).

\[(\text{tail stream})\]

Returns the tail component of a stream, forcing any delayed computations if necessary. Equivalent to \((\text{force (cdr stream)})\).

the-empty-stream

Denotes the empty stream. Equivalent to \(\text{'}()\).

\[(\text{stream-null? stream})\]

Returns \#t if \(\text{stream}\) is the empty stream and \#f otherwise. Equivalent to \((\text{null? stream})\).

For example, here are some infinite streams defined in SCHEME using the stream-processing functions presented in Fig. 5.

\[(\text{define ones (cons-stream 1 ones))}\];Value: ones

\n(take 20 ones)
;Value 1: (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)

\n(\text{define nats (cons-stream 0 (map-inf-stream (lambda (x) (+ x 1)) nats)))}\);Value: nats

\n(take 20 nats)
;Value 3: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)

\n(\text{define fibs (cons-stream 0 (cons-stream 1 (map-inf-stream2 + fibs (tail fibs))))})\);Value: fibs

\n(take 20 fibs)
;Value 5: (0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181)
B C Programming

This appendix presents some simple C programs. For more on C programming, see Scott Anderson’s *C and C++ for Java Programmers* and/or consult the many C programming books available in SCI 173.

B.1 Multiplication Table

```c
// multable.c
// Print an nxn multiplication table
int main () {
    int n = 10;
    int i, j;
    // Print row labels
    for (i=1; i<=n; i++) {
        for (j=1; j<=n; j++) {
            printf("\t%d",i*j);
        }
        printf("\n");
    }
}
```

```
[fturbak@jaguar c] gcc -o multable multable.c
[fturbak@jaguar c] ./multable
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```
B.2 Reading Standard Input

// plus.c
// Read two numbers from user and add them
int main () {
    int a; // storage for first input
    int b; // storage for second input
    printf("a=");
    scanf("%d", &a); // read integer into a
    printf("b=");
    scanf("%d", &b); // read integer into b
    printf("%d+%d=%d\n", a, b, a + b);
}

[fturbak@jaguar c] gcc -o plus plus.c
[fturbak@jaguar c] ./plus
a=3
b=4
3+4=7

B.3 Command Line Arguments

// mainargs.c
// Illustrates command line arguments
int main (int argc, char** argv) {
    int i;
    printf("\n", argc);
    for (i=0; i<argc; i++) {
        printf("%s\n", argv[i]);
    }
}

[fturbak@jaguar c] gcc -o mainargs mainargs.c
[fturbak@jaguar c] ./mainargs foo bar baz
4
./mainargs
foo
bar
baz

// plusargs.c
// Adds up all the numbers in the command line arguments
int main (int argc, char** argv) {
    int i;
    int sum = 0;
    for (i=1; i<argc; i++) {
        sum += atoi(argv[i]); // atoi converts integer to string
    }
    printf("sum=%d\n", sum);
}

[fturbak@jaguar c] gcc -o plusargs plusargs.c
[fturbak@jaguar c] ./plusargs 3 42 17
sum=62
B.4 Reading From a File

// readlines.c
// reads and displays lines from a file
#include <stdio.h>

int main (int argc, char** argv) {
    int i = 0;
    int line = 1;
    char c;
    char buff[128];
    FILE* f = fopen(argv[1], "r"); // open file named in argv[1] for reading
    while ((c = fgetc(f)) != EOF) { // EOF is "end of file" marker
        if (i >= 128) {
            printf("buffer overflow!\n");
            exit(0); // abort program if buffer overflow;
            // there are security problems if this not done!
        } else if (c == '\n') {
            buff[i] = 0;
            printf("%d: %s\n", line, buff);
            i = 0;
            line++;
        } else {
            buff[i++] = c;
        }
    }
}

[fturbak@jaguar c] gcc -o readlines readlines.c
[fturbak@jaguar c] ./readlines tiny-sorted.txt
1: aback
2: babe
3: cab
4: dad
5: each
6: fable
7: gab
8: ha
9: ibex
10: jab
11: kanji
12: lab
13: mace
14: nab
15: oaf
16: pace
CS251 Problem Set 8 Individual Problems
Due Saturday, May 14

Name:

Date & Time Submitted:

By signing below, I attest that I have followed the policy for individual problems set forth in the Course Information handout. In particular, I have not consulted with any person except Lyn about these problems and I have not consulted any materials from previous semesters of CS251.

Signature:

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 design future problem sets. I will fill out the Score column when grading your problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

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.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [45]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [35]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>