CS251 Programming Languages **Handout # 30**
Prof. Lyn Turbak **Wednesday, April 21**
Wellesley College

Problem Set 7
Due: Midnight, Tuesday April 27, 2004

Scheduling:
This assignment is due at midnight on Tuesday, April 27. This assignment will be followed by Exam 2, a take-home exam that will be distributed by Tuesday, April 27 and will be due at midnight on Tuesday, May 4. There will be an optional Problem Set 8 on the final material covered in class distributed during the last week of class.

Overview:
The purpose of this assignment is to give you practice with reasoning about parameter passing, laziness, and memory management in the context of HOILIC, SCHEME, HASKELL, JAVA, and C.

Reading:

- Handout #28: You Can Do More if You're Lazy
- Hughes's paper, "Why Functional Programming Matters"
- Handout #29: C Examples
- Handout #31: Scott Anderson's "C and C++ for Java Programmers"
- Handout #32: HASKELL and HUGS

Working Together:
If you worked with a partner on a previous problem set and want to work with a partner on this assignment, you are encourage to choose a different partner. However, you may also work with someone you worked with in the first half of the semester.

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by midnight on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the 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 pencil-and-paper diagrams and answers to 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. the final version of `sortlines.c` for Problem 3.
5. your pencil-and-paper answers to Problem 4.

Each team should also submit a single softcopy (consisting of your final `ps7` directory) to the drop directory `~cs251/drop/ps7/`*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 the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs251
cp -R ps7 ~cs251/drop/ps7/username/
```

Problem 1 [20]: Parameter Passing

Consider the following HOILIC expression:

```
(bind n 0
  (bind add-twice (abs x
                    (begin (<- x (* 2 x))
                           (<- n (+ n x))
                           n))
    (bind test (abs z
                 (+ (* 100 (add-twice n))
                    (+ (* 10 z) z)))
      (test (add-twice 1)))))
```

For each of the following parameter-pasing mechanisms, (i) draw a diagram that shows how the above expression is evaluated in lexically-scoped HOILIC using that parameter-passing mechanism and (ii) indicate the value of the expression.

- Call-by-value
- Call-by-name
- Call-by-need
- Call-by-reference

Note: Your `~/cs251/hoilic` directory contains interpreters for all four parameter passing mechanisms. You can use them to check your answers, but the diagrams are essential.

Problem 2 [35]: Lazy Data

a. [5]: Why Laziness Matters 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]: Square roots Create a file `~/cs251/ps7/sqrt.scm` in which you translate the Newton-Rhapson square-root example from pp. 27–29 of Hughes's paper into SCHEME using streams (see App. A). Use your procedure to compute the square root of 2 with tolerances of 1, 0.1, and 0.01.

c. [10]: Hamming Numbers in HASKELL Create a file `~/cs251/ps7/Hamming.hs` in which you define the following HASKELL functions. (See Handout #32 for how to write and test HASKELL functions.)

- 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.

d. [10]: Hamming Numbers in JAVA

- In the file `~/cs251/ps7/Hamming.java`, flesh out the skeleton of the `Hamming` class (Fig. 1) that implements the `Enumeration` interface and enumerates the Hamming numbers. Study the `FibEnumeration` class at the end of Handout #28 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 1: Skeleton of the `Hamming` class for enumerating Hamming numbers.

Problem 3 [30]: 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/ps7/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:

```
[fturbak@jaguar ps7] gcc -o sortlines sortlines.c
[fturbak@jaguar ps7] ./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. 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:

- Include the following declarations at the top of your file:

```
#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
```

Problem 4 [15]: 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. **[3]** 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. **[7]** 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. **[5]** 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?

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:

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

(head *stream*)
Returns the head component of a stream. Equivalent to `(car stream)`.

(tail *stream*)
Returns the tail component of a stream, forcing any delayed computations if necessary. Equivalent to `(force (cdr stream))`.

the-empty-stream
Denotes the empty stream. Equivalent to `'()`.

(stream-null? *stream*)
Returns `#t` if *stream* is the empty stream and `#f` otherwise. Equivalent to `(null? stream)`.

For example, here are some infinite streams defined in SCHEME using the stream-processing functions presented in Fig. 2.

```
(define ones (cons-stream 1 ones))
;Value: ones

(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)

(define nats (cons-stream 0 (map-inf-stream (lambda (x) (+ x 1)) nats)))
;Value: nats

(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)

(define fibs (cons-stream 0 (cons-stream 1 (map-inf-stream2 + fibs (tail fibs)))))
;Value: fibs

(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)
```

```
;; Create a (strict) list from the first n elements of a (lazy) stream
(define (take n str)
  (if (or (stream-null? str) (= n 0))
      '()
      (cons (head str) (take (- n 1) (tail str)))))

(define (map-inf-stream f str)
  (cons-stream (f (head str))
               (map-inf-stream f (tail str))))

(define (map-inf-stream2 f str1 str2)
  (cons-stream (f (head str1) (head str2))
               (map-inf-stream2 f (tail str1) (tail str2))))
```

Figure 2: Some SCHEME stream-processing functions.

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

CS251 Problem Set 7
Due Midnitght Tuesday, April 27

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 your problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Problem 2 [35]		
Problem 3 [30]		
Problem 4 [15]		
Total		