CS251 Programming Languages **Handout # 47**
Prof. Lyn Turbak **May 5, 2007**
Wellesley College ***Revised May 16, 2007***

Problem Set 9
Due: Wednesday, May 9

Revisions: (1) Changed the handout date to be 2007 (not 2005) (2) Changed the number of the Hughes paper handout to #53 (#46 was already used for PS7 solutions)

Overview:

This problem set is completely optional. If you do not submit any problems from problem set, there will be no negative impact on your grade. Any problems that you do submit will be included in your group problem set average **only if they would improve your average**. So submitting problems from this assignment can only help your grade.

All of the problems on this assignment are group problems. These problems cover stateful programming, parameter passing, memory management, and lazy data, all of which are possible topics for the final exam. Whether or not you submit any problems from this problem set, you should study the solutions to these problems to prepare for the final exam.

Reading:

- Handout #45: Parameter Passing
- Handout #48: You Can Do More If You're Lazy
- Handout #49: Haskell and HUGS
- Handout #50: Compound Data and Memory Management
- Handout #51: (optional) *Garbage Collection* chapter from Turbak & Gifford with Sheldon's *Design Concepts in Programming Languages*
- Handout #53: John Hughes's article *Why Functional Programming Matters*

Working Together:

On this assignment, students may work together in groups of any size. Also, the rule about all students on a group working at the same time will be relaxed. As long as each student in the group is contributing to every group problem and is communicating with other group members on a frequent basis, students in a group may work in whatever way is best for their schedules.

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 on Wednesday, May 9.

1. a team header sheet indicating the time that you (and your partners, if you working with some) spent on the parts of the assignment.

2. your (i) environment diagram from Problem 1a and (ii) your HOILIC program `diagram.hic` from Problem 1b;

3. your environment diagrams and values for Problem 2;

4. your HOILIC expression file `param.hic` from Problem 3;

5. your HOILIC definition file `cell.hic` from Problem 4;

6. your (i) paragraph for Problem 5a, (ii) final version of `sqrt.hec` for Problem 5b, (iii) final version of `Hamming.hs` for Problem 5c (iv) final version of `Hamming.java` and your answer to the efficiency question for Problem 5d.

7. your pencil-and-paper answers to Problem 6.

Each team should also submit a single softcopy (consisting of your final `ps9-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:

```
cd /students/username/cs251
cp -R ps9-group ~cs251/drop/ps9/username/
```

Group Problems

Group 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 expresssion E_{test} is evaluated in environment frame F_1.

E_{test} ≡
```
(seq (println (list a b))
     (println (list (g 1) (h 1)))
     (h "b") (println (list (g 1) (h 1)))
     (g "a") (println (list (g 1) (h 1)))
     (g "b") (println (list (g 1) (h 1)))
     (h "g") (println (list (g 1) (h 1)))
     (println (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:

```
(bindrec ((I1 E1) ... (In En)) Ebody)
  ⇝ (bindpar ((I1 #f) ... (In #f))
       (seq (<- I1 E1)
                 ⋮
            (<- I1 En)
            Ebody))
```

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

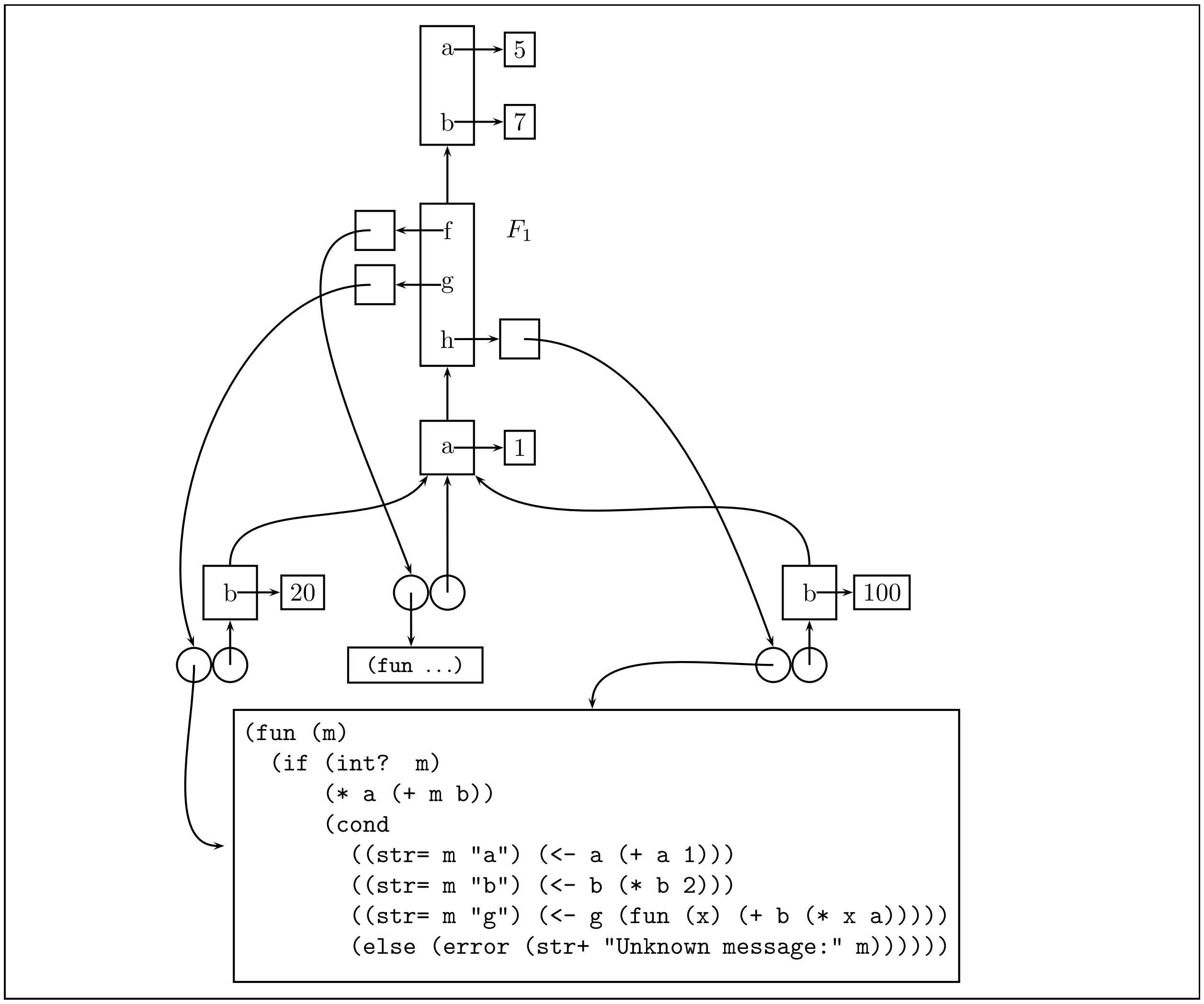
```
(bindrec ((a 1)
          (f (fun () (seq (<- a (* a 10)) a)))
          (b (* 2 a))
          (c (f))
          (d (+ (* 3 a) (+ (* 4 (f)) (* 5 a)))))
  (list a b c d))
```

  evaluates 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/ps9-group/diagram.hic` and can test it by executing the following in the OCAML interpreter:

```
#cd "/students/your-username/cs251/ps9-group";;
#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`.



```
(fun (m)
  (if (int?  m)
      (* a (+ m b))
      (cond
        ((str= m "a") (<- a (+ a 1)))
        ((str= m "b") (<- b (* b 2)))
        ((str= m "g") (<- g (fun (x) (+ b (* x a)))))
        (else (error (str+ "Unknown message:" m))))))
```

Figure 1: A HOILIC environment diagram.

Group Problem 2 [20]: Parameter Passing

Consider the following HOILIC expression:

```
(bind a 1
  (bind inc! (fun () (seq (<- a (+ a 1)) a))
    (bind f (fun (y z)
              (seq (<- y (+ y 3))
                   (+ a (* z z))))
      (f a (inc!)))))
```

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

- Call-by-value
- Call-by-reference
- Call-by-name
- Call-by-lazy (i.e., call-by-need)

Notes:

- Remember that in HOILIC an environment associates names with implicit cells. In your diagrams, every environment name should be associated with a box representing the cell. The contents of the cell may change over time.
- In the environment diagram for call-by-name, represent a thunk as box with named `exp` (expression) and `env` (environment) components.
- In the environment diagram for call-by-lazy, represent a promise as box with named `exp` (expression), `env` (environment), and `memo` (memoized result cell) components.
- The diagrams are essential for getting credit on this problem. However, you can check the value of the expression under the four parameter-passing mechanisms as follows:

```
# #cd "/students/your-username/cs251/ps9-group";;

# #use "load-hoilic-all.ml";;
  ... lots of printout omitted ...

# testHoilicExpFile "prob2.hic";;
```

Group Problem 3 [20]: Parameter-Passing Mechanisms

In the file `~/cs251/ps9-group/param.hic`, write a single HOILIC expression (let's call it E_{param}) such that:

- E_{param} evaluates to the symbol `(sym value)` in call-by-value HOILIC;
- E_{param} evaluates to the symbol `(sym reference)` in call-by-reference HOILIC;
- E_{param} evaluates to the symbol `(sym name)` in call-by-name HOILIC;
- E_{param} evaluates to the symbol `(sym lazy)` in call-by-lazy (i.e., call-by-need) HOILIC.

Notes:

- E_{param} may be built out of any kinds of HOILIC expressions, but the only types of values that these expressions should manipulate are symbols, functions, and the implicit mutable variables of HOILIC. That is, your example should not use any integers, booleans, characters, strings, or lists. Strive to make your expression as simple and understandable as possible.
- If you cannot solve this problem with just symbols, 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:

  ```
  #cd "/students/your-username/cs251/ps9-group";;

  # #use "load-hoilic-all.ml";;
    ... lots of printout omitted ...

  # testHoilicExpFile "param.hic";;
  ```

  The first two lines load the HOILIC interpreter and testing code. These need to be evaluated only once. You can re-evaluate the final expression every time you change `param.hic`. The result of evaluating this expression should be:

  ```
  # testHoilicExpFile "param.hic";;
  Value of expression in CBV Hoilic: (sym value)
  Value of expression in CBR Hoilic: (sym reference)
  Value of expression in CBN Hoilic: (sym name)
  Value of expression in CBL Hoilic: (sym lazy)
  - : unit = ()
  ```

Group Problem 4 [15]: 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 `~/ps9-group/cell.hic`:

(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 #42 (where in this case messages are the booleans `#t` and `#f`). However, your definitions should be considerably simpler than those in the OOP example from Handout #42. Each expression should be at most a few lines long.

- 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:

  ```
  #cd "/students/your-username/cs251/ps9-group";;
  #use "load-cell.ml";;
  testCell();;
  ```

  The first two lines load the HOILIC interpreter and testing code. These need to be evaluated only once. You can re-evaluate `testCell` every time you change `cell.hic`. Here is what the transcript of `testCell()` should look like:

  ```
  # testCell();;
  Creating cell c1 via (cell 17)
  Creating cell c2 via (cell 42)
  Value of (^ c1) is now: 17
  Value of (^ c2) is now: 42
  Value of (:= c1 (* 2 (^ c1))) is: 17
  Value of (^ c1) is now: 34
  Value of (^ c2) is now: 42
  Value of (:= c1 (:= c2 (^ c1))) is: 34
  Value of (^ c1) is now: 42
  Value of (^ c2) is now: 34
  34
  - : unit = ()
  ```

Group Problem 5 [55]: Lazy Data

a. [10]: Why Laziness Matters In his paper, "Why Functional Programming Matters" (Handout #53), John Hughes argues that lazy evaluation is an essential feature of the functional programming paradigm. Briefly summarize his argument in one paragraph.

b. [15]: Square roots Create a file `~/cs251/ps9-group/sqrt.hec` in which you translate the Newton-Rhapson square-root example from pp. 27–29 of Hughes's paper into HOILEC using streams (i.e., lists whose tails are lazy). As described in Handout #48, here is a HOILEC interface to streams:

(**sprep** E_{head} E_{tail})
Create a stream whose head is the value of E_{head} and whose tail is the delayed computation of E_{tail}.

(**shead** *str*)
Returns the head of the stream `str`.

(**stail** *str*)
Returns the stream that is the tail of `str`. Invoking this for the first time on a stream whose delayed tail has not yet been computed causes it to be computed. The result of this computation is returned by any subsequent invocation of `stail` on the same stream.

(**sempty**)
Returns an empty stream.

(**sempty?** *str*)
Returns `#t` if `str` is an empty stream, and `#f` otherwise.

The OCAML modules `HoilecStreams` and `HoilecStreamsEnvInterp` implement a version of the HOILEC language that supports both streams and floating point numbers. The names of floating point operations are obtained by adding `f` in front of the corresponding integer operations: `f+`, `f-`, `f*`, `f/`, `f<`, `f<=`, `f=`, `f!=`, `f>=`, and `f>`. Floating point literals must include an explicit dot. E.g., you must write `(f+ 1.0 3.1459)` rather than `(f+ 1 3.1459)`.

You can test your file as shown in the following transcript, which finds the square root of 2 at various tolerances:

```
#cd "/students/your-username/cs251/ps9-group";;

# #use "load-hoilec-streams.ml";;
  ... lots of printout omitted ...

# repl();;

hoilec-streams> (load "sqrt.hec")
sqrt
repeat
within
fabs
next

hoilec-streams> (sqrt 1.0 1.0 2.0)
1.5

hoilec-streams> (sqrt 1.0 0.1 2.0)
1.41666666667
```

```
hoilec-streams> (sqrt 1.0 0.01 2.0)
1.41421568627

hoilec-streams> (sqrt 1.0 0.001 2.0)
1.41421356237

hoilec-streams> (sqrt 1.0 0.0001 2.0)
1.41421356237
```

c. [15]: Hamming Numbers in **HASKELL** Create a file `~/cs251/ps9-group/Hamming.hs` in which you define the following HASKELL functions. (See Handout #49 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.

For example, here is a transcript of some tests of these functions on `wampeter.wellesley.edu` (the only Linux machine on which HASKELL is installed):

```
[fturbak@wampeter fturbak] hugs

__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2001
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: December 2001  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/usr/share/hugs/lib/Prelude.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Type :? for help

Prelude> :cd ~/cs251/ps9-group

Prelude> :load hamming.hs
Reading file "hamming.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hamming.hs
```

```
Main> take 10 (scale 5 nats) where nats = 0 : (map (1+) nats)
[0,5,10,15,20,25,30,35,40,45]

Main> take 30 (merge (scale 4 nats) (scale 7 nats)) where nats = 0 : (map (1+) nats)
[0,4,7,8,12,14,16,20,21,24,28,32,35,36,40,42,44,48,49,52,56,60,63,64,68,70,72,76,77,80]

Main> take 20 (merge (scale 4 nats) (scale 7 nats)) where nats = 0 : (map (1+) nats)
[0,4,7,8,12,14,16,20,21,24,28,32,35,36,40,42,44,48,49,52]

Main> take 25 (merge (scale 4 nats) (scale 7 nats)) where nats = 0 : (map (1+) nats)
[0,4,7,8,12,14,16,20,21,24,28,32,35,36,40,42,44,48,49,52,56,60,63,64,68]

Main> take 52 hamming
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54,60,64,72,75,80,81
90,96,100,108,120,125,128,135,144,150,160,162,180,192,200,216,225,240,243,250,256]
```

d. [15]: Hamming Numbers in Java

i. In the file `~/cs251/ps9-group/Hamming.java`, flesh out the skeleton of the `Hamming` class (Fig. 2) that implements the `Iterator<String>` interface and enumerates the Hamming numbers. Study the `FibIterator` class at the end of Handout #48 as an example of a Java class that enumerates an infinite sequences of integers. As in `FibIterator`, you will have to iterate integers wrapped in the `Integer` class to satisfy the constraint that `next` must return an `Integer`.[1]

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 iterator, or `java Hamming n`, which will display the first n elements of your iterator.[2] For example:

```
[fturbak@puma ps9-group] javac Hamming.java

[fturbak@puma ps9-group] java Hamming
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 50 54 60 64 72 75
80 81 90 96 100 108 120 125 128 135 144 150 160 162 180 192 200 216 225 240 243
250 256

[fturbak@puma ps9-group] java Hamming 10
1 2 3 4 5 6 8 9 10 12

[fturbak@puma ps9-group] java Hamming 100
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 50 54 60 64 72 75
80 81 90 96 100 108 120 125 128 135 144 150 160 162 180 192 200 216 225 240 243
250 256 270 288 300 320 324 360 375 384 400 405 432 450 480 486 500 512 540 576
600 625 640 648 675 720 729 750 768 800 810 864 900 960 972 1000 1024 1080 1125
1152 1200 1215 1250 1280 1296 1350 1440 1458 1500 1536
```

ii. 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.

[1]Actually, if you forget to wrap the `int` value in an `Integer` object, the *auto-boxing* feature of Java 1.5 will automatically do this for you.

[2]If n is a non-negative integer, then `java Hamming n` will enumerate the first n elements.

```
import java.util.*; // imports Iterator interface

public class Hamming implements Iterator<Integer> {

  // Put instance variable(s) here.

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

  public boolean hasNext () {return true;}

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

  // The Iterator interface requires that this method be implemented
  public void remove() {
    throw new UnsupportedOperationException(
                "This iterator does not support the remove() operation.");
  }

  // 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]);
    }
    Iterator<Integer> h = new Hamming();
    while (i > 0) {
      System.out.print(h.next());
      System.out.print(" ");
      i--;
    }
    System.out.println();
  }

}
```

Figure 2: Skeleton of the `Hamming` class for enumerating Hamming numbers.

Group Problem 6 [40]: Garbage Collection

Consider the list memory shown below, in which the slot at every odd address a represents the head of a list node and the slot at its associated even address $a+1$ represents the tail of a list node. Because all blocks in this memory are list nodes and known to have two slots, there is no need for header blocks in this memory. Assume that entities beginning with n are immediate integers and entities beginning with p are pointers. $p0$ 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. **[10]** 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. **[20]** 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. **[10]** 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?

Extra Credit 1 [80]: More Garbage Collection

Do problems 18.5 and 18.7 in the Turbak & Gifford with Sheldon *Garbage Collection* chapter. Partial credit will be awarded for progress made on any parts of these problems.

Group Problem Header Page
Please make this the first page of your hardcopy submission for group problems.

CS251 Problem Set 9 Group Problems
Due Wednesday, May 9

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.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Problem 2 [20]		
Problem 3 [20]		
Problem 4 [15]		
Problem 5 [55]		
Problem 6 [40]		
Total		