CS251 Programming Languages **Handout # 38**
Randy Shull **April 22, 2013**
Wellesley College

Problem Set 8

Due: 5pm Wednesday, May 8, 2013

Overview:

The individual problem on this assignment tests your understanding of environment diagrams and scoping. The group problems on this assignment cover closure conversion and imperative programming.

Reading:

- Handout #34: FOFL and FOBS: First-Order Functions
- Handout #35: Closure Conversion
- Handout #36: HOILEC: Imperative Programming with Explicit Cells
- Handout #39: HOILIC: Imperative Programming with Implicit Cells
- Handout #40: Parameter Passing

Individual Problem Submission:

Each student should turn in a hardcopy submission packet for the individual problem by slipping it under your instructor's office door by the deadline given above. The packet should include:

1. an individual problem header sheet;
2. your environment diagram from Problem 1.

There is no softcopy submission for this problem.

Working Together:

Reminder: if you worked with a partner on a previous problem set and want to work with a partner on this assignment, you should try to find a different partner if possible.

Group Problem Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Your instructor's door by the deadline given above. The packet should include:

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. your pencil-and-paper solutions to Group Problems 1, 4a, and 6. (including your environment diagrams and values for Problem 6);
3. your final version of `decorate.ffl` from Problem 2.
4. a transcript of running `testDecorate()`.
5. your final version of `Process.java` from Problem 3.
6. a transcript of your tests involving the JAVA `Process` class.
7. your final version of `Counters.java` from Problem 4b.
8. a transcript of your tests involving the JAVA `Counters` class.
9. your HOILIC definition file `cell.hic` from Problem 5;
10. a transcript of running `testCell()`.

Each team should also submit a single softcopy (consisting of your final `ps8-group` directory) to the drop directory `~cs251/drop/ps8/`*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 ps8-group ~cs251/drop/ps8/username/
```

Individual Problems

This is an individual problem. Each student must solve this problem on her own without consulting any other person (except your instructor).

Individual Problem 1 [25]: Environmental Action

Draw an environment diagram that shows all the environments and closures that are created when the following program is run on the input argument list `[3;5]` in statically scoped HOFL:

```
(hofl (i j) (bind q (f p)
              (bind r (q i j)
                (bind answer (p (r m) (r (f m)))
                  answer)))
  (def (f g) (fun (a b) (g b a)))
  (def (p d e) (fun (h) (h d e)))
  (def (m x y) (% x y)))
```

Follow the conventions for drawing environment diagrams used in Problem 1 of Problem Set 6. Think carefully about the parent pointer of each environment. Your diagram should accurately show the sharing of closure values. For example, if the same closure c is named I_1 in environment frame e_1 and and I_2 in environment frame e_2, then c should appear exactly once in your diagram and there should be arrows from I_1 in e_1 to c and from I_2 in e_2 to c.

Group Problems

Group Problem 1 [15]: 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) HOFL and (ii) HOILEC (assume both languages are statically scoped and call-by-value). 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 or signals an error before the transformation, but returns a value after the transformation.
- the program returns a value before the transformation, but loops infinitely or signals an error after the transformation.

For the purposes of this problem, all errors are considered equivalent and programs that loop infinitely are considered equivalent to those that signal an error. In particular, the behavior of an expression should not be considered different if it signals an error both before and after the transformation, but the error messages are different.

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`)` $\Longrightarrow$ `(bind x` E_1 `(+ x` E_2`))`

e. `(+` E_1 E_2`)` $\Longrightarrow$
```
(bindpar ((x (fun () E1))
          (y (fun () 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`))`

Group Problem 2 [15]: Tree Decoration

Consider the following HOILEC function:

```
(def (decorate ornaments) ;; Assume ornaments non-empty
  (bind orns (cell ornaments) ;; orns is a cell holding a non-empty
                              ;; list of ornaments
    (bindrec ((visit (fun (tree)
                       (cond ((not (list? tree))
                              (bind orn (head (^ orns))
                                (seq (:= orns (if (empty? (tail (^ orns)))
                                                  ornaments
                                                  (tail (^ orns))))
                                     orn)))
                             ((empty? tree) #e)
                             (else (prep (visit (head tree))
                                         (visit (tail tree)))))))))
      visit)))
```

Below are some sample invocations of `decorate`:

```
hoilec> (decorate (list 1 2) 17)
1

hoilec> (decorate (list 1 2) #e)
#e

hoilec> (decorate (list 1 2) (list 23 #t 'c' (sym foo) "bar"))
(list 1 2 1 2 1)

hoilec> (decorate (list 4 5 6) (list 23 #t 'c' (sym foo) "bar"))
(list 4 5 6 4 5)

hoilec> (decorate (list 'p' "q" (sym r)) (list 23 #t 'c' (sym foo) "bar"))
(list 'p' "q" (sym r) 'p' "q")

hoilec> (decorate (list 1 2 3 4 5 6 7) (list 23 #t 'c' (sym foo) "bar"))
(list 1 2 3 4 5)

hoilec> (decorate (list 1 2)
                  (list (list "a" "b" (list "c" "d")) "e" (list (list "f") "g")))
(list (list 1 2 (list 1 2)) 1 (list (list 2) 1))

hoilec> (decorate (list 4 5 6)
                  (list (list "a" "b" (list "c" "d")) "e" (list (list "f") "g")))
(list (list 4 5 (list 6 4)) 5 (list (list 6) 4))
```

```
hoilec> (decorate (list 'p' "q" (sym r))
                  (list (list "a" "b" (list "c" "d")) "e" (list (list "f") "g")))
(list (list 'p' "q" (list (sym r) 'p')) "q" (list (list (sym r)) 'p'))

hoilec> (decorate (list 1 2 3 4 5 6 7 8)
                  (list (list "a" "b" (list "c" "d")) "e" (list (list "f") "g")))
(list (list 1 2 (list 3 4)) 5 (list (list 6) 7))
```

a. **[3]: G**ive a English specification for `decorate`. I.e., explain at a high level what it does.

b. **[12]: I**t is possible to translate `decorate` from HOILEC, which has nested functions and mutable cells, to FOFL, which has neither. Here is a skeleton of the corresponding FOFL program:

```
(def (decorate ornaments tree)
  (nth 1 (visit tree ornaments ornaments)))

(def (visit tree ornaments orns)
  (cond ((not (list? tree))
         (bind orn (head orns)
           (list E1 E2)))
        ((empty? tree) (list E3 E4))
        (else (bind visit-head (visit (head tree) E5 E6)
                (bind visit-tail (visit (tail tree) E7 E8)
                  (list E9 E10))))))
```

The nested function `visit` in HOILEC has been translated to a top-level FOFL function that takes extra arguments for `ornaments` and `orns`. The side effects that `visit` performs on the cell `orns` can be simulated by threading the contents of this cell through every call of `visit`. This is done in FOFL by passing the contents of the cell as the `orns` argument to `visit` and having `visit` return a list with two components: (1) the regular tree result of `visit` and (2) the updated contents of the cell.

Your task is to flesh out the skeleton for the FOFL `visit` skeleton in `~/cs251/ps8-group/decorate.ffl` so that any invocation of the FOFL version of `decorate` on two arguments yields the same result as invoking the HOFL version of `decorate` on the same two arguments.

To test your definition, execute the following in OCAML:

```
# #cd "/students/username/cs251/ps8-group"
# #use "load-decorate.ml"
# testDecorate();;
```

Group Problem 3 [25]: Manually Converting OCAML to JAVA

In this problem you will manually translate an OCAML program with block structure and higher-order functions into a JAVA program that has neither block structure nor higher-order functions.

Consider the OCAML list-processing function `process` in Fig. 1. Given an input integer list, `process` generates an output integer list. You should study the definition of `process` carefully to understand what it does. Here are some examples of `process` in action:

```
process [3;4;5;6;7];;
- : int list = [1; 2; 3; 13; 15; 17; 19; 21]

# process [5;7;4;5;7];;
- : int list = [2; 3; 2; 3; 35; 47; 29; 35; 47]

# process [5;7;4;6;5;7];;
- : int list = [2; 3; 2; 3; 15; 17; 14; 16; 15; 17]
```

```
let process xs =
  let rec scan1 ys f =
    match ys with
      [] -> mapxs f
    | (y::ys') ->
        if (y mod 2) = 0 then
          scan2 ys' f
        else
          let y' = y / 2
          in y' :: (scan1 ys' (fun a -> f (a + y')))
  and scan2 zs g =
    match zs with
      [] -> mapxs g
    | (z::zs') ->
        if (z mod 2) = 0 then
          scan1 zs' g
        else
          let z' = z / 2
          in z' :: (scan2 zs' (fun b -> g (b * z')))
  and mapxs q =
    let rec mapq ws =
      match ws with
        [] -> []
      | (w::ws') -> (q w)::(mapq ws')
    in mapq xs
  in scan1 xs (fun x -> x)
```

Figure 1: The OCAML `process` function.

```
# process [5;7;4;6;8;5;7];;
- : int list = [2; 3; 2; 3; 35; 47; 29; 41; 53; 35; 47]

# process [1;2;3];;
- : int list = [0; 1; 1; 2; 3]

# process [3;2;1];;
- : int list = [1; 0; 1; 1; 1]

# process [1;3;5];;
- : int list = [0; 1; 2; 4; 6; 8]

# process [2;4;6];;
- : int list = [2; 4; 6]
```

Your task in this problem is to translate the OCAML `process` function and its internal functions into JAVA methods that perform the same computations. You should do this by filling out the skeleton file `Process.java` (Fig. 2) that can be found in the `ps8-group` directory. In the `Process` class, you should write methods `process`, `scan1`, `scan2`, `mapxs`, and `mapq` that correspond to the five list-processing functions with the same names in Fig. 1. These functions may need to take additional arguments due to the "flattening" of the OCAML block structure that must be done as part of translating the functions into JAVA. The higher-order functions (i.e., closure values) occuring in the OCAML program can be translated into instances of classes that implement the `IntFun` interface in `Process.ml`. One such class, `IdFun`, which implements identity functions, has been provided for you. You will have to define other classes implementing this interface, either explicitly (via JAVA class definitions) or implicitly (using anonymous inner classes).

Notes:

- The OCAML `process` function can be found in the file `~/cs251/ps8-group/process.ml`.
- Appendix A presents an API for the Java `IntList` class. An implementation of this class (which you do not need to study) is provided in `~/cs251/ps8-group/IntList.java`.
- Use `javac` to compile your file and `java` to test it. E.g.:

```
[gdome@jaguar gdome] cd ~/cs251/ps8-group
[gdome@jaguar ps8-group] javac Process.java
[gdome@jaguar ps8-group] java Process [3,4,5,6,7]
[1,2,3,13,15,17,19,21]
[gdome@jaguar ps8-group] java Process [5,7,4,5,7]
[2,3,2,3,35,47,29,35,47]
[gdome@jaguar ps8-group] java Process [1,2,3]
[0,1,1,2,3]
[gdome@jaguar ps8-group] java Process [2,4,6]
[2,4,6]
[gdome@jaguar ps8-group]
```

Group Problem 4 [30]: Counters

Recall that in HOILIC (1) every variable name is bound to an implicit cell; (2) references to a variable implicitly dereference (return the contents of) the cell; and (3) a variable v can be assigned a new value via the assignment construct `(<- v E)`, which changes the contents of the implicit cell associated with v to the value of E.

a. [**21**] Consider the functions in Fig. 3, which are written in call-by-value lexically-scoped HOILIC. For each of the following expressions, (1) show the values displayed when expression is evaluated and (2) draw an environment diagram for the evaluation of the expression. (In your diagram, be sure to show empty environment frames from invoking nullary functions.)

- `(test-counter make-counter1)`
- `(test-counter make-counter2)`
- `(test-counter make-counter3)`

b. [**9**] Let i range over the numbers {1,2,3}. Then each of the HOILIC functions `make-counteri` can be modeled in JAVA by an instance of class `Counteri` that implements the following interface:

```
interface Counter { public int invoke(); }
```

In addition to its single nullary instance method `invoke`, each class `Counteri` should have a single class, instance, or local variable named `count`. The test expression `(test-counter make-counteri)` in HOILIC can be modeled by the JAVA statement:

```
Counters.testCounters(new Counteri(), new Counteri());
```

where `testCounters` is a class method of the `Counters` class with the following definition:

```
public static void testCounters(Counter a, Counter b) {
  System.out.println(a.invoke());
  System.out.println(b.invoke());
  System.out.println(a.invoke());
}
```

```
// Interface for int -> int functions
interface IntFun {
  public int apply (int x);
}

// The identity function
class IdFun implements IntFun {
  public int apply (int x) {
    return x;
  }
}

// Put other classes implementing the IntFun interface here:
// (Alternatively, you can use anonymous inner classes instead.)

// The Process class defines the process, scan1, scan2, mapxs and mapq methods.
public class Process {

  // Handy way of introducing abbreviation IL. for IntList operations:
  // IL.empty, IL.isEmpty, IL.head, IL.tail, IL.prepend.
  public static IntList IL;

  // Define process here:

  // Define scan1 here:

  // Define scan2 here:

  // Define mapxs here:

  // Define mapq here:

  // Testing method. E.g.:
  // [lyn@jaguar ps8-group] java Process [3,4,5,6,7]
  // [1,2,3,13,15,17,19,21]
  // [lyn@jaguar ps8-group] java Process [5,7,4,5,7]
  // [2,3,2,3,35,47,29,35,47]
  public static void main (String [] args) {
    if (args.length == 1) {
      System.out.println(process(IL.fromString(args[0])));
    } else {
      System.out.println("unrecognized main option");
    }
  }
}
```

Figure 2: The JAVA file `Process.java`.

```
(def make-counter1
  (bind count 0
    (fun ()
      (fun ()
        (seq (<- count (+ count 1))
             count)))))

(def make-counter2
  (fun ()
    (bind count 0
      (fun ()
        (seq (<- count (+ count 1))
             count)))))

(def make-counter3
  (fun ()
    (fun ()
      (bind count 0
        (seq (<- count (+ count 1))
             count)))))

(def test-counter
  (fun (make-counter)
    (bindseq ((a (make-counter))
              (b (make-counter)))
      (seq (println (a))
           (println (b))
           (println (a))))))
```

Figure 3: HOILIC counter functions.

In this subproblem your task is to flesh out the definitions of the `Counteri` classes in the file `Counters.java` so that they correctly model `make-counteri`.

Note: To compile and run the `Counters` program, execute the following Linux shell commands:

```
cd "~/cs251/ps8-group"
javac Counters.java
java Counters
```

Group Problem 5 [15]: Explicit Mutable Cells in HOILIC

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-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 #36 (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 #36. 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/ps8-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 6 [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/ps8-group";;

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

# testHoilicExpFile "prob6.hic";;
```

Appendix A: An `IntList` Class for Java

Problem 3 uses a Java `IntList` class that provides list manipulation functions on immutable integer lists. It has the API given below. The file `~/cs251/ps8-group/IntList.java` contains an implementation of this class.

```
public static IntList prepend (int i, IntList L);
// Returns the new list that results from prepending i onto L.

public static IntList empty ();
// Returns an empty list.

public static boolean isEmpty (IntList L );
// Returns true if L is empty and false otherwise.

public static int head (IntList L);
// Returns the head of list L. Throws an exception if L is empty.

public static IntList tail (IntList L);
// Returns the tail of list L. Throws an exception if L is empty.

public String toString ();
// Returns a string representation of this list.

public static String toString (IntList L);
// Returns a string representation of list L. This is a sequence
// of comma-separated integers delimited by square brackets.

public static IntList fromString (String s);
// Returns the integer list whose string representation is s.
```

If a Java program contains the declaration `static IntList IL;` then the abbreviation `IL.` may be used in place of `IntList.` anywhere in the program.. For example, `IL.head` may be used in place of `IntList.head`

Individual Problem Header Page
Please make this the first page of your hardcopy submission of individual problems.

CS251 Problem Set 8 Individual Problems
Due 5pm Wednesday, May 8

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 my instructor 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 you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [25]		
Total		

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

CS251 Problem Set 8 Group Problems
Due 5pm Wednesday, May 8

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 [15]		
Problem 2 [15]		
Problem 3 [25]		
Problem 4 [30]		
Problem 5 [15]		
Problem 6 [20]		
Total		