CS251 Programming Languages **Handout # 25**
Prof. Lyn Turbak **March 17, 2004**
Wellesley College ***Revised April 1, 2004***

Problem Set 5
Due: 6pm Friday, April 2

Revisions:
April 1: In Problem 4, the first classify problem was missing close parens in the third and fourth clauses.

Overview:
The purpose of this assignment is to give you practice with reasoning about the HOFL language (including higher-order functions, scoping, and recursion) and with extending interpreters with primitives, desugarings, and new constructs.

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 6pm 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 pencil-and-paper diagrams and answers to Problem 2
4. your final versions of `ps5/Hofl.ml` and `ps5/HoflEnvInterp.ml` from Problems 3 and 4.
5. your Java program `Process.java` from Problem 5.

Each team should also submit a single softcopy (consisting of your final `ps5` directory) to the drop directory `~cs251/drop/p3/`*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 ps5 ~cs251/drop/ps5/username/
```

Problem 0: Studying HOFL

All of the problems on this problem set involve the HOFL language discussed in class or extensions to this language. Before attempting the problems, you should study the code for the implementation of the HOFL language, which can be found in `~/cs251/hofl` after you perform `cvs update -d`. There are four files to study: `Hofl.ml`, `HoflEnvInterp.ml`, `HoflSubstInterp.ml`, and `Env.ml`. These were distributed as a packet in class.

To use any of the functions defined within files in the `hofl` directory, you should first execute the following directives in OCAML:

```
#cd "/students/username/cs251/hofl"
#use "load-hofl.ml"
```

Having done this, you can now experiment with any functions in the HOFL interpreter. For example:

```
# open HoflEnvInterp;;

# runString "(hofl (n) (bind sq (abs x (* x x)) (sq (sq n))))" [5];;
- : Hofl.valu = Hofl.Int 625

# runFile "even.pgm" [17];;
- : Hofl.valu = Hofl.Bool false

# runFile "even.pgm" [42];;
- : Hofl.valu = Hofl.Bool true

# runFile "expt.pgm" [2;5];;
- : Hofl.valu = Hofl.Int 32

# runFile "expt.pgm" [5;2];;
- : Hofl.valu = Hofl.Int 25

# repl();;

hofl> (#load "simple.def")
Loading simple.def
a
sq
avg
b
Done loading simple.def

hofl> (sq (avg a b))
100

hofl> (def (range lo hi)
        (if (> lo hi)
            (null)
            (prepend lo (range (+ lo 1) hi))))
range

hofl> (range 3 7)
3,4,5,6,7

hofl> (def (zip xs ys)
        (if (|| (null? xs) (null? ys))
            (null)
            (prepend (pair (head xs) (head ys))
                     (zip (tail xs) (tail ys)))))
zip

hofl> (zip (range 1 10) (range 3 7))
<1,3>,<2,4>,<3,5>,<4,6>,<5,7>

hofl> (#quit)
```

Problem 1 [25]: Static and Dynamic Scope in HOFL

a. [**10**] Suppose that the program in Figure 1 is run on the input argument list `[5]`. Draw an environment diagram that shows all of the environments and closures that are created during the evaluation of this program in *statically scoped* HOFL. In order to simplify this diagram:

- you should treat `bind` as if it were a kernel construct and ignore the fact that it desugars into an application of an `abs`. That is, you should treat the evaluation of (`bind` I E_{defn} E_{body}) in environment F as the result of evaluating E_{body} in the environment frame F', where F' binds I_{defn} to V_{defn}, V_{defn} is the result of evaluating E_{defn} in F, and F' is the parent pointer of F.
- you should treat `fun` as if it were a kernel construct and ignore the fact that it desugars into nested abstractions. In particular, (1) the evaluation of (`fun` (`(`I_1 ... I_n`)`) E_{body}) should be a closure consisting of (a) the `fun` expression and (b) the environment of its creation and (2) the application of the closure `<(fun (`I_1 ... I_n`)` E_{body}`), `$F_{\text{creation}}`> to argument values V_1 ...V_n should create a new environment frame F whose parent is F_{creation} and which binds the variables I_1 ...I_n to the values V_1 ...V_n.

```
(hofl (a)
  (bind linear (fun (a b)
                 (fun (x)
                   (+ (* a x) b)))
    (bind line1 (linear 1 2)
      (bind line2 (linear 3 4)
        (bind try (fun (b)
                    (prepend (line1 b)
                             (prepend (line2 (+ b 1))
                                      (prepend (line2 (+ b 2))
                                               (null)))))
          (try (+ a a)))))))
```

Figure 1: A sample HOFL program used to illustrate the difference between static and dynamic scope.

b. [**1**] What is the final value of the program from part (a) in statically scoped HOFL? You should figure out the answer on your own, but may wish to check it using the statically scoped HOFL interpreter.

c. [**7**] Draw an environment diagram that shows all of the environments created in *dynamically scoped* HOFL when running the program from Figure 1 on the input argument list `[5]`.

d. [**1**] What is the final value of the program from part (c) in dynamically scoped HOFL?

e. [**2**] In a programming language with higher-order functions, which supports modularity better: lexical scope or dynamic scope? Explain your answer.

f. [**4**] Suppose that you are given a HOFL interpreter, but you are not told whether it is a statically-scoped or dynamically-scoped version of the interpreter. Write a simple HOFL expression that will evaluate to `true` for a statically-scoped interpreter but will evaluate to `false` for a dynamically-scoped interpreter. The only types of values your expression should manipulate are booleans and functions; it should *not* use integers, pairs, or lists.

Problem 2 [20]: `bindrec`
Consider the following HOFL expression E:

```
(bind f (abs x (+ x 1))
  (bindrec ((f (abs n
                 (if (= n 0)
                     1
                     (* n (f (- n 1)))))))
    (f 3)))
```

a. [**5**] Draw an environment diagram showing the environments created when E is evaluated in *statically scoped* HOFL.

b. [**1**] What is the value of E in *statically scoped* HOFL?

c. [**5**] Consider the expression E' that is obtained from E by replacing `bindrec` by `bindseq`. Draw an environment diagram showing the environments created when E' is evaluated in *statically scoped* HOFL.

d. [**1**] What is the value of E' in *statically scoped* HOFL?

e. [**5**] Draw an environment diagram showing the environments created when E' is evaluated in *dynamically scoped* HOFL.

f. [**1**] What is the value of the expression E' in *dynamically scoped* HOFL?

g. [**2**] Does a dynamically scoped language need a recursive binding construct like `bindrec` in order to support the creation of local recursive procedures? Briefly explain your answer.

Problem 3 [20]: Adding strings to HOFL

Strings

The HOFL language implementation is designed to make it fairly easy to add new primitive dataypes and operations on these datatypes. As an example of this, you will be extending HOFL to handle strings.

Conceptually, a string is just a sequence of characters. We will adopt the convention used in most languages (including C, Java, Ocaml, Scheme, and Haskell) that string literals are denoted by text delimited by double quotes. For example, here are some string literals: `""`, `"a"`, `"cs251"`, `"I do not like them, Sam I am!"`. We want to be able to use string literals in HOFL programs, such as the following:

```
(hofl (n)
  (if (> n 0)
      "positive"
      (if (= n 0)
          "zero"
          "negative")))
```

In addition to including string literals, however, we would also like to extend HOFL with operations that allow analyzing the structure of a string and synthesizing new strings. In particular, consider the following four string operations:

(**strlen** *str*)
Returns the length (number of characters in) the string *str*.

(**strlt** *str1 str2*)
Returns `true` if *str1* is less than *str2* in the lexicographic (dictionary) ordering of strings, and `false` otherwise. For example, the following are arranged in lexicographic order: `""`, `"a"`, `"aa"`, `"ab"`, `"b"`, `"ba"`, `"bb"`.

(**str+** *str1 str2*)
Returns the string that has all the characters of `str1` followed by all of those of `str2`. For example, `(str+ "ab" "bcd")` yields `"abbcd"`.

(**to-string** *val*)
Returns a string representation of the value `val`:

- If `val` is an integer, returns a string of digits representing the integer (preceded by a `-` in the case of negative integers).
- If `val` is a boolean, returns the string `"true"` or `"false"`, depending on the boolean.
- If `val` is a function, returns the string `"<fun>"`.
- If `val` is a string, returns the string.
- If `val` is a pair, returns the string representations of the two pair components separated by a comma and delimited by angle brackets. E.g. `"<1,true>"`.
- If `val` is a list, returns the string representations of the string elements separated by a commas and delimited by squiggly brackets. E.g. `"{2,false,\"foobar\",<1,true>,<fun>}"`.

Your Task

Your task is to extend the HOFL interpreter in the `ps5` directory with string literals and the above four string operations by making appropriate changes to `Hofl.ml` and `HoflEnvInterp.ml`.

Notes:

- You should change the HOFL implementation in the `ps5` directory, *not* in the `hofl` directory.
- To use any parts of the HOFL interpreter, you must first execute the following in OCAML:

```
#cd "/students/username/cs251/ps5"
#use "load-hofl.ml"
```

- You will need to use OCAML string operations in your implementation. For documentation on these, consult the documentation on the OCAML `String` module, available on the CS251 home page.

Problem 4 [15]: Desugaring `classify`

The `classify` *construct*

You are a summer programming intern at Sweetshop Coding, Inc. Your supervisor, Dexter Rose, has been studying the syntactic sugar for HOFL and is very impressed by the `cond` construct. He decides that it would be neat to extend HOFL with a related `classify` construct that classifies an integer relative to a collection of ranges. For instance, using his construct, Dexter can write the following grade classification program:

```
(hofl (grade)
  (classify grade
    ((90 100) "A")
    ((80 89) "B")
    ((70 79) "C")
    ((60 69) "D")
    (otherwise "E")))))
```

This program takes an integer grade value and returns a string indicating which range the grade falls in. (Dexter is of course using the version of HOFL from Problem 3 that supports strings!)

In general, the `classify` construct has the following form:

```
(classify E_disc
  ((E_lo1  E_hi1) E_body1)
     ⋮
  ((E_lon  E_hin) E_bodyn)
   (otherwise E_dflt))
```

The evaluation of `classify` should proceed as follows. First the **discriminant** expression E_{disc} should be evaluated to the value V_{disc}. Then V_{disc} should be matched against each of the clauses $((E_{lo_i}\ E_{hi_i})\ E_{body_i})$ from top to bottom until one matches. The value matches a clause if it lies in the range between V_{lo_i} and V_{hi_i}, inclusive, where V_{lo_i} is the value of E_{lo_i}, and V_{hi_i} is the value of E_{hi_i}. When the first matching clause is found, the value of the associated expression E_{body_i} is returned. If none of the clauses matches V_{disc}, the value of the **default** expression E_{dflt} is returned.

Here are a few more examples of the the `classify` construct in action:

```
; Program 2
(hofl (a b c d)
  (classify (* c d)
    ((a (- (/ (+ a b) 2) 1)) (* a c))
    (((+ (/ (+ a b) 2) 1) b) (* b d))
    (otherwise (- d c))))
```

```
; Program 3
(hofl (a)
  (classify a
    ((0 9) a)
    (((/ 20 a) 20) (+ a 1))
    (otherwise (/ 100 (- a 5)))))
```

Program 2 emphasizes that any of the subexpressions of `classify` may be an arbitrary expression that requires evaluations. In particular, the upper and lower bound expressions need not be integer literals. For instance, here are some examples of the resulting value returned by Program 2 for some sample inputs.

a	b	c	d	result
10	20	3	4	30
10	20	3	6	120
10	20	3	5	2

Program 3 emphasizes that (1) ranges may overlap (in which case the first matching range is chosen) and (2) expressions in clauses after the matching one are not evaluated. For instance, here are here are some examples of the resulting value returned by Program 3 for some sample inputs.

a	result
0	0
5	5
10	11
20	21
25	5
30	4

Your Task

Dexter has asked you to implement the `classify` construct in HOFL as syntactic sugar. You should begin by writing *on paper* desugaring rules that desugar `classify` into other HOFL constructs. Then you should implement your rule(s) by extending the `desugarRules` function in `Hofl.ml` with clauses for `classify`. Your desugaring should only evaluate E_{disc} once; to guarantee this, you will need to name the value with a "fresh" variable (one that does not appear elsewhere in the program). Use `StringUtils.fresh` to create a fresh variable.

Problem 5 [20]: 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. 2. Given an input integer list, `process` generates an output integer list. You should study the definition of `process` carefully to see 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]

# 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. 3) that can be found in the `ps5` 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. 2. 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.

```
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 2: The OCAML `process` function.

Notes:

- The OCAML `process` function can be found in the file `process.ml`.
- The `IntegerList` class provides the following list manipulation functions on immutable integer lists:

```
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.
```

- The abbreviation `IL.` may be used in place of `IntList.`; For example, `IL.head` rather than `IntList.head`.
- Use `javac` to compile your file and `java` to test it. E.g.:

```
[fturbak@jaguar ps5] cd ~/cs251/private/ps5
[fturbak@jaguar ps5] javac Process.java
[fturbak@jaguar ps5] java Process [3,4,5,6,7]
[1, 2, 3, 13, 15, 17, 19, 21]
[fturbak@jaguar ps5] java Process [5,7,4,5,7]
[2, 3, 2, 3, 35, 47, 29, 35, 47]
[fturbak@jaguar ps5] java Process [1,2,3]
[0, 1, 1, 2, 3]
[fturbak@jaguar ps5] java Process [2,4,6]
[2, 4, 6]
[fturbak@jaguar ps5]
```

```
// 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:


// 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 ps5] java Process [3,4,5,6,7]
  // [1, 2, 3, 13, 15, 17, 19, 21]
  // [lyn@jaguar ps5] 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(IntList.fromString(args[0])));
    } else {
      System.out.println("unrecognized main option");
    }
  }
}
```

Figure 3: The JAVA file `Process.java`.

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

CS251 Problem Set 5
Due 6pm Friday, April 2

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