CS251 Programming Languages **Handout # 22**
Prof. Lyn Turbak **Friday, March 1**
Wellesley College

Problem Set 5
Due: Friday, March 8, 2002

Reading:

- Handouts: HOFL (# 18), Recursive Binding (# 23), FOFL/FOBS (# 24)
- Environment model: *SICP* 3.2 – 3.2.2
- Interpretation of a higher-order functional language (i.e., Scheme, which is pretty similar to HOFL): *SICP* 4 – 4.2.1.

Submission:

- Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- For Problems 3, 4, and 5, your softcopy submission should include a copy of your entire `ps5` directory.
- Your hardcopy submission for Problem 3 should be the files `~/cs251/ps5/free-vars.scm` and `~/cs251/ps5/desugar.scm`.
- Your hardcopy submission for Problem 4 should be the file `~/cs251/ps5/mystery.scm`.
- Your hardcopy submission for Problem 5 should be the file `~/cs251/ps5/pythag.scm`.

Problem 0: Getting Started

To obtain the code you need for this assignment, you should execute the following command in a Unix shell:

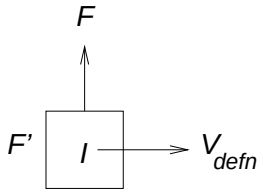
```
cd ~/cs251; cvs update -d
```

This will install four new directories: `fofl`, `fobs`, `hofl`, and `ps5`.

The problems on this problem set involve the FOFL, FOBS, and HOFL languages discussed in class. As part of working on these problems, you may want/need to study and experiment with interpreters for these languages. Appendix A contains a description of how to use the implementations for these three languages.

Problem 1 [30]: Static and Dynamic Scope in HOFL

a. **[12]** 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 the 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' shown below (where V_{defn} is the result of evaluating E_{defn} in F):



b. **[2]** 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. See Appendix A.3 for details.)

c. **[10]** 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. **[2]** What is the final value of the program from part (c) in dynamically scoped HOFL?

e. **[4]** In a programming language with higher-order functions, which supports modularity better: lexical scope or dynamic scope? Explain your answer.

```
(program (a)
  (bind linear (abs (a b)
                  (abs (x)
                    (+ (* a x) b)))
    (bind line1 (linear 1 2)
      (bind line2 (linear 3 4)
        (bind try (abs (b)
                    (prepend (line1 b)
                             (prepend (line2 (+ b 1))
                                      (prepend (line2 (+ b 2))
                                               (empty)))))
          (try (+ a a)))))))
```

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

Problem 2 [15]: `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. [**4**] What is the value of E in statically scoped HOFL?

b. [**4**] Consider the expression E' that is obtained from E by replacing `bindrec` by `bindseq`. What is the value of E' in statically scoped HOFL?

c. [**4**] What is the value of the expression E' from part b in dynamically scoped HOFL?

d. [**3**] 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]: `recur`

Local recursive functions defined via HOFL's `bindrec` or Scheme's `letrec` are often hard to read. For example, consider the following HOFL program using a tail-recursive factorial function:

```
(program (n)
  (bindrec ((factloop (abs (num ans)
                           (if (= num 0)
                               ans
                               (factloop (- num 1) (* num ans))))))
    (factloop n 1)))
```

This program illustrates two problems:

1. The declaration of the state variables of the iteration (`num` and `ans`) is textually far away from the initialization of these variables (to `n` and `1` in the invocation `(factloop n 1)`).
2. Following the usual indentation conventions causes much of the program to be shifted far to the right.

These problems can be addressed by using a more readable construct for simple local recursions:

(recur I_{name} $((I_1\ E_1)\ \ldots\ (I_n\ E_n))$ E_{body}**)**
Create a local function named I_{name} whose formal parameters are I_1 through I_n and whose body is E_{body}. The value returned by `recur` is the result of applying this local function to the values that result from evaluating the initial value expressions E_1 through E_n. E_{body} is in the scope of I_{name} as well as I_1 through I_n, so I_{name} may be recursive. However, E_1 through E_n are not in the scope of I_{name} or I_1 through I_n.

For example, here is a version of the above factorial example using `recur`:

```
(program (n)
  (recur factloop ((num n) (ans 1))
    (if (= num 0)
        ans
        (factloop (- num 1) (* num ans)))))
```

In this problem, you will extend the HOFL implementation in `~/cs251/ps5` to support the `recur` construct. In your extensions, you should use the following abstract syntax operations for the recur construct, which have already been implemented for you:

```
(make-recur name formals inits body)
(recur-name recur-exp)  ; I_name
(recur-formals recur-exp) ; I_1 through I_n
(recur-inits recur-exp) ; E_1 through E_n
(recur-body recur-exp) ; E_body
(recur? s-expression)
```

Before starting this problem, you should evaluate the following Scheme expressions:

```
(cd "~/cs251/ps5")
(load "load-hofl.scm")
```

a. [**5**] Based on the above specification of the `recur` construct, extend the definition of the `free-vars` function in `~/cs251/ps5/free-vars.scm` to handle the `recur` construct. Test your definition by evaluating `(test-recur-free-vars)`.

b. [**15**] Although it is possible to implement `recur` by extending `env-eval` and other functions in the HOFL interpreter, it is easier to implement `recur` via desugaring. Modify the `desugar` function in `~/cs251/ps5/desugar.scm` to desugar `recur` into other HOFL constructs. You should use `bindrec` in your desugaring. There is already a "stub" clause for `recur` in `desugar.scm` that just desugars the parts of a `recur` but leave the `recur` in place. You should replace this stub with a clause that removes the `recur` altogether. Test your extension by evaluating `(test-recur-desugar)`.

Problem 4 [20]: Variable and Function Scoping in FOBS

As discussed in lecture, the first-order block-structured language FOBS has two namespaces: one for functions and one for variables. It is possible to independently choose between static and dynamic scoping in each of the two namespaces. Indeed, the FOBS implementation provides four distinct interpreters for the four possibilities:

- `fobs-run-statfun-statvar` is statically scoped in both the function and variable namespaces.
- `fobs-run-statfun-dynvar` is statically scoped in the function namespace but dynamically scoped in the variable namespace.
- `fobs-run-dynfun-statvar` is dynamically scoped in the function namespace but statically scoped in the variable namespace.
- `fobs-run-dynfun-dynvar` is dynamically scoped in both the function and variable namespaces.

Suppose you are given a *mystery interpreter* that is one of the above four interpreters, but you aren't told which one. By evaluating various FOBS programs with the mystery interpreter, you can determine which of the four interpreters it is. In particular, it is possible to write a *single* FOBS program `solve-mystery` that takes zero arguments and returns a two-element list of strings (*fstring vstring*), where

- *fstring* is `"statfun"` if function names are statically scoped and `"dynfun"` if they are statically scoped.
- *vstring* is `"statvar"` if variable names are statically scoped and `"dynvar"` if they are dynamically scoped.

For example, if running the mystery interpreter on the `solve-mystery` program returns the result `("statfun" "dynvar")`, the mystery interpreter must be `env-run-statfun-dynvar`.

In this problem you are to write the `solve-mystery` program in FOBS. You should do so by fleshing out the skeleton in `~/cs251/ps5/mystery.scm`. The only types of values that your program should manipulate are strings, functions, and lists of strings. That is, your program should not involve any integers, booleans, or symbols. Strive to make your program as simple and understandable as possible.

You can test your program by invoking `(test-mystery)` after you have evaluated the following Scheme expressions:

```
(cd "~/cs251/ps5")
(load "load-mystery.scm")
```

Note: If you cannot solve this problem with just strings, functions, and list of strings, you can get partial credit by solving the problem using other types of values.

Problem 5 [15]: Block Structure

It is possible to translate any program in a first-order block-structured language (such as FOBS) into a first-order language without block structure (such as FOFL). This can be done by "lifting" all function declarations to top-level, possibly renaming some of the functions and adding some extra formal and actual parameters to some functions.

In this problem, you will illustrate this fact by manually translating the FOBS (first-order, block structured) program in Figure 2 into a FOFL (first-order, no block-structure) program that has the same meaning. You should write your FOFL program in the file `~/cs251/ps5/pythag.scm`. Your FOFL program should have exactly the same number of function declarations as the FOBS program, but all of the FOFL declarations should be top-level. You should assume that the FOBS program is statically scoped and the FOFL program is globally scoped.

Notes:

1. Given a non-negative integer n, the program in Figure 2 returns a list of pythagorean triples, where each such triple is a three-element list of the form `(a b c)`, where a, b, c are positive integers such that $a < b < c \leq n$ and $a^2+b^2 = c^2$. The program appears as `pythagorean-triples` in the FOBS test suite in the file `~cs251/fobs/fobs-examples.scm`.

2. You can test your FOFL program by evaluating `(test-pythag)` after you have evaluated (once per session) the following:

   ```
   (cd "~/cs251/ps5")
   (load "load-pythag.scm")
   ```

   Note that running the test program can take a long time.

```
(program (n)
  (funrec
    ((sq (x) (* x x))
     (triple (x y z) (prepend x (prepend y (prepend z (empty)))))
     (from (lo)
       (if (> lo n) ; only generate numbers up to (and including) n
           (empty)
           (prepend lo (from (+ lo 1)))))
     (a-loop (as ans1)
       (if (empty? as)
           ans1
           (bind a (head as)
             (funrec
               ((b-loop (bs ans2)
                   (if (empty? bs)
                       (a-loop (tail as) ans2)
                       (bind b (head bs)
                         (funrec
                           ((c-loop (cs ans3)
                              (if (empty? cs)
                                  (b-loop (tail bs) ans3)
                                  (bind c (head cs)
                                    (if (= (+ (sq a) (sq b)) (sq c))
                                        (b-loop (tail bs)
                                                (prepend (triple a b c)
                                                         ans3))
                                        (c-loop (tail cs) ans3))))))
                           (c-loop (from (+ b 1)) ans2))))))
                (b-loop (from (+ a 1)) ans1))))))
     (a-loop (from 1) (empty))
     )))
```

Figure 2: A FOBS program for computing Pythagorean triples $(a\ b\ c)$, where a, b, c are positive integers such that $a < b < c \leq n$ and $a^2 + b^2 = c^2$.

A FOFL, FOBS, and HOFL Interpreters

This section documents the interpreters for the FOFL, FOBS, and HOFL languages. To obtain the code for all of these intepreters, perform a CVS update by executing the following command in a Unix shell:

```
cd ~/cs251; cvs update -d
```

A.1 FOFL Interpreters

FOFL is a language that extends IBEX with top-level first-order functions. To use the FOFL interpreters, evaluate the following Scheme expressions:

```
(cd "~/cs251/fofl")
(load "load-fofl")
```

There are three distinct FOFL interpreters:

- `fofl-run-global`: uses global scope.
- `fofl-run-dynamic`: uses dynamic scope.
- `fofl-run-flat`: uses flat scope.

A.2 FOBS Interpreters

FOBS is a language that extends FOFL with block structure. To use the FOBS interpreters, evaluate the following Scheme expressions:

```
(cd "~/cs251/fobs")
(load "load-fobs")
```

There are four distinct FOFL interpreters:

- `fobs-run-statfun-statvar` uses statical scope in both the function and variable namespaces.
- `fobs-run-statfun-dynvar` uses static scope in the function namespace but dynamic scope in the variable namespace.
- `fobs-run-dynfun-statvar` uses dynamic scope in the function namespace but static scope in the variable namespace.
- `fobs-run-dynfun-dynvar` uses dynamic scope in both the function and variable namespaces.

A.3 HOFL Interpreters

HOFL is a language that extends FOBS with higher-order functions. To use the HOFL interpreters, evaluate the following Scheme expressions:

```
(cd "~/cs251/hofl")
(load "load-hofl")
```

There are two distinct HOFL interpreters:

- `hofl-run-static` uses static scope.
- `hofl-run-dynamic` uses dynamic scope.

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

CS251 Problem Set 5
Due Friday, March 8

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

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