CS251 Programming Languages **Handout # 38**
Prof. Lyn Turbak **April 14, 2007**
Wellesley College

Problem Set 7

Due: 6pm Friday, April 20

Overview:

The individual problem on this assignment tests your understanding of interpreter extensions and desugaring. The group problems on this assignment cover static and dynamic scope and recursive scoping in Hofl, Fofl, and Fobs.

Reading:

- Handout #35: An Introduction to Hofl: A Higher-order Functional Language
- Handout #36: Scoping in Hofl (make sure you have the April 14 revised version of this handout!)
- Handout #37: Fofl and Fobs: First-Order Functions

Individual Problem Submission:

Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn's office door by 6pm Wed. April 20. The packet should include:

1. an individual problem header sheet;
2. your "wiring diagram" from Problem 1a.
3. your pencil-and-paper answers to Problems 1c and 1d.i.
4. your final versions of `Loopex.ml`, `LoopexEnvInterp.ml`, and `LoopexSubstInterp.ml`.

Each student should also submit a softcopy (consisting of your final `ps7-individual` directory) to the drop directory by executing:

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

Working Together:

If you want to work with a partner on this assignment, you should try to find a different partner than you worked with on a previous assignment. If this proves difficult, please email Lyn describing your situation.

Group Problem 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 Fri. Apr. 15. 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, 2, and 4a.
3. your final version of `exp3a.hfl`, `exp3b.fbs`, and `pgm3c.ffl` from Problem 3.
4. your final version of `exp4b.vlx` from Problem 4.

Each team should also submit a single softcopy (consisting of your final `ps7-group` 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:

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

Individual Problem [50]: Going Loopy

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

In this problem you will extend VALEX with a looping construct and explore desugarings involving this construct.

The `loop` *construct*

Due to your extensive experience with VALEX in CS251, you have been elected head of the VALEX Users Group, a worldwide consortium of VALEX programmers. At your most recent consortium meeting, there was much grumbling from attendees about the lack of expressiveness of VALEX. As one dissatisfied VALEX programmer put it, "Sure, `simprec` helps a little bit. But how can we be expected to write general programs in this language if it doesn't even have a real looping construct?"

You decide it's high time to pay a visit to Ida Ray-Sun, the CTO of Loopster, a company that specializes in loop constructs for programming languages. Ida agrees to help develop a looping construct for VALEX if you will help with the implementation.

Ida quickly designs a looping construct for VALEX and christens the extended language LOOPEX. Here is Ida's looping construct:

$$
\begin{array}{l}
\texttt{(loop ((}I_{sv_1}\ \ E_{init_1}\ \ E_{update_1}\texttt{)} \\
\qquad\qquad\quad \vdots \\
\qquad\qquad \texttt{(}I_{sv_n}\ \ E_{init_n}\ \ E_{update_n}\texttt{))} \\
\qquad\quad E_{test} \\
\quad E_{body}\texttt{)}
\end{array}
$$

The `loop` construct describes an iteration over the **state variables** $I_{sv_1} \ldots I_{sv_n}$, which are assumed to be pairwise distinct. The iteration consists of a sequence of steps between abstract units of time starting with 0, where the **state** of the iteration at time t is characterized by the values of the state variables at time t. The state variables are initialized at time $t = 0$ to the corresponding values of the **initializers** $E_{init_1} \ldots E_{init_n}$. On each step of the iteration, the **updaters** $E_{update_1} \ldots E_{update_n}$ are evaluated relative to the state at time t to determine the state at time $t + 1$. The iteration continues as long as the **test expression** E_{test} gives any non-false value when evaluated relative to the current state. If E_{test} yields false for the initial state, the updaters are never evaluated. The `loop` construct returns the value of E_{body} relative to the first state for which E_{test} yields false.

The scope of state variables declared in `loop` includes the updater expressions, the test expression, and the body expression. The scope does *not* include the initializer expressions.

For example, the following LOOPEX program calculates the factorial of `n`:

```
(loopex (n)
  (loop ((i n (- i 1))
         (prod 1 (* i prod)))
        (> i 0)
    prod))
```

Below is an **iteration table** that shows the values of the state variables of the `loop` iteration at each point in time when the factorial of 5 is computed. Note that the values in a given row are the "state" of the iteration at that time.

t	i	prod
0	5	1
1	4	5
2	3	20
3	2	60
4	1	120
5	0	120

As another example, here are an LOOPEX program that calculates the nth Fibonacci number, and an iteration table that summarizes the iteration for $n = 6$.

```
(loopex (n)
  (loop ((i 0 (+ 1 i))
         (fib_i 0 fib_i+1)
         (fib_i+1 1 (+ fib_i fib_i+1)))
        (< i n)
    fib_i))
```

t	i	fib_i	fib_i+1
0	0	0	1
1	1	1	1
2	2	1	2
3	3	2	3
4	4	3	5
5	5	5	8
6	6	8	13

Note that when evaluating the updater expressions `fib_i+1` and `(+ fib_i fib_i+1)` to determine the state for time $t + 1$, *both* of these expressions are evaluated with respect to the values of the state variables `fib_i` and `fib_i+1` at time t. Because the updaters are effectively evaluated "in parallel", there is no need for "temporary variables" that would often be necessary if such iteration were expressed via a **while** or **for** loop in a language like Java or C.

Your Task

Your task is to solve the following problems related to the `loop` construct. Parts (a), (c), and (d.i) are pencil-and-paper problems; parts (b) and (d.ii) require fleshing out parts of the LOOPEX interpreter in `~/cs251/ps7-individual`. To use any parts of the LOOPEX interpreter, you must first evaluate the following in an OCAML interpreter:

```
#cd "/students/your-username/ps7-individual"
#use "load-loopex.ml"
```

Parts (a) – (d.i) are independent and can be done in any order. The code for part (d.ii) can be written independently of the other parts, but testing it requires the completion of one of parts (b.iii) or (b.iv).

a. [5]: Variable Scoping

Fig. 1 shows a (contrived) LOOPEX expression. In this expression, (1) circle every free variable reference occurrence and (2) draw a line from every bound variable reference occurrence to the binding occurence of that reference.

b. [20]: Implementing `loop` in OCAML

To implement LOOPEX, Ida begins by making a copy of the VALEX interpreter described in Handout #33. The abstract syntax of LOOPEX is the same as that as VALEX except that the `exp` data type has been extended with the following clause to handle the `loop` construct:

```
(loop ((    a     a     (+    a    1   ))
       (    b     b     (-    b    a   )))
      (<=    a    b   )
  (loop ((   a    b    (*    a    2   ))
         (   b    0
             (+    b
                   (loop ((   a    a    (/    a    2))
                          (   b    1    (*    b    a   )))
                         (=    a    0)
                     b   ))))
        (>    a    b   )
    (+    a    b   )))
```

Figure 1: Sample Loopex expression for part a.

```
| Loop of var list (* state variable names *)
          * exp list (* initializer expressions *)
          * exp list (* updater expressions *)
          * exp (* test expression *)
          * exp (* body expression *)
```

Note that the state variables, initializers, and updaters are stored in "unzipped form" rather than being zipped together in some sort of binding structure.

Ida modifies `sexpToExp` and `expToSexp` to correctly handle the parsing and unparsing of `loop` expressions. However, she asks you to modify the rest of the interpreter to handle the `loop` construct.

i [2]: `freeVarsExp` In `Loopex.ml`, add a `loop` clause to the `freeVarsExp` function that calculates the free variables of a `loop` expression. Test `freeVarsExp` via the `testFreeVarsExp` function, which takes a string representation of an expression and returns a list of the free variables in the expression. E.g.:

```
# testFreeVarsExp "(+ b (* a b))";;
- : Loopex.S.elt list = ["a"; "b"]
```

ii [4]: `subst` In `Loopex.ml`, add a `loop` clause to the `subst` function that performs substitutions on a `loop` expression. You can test `subst` via the `testSubst` function, which takes (1) a string representation of an expression and (2) a string representation of an s-expression list of name/expression bindings; it prints the result of performing substitution on the expression using an environment made from the bindings. E.g.:

```
# testSubst "(bind b (/ a b) (+ a b))" "((a (* a b)) (b (- a b)))";;
(bind b.0 (/ (* a b) (- a b)) (+ (* a b) b.0))
- : unit = ()
```

iii [8]: Environment model `eval` In `LoopexEnvInterp.ml`, add a `loop` clause to the `eval` function that correctly specifies the evaluation of the `loop` construct in the environment model. You should do this by fleshing out the three OCaml expressions E_1, E_2, and E_3 in the following skeleton:

```
| Loop(vars, inits, updates, test, body) ->
    eval body (iterate E1 E2 E3)
```

This skeleton uses the following higher-order `iterate` function from the `ListUtils` module (discussed in Section 4.2 of Handout #20):

```
let rec iterate next isDone state =
  if isDone state then
    state
  else
    iterate next isDone (next state)
```

Notes:

- Think carefully about types when doing this problem. What type of value should be returned by the call to `iterate` in the skeleton? What does this imply about the types of values returned by E_1, E_2, and E_3?
- Keep in mind that `loop` treats any non-false test value as true. So any non-boolean value is treated like `#t` in a `loop`.
- Use the environment operations in `~/cs251/utils/Env.ml` to manipulate environments.
- You can test your `loop` clause for `eval` by evaluating `testEnvInterp()`. This runs the factorial and Fibonacci examples described earlier. You are encouraged to add

more test programs containing `loop` to the list of entries `loopexEntries` in the file `LoopexInterpTest.ml`.

iv [6]: Substitution model `eval` In `LoopexSubstInterp.ml`, add a `loop` clause to the `eval` function that correctly specifies the evaluation of the `loop` construct in the substitution model. You should do this by fleshing out the three OCAML expressions E_1, E_2, and E_3 in the following skeleton:

```
| Loop(vars, inits, updates, test, body) ->
       let state = E1 in
         match eval (substAll state vars test) with
           Bool false -> eval E2
         | _ -> eval E3
```

Notes:

- Think carefully about types when doing this problem. From the way that `state` is used in the skeleton, what type must E_1 be? Similarly use the contexts of E_2 and E_3 to determine what types they must have.
- You can test your `loop` clause for `eval` by evaluating `testSubstInterp()`. This will test the substitution model `eval` function on the entries `loopexEntries` in the file `LoopexInterpTest.ml`.

c. [10]: Desugaring `least` into `loop`

Ida notes that many iteration constructs can be desugared into an appropriate `loop` expression. As an example, she invents a `least` construct defined by the following desugaring rule:

(`least` I_{var} E_{pred}) $\rightsquigarrow$ (`loop` ((`(`I_{var} `0` (`+` I_{var} `1`))) (`not` E_{pred}) I_{var})

i [2] Based on the above desugaring, give an English description for the meaning of (`least` I_{var} E_{pred}). Your description should be very concise.

ii [6] What are the values of the following expressions using `least`? Show your work in order to receive partial credit.

- `(least x (> (* x x) 100))`

-
  ```
  (least i (>= (* i (least j (<= (/ 100 (+ j 1))
                                  i)))
               80))
  ```

Recall that `/` denotes **integer division**; it gives the integer quotient of dividing two numbers. For example `(/ 100 50)` yields 2 but `(/ 100 51)` yields 1.

iii [2] Briefly explain the key advantage of implementing `least` as syntactic sugar rather than as a kernel VALEX construct (like `if`, `bind`, or `loop`).

d. [10]: Desugaring `simprec` into `loop`

Inspired by Ida's `least` construct, you decide to extend LOOPEX with the `simprec` construct from Problem Set 5. Rather than implementing `simprec` "from scratch", as you did in Problem Set 5, you instead implement it as syntactic sugar by rewriting all `simprec` expressions into expressions using `loop`.

You should do this problem in two parts:

i [6] Write a high-level desugaring rule or rules (like those in Handout #31), that specifies how to rewrite the expression (`simprec` E_{zero} (I_{num} I_{ans} $E_{combine}$) E_{arg}) into an expression that uses `loop` in addition to any other kernel LOOPEX constructs that you need. The desugared expression should evaluate each of E_{zero} and E_{arg} exactly once. You will need to introduce one or more new names as part of your desugaring. You should specify which of your new names needs to be "fresh" in order to avoid accidental variable capture.

ii [4] Extend the `desugarRules` function in the file `Loopex.ml` so that it correctly desugars `simprec` into `loop` by implementing your high-level desugaring rule(s). Use `StringUtils.fresh` to introduce fresh variable names. You can test your desugaring by adding examples containing `simprec` to the list of entries `loopexEntries` in the file `LoopexInterpTest.ml` and executing `testEnvInterp()` or `testSubstInterp()`.

Group Problems

Group Problem 1 [30]: Static and Dynamic Scope in Hofl

a. [**12**] Suppose that the program in Figure 2 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_{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) (list (line1 b) (line2 (+ b 1)) (line2 (+ b 2))))
          (try (+ a a)))))))
```

Figure 2: A sample Hofl program used to illustrate the difference between static and dynamic scope.

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.

c. [**10**] Draw an environment diagram that shows all of the environments created in *dynamically scoped* HOFL when running the program from Figure 2 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.

Group 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. [**6**] Draw an environment diagram showing the environments created when E is evaluated in *statically scoped* HOFL, and show the final value of evaluating E.

b. [**6**] 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, and show the final value of evaluating E'.

c. [**6**] Draw an environment diagram showing the environments created when E' is evaluated in *dynamically scoped* HOFL, and show the final value of evaluating E'.

d. [**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.

Group Problem 3 [30]: Distinguishing Scopes

In this problem, you will write programs and expressions that distinguish the various scoping mechanisms of HOFL, FOFL, and FOBS.

a. [**6**] In the file `~/ps7-group/exp3a.hfl`, write a simple HOFL expression E_{3a} that evaluates to `(sym static)` in a statically-scoped HOFL interpreter but evaluates to `(sym dynamic)` in dynamically-scoped interpreter. The only types of values that E_{3a} should manipulate are symbols and functions; it should *not* use integers, booleans, characters, strings, or lists. You can test your expression in the OCAML interpreter as follows:

```
#cd /students/your-username/ps7-group
#use "load-hofl.ml"
#testHoflFile "exp3a.hfl"
```

The `testHoflFile` function will evaluate the expression in the given file in both scopes and display the results. A correct solution should have the following behavior:

```
# testHoflFile "exp3a.hfl";;
Value of expression in static scope: (sym static)
Value of expression in dynamic scope: (sym dynamic)
- : unit = ()
```

b. [**10**] In the file `~/ps7-group/exp3b.fbs`, write a simple FOBS expression E_{3b} that evaluates to a list `(list` $V_{varscope}$ $V_{funscope}$`)`, where

- $V_{varscope}$ is `(sym static)` if FOBS uses static variable scoping and `(sym dynamic)` if FOBS uses dynamic variable scoping; and
- $V_{funscope}$ is `(sym static)` if FOBS uses static function scoping and `(sym dynamic)` if FOBS uses dynamic function scoping.

The only types of values that E_{3b} should manipulate are symbols; it should *not* use integers, booleans, characters, strings, or lists. You can test your expression in the OCAML interpreter as follows:

```
#cd /students/your-username/ps7-group
#use "load-fobs.ml"
#testFobsFile "exp3b.fbs"
```

A correct solution should have the following behavior:

```
# testFobsFile "exp3b.fbs";;
Value of expression with static variable scope and static function scope:
  (list (sym static) (sym static))
Value of expression with dynamic variable scope and static function scope:
  (list (sym dynamic) (sym static))
Value of expression with static variable scope and dynamic function scope:
  (list (sym static) (sym dynamic))
Value of expression with dynamic variable scope and dynamic function scope:
  (list (sym dynamic) (sym dynamic))
- : unit = ()
```

If you cannot solve this problem using only symbol variables, you can receive partial credit for writing an expression that uses other types of values and has four different values for the four scope combinations.

c. [**10**] In the file `~/ps7-group/pgm3c.ffl`, write a simple FOFL *one-parameter program* (not an expression) P_{3c} that has a different behavior under each of the four FOFL scoping mechansims (static, dynamic, empty, and merged) when run on the argument list `[5]`. The results can be distinguished not only by values but by error messages. That is, each of the four evalations should have a different value or error message than the others.

You can test your program in the OCAML interpreter as follows:

```
#cd /students/your-username/ps7-group
#use "load-fofl.ml"
#testFoflFile "pgm3c.ffl"
```

d. [**4**] Bud Lojack claims that he can write a FOFL program that, when executed on the correct number of arguments, does not signal an error but returns three different values under the static, dynamic, and merged scoping mechanisms. Explain why Bud's claim is bogus.

Group Problem 4 [20]: VALEX expressions in HOFL

HOFL is designed as an extension to VALEX— every VALEX program is a legal HOFL program if the `valex` keyword is changed to `hofl`.

a. [**10**] Suppose that E_{4a} is any closed VALEX expression. (Recall that an expression is closed if it has no free variables.) Carefully explain why E_{4a} must have the same value in *statically-scoped* HOFL as it does in VALEX. *Hint:* Compare the evaluation rules and desugaring rules for the two languages. Where do they differ for VALEX expressions? In the places where they differ, explain why they still must have the same value.

b. [**10**] In the file `exp4b.vlx`, write a simple closed VALEX expression E_{4b} that evaluates to a different value in *dynamically-scoped* HOFL than it does in VALEX. Explain why the values are different. You can test your expression using the `testHoflFile` function from Problem 3a. (By the result of Problem 4a, the statically-scoped HOFL interpreter will give the same value as the VALEX interpreter.)

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

CS251 Problem Set 7 Individual Problems
Due 6pm Friday, April 20

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 Lyn 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 1a [10]		
Problem 1b [20]		
Problem 1c [10]		
Problem 1d [10]		
Total		

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

CS251 Problem Set 7 Group Problems
Due 6pm Friday, April 20

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