CS251 Programming Languages **Handout # 26**
Prof. Lyn Turbak **Monday, April 5, 2004**
Wellesley College ***Revised Friday, April 9, 2004***

Problem Set 6
Due: 6pm Friday, April 16

Revisions:
This is the final draft of PS6, which includes Problems 3, 4, and 5.

Overview:
The purpose of this assignment is to give you practice with reasoning about state in the context of HOILEC, HOILIC, JAVA, and SCHEME. You will also get some practice programming in Scheme.

Reading:

- Skim *R5RS*.
- Reading sections 3.1–3.3 of Abelson & Sussman's *Structure and Interpretation of Computer Programs* is strongly recommended. This is a good way to learn about Scheme, especially in the context of its imperative features and environment diagrams.

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 2a and your final version of `Counters.java` for Problem 2b.
4. for Problem 3, the final versions of `Hoilic.ml` and `HoilicEnvInterp.ml` for parts (a) and (b) and `cell.def` for part (c). You should also include the evaluation answers for part (b).
5. the final version of `int-table.scm` for Problem 4.
6. the final version of `memoize.scm` for Problem 5.

Each team should also submit a single softcopy (consisting of your final `ps6` directory) to the drop directory `~cs251/drop/ps6/`*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 ps6 ~cs251/drop/ps6/username/
```

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. 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 before the transformation, but returns a value after the transformation.
- the program returns a value before the transformation, but loops infinitely after the transformation.

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. `(+ E1 E2)` $\Longrightarrow$ `(+ E2 E1)`

d. `(+ E1 E2)` $\Longrightarrow$ `(bind x E1 (+ x E2))`

e.
```
(+ E1 E2) ⟹  (bindpar ((x (fun () E1))
                       (y (fun () E2)))
                (+ (x) (y)))
```

f. `(if true E1 E2)` $\Longrightarrow$ `E1`

g. `(if E1 E2 E2)` $\Longrightarrow$ `E2`

h. `(if (if E1 E2 E3) E4 E5)` $\Longrightarrow$ `(if E1 (if E2 E4 E5) (if E3 E4 E5))`

Problem 2 [25]: 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. [15]

Consider the HOILIC functions in Fig. 1. For each of the following expressions, (1) give the value of the expression and (2) draw an environment diagram that justifies why the expression has that value. You should assume that all operands are evaluated in left-to-right order.

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

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

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

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

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

Figure 1: HOILIC counter functions.

b. [10]

Let *x* range over the numbers {1,2,3,}. Then each of the HOILIC functions `make-counterx` can be modeled in JAVA by an instance of class `Counterx` that implements the following interface:

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

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

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

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

```
public static void testCounters (Counter a, Counter b) {
  return IL.prepend(a.invoke(),
                    IL.prepend(b.invoke(),
                               IL.prepend(a.invoke(),
                                          IL.empty())));
}
```

Here `IL.` is a prefix for operations manipulating integer lists.

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

Problem 3 [30]: HOILIC Extensions

In this problem, you will add three extensions to the HOILIC interpreter. This interpreter can be found in the files `Hoilic.ml` and `HoilicEnvInterp.ml` in the `ps6` directory. It can be loaded into OCAML via `#use "load-hoilic.ml"`. In addition to the usual imperative features of HOILIC, this version of HOILIC supports strings and the string operations from PS5 (`strlen`, `strlt`, `str+`, and `to-string`). It also supports a string equality operator `str=`.

a. [10]: Mutable Lists

In the version of HOILIC you are given, lists are immutable; there is no way to change the head or tail component of a list node. In this suproblem, your task is to make the HOILIC lists mutable, like those in SCHEME. In particular, you should add the following two primitive operators to HOILIC:

(set-head! ***list new-head*****)**
Changes the head component of *list* to be *new-head*. Returns the old head component. Signals an error if *list* is empty or if *list* is not a list.

(set-tail! ***list new-tail*****)**
Changes the head component of *list* to be the list *new-tail*. Returns the old tail list. Signals an error if *list* is empty or if *list* or *new-tail* are not lists.

Make whatever changes are necessary to `Hoilic.ml` and `HoilicEnvInterp.ml` to implement mutable lists. Note that you may have to change the way some existing list operations are handled.

```
# HoilicEnvInterp.repl();;

hoilic> (def lst (list 1 2 3))

hoilic> lst
{1,2,3}

hoilic> (head lst)
1

hoilic> (set-head! lst 4)
1

hoilic> (head lst)
4

hoilic> lst
{4,2,3}

hoilic> (tail lst)
{2,3}

hoilic> (set-tail! lst (list 5 6 7))
{2,3}

hoilic> (tail lst)
{5,6,7}

hoilic> lst
{4,5,6,7}
```

Figure 2: Mutable list examples in the HOILIC Read/Eval/Print loop.

b. [10]: `fluid-bind`

Renowned naming expert Dan Emmet Schoop has hired you to implement in HOILIC a new binding construct he calls `fluid-bind`. Here is Dan's specification for his construct:

> **(fluid-bind** I E_{def} E_{body}**)**
> Temporarily assigns to I the value of E_{def} during the evaluation of E_{body} and then resets I to its original value. Returns the value of E_{body}. Signals an error if I is not already bound in the enclosing lexical context.

1. Dan claims that `fluid-bind` gives much of the behavior associated with dynamic scoping within a statically-scoped language like HOILIC. What are the values of the following two HOILIC expressions?

```
(bind a 1
  (bind f (abs x (+ x a))
    (+ (fluid-bind a 20 (f 300))
       (f 4000))))

(bind a 1
  (+ (fluid-bind a 20
       (begin
         (<- a (+ a 300))
         a))
     a))
```

2. `fluid-bind` can be implemented in HOILIC via desugaring. Extend the desugaring rules of HOILIC to implement `fluid-bind`. Show that your desugaring-based implementation of `fluid-bind` yields the values you predicted for the above expressions.

Notes:

- You can test your implementation on the examples via `(#load "fluid.def")`. This defines the name `fluid1` to be the value of the first expression and `fluid2` to be the value of the second expression.
- MIT Scheme has a construct called `fluid-let` that is similar to the `fluid-bind` construct described above.

c. [10]: 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 defining the user-level functions `cell`, `$`, and `$=`. Show this by creating a HOILIC file `cell.def` that defines these three functions using the syntax `(def` I E`)` for each definition. Show that your functions behave as expected – e.g., by trying the examples in Fig. 3.

Notes:

- *Hint:* Consider the message-passing approach to implementing stateful objects covered in class and in Problem 4 on this assignment.
- Unlike the HOILEC `$=` construct, your HOILIC `$=` function will be curried. I.e., `($= a 5)` is equivalent to `(($= a) 5)`.

```
hoilic> (#load "cell.def")
Loading cell.def
Done loading cell.def

hoilic> (def c1 (cell 5))

hoilic> (def c2 (cell 17))

hoilic> ($ c1)
5

hoilic> ($ c2)
17

hoilic> ($= c1 (+ ($ c1) 1))
5

hoilic> ($ c1)
6

hoilic> ($ c2)
17

hoilic> ($= c1 ($= c2 ($ c1)))
6

hoilic> ($ c1)
17

hoilic> ($ c2)
6
```

Figure 3: Examples involving explicit cells in HOILIC.

Problem 4 [20]: Mutable Tables in Scheme

Fig. 4 shows a Scheme implementation of mutable tables in which keys and values may be of any type. In this implementation, the key/value bindings are represented as a so-called **association list** (**alist** for short) of binding pairs of the form `(`*`key . value`*`)`. The order of the bindings depends on the insertion order, not the order of the keys. The `assoc` function performs a linear lookup of a key k in a list of binding pairs and returns the first binding pair whose key is `equal?` to k. If no such binding pair is found, `assoc` returns `#f`.

A table returned by the invocation `(new-alist-table)` is a stateful message-passing procedure that responds to the messages `'insert` and `'lookup`. The `'insert` message returns a procedure that takes a key and value and inserts them into the table. The `'lookup` message returns a procedure that looks up a key in the table. If the key is found, the associated value is returned; otherwise the distinguished `none` value is return, as defined below:

```
(define none '(*none*))
(define none? (lambda (x) (eq? x none)))
```

Here is a transcript showing these tables in action:

```
(define t (new-alist-table))
;Value: t

((t 'insert) 'foo 1)
;Value: inserted

((t 'lookup) 'foo)
;Value: 1

((t 'lookup) 'bar)
;Value 4: (*none*)

((t 'insert) 2 'baz)
;Value: inserted

((t 'insert) "quux" #t)
;Value: inserted

((t 'insert) 2 #\c)
;Value: inserted

((t 'lookup) 'foo)
;Value: 1

((t 'lookup) 2)
;Value: #\c

((t 'lookup) "quux")
;Value: #t

((t 'lookup) "foo")
;Value 4: (*none*)

((t 'lookup) 'bar)
;Value 4: (*none*)
```

```
(define new-alist-table
  (lambda ()
    (let ((alist '()))
      ;; Create message dispatcher named SELF (like THIS in Java)
      (define self
        (lambda (msg)
          (cond
           ((eq? msg 'lookup)
            (lambda (key)
              (let ((probe (assoc key alist)))
                (if probe
                    (cdr probe)
                    none))))
           ((eq? msg 'insert)
            (lambda (key val)
              (let ((probe (assoc key alist)))
                (begin
                  (if probe
                      (set-cdr! probe val)
                      ;; Didn't find pair; add new pair to front.
                      (set! alist (cons (cons key val) alist)))
                  'inserted))))
           (else (error "Unrecognized message" msg))
          ))) ; end of DEFINE SELF ...
      ;; Return message dispatcher
      self)))
```

Figure 4: Scheme implementation of mutable tables as association lists of mutable pairs.

a. [5]: Environment/Box-and-Pointer Diagram

Draw a diagram showing the state of the table `t` at the end of the above transcript. Your diagram should include any closures and environments needed to show the final state of the table, but not any intermediate closures and frames created in the process of creating the table. Use standard box-and-pointer diagrams to represent lists and pairs.

b. [15]: An Efficient Integer Table

Because it uses linear search, the above table representation is not particularly efficient. This can be shown using the testing procedures in Fig. 5. The `test-table` procedure [1] takes a table-making function and a non-negative integer `n` and creates a table whose bindings are `(`*`i . 2i`*`)` for the integers *i* between 0 and `n`. The `timed` procedure[2] takes any *n*-ary function `f` and returns another *n*-ary function that, when called on *n* arguments, returns the 2-element list `(`*`time result`*`)`, where *result* is the result of calling `f` on the *n* arguments and *time* is the time in seconds to compute *result*.

```
(define test-table
  (lambda (table-maker n)
    (let ((table (table-maker)))
      (let insert-loop ((i 0))
        (if (<= i n)
            (begin ((table 'insert) i (* 2 i))
                   (insert-loop (+ i 1)))
            (let lookup-rec ((j 0))
              (if (> j n)
                  '()
                  (cons (cons j ((table 'lookup) j))
                        (lookup-rec (+ j 1)))))))))))

(define timed
  (lambda (f)
    (lambda args ;; args is bound to list of all arguments
      (let* ((start (system-clock)) ; Initial clock
             (result (apply f args))
             (stop (system-clock))) ; Final clock
        ;; Return a 2-elements list of (1) time taken and (2) function result
        (list (- stop start) result)))))
```

Figure 5: Scheme procedures for testing the table implementation.

[1] In the body of `test-table`, the recursion-defining sugar `(let` *I* `((`I_1 E_1`) ...)` E_{body}`)` is used twice. See *R5RS* for details.

[2] In the notation `(lambda args ...)`, the name `args` is bound to the *list* of all the parameters. This is how procedures of arbitrary numbers of arguments are defined in SCHEME; see *R5RS* for details. The `system-clock` timing procedure is specific to MIT Scheme.

For example, consider the following:[3]

```
(map (lambda (n) (list n (car ((timed test-table) new-alist-table n))))
     '(100 200 400 800 1600 3200 6400 12800))
;Value 19: ((100 1.0000000000005116e-2)
            (200 3.0000000000001137e-2)
            (400 .09000000000000341)
            (800 .3100000000000023)
            (1600 1.1400000000000006)
            (3200 4.439999999999998)
            (6400 18.840000000000003)
            (12800 90.47))
```

Doubling the input size cause the time to increase by a factor of about 4, a feature of an algorithm that is quadratic.

In this problem your task is to implement an (amortized) linear-time table for the special case where keys are non-negative integers. Your implementation should represent a collection of key/value bindings as a vector (i.e., a Scheme array) in which indices are the (implicit) keys and the value at index i is the value associated with key i. If there is no binding for a given key i, then slot i of the vector should contain the distinguished `none` value.

As with JAVA arrays, SCHEME vectors have fixed size, so some accomodation must be made to handle the insertion of a value for key i when i is $\geq$ the length *len* of the vector. You should implement the following strategy:

- if $len \leq i < 2 \cdot len$, then make a new vector of size $2 \cdot len$, and copy the contents of the original vector into the first half of the new vector.
- if $i \geq 2 \cdot len$, then make a new vector of size i, and copy the contents of the original vector into the first *len* slots of the new vector.

Your table-creating procedure should be named `new-int-table`. As with `new-alist-table`, the table itself should be a stateful message-passing procedure responding to the messages `'insert` and `'lookup`. Additionally, it should respond to the following two messages:

- `'capacity`: returns the length of the underlying vector.
- `'embiggen`: returns a procedure of one argument, n. If n is greater than the length *len* of the underlying vector, installs a new underlying vector whose length is n and whose first *len* slots are filled from the original vector. Otherwise does nothing.

Notes:

- Write your definition of `new-int-table` in the file `int-table.scm`.
- Use `(load "load-tables.scm")` to load into Scheme the table procedures and associated utility and testing procedures.[4]
- Test your definition using `(test-table new-int-table n)` and

  ```
  (map (lambda (n) (list n (car ((timed test-table) new-int-table n))))
       '(100 200 400 800 1600 3200 6400 12800))
  ```

  For the latter test, you should see a marked improvement over the timings for the alist table.

[3] The notation `ne-2` means $n \times 10^{-2}$.

[4] If this complains that the file is not found, you may have to first execute the following in Scheme to set the current directory to be correct: `(cd "/students/your-username/cs251/ps6")`

Problem 5 [10]: Function Memoization in Scheme

Consider the standard recursive definition of a Scheme Fibonacci function:

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

Such a definition is *extremely* inefficient (exponential time) due to the repeated calculation of subcomputations. Indeed, while SCHEME's big integers can easily express the 100th Fibonacci number, calculating `(fib 100)` would take more time than our sun is expected to "live".

The running time of the above `fib` function can be dramatically improved by a simple strategy known as **memoization**. The idea is to save the result of `(fib n)` in a table the first time we compute it, and then simply look up the answer on every subsequent invocation of `(fib n)`.

This strategy can be captured in `memoize!` procedure that takes as its input a procedure of a single integer argument and returns a memoized version of that procedure. The above Fibonacci function can be memoized as follows:

```
(define fib
  (memoize!
    (lambda (n)
      (if (< n 2)
          n
          (+ (fib (- n 1))
             (fib (- n 2)))))))
```

If we use the fast integer tables from Problem 4, it is easy to compute the 100th, and even the 1000th, Fibonacci numbers in less than a second:

```
((timed fib) 100)
;Value 20: (.01999999999998181 354224848179261915075)

((timed fib) 1000)
;Value 21: (.13999999999998636 434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733247116164299644090653318793829896964992851600370447613779516684922887 5)
```

Your task is to define the `memoize!` procedure in Scheme.

Notes:

- Write your definition in the file `memoize.scm`, which initially contains the unmemoized version of `fib`.
- Use `(load "load-memoize.scm")` to load `"memoize.scm"` and other files on which it depends.
- You may use either alist tables or the fast integer tables from Problem 4.
- The definition of `memoize!` can be very short (mine is 10 lines). Draw environment diagrams to help you *design* the `memoize!` procedure.
- Test your `memoize!` function by redefinining `fib` to be a memoized function and using `(timed fib)` as shown above.

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

CS251 Problem Set 6
Due 6pm Friday, April 16

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 [15]		
Problem 2 [25]		
Problem 3 [30]		
Problem 4 [20]		
Problem 5 [10]		
Total		