

# Imperative Programming

Handout #35

CS251 Lecture 29

April 20, 2000

# Functional vs. Imperative Programming

- Functional Programming (e.g., Scheme, ML, Haskell)
  - Heavy use of first-class functions
  - Immutability/persistence: variables and data structures do not change over time.
  - Expressions denote values
- Imperative Programming (e.g., C, Pascal, Fortran, Ada; core of C++, Java)
  - Mutability/side effects: variables, data structures, procedures, input/output streams can change over time:
  - Often a distinction between expressions (which denote values) and statements (which perform actions). In some languages, expressions do both.
  - Imperative languages often have non-local control flow features (gotos, non-local exits, exceptions). We will study these soon.
- Combining functional and imperative programming
  - Scheme and ML *do* have imperative features, but used sparingly. They are “mostly functional” languages.
  - First-class functions + side effects are at the core of many important programming idioms.

# HOILEC = HOFL + Explicit Mutable Cells

HOILEC is HOFL extended with the following constructs:

`(cell  $E$ )`

Return a cell whose contents is the value of  $E$ .

`(cell-ref  $E$ )` or `( $\wedge$   $E$ )`

Return the current contents of the cell designated by  $E$ .

`(cell-set!  $E_{cell}$   $E_{new}$ )` or `( $:=$   $E_{cell}$   $E_{new}$ )`

Change the contents of the cell designated by  $E_{cell}$  to be the value of  $E_{new}$ .  
Returns `()` (the unit value)

`(cell-equal?  $E_1$   $E_2$ )`

Return true if  $E_1$  and  $E_2$  are the same cell and false otherwise.

`(cell?  $E$ )`

Return true if the value of  $E$  is a cell and false otherwise.

HOILEC cells model ML's refs:

`(ref  $E$ )`, `(!  $E$ )`, `( $E_{cell}$  :=  $E_{new}$ )`, `( $E_1$  =  $E_2$ )`

(ML has no operation corresponding to `cell?`)

# Sequential Execution

In the presence of side effects, order of evaluation is important!  
HOILEC has the following for sequentializing expressions:

$(\text{seq } E_1 \dots E_n)$   
Evaluate  $E_1 \dots E_n$  in order and return the value of  $E_n$ .

*Notes:*

- `seq` can be considered sugar for `bindseq`:

$(\text{bindseq } ((I_1 E_1) \dots (I_n E_n)) I_n)$   
;  $I_i$  must be fresh!

- HOILEC's  $(\text{seq } E_1 \dots E_n)$  corresponds to:
  - Scheme's  $(\text{begin } E_1 \dots E_n)$
  - ML's  $(E_1; \dots; E_n)$

# Mutable Cells: Example

```
(bind a (cell (+ 3 4)))
  (seq (print (^ a))
        (:= a (* 2 (^ a)))
        (print (^ a))
        (:= a (+ 1 (^ a)))
        (print (^ a))
        (bind b (cell (^ a))
              (bind c b
                    (seq (print (cell-equal? a b))
                          (print (cell-equal? b c))
                          (:= c (div (^ c) 5))
                          (print (^ a))
                          (print (^ b))
                          (^ c)))))))
```

# Imperative Factorial in Java

```
public static int fact (int n) {  
    int ans = 1;  
    while (n > 0) {  
        // Order of assignments is critical!  
        ans = n * ans;  
        n = n - 1;  
    }  
    return ans;  
}
```

# Imperative Factorial in HOILEC

```
(bindrec
  ((fact (abs (n)
            (bindpar ((num (cell n))
                      (ans (cell 1)))
                    (bindrec
                      ((loop (abs ()
                            (if (= (^ num) 0)
                                (^ ans)
                                (seq
                                   (:= ans (* (^ num) (^ ans)))
                                   (:= num (- (^ num) 1))
                                   (loop))))))
                    (loop))))))
  . . . body of outer bindrec . . . )
```

# Mutable Stacks in HOILEC

```
(bindpar
  ((stack-create (abs ()) (cell (empty))))
  (stack-empty? (abs (stk) (empty? (^ stk))))
  (top (abs (stk) (head (^ stk))))
  (push! (abs (val stk)
              (:= stk (prepend val (^ stk)))))
  (pop! (abs (stk)
            (if (stack-empty? stk)
                (error "Attempt to pop empty stack")
                (bind elt (top stk)
                          (seq (:= stk (tail (^ stk)))
                                elt)))))))
(bind ((s (stack-create))
      (seq (push! 2 s) (push! 3 s) (push! 5 s)
           (+ (pop! s) (pop! s)))))
```



# Input/Output in HOILEC

`(read-char)`

Consumes and returns the next character from the standard input stream. Returns the distinguished end-of-file value if the standard input stream is empty.

`(read-line)`

Consumes the sequence of characters up to and including the next newline character, and returns a string of those characters (excluding the final newline). Returns the distinguished end-of-file value if the standard input stream is empty.

`(read-int)`

Consumes any whitespace followed by an optional + or - sign and a nonempty maximal sequence of digits, and returns the integer corresponding to those digits. Returns the distinguished end-of-file value if the standard input stream is empty.

`(eof? val)`

Returns true for the distinguished end-of-file value and false for all other values.

`(write-char val)`

Writes the character *val* to the standard output stream.

`(write-int val)`

Writes the character representation of the integer *val* to the standard output stream.

`(write-string val)`

Writes the character representation of the string *val* to the standard output stream.

# I/O Example: Uppercasing all chars in a file

HOILEC program:

```
(program ()
  (bindrec ((loop ()
            (bind c (read-char)
                  (if (eof? c)
                      ()
                      (seq ;; Assume char-uppercase fcn
                          (write-char (char-uppercase c))
                          (loop)))))))
  (loop)))
```

C program:

```
char c;
while ((c = getchar()) != EOF) {
  // Assumes auxiliary char_upper function
  putchar(char_upper(c));
}
```

# “Functions” with State: Counters

How can we use cells to program the following behavior?:

```
(bind make-counter definition-goes-here
  (bind a (make-counter)
    (seq (write-int (a)) ; prints 1
         (write-int (a)) ; prints 2
    (bind b (make-counter)
      (seq (write-int (b)) ; prints 1
           (write-int (a)) ; prints 3
           (write-int (b)) ; prints 2
      )))))
```

Each call to `make-counter` returns what is effectively a new object (in the object-oriented sense). Functions + side effects give much of the power of object-oriented programming -- something we explore later in the semester

# Definition of make-counter

```
(abs () ; This abstraction called to create counter
  (bind count ((cell 0))
    (abs () ; This abstraction called to increment counter
      (seq (:= count (+ (^ count) 1))
        (^ count))))))
```

# Environment diagram for `make-counter` example

Draw the environment diagram here:



# Promise Implementation 1

```
(bindpar
  ((delayed
    (abs (thunk)
      (list thunk (cell false) (cell ())))))
  (force
    (abs (promise)
      (if (^ (second promise))
          (^ (third promise))
          (bind value ((first-promise)) ; dethunk!
            (seq (:= (second promise) true)
                  (:= (third promise) value)
                  value))))))
  ... body of bindpar ... )
```

# Promise Implementation 2

```
(bindpar
  ((delayed
    (abs (thunk)
      (bindpar ((flag (cell false))
                (value (cell ())))))
    (abs ()
      (if (^ flag)
          (^ value)
          (seq (:= value (thunk))
                (:= flag true)
                (^ value)))))))
  (force (abs (promise) (promise)))
  ... body of bindpar ... )
```



## Other Mutable Structures

- Scheme:

- All variables are implicit mutable cells: can be changed with `set!`

```
(let ((a 2) (b 3))
  (begin (set! a (+ a b)) a))
```

```
(define make-counter
```

```
  (lambda ()
```

```
    (let ((count 0))
```

```
      (lambda ()
```

```
        (begin (set! count (+ count 1)) count))))))
```

- Mutable list node slots: can be changed via `set-car!`, `set-cdr!`
- Vectors with mutable slots: can be changed by `vector-set!`

- ML: In addition to ref cells, supports arrays with mutable slots. But all variables and list nodes are *immutable!*
- C and Pascal support mutable records and array variables, which can be stored either on the stack or on the heap. Stack-allocated variables are sources of big headaches.

# Advantages of Side Effects

- Can maintain and update information in a modular way.

## Examples:

- Report the number of times the base case is reached in a recursive SML Fibonacci function. Much easier with cells than without!
- Using `fresh()` to generate new type variables in the type reconstructor, rather than (1) single-threading counter through computation or (2) using finding identifier not in set.
- Tracing/untracing functions in Scheme.
- Organizing interpreter to allow modular addition of new constructs. E.g: in Scheme implementations of interpreters, could have:

```
(define-desugarer! `scand
  (lambda (sexp)
    (list `if (second sexp) (third sexp) falsity)))
```

- Computational objects with local state are nice for modeling the real world. E.g., gas molecules, digital circuits, bank accounts

## Disadvantages of Side Effects

- Lack of referential transparency makes reasoning harder:
  - *Referential transparency*: evaluating the same expression in the same environment always gives the same result.
  - In language without side effects,  $(+ E E)$  can always be safely transformed to  $(* 2 E)$ . But not true in the presence of side effects!
  - Even in a purely functional call-by-value language, non-termination is a kind of side effect. Are the following Scheme expressions always equal?

`(let ((I E3)) (if E1 E2 I)) <=?=> (if E1 E2 E3)`

- Aliasing makes reasoning in the presence of side effects particularly tricky. E.g. HOILEC example:

`(+ (^ a) (seq (:= b (+ 1 (^ b))) (^ a)))  
<=?=> (seq (:= b (+ 1 (^ b))) (* 2 (^ a))))`

- Harder to make persistent structures (e.g., aborting a transaction, rolling back a database to a previous saved point).