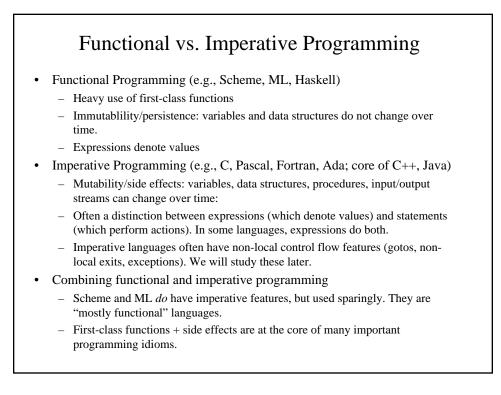
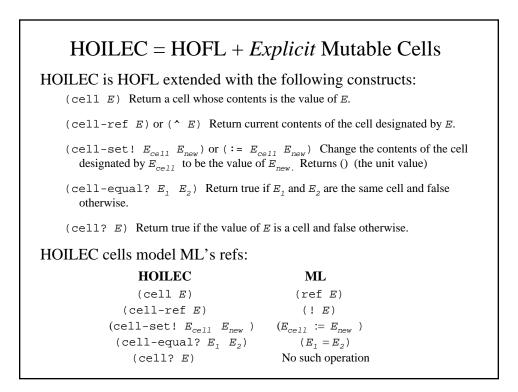
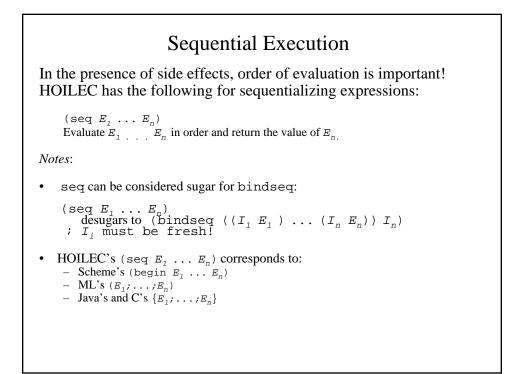
# **Imperative Programming**

Handout #37 CS251 Lecture 28 April 9, 2002







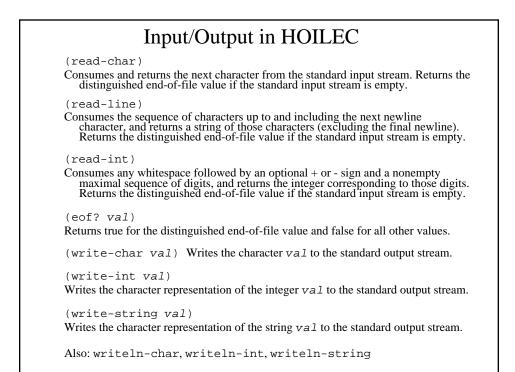
#### Mutable Cells: Example

```
(bind a (cell (+ 3 4))
 (seq (writeln-int (^ a))
  (:= a (* 2 (^ a)))
  (writeln-int (^ a))
  (writeln-int (^ a))
  (bind b (cell (^ a))
    (bind c b
      (seq (writeln-int (cell-equal? a b))
           (writeln-int (cell-equal? b c))
           (:= c (div (^ c) 5))
           (writeln-int (^ a))
           (writeln-int (^ 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;
}
```

#### 

```
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))))
```



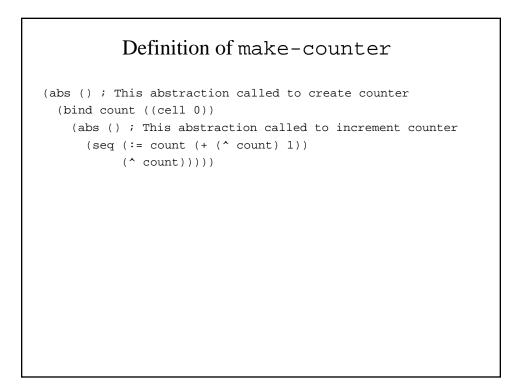
```
I/O Example: Uppercasing all chars in a file
HOILEC program:
  (program ()
    (bindrec ((loop ()
                (bind c (read-char)
                   (if (eof? c)
                       ()
                       (seq ;; Assume char-uppercase fcn
                         (write-char (char-uppercase 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
```

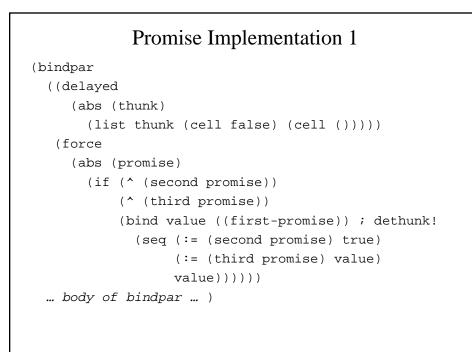
(in the object-oriented sense). Functions + side effects give much of the power of object-oriented programming -something we explore later in the semester



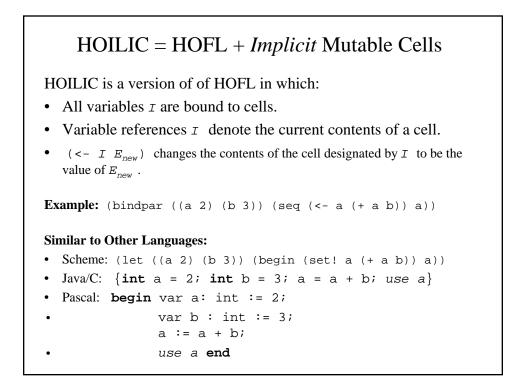
Environment diagram for make-counter example

Draw the environment diagram here:

## 



```
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 ... )
```



```
make-counter Revisited
• HOILIC:
    (bind make-counter
           (abs ()
             (bind ((count 0))
               (abs (lambda ()
                 (seq (<- count (+ count 1))
                      count))))
      body of bind)
• Scheme:
     (define make-counter
      (lambda ()
        (let ((count 0))
          (lambda ()
            (begin (set! count (+ count 1))
                   count))))))
```

#### Other Mutable Structures

- Scheme:
  - 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 and file operations. 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. Stackallocated variables are sources of big headaches (we shall see this later in the semester).
- Almost every language has stateful operations for reading from/writing to files.

Advantages of Side Effects
• Can maintain and update information in a modular way. Examples:
<ul> <li>Report the number of times the base case is reached in a recursive SML</li> <li>Fibonacci function. Much easier with cells than without!</li> </ul>
<ul> <li>Using fresh() to generate new type variabes in the type reconstructor, rather than (1) single-threading counter through computation or (2) using finding identifier not in set.</li> </ul>
<ul> <li>Tracing/untracing functions in Scheme.</li> </ul>
<ul> <li>Organizing interpreter to allow modular addition of new constructs. E.g. in Scheme implementations of interpreters, could have:</li> </ul>
(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

