We have introduced imperative programming in the context of HOILEC, a language with explicit cells. In HOILEC, all variables have immutable bindings to values, but one of the values is a mutable explicit cell. OCAML is a real-world language that uses this model of state.

However, in most real-world languages with imperative and/or object-oriented features (e.g., C, C++, JAVA, ADA, PASCAL, and even SCHEME and COMMON LISP), all variables have mutable bindings to values. In these languages, each variable names an implicit cell whose contents can change over time.

For example, here are imperative versions of the factorial function written in C and in SCHEME:

```c
// C version of imperative factorial
int fact (int n) {
    int ans = 1;
    while (n > 0) {
        ans = n*ans;
        n = n-1;
    }
    return ans;
}

;; Scheme version of imperative factorial
(define (fact n)
    (let ((ans 1))
        (letrec ((loop (lambda ()
                                (if (<= n 0)
                                    ans
                                    (begin (set! ans (* n ans))
                                           (set! n (- n 1))
                                           (loop))))))
            (loop))))
```

In both examples, the variables `n` and `ans` name implicit cells with time-varying integer contents. The contents of an implicit cell are accessed simply by referring to the variable name (which implicitly dereferences the cell — i.e., extracts its contents). The contents of an implicit cell are changed by performing an assignment (written `I_var = E_newval` in C and `(set! I_var E_newval)` in SCHEME).

In this handout, we explore imperative programming with implicit cells in the context of the mini-language HOILIC = HOFL + Implicit Cells.

## 1 HOILIC Overview

HOILIC is like HOILEC except for the following differences:

- Every variable in HOILIC names an implicit cell. In HOILIC, the contents of a cell can be changed by the assignment expression `(<= I_var E_newval)`. Evaluating this expression (1) replaces the contents of the implicit cell named by `I_var` with the value of the expression `E_newval` and (2) returns the previous contents of `I_var`. For example.\(^1\)

\(^1\)The call-by-value HOILIC interpreter uses the prompt `hoilic-cbv` to distinguish it from the interpreters for versions of HOILIC that use other parameter-passing mechanisms. See Handout #45 for a discussion of parameter-passing mechanisms in HOILIC.
\(\text{hoilic-cbv}\)\> (def a 17)

\(a\)

\(\text{hoilic-cbv}\)\> (def b a)

\(b\)

\(\text{hoilic-cbv}\)\> (list a b)

\((\text{list} \ 17 \ 17)\)

\(\text{hoilic-cbv}\)\> (\(-\ a \ 42)\)

\(17\)

\(\text{hoilic-cbv}\)\> (list a b)

\((\text{list} \ 42 \ 17)\)

\(\text{hoilic-cbv}\)\> (\(-\ b \ (\text{-} a \ b))\) \(\text{; Swaps the contents of variables } a \text{ and } b\)

\(17\)

\(\text{hoilic-cbv}\)\> (list a b)

\((\text{list} \ 17 \ 42)\)

- Unlike \textsc{hoilec}, \textsc{hoilic} does not include explicit cell values or primitive operations on these values. The reason is that explicit cells are easy to construct in a language with implicit cells (see Problem 4 of Problem Set 9).

- In \textsc{hoilic}, the \textsc{bindrec} construct can be expressed as syntactic sugar rather than as a kernel construct:

\[
\begin{align*}
\text{bindrec} \ ((I_1 \ E_1) \ \ldots \ \ (I_n \ E_n)) \ E_{body} \\
\leadsto \ (\text{bindpar} \ ((I_1 \ \text{*undefined*}) \ \ldots \ \ (I_n \ \text{*undefined*}))) \\
\quad \ (\text{seq} \ \ (-\ I_1 \ E_1)) \\
\quad \ \vdots \\
\quad \ (\ (-\ I_n \ E_n)) \\
\quad \ E_{body})
\end{align*}
\]

Not only does this guarantee that the identifiers \(I_1 \ldots I_n\) are defined in a single mutual recursive scope, but it also allows the expression \(E_i\) to directly reference the identifiers \(I_1 \ldots I_{i-1}\). (In \textsc{hofl} and \textsc{hoilec}, any such references would denote “black holes”.) For example, the expression

\[
\text{bindrec} \ ((a \ 1) \\
\quad \ (f \ (\text{fun} \ () \ (\text{seq} \ (-\ a \ (* \ a \ 10) \ a)))) \\
\quad \ (b \ (* \ 2 \ a)) \\
\quad \ (c \ (f)) \\
\quad \ (d \ (+ \ (* \ 3 \ a) \ (+ \ (* \ 4 \ (f)) \ (* \ 5 \ a))))
\]

\((\text{list} \ a \ b \ c \ d)\)

evaluates to the value \((\text{list} \ 100 \ 2 \ 10 \ 930)\).

If the \(E_i\) directly references any identifiers \(I_i \ldots I_n\), these will appear to have the value \((\text{sym \ *undefined*})\). For example,

\[
\text{bindrec} \ ((a \ (+ \ 1 \ 2)) \\
\quad \ (b \ (\text{list} \ a \ b \ c)) \\
\quad \ (c \ (* \ 4 \ a))
\]

\((\text{list} \ a \ b \ c)\)

has the value \((\text{list} \ 3 \ (\text{list} \ 3 \ \text{sym \ *undefined*} \ \text{sym \ *undefined*}) \ 12)\) and
(bindrec ((a (+ 1 2))
  (b (- c a))
  (c (* 4 a)))
  (list a b c))

signals an error, because it is not able to subtract 3 from (sym *undefined*).

In all other respects, HOILIC is like HOILEC. In particular, HOILIC includes HOILEC’s syntactic sugar constructs (seq \( E_1 \ldots E_n \)) and (while \( E_{\text{test}} E_{\text{body}} \)).

### 2 HOILIC Examples

This section presents HOILIC verisons of several examples we considered earlier in the context of HOILEC.

#### 2.1 Factorial

```lisp
(defun (fact n)
  (bind ans 1
    (seq (while (> n 0)
          (seq (<- ans (* ans n))
               (<- n (- n 1)))
        ans)))

hoilic-cbv> (fact 4)
24

hoilic-cbv> (fact 5)
120
```

#### 2.2 Fresh Variables

```lisp
(defun fresh
  (bind count 0
    (fun (s)
      (str+ (str+ s ".")
            (toString (<- count (+ count 1)))))

hoilic-cbv> (fresh "a")
"a.0"

hoilic-cbv> (fresh "b")
"b.1"

hoilic-cbv> (fresh "a")
"a.2"
```
2.3 Promises

(def (make-promise thunk)
  (bindpar ((flag #f)
            (memo #f))
    (fun ()
      (if flag
          memo
          (seq (<- flag #t)
               (<- memo (thunk))
               memo))))))

(def (force promise) (promise))

hoilic-cbv> (def p (make-promise (fun () (println (+ 1 2))))))
p
hoilic-cbv> (* (force p) (force p))
3
9

2.4 Message-Passing Stacks

(def (new-stack)
  (bind elts (empty)
    ;; Dispatch function representing stack instance
    (fun (msg)
      (cond
        ((str= msg "empty?") (empty? elts))
        ((str= msg "push")
         (fun (val)
           (seq (<- elts (prep val elts))
                val))) ; Return pushed val
        ((str= msg "top")
         (if (empty? elts)
             (error "Attempt to top an empty stack!" elts)
             (head elts)))
        ((str= msg "pop")
         (if (empty? elts)
             (error "Attempt to pop an empty stack!" elts)
             (bind result (head elts)
                        (seq (<- elts (tail elts))
                             result))))
        (else (error "Unknown stack message:" msg))
      ))))

hoilic-cbv> (def s1 (new-stack))
s1

hoilic-cbv> (def s2 (new-stack))
s2

hoilic-cbv> ((s1 "push") 17)
17

hoilic-cbv> ((s1 "push") 42)
42
hoilic-cbv> ((s1 "push") 23)
 23

hoilic-cbv> (while (not (s1 "empty?"))
    (println ((s2 "push") (s1 "pop"))))
23
42
17
#f

hoilic-cbv> (while (not (s2 "empty?"))
    (println (s2 "pop")))
17
42
23
#f

hoilic-cbv> (s2 "top")
EvalError: Hoilic Error -- Attempt to top an empty stack!:#e

2.5 Message-Passing Points

(def my-point
  (bind num-points 0 ; class variable
    (fun (cmsg) ; class message
      (cond
        ((str= cmsg "count") (fun () num-points)); acts like a class method
        ((str= cmsg "new") ; acts like a constructor method
          (fun (ix iy)
            (bindpar ((x 0) (y 0)); instance variables
              (seq (<- num-points (+ num-points 1)); count points
                (<- x ix) (<- y iy)
              )
            )
            (bindrec ; create and return instance dispatcher function.
              ((this ; gives the name "this" to instance = instance method dispatcher
                (fun (imsg) ; instance message
                  (cond
                    ;; the following are instance methods
                    ((str= imsg "get-x") (fun () x))
                    ((str= imsg "get-y") (fun () y))
                    ((str= imsg "set-x") (fun (new-x) (<- x new-x)))
                    ((str= imsg "set-y") (fun (new-y) (<- y new-y)))
                    ((str= imsg "translate")
                      (fun (dx dy) (seq ((this "set-x") (+ x dx))
                                      ((this "set-y") (+ y dy)))))))))
                    (else "error: unknown instance message" imsg))))
              this)))))) ; return instance as the result of "new"
  (else "error: unknown class message" cmsg)
)))))
3 Implementing HOILIC

Like HOILEC, HOILIC has seven types of expressions:

\[
\text{and exp} = \\
\text{Lit of valu (* value literals *)} \\
\text{Var of var (* variable reference *)} \\
\text{PrimApp of primop * exp list (* primitive application with rator, rands *)} \\
\text{If of exp * exp * exp (* conditional with test, then, else *)} \\
\text{Abs of var * exp (* function abstraction *)} \\
\text{App of exp * exp (* function application *)} \\
\text{Assign of var * exp (* variable assignment (new in HOILIC) *)}
\]

Assignment expressions, introduced by the Assign constructor, are new in HOILIC. These are the abstract form of the concrete \((I_{var} E_{newval})\) syntax. The reason why HOILIC still has the same number of expression types as HOILEC is that the kernel bindrec expression in HOILEC is syntactic sugar in HOILIC.

HOILIC values are similar to HOILEC values:

\[
\text{and valu} = \\
\text{Int of int} \\
\text{Bool of bool} \\
\text{Char of char} \\
\text{String of string} \\
\text{Symbol of string} \\
\text{List of valu list} \\
\text{Fun of var * exp * valu ref Env.env (* In HOILIC, vars bound to implicit cells *)}
\]

There are two differences:

1. In HOILIC, environments associate names with implicit cells, so the environments in HOILIC closures have type valu ref Env.env. In contrast, HOILIC closure environments have type

...
valu Env.env.

2. In Hoilic, the valu type does not include the Cell values of Hoilec.

The environment-model interpreter for Hoilic differs in a few ways from the Hoilec interpreter:

- In the run function for executing a program, the evaluation of the program body must wrap the integer arguments of the program in implicit cells, which are represented as OCAML cells:
  
  \[
  \text{eval body (Env.make fmls (map (fun i -> ref (Int i)) ints))}
  \]

- The type of the eval function indicates that the environments map names to implicit cells:
  
  \[
  \text{val eval : Hoilic.exp \to valu \text{ ref Env.env} \to valu}
  \]

- Because variable names are bound to implicit cells, a variable reference must implicitly dereference the value from the implicit cell:
  
  \[
  \begin{align*}
  \text{and eval exp env =} \\
  \text{match exp with} \\
  \quad : \\
  \quad | \text{Var name} -> \\
  \quad \quad (\text{match Env.lookup name env with} \\
  \quad \quad \quad \quad \text{Some implicitCell} -> (! implicitCell) (* Implicit variable dereference *) \\
  \quad \quad \quad | \text{None} -> \text{raise (EvalError("Unbound variable: " \^ name))}) \\
  \end{align*}
  \]

- The eval function must handle the new assignment expression:
  
  \[
  \begin{align*}
  \text{| Assign(name,rhs) ->} \\
  \quad (* \text{ Store value of rhs in name and return old value. } *) \\
  \quad (\text{match Env.lookup name env with} \\
  \quad \quad \text{Some implicitCell} -> \\
  \quad \quad \quad \text{let oldValu = (! implicitCell)} \\
  \quad \quad \quad \text{and newValu = eval rhs env in} \\
  \quad \quad \quad \text{let _ = implicitCell := newValu in} \\
  \quad \quad \quad \text{oldValu} \\
  \quad \quad | \text{None} -> \text{raise (EvalError("Unbound variable: " \^ name))})
  \end{align*}
  \]

Note that assignment in Hoilic returns the previous value stored in the implicit cell. In languages with assignment expressions (which return a value) as opposed to assignment statements (which don’t), there is a design choice as to what value should be returned:

- C and JAVA specify that the new value assigned to the variable should be returned. This facilitates sequences of assignment statements involving the new variable. For example, in C and JAVA, \text{a=b=c=E}; assigns the result of \text{E} to the three variables \text{a}, \text{b}, and \text{c}. This also facilitates the C and JAVA idiom of performing assignments in loop tests, e.g.:
  
  \begin{verbatim}
  \text{while ((c = readChar()) != '\n') \{ ... body using c ... \}}
  \end{verbatim}

- OCAML specifies that the trivial unit value, (), is returned by explicit cell assignment.

- The SCHEME language definition does not specify what value is returned by a variable assignment — it could be any value whatsoever!

Hoilic returns the value stored in the cell before the assignment. This facilitates several programing idioms, such as swapping variable values and enumerating values from a state-based process.
(<- a (<- b a)) ; Swap the contents of a and b

(bind c 0 (fun () (<- c (+ c 1)))) ; Function enumerating nonnegative integers

- Because `bindrec` is sugar in HOILIC, the `eval` function has no clause for a `Bindrec` case.
- The `apply` function is modified to wrap each argument value in an implicit cell when a new environment binding is created:
  ```ml
  (* val apply: valu -> valu -> valu *)
  and apply fcn arg =
  match fcn with
    Fun(fml,body,env) -> eval body (Env.bind fml (ref arg) env)
  | _ -> raise (EvalError ("Non-function rator in application: "
                             ^ (valuToString fcn)))
  ```
- Finally, the read-eval-print loop for HOILIC must associate the name of each top-level `def` with an implicit cell containing the defined value.