CS251 Programming Languages **Handout # 35**
Prof. Lyn Turbak **April 3, 2007**
Wellesley College

An Introduction to HOFL, a Higher-order Functional Language

HOFL (Higher Order Functional Language) is a language that extends VALEX with first-class functions and a recursive binding construct. We study HOFL to understand the design and implementation issues involving first-class functions, particularly the notions of static vs. dynamic scoping and recursive binding. Later, we will consider languages that support more restrictive notions of functions than HOFL.

Although HOFL is a "toy" language, it packs a good amount of expressive punch, and could be used for many "real" programming purposes. Indeed, it is very similar to the Scheme programming language, and it is powerful enough to write interpreters for all the mini-languages we have studied, including HOFL itself!

In this handout, we introduce the key features of the HOFL language in the context of examples. We will study the implementation of HOFL, particularly with regard to scoping issues, in a separate handout.

1 An Overview of HOFL

The HOFL language extends VALEX with the following features:

1. Anonymous first-class curried functions and a means of applying these functions;
2. A `bindrec` form for defininining mutually recursive values (typically functions);
3. A `load` form for loading definitions from files.

The full grammar of HOFL is presented in Fig. 1. The syntactic sugar of HOFL is defined in Fig. 2.

2 Abstractions and Function Applications

In HOFL, anonymous first-class functions are created via

`(abs` I_{formal} E_{body}`)`

This denotes a function of a single argument I_{formal} that computes E_{body}. It corresponds to the OCAML notation `fun` I_{formal} `->` E_{body}.

Function application is expressed by the parenthesized notation `(`E_{rator} E_{rand}`)`, where E_{rator} is an arbitrary expression that denotes a function, and E_{rand} denotes the operand value to which the function is applied. For example:

```
hofl> ((abs x (* x x)) (+ 1 2))
9

hofl> ((abs f (f 5)) (abs x (* x x)))
25

hofl> ((abs f (f 5)) ((abs x (abs y (+ x y))) 12))
17
```

The second and third examples highlight the first-class nature of HOFL function values.

The notation `(fun (`$I_1 \ldots I_n$`)` E_{body}`)` is syntactic sugar for curried abstractions and the notation `(`E_{rator} $E_1 \ldots E_n$`)` for $n \geq 2$ is syntactic sugar for curried applications. For example,

$P \in$ *Program*	
$P \rightarrow$ `(hofl (`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`	Kernel Program
$P \rightarrow$ `(hofl (`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body} $D_1 \ldots D_k$`)`	Sugared Program
$D \in$ *Definition*	
$D \rightarrow$ `(def` I_{name} E_{body}`)`	Basic Definition
$D \rightarrow$ `(def (`$I_{fcnName}$ $I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`	Sugared Function Definition
$D \rightarrow$ `(load filename)`	File Load
$E \in$ *Expression*	
Kernel Expressions:	
$E \rightarrow L$	Literal
$E \rightarrow I$	Variable Reference
$E \rightarrow$ `(if` E_{test} E_{then} E_{else}`)`	Conditional
$E \rightarrow$ `(`O_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Primitive Application
$E \rightarrow$ `(abs` I_{formal} E_{body}`)`	Function Abstraction
$E \rightarrow$ `(`E_{rator} E_{rand}`)`	Function Application
$E \rightarrow$ `(bindrec ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Local Recursion
Sugar Expressions:	
$E \rightarrow$ `(fun (`$I_1 \ldots I_n$`)` E_{body}`)`	Curried Function
$E \rightarrow$ `(`E_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`, where $n \geq 2$	Curried Application
$E \rightarrow$ `(`E_{rator}`)`	Nullary Application
$E \rightarrow$ `(bind` I_{name} E_{defn} E_{body}`)`	Local Binding
$E \rightarrow$ `(bindseq ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Sequential Binding
$E \rightarrow$ `(bindpar ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Parallel Binding
$E \rightarrow$ `(&&` E_1 E_2`)`	Short-Circuit And
$E \rightarrow$ `(||` E_1 E_2`)`	Short-Circuit Or
$E \rightarrow$ `(cond (`E_{test_1} E_{body_1}`) ... (`E_{test_n} E_{body_n}`) (else` $E_{default}$`))`	Multi-branch Conditional
$E \rightarrow$ `(list` E_1 ... E_n`)`	List
$E \rightarrow$ `(quote` S`)`	Quoted Expression
$S \in$ *S-expression*	
$S \rightarrow N$	S-expression Integer
$S \rightarrow C$	S-expression Character
$S \rightarrow R$	S-expression String
$S \rightarrow I$	S-expression Symbol
$S \rightarrow$ `(`S_1 ... S_n`)`	S-expression List
$L \in$ *Literal*	
$L \rightarrow N$	Numeric Literal
$L \rightarrow B$	Boolean Literal
$L \rightarrow C$	Character Literal
$L \rightarrow R$	String Literal
$L \rightarrow$ `(sym` I`)`	Symbolic Literal
$L \rightarrow$ `#e`	Empty List Literal

$O \in$ *Primitive Operator*: e.g., `+`, `<=`, `and`, `not`, `prep`
$F \in$ *Function Name*: e.g., `f`, `sqr`, `+-and-*`
$I \in$ *Identifier*: e.g., `a`, `captain`, `fib_n-2`
$N \in$ *Integer*: e.g., `3`, `-17`
$B \in$ *Boolean*: `#t` and `#f`
$C \in$ *Character*: `'a'`, `'B'`, `'7'`, `'\n'`, `''''\\'`
$R \in$ *String*: `"foo"`, `"Hello there!"`, `"The string \"bar\""`

Figure 1: Grammar for the HOFL langauge.

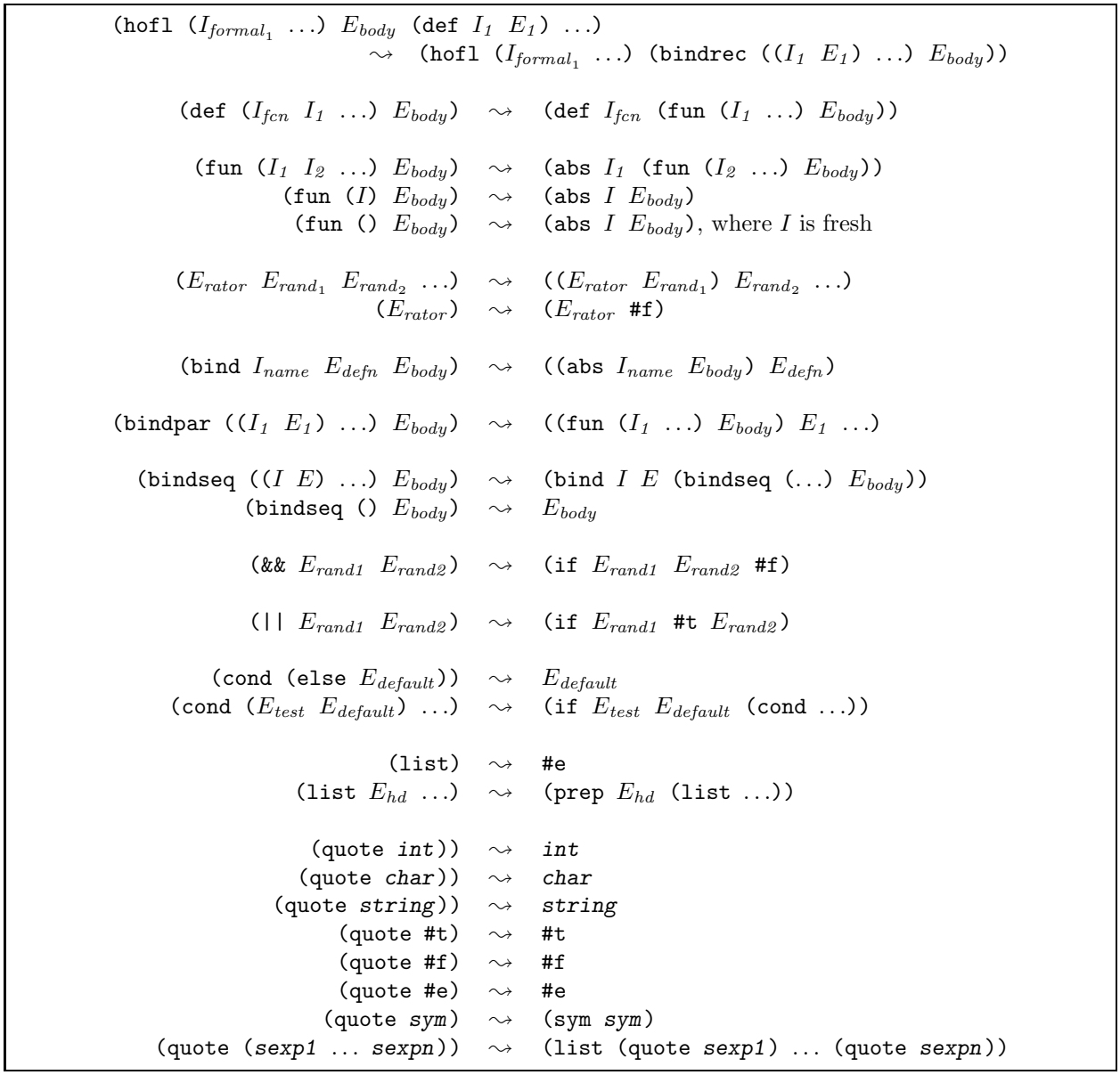

Figure 2: Desugaring rules for HOFL.

```
((fun (a b x) (+ (* a x) b)) 2 3 4)
```

is syntactic sugar for

```
((((abs a (abs b (abs x (+ (* a x) b)))) 2) 3) 4)
```

Nullary functions and applications are also defined as sugar. For example, `((fun () `E`))` is syntactic sugar for `((abs `I` `E`) #f)`, where I is a fresh variable. Note that `#f` is used as an arbitrary argument value in this desugaring.

In HOFL, `bind` is not a kernel form but is syntactic sugar for the application of a manifest abstraction. Unlike in VALEX, in HOFL the `bindpar` desugaring can be expressed via a high-level rule (also involving the application of a manifest abstraction). For example,

```
(bind c (+ a b) (* c c))
```

is sugar for

```
((abs c (* c c)) (+ a b))
```

and

```
(bindpar ((a (+ a b)) (b (- a b))) (* a b))
```

is sugar for

```
((fun (a b) (* a b)) (+ a b) (- a b)),
```

which is itself sugar for

```
(((abs a (abs b (* a b))) (+ a b)) (- a b)).
```

3 Local Recursive Bindings

Singly and mutually recursive functions can be defined anywhere (not just at top level) via the `bindrec` construct:

`(bindrec ((`I_{name_1} E_{defn_1}`)` ... `(`I_{name_n} E_{defn_n}`))` E_{body}`)`

The `bindrec` construct is similar to `bindpar` and `bindseq` except that the scope of $I_{name_1} \ldots I_{name_n}$ includes *all* definition expressions $E_{defn_1} \ldots E_{defn_n}$ as well as E_{body}. For example, here is a definition of a recursive factorial function:

```
(hofl (x)
  (bindrec ((fact (abs (n)
                    (if (= n 0)
                        1
                        (* n (fact (- n 1)))))))
    (fact x)))
```

Here is an example involving the mutual recursion of two functions, `even?` and `odd?`:

```
(hofl (n)
  (bindrec ((even? (abs (x)
                     (if (= x 0)
                         #t
                         (odd? (- x 1)))))
            (odd? (abs (y)
                    (if (= y 0)
                        #f
                        (even? (- y 1))))))
    (list (even? n) (odd? n))))
```

To emphasize that `bindrec` need not be at top-level, here is program that abstracts over the `even?/odd?` example from above:

```
(hofl (n)
  (bind tester (abs (bool)
                 (bindrec ((test1 (fun (x)
                                    (if (= x 0)
                                        bool
                                        (test2 (- x 1)))))
                           (test2 (abs (y)
                                    (if (= y 0)
                                        (not bool)
                                        (test1 (- y 1))))))))
    (list ((tester #t) n) ((tester #f) n))))
```

To simplify the definition of values, especially functions, at top-level HOFL supports syntactic sugar for top-level program definitions. For example, the `fact` and `even?/odd?` examples can also

be expressed as follows:

```
(hofl (x) (fact x)
  (def (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))

(hofl (n) (list (even? n) (odd? n))
  (def (even? x)
    (if (= x 0)
        #t
        (odd? (- x 1))))
  (def (odd? y)
    (if (= y 0)
        #f
        (even? (- y 1)))))
```

The HOFL read-eval-print loop (REPL) accepts definitions as well as expressions. All definitions are considered to be mutually recursive. Any expression submitted to the REPL is evaluated in the context of a `bindrec` derived from all the definitions submitted so far. If there has been more than one definition with a given name, the most recent definition with that name is used. For example, consider the following sequence of REPL interactions:

```
hofl> (def three (+ 1 2))
three
```

For a definition, the response of the interpreter is the defined name. This can be viewed as an acknowledgement that the definition has been submitted. The body expression of the definition is *not* evaluated yet, so if it contains an error or infinite loop, there will be no indication of this until an expression is submitted to the REPL later.

```
hofl> (+ three 4)
7
```

When the above expression is submitted, the result is the value of the following expression:

```
(bindrec ((three (+ 1 2)))
  (+ three 4))
```

Now let's define a function and then invoke it:

```
hofl> (def (sq x) (* x x))
sq
hofl> (sq three)
9
```

The value *9* is the result of evaluating the following expression:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x))))
  (sq three))
```

Let's define one more function and invoke it:

```
hofl> (def (sos a b) (+ (sq a) (sq b)))
sos
hofl> (sos three 4)
25
```

The value *25* is the resulf of evaluating the following expression:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x)))
          (sos (abs a (abs b (+ (sq a) (sq b))))))
  ((sos three) 4))
```

Note that it wasn't necessary to define `sq` before `sos`. They could have been defined in the opposite order, as long as no attempt was made to evaluate an expression containing `sos` before `sq` was defined.

4 Loading Definitions From Files

Typing sequences of definitions into the HOFL REPL can be tedious for any program that contains more than a few definitions. To facilitate the construction and testing of complex programs, HOFL supports the loading of definitions from files. Suppose that *filename* is a string literal (i.e., a character sequence delimited by double quotes) naming a file that contains a sequence of HOFL definitions. In the REPL, entering the directive `(load` *`filename`*`)` has the same effect as manually entering all the definitions in the file named *filename*. For example, suppose that the file named `"option.hfl"` contains the definitions in Fig. 3 and `"list-utils.hfl"` contains the definitions in Fig. 4. Then we can have the following REPL interactions:

```
hofl> (load "option.hfl")
none
none?
some?
```

When a `load` directive is entered, the names of all definitions in the loaded file are displayed. These definitions are not evaluated yet, only collected for later.

```
hofl> (none? none)
#t

hofl> (some? none)
#f

hofl> (load "list-utils.hfl")
length
rev
nth
first
second
third
fourth
map
filter
gen
range
foldr
foldr2

hofl> (range 3 7)
(list 3 4 5 6 7)

hofl> (map (fun (x) (* x x)) (range 3 7))
(list 9 16 25 36 49)

hofl> (foldr (fun (a b) (+ a b)) 0 (range 3 7))
25
```

```
(def none (sym *none*)) ; Use the symbol *NONE* to represent the none value.

(def (none? v)
  (if (sym? v)
      (sym= v none)
      #f))

(def (some? v) (not (none? v)))
```

Figure 3: The contents of the file "option.hfl" which contains an OCAML-like option data structure express in HOFL.

```
hofl> (filter some? (map (fun (x) (if (= 0 (% x 2)) x none)) (range 3 7)))p
(list 4 6)
```

In HOFL, a load directive may appear whereever a definition may appear. It denotes the sequence of definitions contains in the named file. For example, loaded files may themselves contain load directives for loading other files. The environment implementation in Fig. 5 loads the files "option.hfl" and "list-utils.hfl". load directives may also appear directly in a HOFL program. For example:

```
(hofl (a b)
  (filter some? (map (fun (x) (if (= 0 (% x 2)) x none)) (range a b)))
  (load "option.hfl")
  (load "list-utils.hfl"))
```

When applied to the argument list [3;7], this program yields a HOFL list containing the integers 4 and 6

5 A BINDEX Interpreter Written in HOFL

To illustrate that HOFL is suitable for defining complex programs, in Figs. 6 and 7 we present a complete interpreter for the BINDEX language written in HOFL. BINDEX expressions and programs are represented as tree structures encoded via HOFL lists, symbols, and integers. For example, the BINDEX averaging program can be expressed as the following HOFL list:

```
(list (sym bindex)
      (list (sym a) (sym b)) ; formals
      (list (sym /) ; body
            (list (sym +) (sym a) (sym b))
            2))
```

HOFL's LISP-inspired quote sugar allows such BINDEX programs to be written more perspicuously. For example, the above can be expressed as follows using quote:

```
(quote (bindex (a b) (/ (+ a b) 2)))
```

Here are some examples of the HOFL-based BINDEX interpreter in action:

```
hofl> (load "bindex.hfl")
... lots of names omitted ...
hofl> (run (quote (bindex (x) (* x x))) (list 5))
25
hofl> (run (quote (bindex (a b) (/ (+ a b) 2))) (list 5 15))
10
```

```
(def (length xs)
  (if (empty? xs)
      0
      (+ 1 (length (tail xs)))))

(def (rev xs)
  (bindrec ((loop (fun (old new)
                    (if (empty? old)
                        new
                        (loop (tail old) (prep (head old) new))))))
    (loop xs #e)))

(def (nth n xs) ; Returns the nth element of a list (1-indexed)
  (if (= n 1)
      (head xs)
      (nth (- n 1) (tail xs))))

(def first (nth 1))
(def second (nth 2))
(def third (nth 3))
(def fourth (nth 4))

(def (map f xs)
  (if (empty? xs)
      #e
      (prep (f (head xs))
            (map f (tail xs)))))

(def (filter pred xs)
  (cond ((empty? xs) #e)
        ((pred (head xs))
         (prep (head xs) (filter pred (tail xs))))
        (else (filter pred (tail xs)))))

(def (gen next done? seed)
  (if (done? seed)
      #e
      (prep seed (gen next done? (next seed)))))

(def (range lo hi) (gen (fun (x) (+ x 1)) (fun (y) (> y hi)) lo))

(def (foldr binop null xs)
  (if (empty? xs)
      null
      (binop (head xs)
             (foldr binop null (tail xs)))))

(def (foldr2 ternop null xs ys)
  (if (|| (empty? xs) (empty? ys))
      null
      (ternop (head xs)
              (head ys)
              (foldr2 ternop null (tail xs) (tail ys)))))
```

Figure 4: The contents of the file `"list-utils.hfl"` which contains some classic list functions expressed in HOFL.

```
(load "option.hfl")
(load "list-utils.hfl")

(def env-empty (fun (name) none))

(def (env-bind name val env)
  (fun (n)
    (if (sym= n name)
        val
        (env n))))

(def (env-bind-all names vals env)
  (foldr2 env-bind env names vals))

(def (env-lookup name env) (env name))
```

Figure 5: The contents of the file `"env.hfl"`, which contains an functional implementation of environments expressed in HOFL.

It is not difficult to extend the BINDEX interpreter to be a full-fledged HOFL interpreter. If we did this, we would have a HOFL interpreter defined in HOFL. An interpreter for a language written in that language is called a **meta-circular** interpreter. There is nothing strange or ill-defined about a meta-circular interpreter. Keep in mind that in order to execute a meta-circular interpreter for a language L, we must have an existing working implementation of L. For example, to execute a meta-circular HOFL interpreter, we could use a HOFL interpreter defined in OCAML.

```
(load "env.hfl") ; This also loads list-utils.hfl and option.hfl

(def (run pgm args)
  (bind fmls (pgm-formals pgm)
    (if (not (= (length fmls) (length args)))
        (error "Mismatch between expected and actual arguments"
               (list fmls args))
        (eval (pgm-body pgm)
              (env-bind-all fmls args env-empty)))))

(def (eval exp env)
  (cond ((lit? exp) (lit-value exp))
        ((var? exp)
         (bind val (env-lookup (var-name exp) env)
           (if (none? val)
               (error "Unbound variable" exp)
               val)))
        ((binapp? exp)
         (binapply (binapp-op exp)
                   (eval (binapp-rand1 exp) env)
                   (eval (binapp-rand2 exp) env)))
        ((bind? exp)
         (eval (bind-body exp)
               (env-bind (bind-name exp)
                         (eval (bind-defn exp) env)
                         env)))
        (else (error "Invalid expression" exp))))

(def (binapply op x  y)
  (cond ((sym= op (sym +)) (+ x y))
        ((sym= op (sym -)) (- x y))
        ((sym= op (sym *)) (* x y))
        ((sym= op (sym /)) (if (= y 0) (error "Div by 0" x) (/ x y)))
        ((sym= op (sym %)) (if (= y 0) (error "Rem by 0" x) (% x y)))
        (else (error "Invalid binop" op))))
```

Figure 6: Environment model interpreter for BINDEX expressed in HOFL, part 1.

```
;;;-------------------------------------------------------------------------
;;; Abstract syntax

;;; Programs

(def (pgm? exp)
 (&& (list? exp)
     (&& (= (length exp) 3)
         (sym= (first exp) (sym bindex)))))

(def (pgm-formals exp) (second exp))
(def (pgm-body exp) (third exp))

;;; Expressions

;; Literals
(def (lit? exp) (int? exp))
(def (lit-value exp) exp)

;; Variables
(def (var? exp) (sym? exp))
(def (var-name exp) exp)

;; Binary Applications
(def (binapp? exp)
  (&& (list? exp)
      (&& (= (length exp) 3)
          (binop? (first exp)))))

(def (binapp-op exp) (first exp))
(def (binapp-rand1 exp) (second exp))
(def (binapp-rand2 exp) (third exp))

;; Local Bindings
(def (bind? exp)
  (&& (list? exp)
      (&& (= (length exp) 4)
          (&& (sym=? (first exp) (sym bind))
              (sym? (second exp))))))

(def (bind-name exp) (second name))
(def (bind-defn exp) (third name))
(def (bind-body exp) (fourth name))

;; Binary Operators
(def (binop? exp)
  (|| (sym= exp (sym +))
      (|| (sym= exp (sym -))
          (|| (sym= exp (sym *))
              (|| (sym= exp (sym /))
                  (sym= exp (sym %)))))))
```

Figure 7: Environment model interpreter for BINDEX expressed in HOFL, part 2.