CS251 Programming Languages **Handout # 1**
Prof. Lyn Turbak **May 09, 2018**
Wellesley College

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.

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 RACKET 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 also study the implementation of HOFL, particularly with regard to scoping issues.

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 figure 1. The syntactic sugar of HOFL is defined in figure 2.

2 Abstractions and Function Applications

In HOFL, anonymous first-class functions are created via

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

This denotes a function of a single argument Id_{formal} that computes E_{body}. It corresponds to the SML notation `fn` Id_{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 (`Id_1 ... Id_n`)` E_{body}`)` is syntactic sugar for curried abstractions and the notation `(`E_{rator} E_1 ... E_n`)` for $n \geq 2$ is syntactic sugar for curried applications. For example,

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

$P \in$ *Program*

$P \rightarrow$ `(hofl (`Id_{formal_1} `...` Id_{formal_n}`)` E_{body}`)`	Kernel Program
$P \rightarrow$ `(hofl (`Id_{formal_1} `...` Id_{formal_n}`)` E_{body} $D_1 \ldots D_k$`)`	Sugared Program

$D \in$ *Definition*

$D \rightarrow$ `(def` Id_{name} E_{body}`)`	Basic Definition
$D \rightarrow$ `(def (`$Id_{fcnName}$ Id_{formal_1} `...` Id_{formal_n}`)` E_{body}`)`	Sugared Function Definition
$D \rightarrow$ `(load filename)`	File Load

$E \in$ *Expression*

Kernel Expressions:

$E \rightarrow L$	Literal
$E \rightarrow Id$	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` Id_{formal} E_{body}`)`	Function Abstraction
$E \rightarrow$ `(`E_{rator} E_{rand}`)`	Function Application
$E \rightarrow$ `(bindrec ((`Id_{name_1} E_{defn_1}`) ...` $(Id_{name_n}$ E_{defn_n}`))` E_{body}`)`	Local Recursion

Sugar Expressions:

$E \rightarrow$ `(fun (`Id_1 `...` Id_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` Id_{name} E_{defn} E_{body}`)`	Local Binding
$E \rightarrow$ `(bindseq ((`Id_{name_1} E_{defn_1}`) ...` $(Id_{name_n}$ E_{defn_n}`))` E_{body}`)`	Sequential Binding
$E \rightarrow$ `(bindpar ((`Id_{name_1} E_{defn_1}`) ...` $(Id_{name_n}$ 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
$S \rightarrow Id$	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` Id`)`	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-*`
$Id \in$ *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.

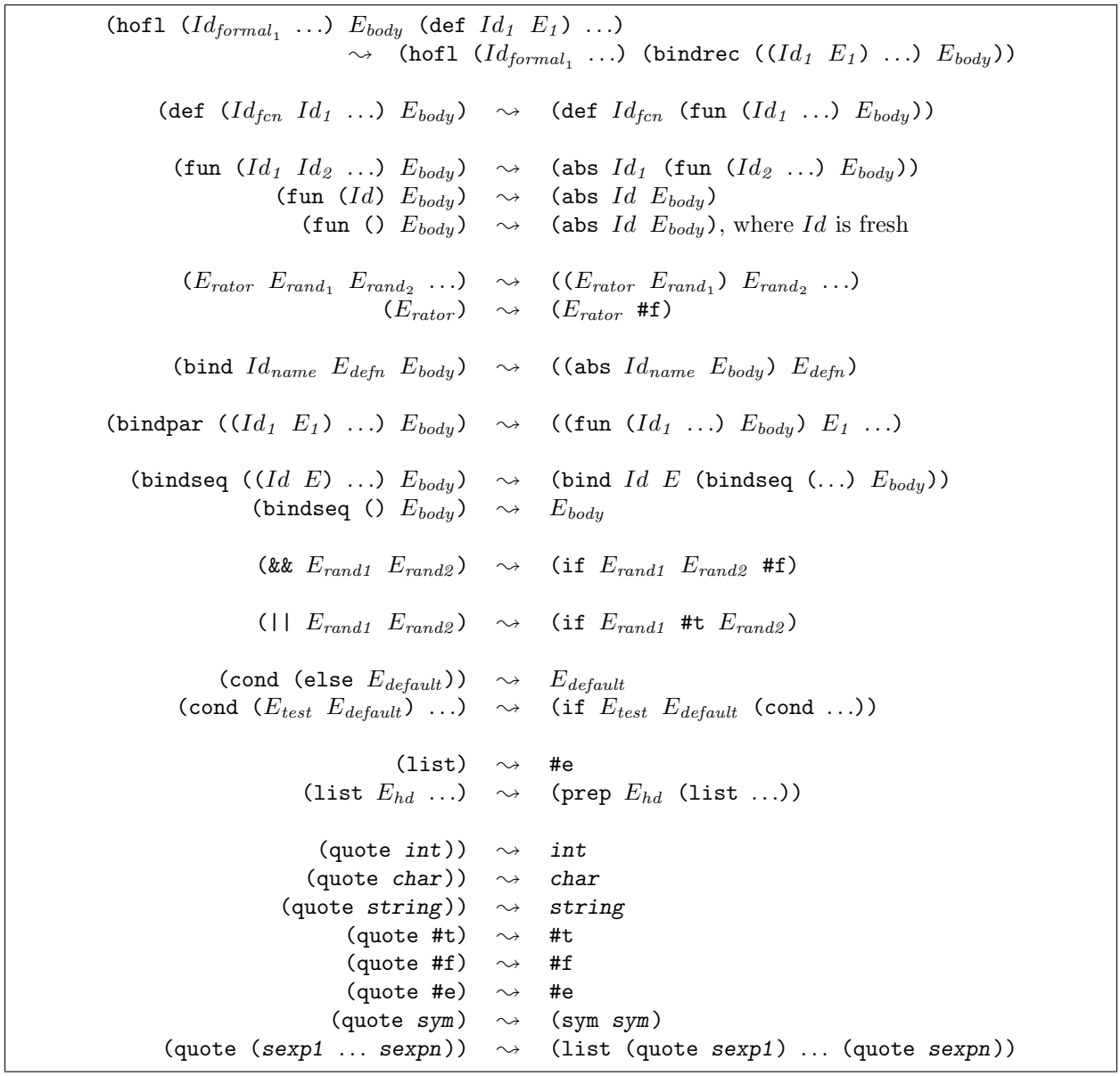


Figure 2: Desugaring rules for HOFL.

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 Id E) #f)`, where *Id* 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. For example,

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

is sugar for

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

Unlike in VALEX, in HOFL the `bindpar` desugaring need not be handled by first collecting the definitions into a list and then extracting them. Instead, like RACKET's `let` construct, HOFL's `bindpar`

be expressed via a high-level desugaring rule involving the application of a manifest abstraction. For example:

```
(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` ((*Id*$_{name_1}$ *E*$_{defn_1}$) ... (*Id*$_{name_n}$ *E*$_{defn_n}$)) *E*$_{body}$)

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

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

The scope of the names bound by `bindrec` (`even?` and `odd?` in the latter case) includes not only the body of the `bindrec` expression, but also the definition expressions bound to the names. This distinguishes `bindrec` from `bindpar`, where the scope of the names would include the body, but not the definitions. The difference between the scoping of `bindrec` and `bindpar` can be seen in the two contour diagrams in figure 16. In the `bindrec` expresion, the reference occurrence of `odd?` within the `even?` abstraction has the binding name `odd?` as its binding occurrence; the case is similar for `even?`. However, when `bindrec` is changed to `bindpar` in this program, the names `odd?` and `even?` within the definitions become unbound variables. If `bindrec` were changed to `bindseq`, the occurrence of `even?` in the second binding would reference the declaration of `even?` in the first, but the occurrence of `odd?` in the first binding would still be unbound.

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 (fun (bool)
```

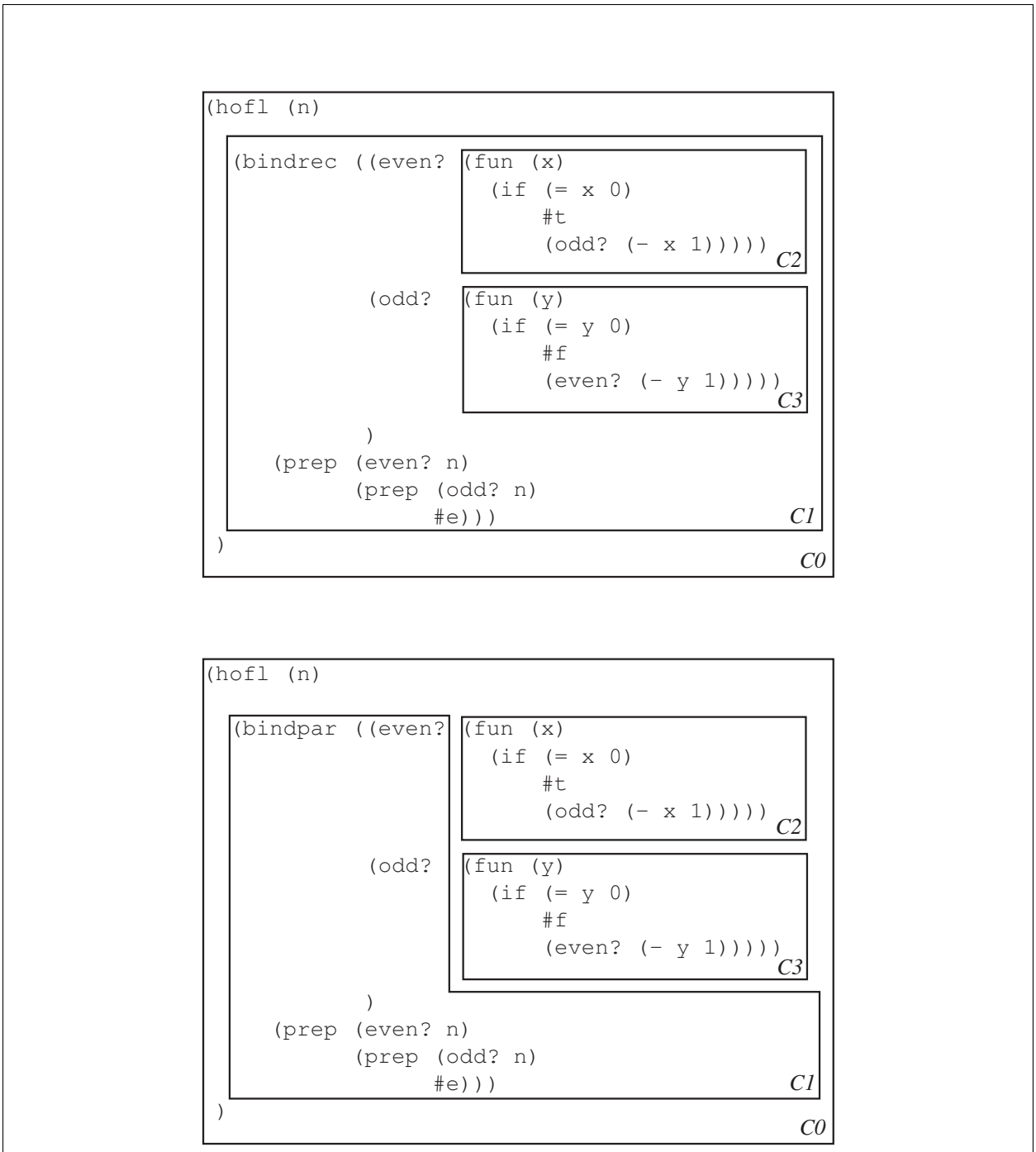


Figure 3: Lexical contours for versions of the `even?`/`odd?` program using `bindrec` and `bindpar`.

```
(hofl (a b)
  (bindrec
      (
       (map (fun (f xs)
              (if (empty? xs)
                  #e
                  (prep (f (head xs))
                        (map f (tail xs))))))

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

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

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

       (range (fun (lo hi) ; includes lo but excludes hi
                (gen (fun (x) (+ x 1)) (fun (y) (>= y hi)) lo)))

       (sq (fun (i) (* i i)))

       (even? (fun (n) (= (% n 2) 0)))
       )
    (foldr (fun (x y) (+ x y))
      0
      (map sq (filter even? (range a (+ b 1)))))))
```

Figure 4: A sample HOFL program with higher-order list-processing functions.

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

Because HOFL supports recursion and higher-order functions, we can use it to define all the higher-order functions and list-processing functions that we've investigated in SML. For example, what does the HOFL program in figure 4 do?

4 Definitions in HOFL Programs

To simplify the definition of values, especially functions, in HOFL programs and in the interactive HOFL interpreter, 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, which results from collecting the original expression and two definitions into a `bindrec` and desugaring:

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

Let's define one more function and invoke it:

```
hofl> (def (sum-squares-between lo hi)
        (if (> lo hi)
            0
            (+ (sq lo) (sum-squares-between (+ lo 1) hi))))
sum-squares-between

hofl> (sum-squares-between three 5)
50
```

The value *50* is the result of evaluating the following expression, which results from collecting the original expression and three definitions into a `bindrec` and desugaring:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x)))
          (sum-squares-between
            (abs lo
              (abs hi
                (if (> lo hi)
                    0
                    (+ (sq lo) ((sum-squares-between (+ lo 1)) hi)))))))
  ((sum-squares-between three) 5))
```

It isn't necessary to define `sq` before `sum-square-between`. The definitions can appear in any order, as long as no attempt is made to find the value of a defined name before it is defined.

5 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 figure 5 and `"list-utils.hfl"` contains the definitions in figure 6. 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
first
second
third
fourth
map
```

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

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

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

Figure 5: The contents of the file `"option.hfl"` which contains an SML-like option data structure expressed in HOFL.

```
filter
gen
range
foldr
foldr2

hofl> (range 3 8)
(list 3 4 5 6 7)

hofl> (map (fun (x) (* x x)) (range 3 8))
(list 9 16 25 36 49)

hofl> (foldr (fun (a b) (+ a b)) 0 (range 3 8))
25

hofl> (filter some? (map (fun (x) (if (= 0 (% x 2)) x none)) (range 3 8)))
(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 figure 7 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 1))))
  (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`.

HOFL is even powerful enough for writing interpreters. For example, we can write an INTEX, BINDEX, and VALEX interpreters in HOFL. We can even (gasp!) we write a HOFL interpreter in HOFL. Such an interpreter is known as a **metacircular interpreter**.

6 A BINDEX Interpreter Written in HOFL

To illustrate that HOFL is suitable for defining complex programs, in figures 8 and 9 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:

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

(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) ; includes lo but excludes 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 6: 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 7: The contents of the file `"env.hfl"`, which contains a functional implementation of environments expressed in HOFL.

```
(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
```

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 SML.

```
(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 8: 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 exp))
(def (bind-defn exp) (third exp))
(def (bind-body exp) (fourth exp))

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

Figure 9: Environment model interpreter for BINDEX expressed in HOFL, part 2.

7 Scoping Mechanisms

In order to understand a program, it is essential to understand the meaning of every name. This requires being able to reliably answer the following question: given a reference occurrence of a name, which binding occurrence does it refer to?

In many cases, the connection between reference occurrences and binding occurrences is clear from the meaning of the binding constructs. For instance, in the HOFL abstraction

```
(fun (a b) (bind c (+ a b) (div c 2)))
```

it is clear that the `a` and `b` within `(+ a b)` refer to the parameters of the abstraction and that the `c` in `(div c 2)` refers to the variable introduced by the `bind` expression.

However, the situation becomes murkier in the presence of functions whose bodies have free variables. Consider the following HOFL program:

```
(hofl (a)
  (bind add-a (fun (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a)))))
```

The `add-a` function is defined by the abstraction `(fun (x) (+ x a))`, which has a free variable `a`. The question is: which binding occurrence of `a` in the program does this free variable refer to? Does it refer to the program parameter `a` or the `a` introduced by the `bind` expression?

A **scoping mechanism** determines the binding occurrence in a program associated with a free variable reference within a function body. In languages with block structure[1] and/or higher-order functions, it is common to encounter functions with free variables. Understanding the scoping mechanisms of such languages is a prerequisite to understand the meanings of programs written in these languages.

We will study two scoping mechanisms in the context of the HOFL language: **static scoping** (section 8) and **dynamic scoping** (section 9) . To simplify the discussion, we will initially consider HOFL programs that do not use the `bindrec` construct. Then we will study recursive bindings in more detail in section **??**).

8 Static Scoping

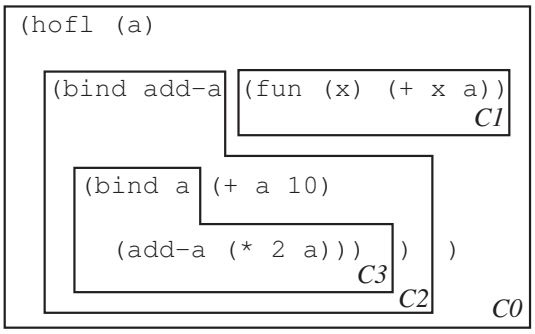
8.1 Contour Model

In **static scoping**, the meaning of every variable reference is determined by the lexical contour boxes introduced in the slides on BINDEX. To determine the binding occurrence of any reference occurrence of a name, find the innermost contour enclosing the reference occurrence that binds the name. This is the desired binding occurrence.

For example, below is the contour diagram associated with the `add-a` example. The reference to `a` in the expression `(+ x a)` lies within contour boxes C_1 and C_0. C_1 does not bind `a`, but C_0 does, so the `a` in `(+ x a)` refers to the `a` bound by `(hofl (a) ... )`. Similarly, it can be determined that:

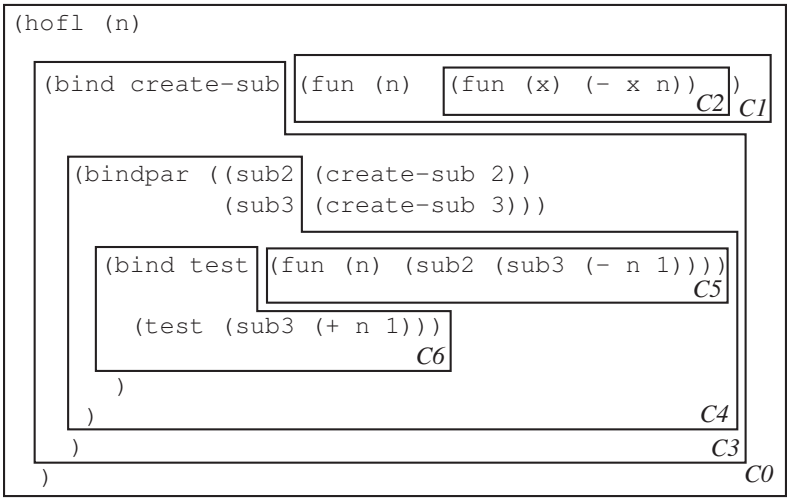
- the `a` in `(+ a 10)` refers to the `a` bound by `(hofl (a) ... )`;
- the `a` in `(* 2 a)` refers the `a` bound by `(bind a ... )`;
- the `x` in `(+ x a)` refers to the `x` bound by `(abs (x) ... )`.
- the `add-a` in `(add-a (* 2 a))` refers to the `add-a` bound by `(bind add-a ... )`.

[1] A language has block structure if functions can be declared locally within other functions. As we shall see later in this course, a language can have block structure without having first-class functions.



Static scoping is also known as **lexical scoping** because the meaning of any reference occurrence is apparent from the lexical structure of the program.

As another example of a contour diagram, consider the contours associated with the following program containing a `create-sub` function:



By the rules of static scope:

- the `n` in `(- x n)` refers to the `n` bound by the `(fun (n) ... )` of `create-sub`;
- the `n` in `(- n 1)` refers to the `n` bound by the `(fun (n) ... )` of `test`;
- the `n` in `(+ n 1)` refers to the `n` bound by `(hofl (n) ... )`.

8.2 Substitution Model

The same substitution model used to explain the evaluation of OCAML, BINDEX, and VALEX can be used to explain the evaluation of statically scoped HOFL expressions that do not contain `bindrec`. (Handling `bindrec` is tricky in the substitution model, and will be considered later.)

For example, suppose we run the program containing the `add-a` function on the input 3. Then the substitution process yields:

```
(hofl (a)
  (bind add-a (fun (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a))))) run on [3]
; Here and below, assume a ''smart'' substitution that
; performs renaming only when variable capture is possible.
⇒ (bind add-a (fun (x) (+ x 3))
    (bind a (+ 3 10)
```

```
      (add-a (* 2 a))))
⇒* (bind a 13 ((fun (x) (+ x 3)) (* 2 a)))
⇒ ((fun (x) (+ x 3)) (* 2 13))
⇒ ((fun (x) (+ x 3)) 26)
⇒ (+ 26 3)
⇒ 29
```

As a second example, suppose we run the program containing the `create-sub` function on the input 12. Then the substitution process yields:

```
(hofl (n)
  (bind create-sub (fun (n) (fun (x) (- x n)))
    (bindpar ((sub2 (create-sub 2))
              (sub3 (create-sub 3)))
      (bind test (fun (n) (sub2 (sub3 (- n 1))))
        (test (sub3 (+ n 1)))))))) run on [12]
⇒ (bind create-sub (fun (n) (fun (x) (- x n)))
    (bindpar ((sub2 (create-sub 2))
              (sub3 (create-sub 3)))
      (bind test (fun (n) (sub2 (sub3 (- n 1))))
        (test (sub3 (+ 12 1))))))
⇒* (bindpar ((sub2 ((fun (n) (fun (x) (- x n))) 2))
             (sub3 ((fun (n) (fun (x) (- x n))) 3)))
    (bind test (fun (n) (sub2 (sub3 (- n 1))))
      (test (sub3 13))))
⇒ (bindpar ((sub2 (fun (x) (- x 2)))
             (sub3 (fun (x) (- x 3))))
    (bind test (fun (n) (sub2 (sub3 (- n 1))))
      (test (sub3 13))))
⇒ (bind test (fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1))))
    (test ((fun (x) (- x 3)) 13))))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1)))) ((fun (x)
   (- x 3)) 13))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1)))) (- 13 3))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1)))) 10)
⇒ ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- 10 1)))
⇒ ((fun (x) (- x 2)) ((fun (x) (- x 3)) 9))
⇒ ((fun (x) (- x 2)) (- 9 3))
⇒ ((fun (x) (- x 2)) 6)
⇒ (- 6 2)
⇒ 4
```

We could formalize the HOFL substitution model by defining a substitution model evaluator in SML, but for the time being we omit that to focus on an environment model interpreter.

8.3 Environment Model

We would like to be able to explain static scoping within the environment model of evaluation. In order to explain the structure of environments in this model, it is helpful to draw an environment as a linked chain of **environment frames**, where each frame has a set of name/value bindings and each frame has a single **parent frame**. There is a distinguished **empty frame** that terminates the chain, much as an empty list terminates a linked list. See figure 10 for an example. In practice, we will often omit the empty frame, and instead indicate the last frame in a chain as a frame with no parent frame.

Intuitively, name lookup in an environment represented as a chain of frames is performed as follows:

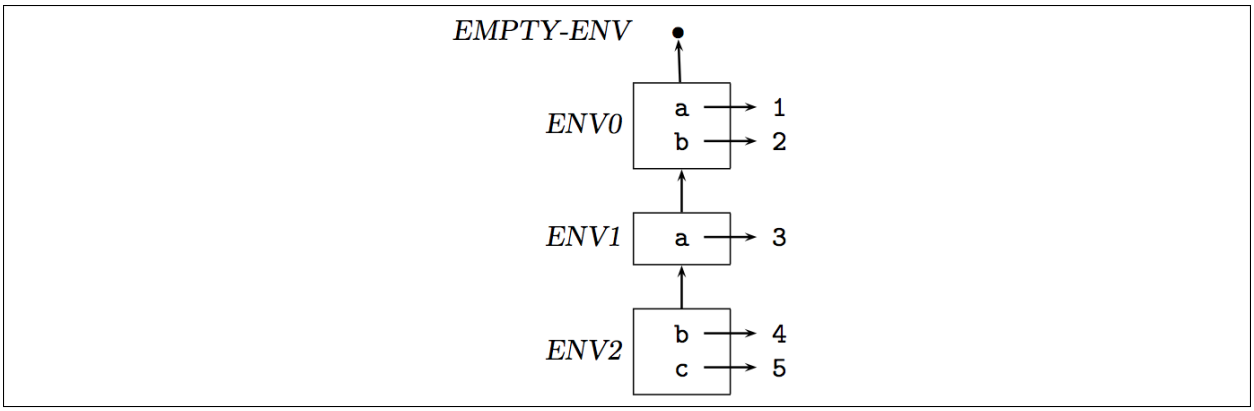


Figure 10: An example chain of environment frames.

- if the name appears in a binding in the first frame of the chain, a `Some` option of its associated value is returned;
- if the name does not appear in a binding in the first frame of the chain, the lookup process continues starting at the parent frame of the first frame;
- if the empty frame is reached, a `None` option is returned, indicated that the name was not found.

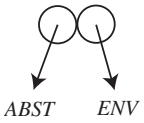
Most evaluation rules of the environment model are independent of the scoping mechanism. Such rules are shown in figure 11.

It turns out that any scoping mechanism is determined by how the following two questions are answered within the environment model:

1. What is the result of evaluating an abstraction in an environment?
2. When creating a frame to model the application of a function to arguments, what should the parent frame of the new frame be?

In the case of static scoping, answering these questions yields the following rules:

1. Evaluating an abstraction *ABS* in an environment *ENV* returns a closure that pairs together *ABS* and *ENV*. The closure "remembers" that *ENV* is the environment in which the free variables of *ABS* should be looked up; it is like an "umbilical cord" that connects the abstraction to its place of birth. We shall draw closures as a pair of circles, where the left circle points to the abstraction and the right circle points to the environment:



2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment remembered by the closure. That is, the new frame should extend the environment where the closure was born, not (necessarily) the environment in which the closure was called. This creates the right environment for evaluating the body of the abstraction as implied by static scoping: the first frame in the environment contains the bindings for the formal parameters, and the rest of the frames contain the bindings for the free variables.

Program Running Rule

- To run a HOFL program `(hofl (`Id_1 ...Id_n`)` E_{body}`)` on integers $i_1, \ldots, i_k$, return the result of evaluating E_{body} in an environment that binds the formal parameter names Id_1 ...Id_n respectively to the integer values $i_1, \ldots, i_k$.

Expression Evaluation Rules

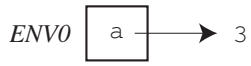
- To evaluate a literal expression in any environment, return the value of the literal.
- To evaluate a variable reference expression Id expression in environment ENV, return the value of looking up Id in ENV. If Id is not bound in ENV, signal an unbound variable error.
- To evaluate the conditional expression `(if` E_1 E_2 E_3`)` in environment ENV, first evaluate E_1 in ENV to the value V_1. If V_1 is true, return the result of evaluating E_2 in ENV; if V_1 is false, return the result of evaluating E_3 in ENV; otherwise signal an error that V_1 is not a boolean.
- To evaluate the primitive application `(`O_{rator} E_1 ... E_n`)` in environment ENV, first evaluate the operand expressions E_1 through E_n in ENV to the values V_1 through V_n. Then return the result of applying the primitive operator O_{primop} to the operand values V_1 through V_n. Signal an error if the number or types of the operand values are not appropriate for O_{primop}.
- To evaluate the function application `(`E_{fcn} E_{rand}`)` in environment ENV, first evaluate the expressions E_{fcn} and E_{rand} in ENV to the values V_{fcn} and V_{rand}, respectively. If V_{fcn} is a function value, return the result of applying V_{fcn} to the operand value V_{rand}. (The details of what it means to apply a function is at the heart of scoping and, as we shall see, differs among scoping mechanisms.) If V_{fcn} is not a a function value, signal an error.

Although `bind`, `bindrec`, and `bindseq` can all be "desugared away", it is convenient to imagine that there are rules for evaluating these constructs directly:

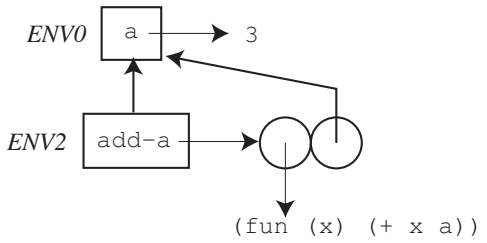
- Evaluating `(bind` Id_{name} E_{defn} E_{body}`)` in environment ENV is the result of evaluating E_{body} in the environment that results from extending ENV with a frame containing a single binding between Id_{name} and the value V_{defn} that results from evaluating E_{defn} in ENV.
- A `bindpar` is evaluated similarly to `bind`, except that the new frame contains one binding for each of the name/defn pairs in the `bindpar`. As in `bind`, all defns of `bindpar` are evaluated in the original frame, not the extension.
- A `bindseq` expression should be evaluated as if it were a sequence of nested `bind`s.

Figure 11: Environment model evaluation rules that are independent of the scoping mechanism.

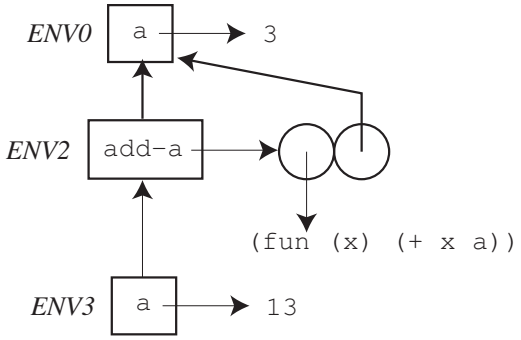
We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



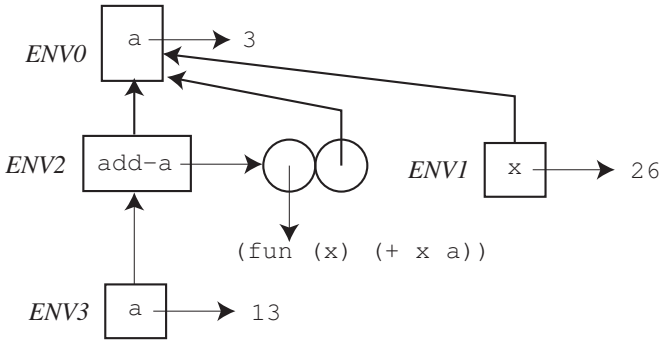
To evaluate the `(bind add-a ... )` expression, we first evaluate `(fun (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



Next the expression `(bind a ...)` is evaluated in ENV_2. First the definition `(+ a 10)` is evaluated in ENV_1, yielding 13. Then a new frame ENV_3 is created that binds a to 13:

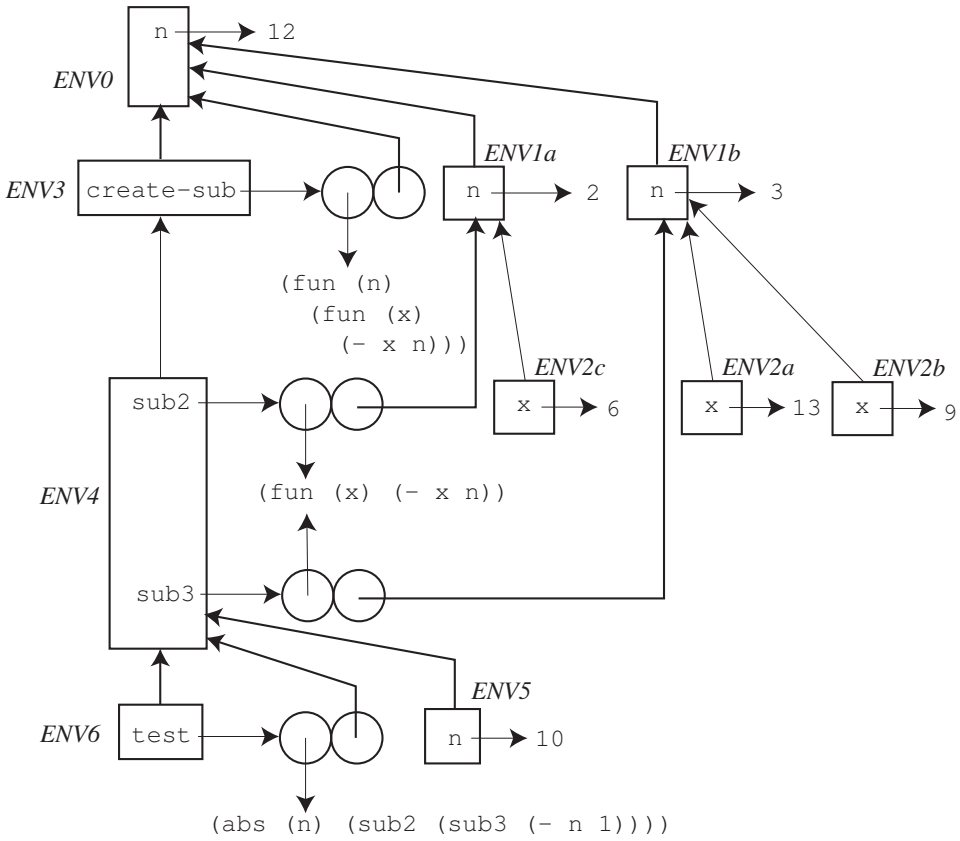


Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3. First, the subexpressions `add-a` and `(* 2 a)` must be evaluated in ENV_3; these evaluations yield the add-a closure and 26, respectively. Next, the closure is applied to 26. This creates a new frame ENV_1 binding `x` to 26; by rule 2 from above, the parent of this frame is ENV_0, the environment of closure; the environment ENV_3 of the function application is simply not involved in this decision.



As the final step, the abstraction body `(+ x a)` is evaluated in ENV_1. Since `x` evaluates to 26 in ENV_3 and a evaluates to 3, the final answer is 29.

As a second example of static scoping in the environment model, consider running the `create-sub` program from the previous section on the input 12. Below is an environment diagram showing all environments created during the evaluation of this program. You should study this diagram carefully and understand why the parent pointer of each environment frame is the way it is. The final answer of the program (which is not shown in the environment model itself) is 4.



In both of the above environment diagrams, the environment names have been chosen to underscore a critical fact that relates the environment diagrams to the contour diagrams. Whenever environment frame ENV_{i} has a parent pointer to environment frame ENV_{j} in the environment model, the corresponding contour C_{i} is nested directly inside of C_{j} within the contour model. For example, the environment chain $ENV_6 \rightarrow ENV_4 \rightarrow ENV_3 \rightarrow ENV_0$ models the contour nesting $C_6 \rightarrow C_4 \rightarrow C_3 \rightarrow C_0$, and the environment chains $ENV_{2\mathrm{c}} \rightarrow ENV_{1\mathrm{a}} \rightarrow ENV_0$, $ENV_{2\mathrm{a}} \rightarrow ENV_{1\mathrm{b}} \rightarrow ENV_0$, and $ENV_{2\mathrm{b}} \rightarrow ENV_{1\mathrm{b}} \rightarrow ENV_0$ model the contour nesting $C_2 \rightarrow C_1 \rightarrow C_0$.

These correspondences are not coincidental, but by design. Since static scoping is defined by the contour diagrams, the environment model must somehow encode the nesting of contours. The environment component of closures is the mechanism by which this correspondence is achieved. The environment component of a closure is guaranteed to point to an environment ENV_{birth} that models the contour enclosing the abstraction of the closure. When the closure is applied, the newly constructed frame extends ENV_{birth} with a new frame that introduces bindings for the parameters of the abstraction. These are exactly the bindings implied by the contour of the abstraction. Any expression in the body of the abstraction is then evaluated relative to the extended environment.

8.4 SML Interpreter Implementation of HOFL Environment Model

Now we show how to implement the environment model for HOFL in an interpreter written in SML. Figure 12 shows the SML datatypes used in the interpreter. These are similar to VALEX except:

- There are new constructors `Abs` (for unary abstractions), `App` (for unary applications), and `Bindrec` (for the recursively-scoped `bindrec` construct).
- There is no need for a `Bind` constructor, because `bind` is sugar in HOFL.
- There is a new `Fun` value for function closures. The third component of a `Fun` value, an environment, plays a very important role in the environment model.

```
type var = string

datatype pgm = Hofl of var list * exp (* param names, body *)

and exp =
    Lit of value (* integer, boolean, character, string, symbol, and
       list literals *)
  | Var of var (* variable reference *)
  | PrimApp of primop * exp list (* primitive application with rator,
     rands *)
  | If of exp * exp * exp (* conditional with test, then, else *)
  (*** New in HOFL ***)
  | Abs of var * exp (* function abstraction *)
  | App of exp * exp (* function application *)
  | Bindrec of var list * exp list * exp (* recursive bindings *)

and value = (* use value rather than val b/c val is an SML keyword *)
    Int of int
  | Bool of bool
  | Char of char
  | String of string
  | Symbol of string
  | List of value list (* Recursively defined value *)
  | Fun of var * exp * value Env.env (*** New in HOFL ***
                                         Closure value combines
                                            abstraction and
                                            environment *)

and primop = Primop of var * (value list -> value)

fun primopName (Primop(name,_)) = name
fun primopFunction (Primop(_,fcn)) = fcn
```

Figure 12: SML data types for the abstract syntax of HOFL.

Rules 1 and 2 of the previous section are easy to implement in an environment model interpreter. The implementation is shown in Figure 13. Note that it is not necessary to pass `env` as an argument to `funapply`, because static scoping dictates that the call-time environment plays no role in applying the function.

9 Dynamic Scoping

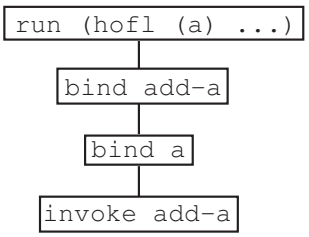
9.1 Environment Model

In dynamic scoping, environments follow the shape of the invocation tree for executing the program. Recall that an invocation tree has one node for every function invocation in the program, and that each node has as its children the nodes for function invocations made directly within in its body, ordered from left to right by the time of invocation (earlier invocations to the left). Since `bind` desugars into a function application, we will assume that the invocation tree contains nodes for `bind` expressions as well. We will also consider the execution of the top-level program to be a kind of function application, and its corresponding node will be the root of the invocation tree. For example, here is the invocation tree for the `add-a` program:

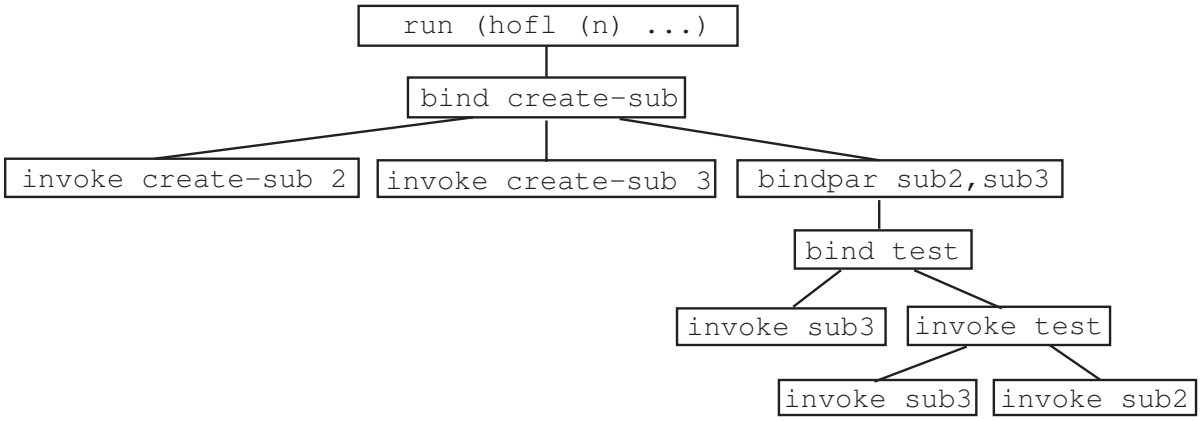
```
(* val eval : Hofl.exp -> value Env.env -> value *)
and eval (Lit v) env = v
      ⋮
  | eval (Abs(fml,body)) env = Fun(fml,body,env) (* make a closure *)
  | eval (App(rator,rand)) env = apply (eval rator env) (eval rand env)
      ⋮

and apply (Fun(fml,body,env)) arg = eval body (Env.bind fml arg env)
  | apply  fcn arg = raise (EvalError ("Non-function rator in
     application: " ^ (valueToString fcn)))
```

Figure 13: Essence of static scoping in HOFL.



As a second example, here is the invocation tree for the `create-sub` program:



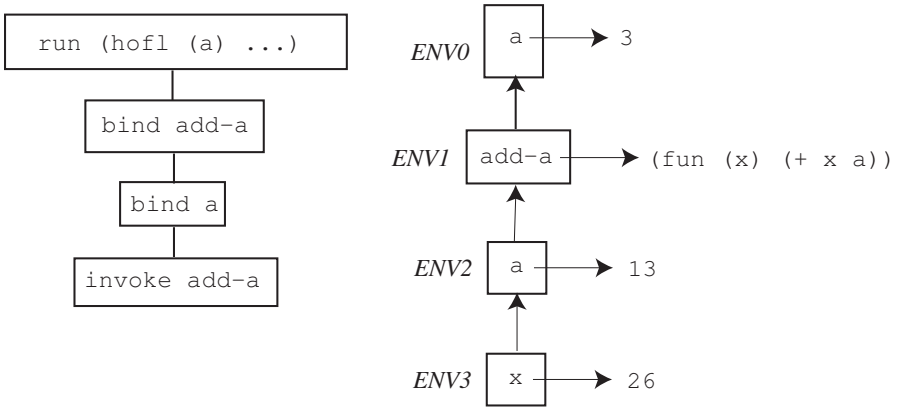
Note: in some cases (but not the above two), the shape of the invocation tree may depend on the values of the arguments at certain nodes, which in turn depends on the scoping mechanism. So the invocation tree cannot in general be drawn without fleshing out the details of the scoping mechanism.

The key rules for dynamic scoping are as follows:

1. Evaluating an abstraction *ABS* in an environment *ENV* just returns *ABS*. In dynamic scoping, there there is no need to pair the abstraction with its environment of creation.

2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment in which the function application is being evaluated - that is, the environment of the invocation (call), not the environment of creation. This means that the free variables in the abstraction body will be looked up in the environment where the function is called.

Consider the environment model showing the execution of the `add-a` program on the argument 3 in a dynamically scoped version of HOFL. According to the above rules, the following environments are created:



The key differences from the statically scoped evaluation are (1) the name `add-a` is bound to an abstraction, not a closure and (2) the parent frame of ENV_3 is ENV_2, not ENV_0. This means that the evaluation of `(+ x a)` in ENV_3 will yield 39 under dynamic scoping, as compared to 29 under static scoping.

Figure 14 shows an environment diagram showing the environments created when the `create-sub` program is run on the input 12. The top of the figure also includes a copy of the invocation tree to emphasize that in dynamic scope the tree of environment frames has *exactly* the same shape as the invocation tree. You should study the environment diagram and justify the target of each parent pointer. Under dynamic scoping, the first invocation of `sub3` (on 13) yields 1 because the `n` used in the subtraction is the program parameter `n` (which is 12) rather than the 3 used as an argument to `create-sub` when creating `sub3`. The second invocation of `sub3` (on 0) yields -1 because the n found this time is the argument 1 to test. The invocation of `sub2` (on -1) finds that `n` is this same 1, and returns -2 as the final result of the program.

9.2 Interpreter Implementation of Dynamic Scope

The two rules of the dynamic scoping mechanism are easy to encode in the environment model. The implementation is shown in Figure 13. For the first rules, the evaluation of an abstraction just returns the abstraction. For the second rules, the application of a function passes the call-time environment to funapply-dynamic, where it is used as the parent of the environment frame created for the application.

9.3 Comparing Static and Dynamic Scope

SNOBOL4, APL, most early LISP dialects, and many macro languages are dynamically scoped. In each of these languages, a free variable in a function (or macro) body gets its meaning from the environment at the point where the function is called rather than the environment at the point where the function is created. Thus, in these languages, it is not possible to determine a unique declaration corresponding to a given free variable reference; the effective declaration depends on where the function is called. It is therefore generally impossible to determine the scope of a declaration simply by considering the abstract syntax tree of the program.

By and large, however, most modern languages use static scoping because, in practice, static scoping is often preferable to dynamic scoping. There are several reasons for this:

- Static scoping has better modularity properties than dynamic scoping. In a statically scoped language, the particular names chosen for variables in a function do not affect its behavior, so

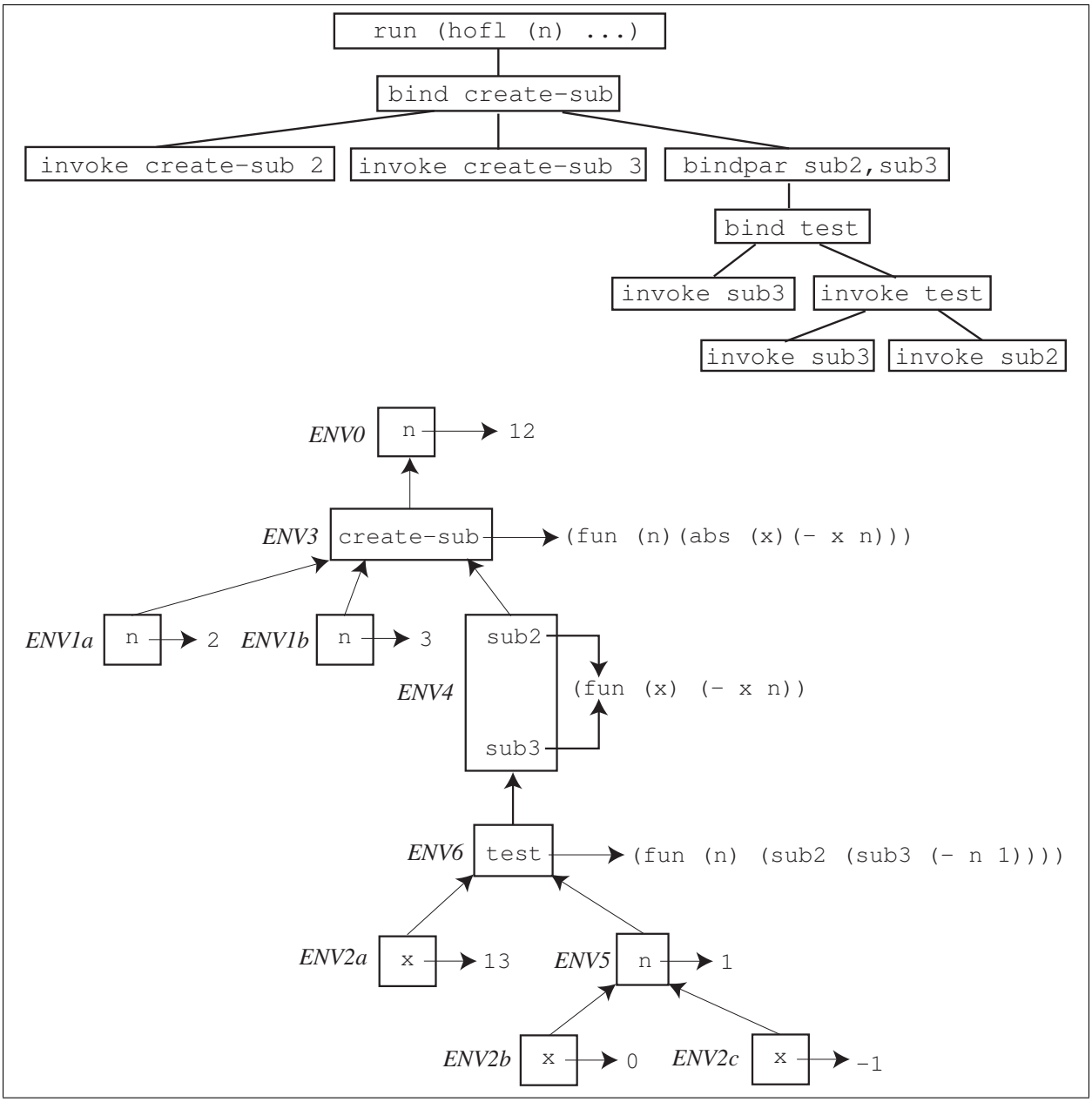


Figure 14: Invocation tree and environment diagram for the `create-sub` program run on 12.

it is always safe to rename them in a consistent fashion. In contrast, in dynamically scoped systems, the particular names chosen for variables matter because a local variable name can interact with a free variable name of a function invoked in its scope. Function interfaces are more complex under dynamic scoping because they must mention the free variables of the function.

- Static scoping works nicely with block structure to create higher-order functions that "remember" information from outer scopes. Many of the functional-programming idioms we have studied depend critically on this "memory" to work properly. As a simple example, consider the HOFL definition

  ```
  (def add (abs x (abs y (+ x y))))
  ```

  Under static scope, `(add 1)` stands for an incrementing function because the returned func-

```
(* val eval : Hofl.exp -> value Env.env -> value *)
and eval (Lit v) env = v
      ⋮
  | eval (Abs(fml,body)) env = Fun(fml,body,env) (* make a closure *)
  | eval (App(rator,rand)) env = apply (eval rator env) (eval rand env)
      env
      ⋮

and apply (Fun(fml,body,senv)) arg denv = eval body (Env.bind fml arg
    denv) (* extend dynamic env *)
  | apply  fcn arg = raise (EvalError ("Non-function rator in
      application: " ^ (valueToString fcn)))
```

Figure 15: Essence of dynamic scoping in HOFL.

tion "remembers" that `x` is 1. But under dynamic scope, `(add 1)` "forgets" that `x` is 1. The returned function is equivalent to `(abs y (+ x y))` and will use whatever value for `x` it finds (if there is one) in the context where it is called. Clearly, dynamic scope and higher-order functions do not mix well!

In particular, in HOFL, multi-parameter functions are desugared into single-parameter functions using currying, whose proper behavior depends critically on the sort of "remembering" described above. So multi-parameter functions will simply not work as expected in a dynamically-scoped version of HOFL! For this reason, multi-parameter functions must be kernel constructs in a dynamically scoped lanuage.

- Statically scoped variables can be implemented more efficiently than dynamically scoped variables. In a compiler, references to statically scoped variables can be compiled to code that accesses the variable value efficiently using its **lexical address**, a description of its location that can be calculated from the program's abstract syntax tree. In contrast, looking up dynamically scoped variables implies an inefficient search through a chain of bindings for one that has the desired name.

Is dynamic scoping ever useful? Yes! There are at least two situations in which dynamic scoping is important:

- *Exception Handling*: In the languages we have studied so far, computations cannot proceed after encountering an error. However, later we will study ways to specify so-called **exception handlers** that describe how a computation can proceed from certain kinds of errors. Since exception handlers are typically in effect for certain subtrees of a program's execution tree, dynamic scope is the most natural scoping mechanism for the namespace of exception handlers.

- *Implicit Parameters*: Dynamic scope is also convenient for specifying the values of **implicit parameters** that are cumbersome to list explicitly as formal parameters to functions. For example, consider the following `derivative` function in a version of HOFL with floating point operations (prefixed with `fp`):

```
(def derivative
  (fun (f x)
    (fp/ (fp- (f (fp+ x epsilon))
              (f x))
         epsilon)))
```

Note that `epsilon` appears as a free variable in `derivative`. With dynamic scoping, it is possible to dynamically specify the value of `epsilon` via any binding construct. For example, the expression

```
(bind epsilon 0.001
  (derivative (abs x (fp* x x)) 5.0))
```

would evaluate `(derivative (abs x (fp* x x)) 5.0)` in an environment where `epsilon` is bound to 0.001.

However, with lexical scoping, the variable `epsilon` must be defined at top level, and, without using mutation, there is no way to temporarily change the value of `epsilon` while the program is running. If we really want to abstract over `epsilon` with lexical scoping, we must pass it to `derivative` as an explicit argument:

```
(def derivative
  (fun (f x epsilon)
    (fp/ (fp- (f (fp+ x epsilon))
              (f x))
         epsilon)))
```

But then any procedure that uses `derivative` and wants to abstract over `epsilon` must also include `epsilon` as a formal parameter. In the case of `derivative`, this is only a small inconvenience. But in a system with a large number of tweakable parameters, the desire for fine-grained specification of variables like `epsilon` can lead to an explosion in the number of formal parameters throughout a program.

As an example along these lines, consider the huge parameter space of a typical graphics system (colors, fonts, stippling patterns, line thicknesses, etc.). It is untenable to specify each of these as a formal parameter to every graphics routine. At the very least, all these parameters can be bundled up into a data structure that represents the graphics state. But then we still want a means of executing window routines in a temporary graphics state in such a way that the old graphics state is restored when the routines are done. Dynamic scoping is one technique for achieving this effect; side effects are another (as we shall see later).

10 Recursive Bindings

10.1 The `bindrec` Construct

HOFL's `bindrec` construct allows creating mutually recursive structures. For example, here is the classic `even?`/`odd?` mutual recursion example expressed in HOFL:

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

The scope of the names bound by `bindrec` (`even?` and `odd?` in this case) includes not only the body of the `bindrec` expression, but also the definition expressions bound to the names. This

distinguishes `bindrec` from `bindpar`, where the scope of the names would include the body, but not the definitions. The difference between the scoping of `bindrec` and `bindpar` can be seen in the two contour diagrams in figure 16. In the `bindrec` expresion, the reference occurrence of `odd?` within the `even?` abstraction has the binding name `odd?` as its binding occurrence; the case is similar for `even?`. However, when `bindrec` is changed to `bindpar` in this program, the names `odd?` and `even?` within the definitions become unbound variables. If `bindrec` were changed to `bindseq`, the occurrence of `even?` in the second binding would reference the declaration of `even?` in the first, but the occurrence of `odd?` in the first binding would still be unbound.

10.2 Environment-Model Evaluation of `bindrec`

10.2.1 High-level Model

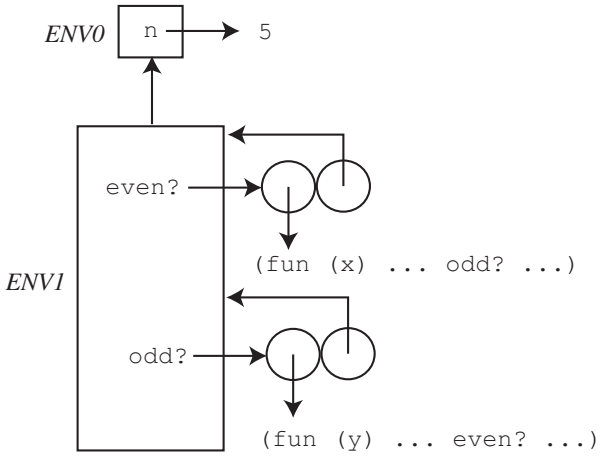
How is `bindrec` handled in the environment model? We do it in three stages:

1. Create an empty environment frame that will contain the recursive bindings, and set its parent pointer to be the environment in which the `bindrec` expression is evaluated.

2. Evaluate each of the definition expressions with respect to the empty environment. If evaluating any of the definition expressions requires the value of one of the recursively bound variables, the evaluation process is said to encounter a **black hole** and the `bindrec` is considered ill-defined.

3. Populate the new frame with bindings between the binding names and the values computed in step 2. Adding the bindings effectively "ties the knot" of recursion by making cycles in the graph structure of the environment diagram.

The result of this process for the `even?`/`odd?` example is shown below, where it is assumed that the program is called on the argument 5. The body of the program would be evaluated in environment ENV_1 constructed by the `bindrec` expression. Since the environment frames for containing `x` and `y` would all have ENV_1 as their parent pointer, the references to odd? and even? in these environments would be well-defined.



In order for `bindrec` to be meaningful, the definition expressions cannot require immediate evaluation of the `bindrec`-bound variables (else a black hole would be encountered). For example, the following `bindrec` example clearly doesn't work because in the process of determining the value of `x`, the value `x` is required before it has been determined:

```
(bindrec ((x (+ x 1)))
  (* x 2))
```

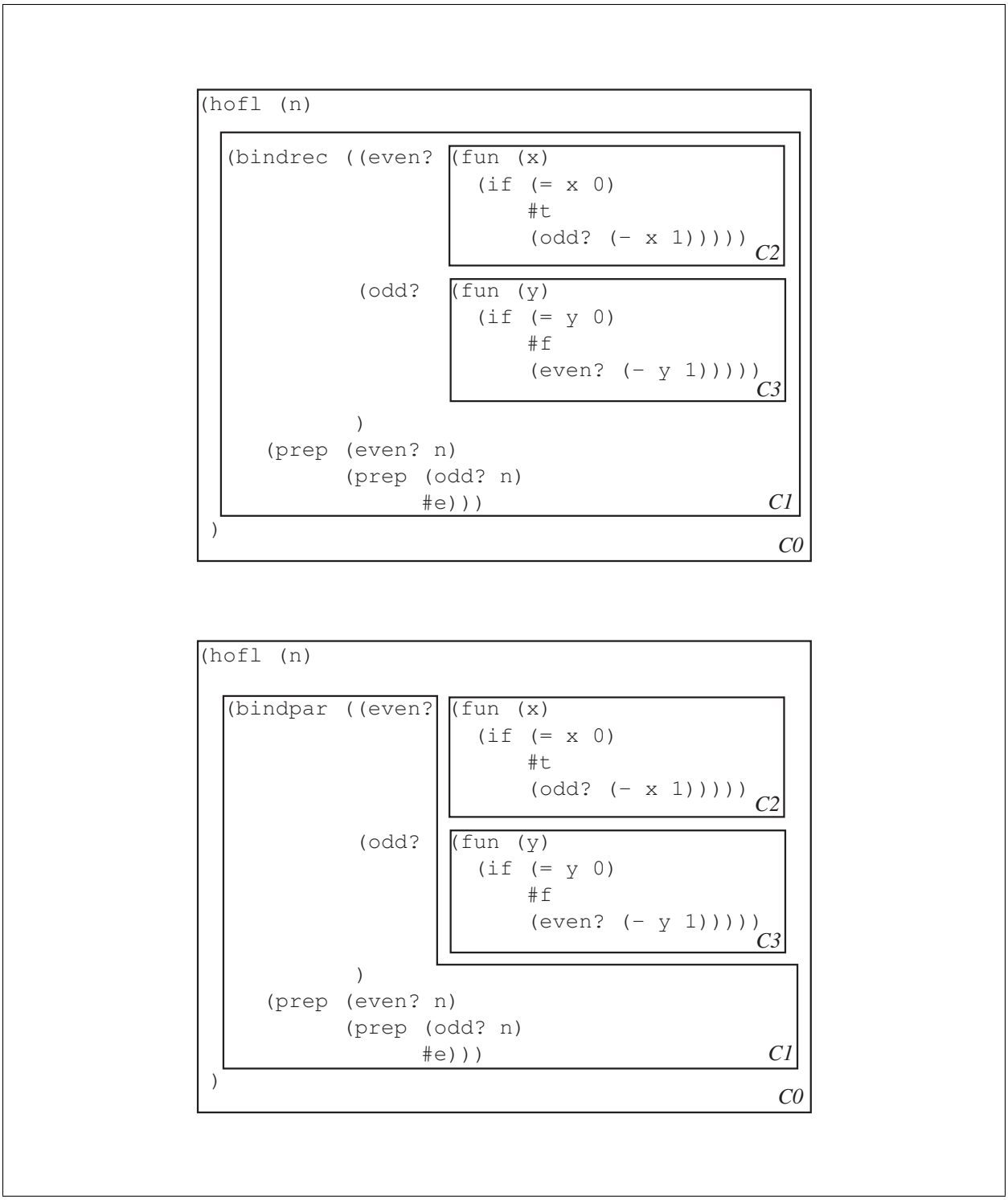


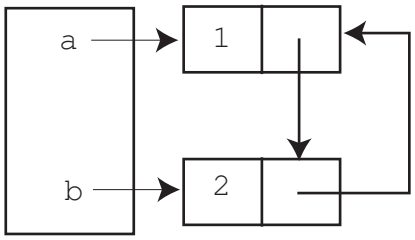
Figure 16: Lexical contours for versions of the `even?`/`odd?` program using `bindrec` and `bindpar`.

In contrast, in the `even?/odd?` example we are not asking for the values of `even?` and `odd?` in the process of evaluating the definitions. Rather the definitions are abstractions that will refer to `even?` and `odd?` at a later time, when they are invoked. Abstractions serve as a sort of delaying mechanism that make the recursive bindings sensible.

As a more subtle example of a meaningless `bindrec`, consider the following:

```
(bindrec ((a (prep 1 b))
          (b (prep 2 a)))
  b)
```

Unlike the above case, here we can imagine that the definition might mean something sensible. Indeed in so-called call-by-need (a.k.a lazy) languages (such as Haskell), definitions like the above are very sensible, and stand for the following list structure:



However, call-by-value (a.k.a. strict or eager) languages (such as HOFL, OCAML, SCHEME, JAVA, C, etc.) require that all definitions be completely evaluated to values before they can be bound to a name or inserted in a data structure. In this class of languages, the attempt to evaluate `(preps 1 b)` fails because the value of `b` cannot be determined.

Nevertheless, by using the delaying power of abstractions, we can get something close to the above cyclic structure in HOFL. In the following program, the references to the recursive bindings `one-two` and `two-one` are "protected" within abstractions of zero variables (which are known as **thunks**). Any attempt to use the delayed variables requires applying the thunks to zero arguments (as in the expression `((snd stream))` within the `prefix` function).

```
(hofl (n)
  (bindpar ((pair (fun (a b) (list a b)))
            (fst (fun (pair) (head pair)))
            (snd (fun (pair) (head (tail pair)))))
    (bindrec ((one-two (pair 1 (fun () two-one)))
              (two-one (pair 2 (fun () one-two)))
              (prefix (fun (num stream)
                        (if (= num 0)
                            (empty)
                            (prep (fst stream)
                                  (prefix (- num 1)
                                          ((snd stream))))))))
      (prefix n one-two))))
```

When the above program is applied to the input 5, the result is `(list 1 2 1 2 1)`.