CS251 Programming Languages **Handout # 20**
Prof. Lyn Turbak **Feb. 27, 2002**
Wellesley College

HOFL

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

Although HOFL supports a list datatype, we will not study lists in the context of HOFL. Instead, we will embark on a more general study of compound data later. later. Lists in HOFL are provided mainly for the convenience of expressing Scheme-like recursive functions.

1 HOFL

The HOFL language extends IBEX with the following features:

1.1 Abstractions and Function Applications

In HOFL, anonymous first-class functions are created via:

(abs (I_{formal_1} ... I_{formal_n}) E_{body})

The `abs` construct corresponds directly to Scheme's `lambda`. It is introduced by a different keyword (`abs` vs. `lambda`) to help us maintain the distinction between HOFL programs and Scheme programs.

As in Scheme, functions applications are expressed by the parenthesized notation

(E_{rator} E_{rand}1 ... E_{rand}n)

where E_{rator} is an arbitrary expression that denotes a function, and $E_{rand}1 \ldots E_{rand}$n denote the n operand values to which the function is applied. For example:

```
hofl> ((abs (a b) (div (+ a b) 2)) 3 5)
4

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.

One difference between HOFL and Scheme is that the names of HOFL primitive operators that appear in the rator position of an application *cannot* be shadowed by outer bindings of the same name. However, names of HOFL primitive operators can be used as regular variables in other positions:

```
scheme> ((lambda (+) (+ 3 4)) (lambda (a b) (* a b)))
12

hofl> ((abs (+) (+ 3 4)) (abs (a b) (* a b)))
7

hofl> ((abs (*) (+ 3 *)) 5)
8
```

1.2 Top-Level Definitions

In HOFL, values and functions may be defined at the "top level" of a program via the `def` construct. For example:

```
(program (x)
  (def five (+ 2 3))
  (def sqr (abs (x) (* x x)))
  (def (sos a b) (+ (sqr a) (sqr b)))
  (sos (* 2 x) five))
```

In general, a HOFL program has the form:

(program (I_{formal_1} ... I_{formal_n})
D_1
⋮
D_k
E_{body})

where each D_i is a definition that is defined by the following grammar:

$D \rightarrow$ (def I_{name} E_{defn})
$D \rightarrow$ (def ($I_{funname}$ I_{formal_1} ... I_{formal_n}) E_{body})

The first form of definition associates I_{name} with the value of E_{defn}. The second form of definition associates $I_{funname}$ with a function that takes formal parameters $I_{funname}$ I_{formal_1} ...I_{formal_n}) and whose body is E_{body}. The second form of definition is just syntactic sugar for

(def $I_{funname}$ (abs (I_{formal_1} ... I_{formal_n}) E_{body})).

Running a HOFL program returns the result of evaluating its body expression in a context where the formal parameters of the program and the names introduced by the definitions are appropriately bound.

Functions define via `def` are mutually recursive with each other. For example:

```
;; Singly recursive function
(program (x)
  (def fact
    (abs (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
  (fact x))
```

```
;; Mutually recursive function
(program (x)
  (def (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (def (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (even? x))
```

1.3 Recursive Local 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:

```
(program (x)
  (def tester
    (abs (bool)
      (bindrec ((test1 (abs (n)
                         (if (= n 0)
                             bool
                             (test2 (- n 1)))))
                (test2 (abs (n)
                         (if (= n 0)
                             (not bool)
                             (test1 (- n 1))))))
        test1)))
  ((tester false) x))
```

1.4 Lists

HOFLsupports the following list operators, which are shown side-by-side with their Scheme analogs:

HOFL	**Scheme**
prepend	cons
head	car
tail	cdr
empty	'()
empty?	null?

```
(program (hi)
  (def (map f lst)
    (if (empty? lst)
        (empty)
        (prepend (f (head lst))
                 (map f (tail lst)))))
  (def (from lo)
    (if (> lo hi)
        (empty)
        (prepend lo (from (+ lo 1)))))
  (bind test-list (from 1)
    (map (abs (f) (map f test-list))
         (list (abs (x) (* x x))
               (abs (y) (= (mod y 2) 0))
               (abs (z) (prepend z
                          (prepend (* 2 z)
                            (empty))))))))
```

The HOFL construct `(list` E_1 E_2 ... E_n`)` desugars to:

```
(prepend E1
  (prepend E2
    ⋮
    (prepend En
      (empty)) ...))
```

This is similar to Scheme, except that Scheme's `list` is a first-class function whereas HOFL's `list` is a special syntactic construct defined by desugaring.

2 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

```
(abs (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:

```
(program (a)
  (bind add-a (abs (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a)))))
```

The `add-a` function is defined by the abstraction `(abs (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** and **dynamic scoping**. To simplify the discussion, here will will only consider HOFL programs that do not use the `bindrec` construct. We will study recursive bindings in more detail later.

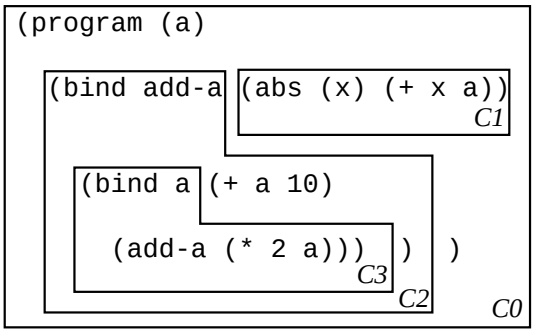
3 Static Scoping

3.1 Contour Model

In **static scoping**, the meaning of every variable reference is determined by the contour boxes introduced in Section 2. 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 `(program (a) ...)`. Similarly, it can be determined that:

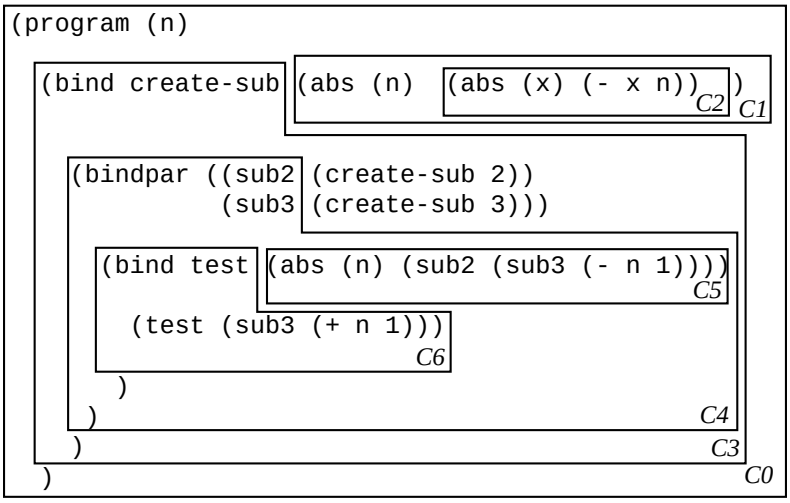
- the `a` in `(+ a 10)` refers to the `a` bound by `(program (a) ...)`;
- the `a` in `(* 2 a)` refers the `a` bound by `(bind a ...)`;
- the `x` in `(+ x a)` refers to the `x` bound by `(abs (x) ...)`.
- the `add-a` in `(add-a (* 2 a))` refers to the `add-a` bound by `(bind add-a ...)`.



Because the meaning of any reference occurrence is apparent from the lexical structure of the program, static scoping is also known as **lexical scoping**.

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

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



By the rules of static scope:

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

3.2 Substitution Model

The same substitution model used to explain the evaluation of Scheme 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.)

Below, use the substitution model to explain the evaluation of the two examples from the previous section:

3.3 Environment Model

We would like to be able to explain static scoping within the environment model of evaluation. Most evaluation rules of the environment model are independent of the scoping mechanism. Such rules are shown in Fig. 1.

Program Running Rule

- To run a HOFL program `(program (`I_1 ... I_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 $I_1 \ldots I_n$ 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 I expression in environment ENV, return the value of looking up I in ENV. If I 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, we first evaluated E_1 in ENV. If the result is true, we return the result of evaluating E_2 in ENV; else we return the result of evaluating E_3 in ENV.
- To evaluate the primitive application `(`*primop* E_1 ... E_n`)` in environment ENV, we must first evaluate the operand expressions E_1 through E_n in ENV. We then return the result of applying the primitive operator primop to the resulting operand values.
- To evaluate the function application `(`E_0 E_1 ... E_n`)` in environment ENV, we must first evaluate the expressions E_0 through E_n in ENV. We then return the result of applying the function value to the operand values. (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.)

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` E_{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 E_{name} and the result of 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.

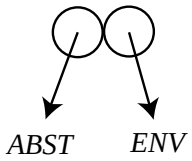
Figure 1: Environment model evaluation rules that are independent of the scoping mechanism.

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?

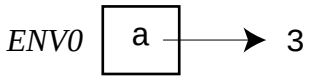
In the case of static scoping, answering these questions yields the following rules:

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:

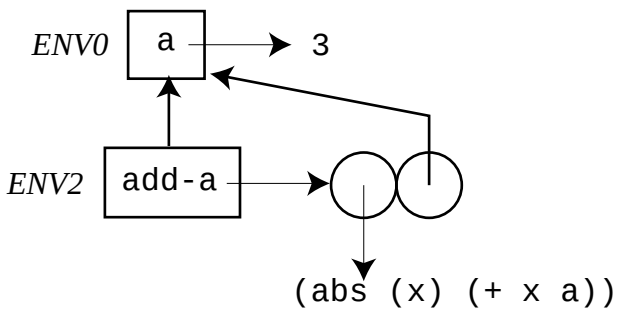


2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment remembered by the closure. That is, the new frame should extend the environment where the closure 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.

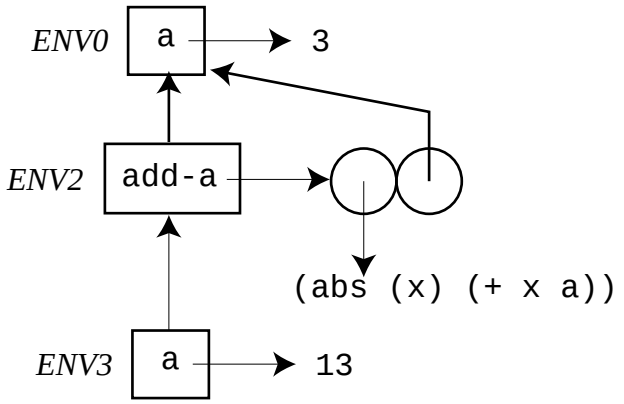
We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



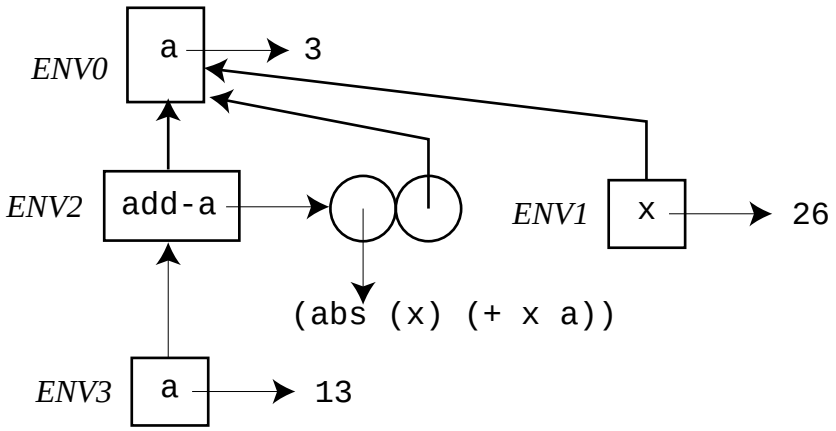
To evaluate the `(bind add-a ...)` expression, we must first evaluation the definition `(abs (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



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:

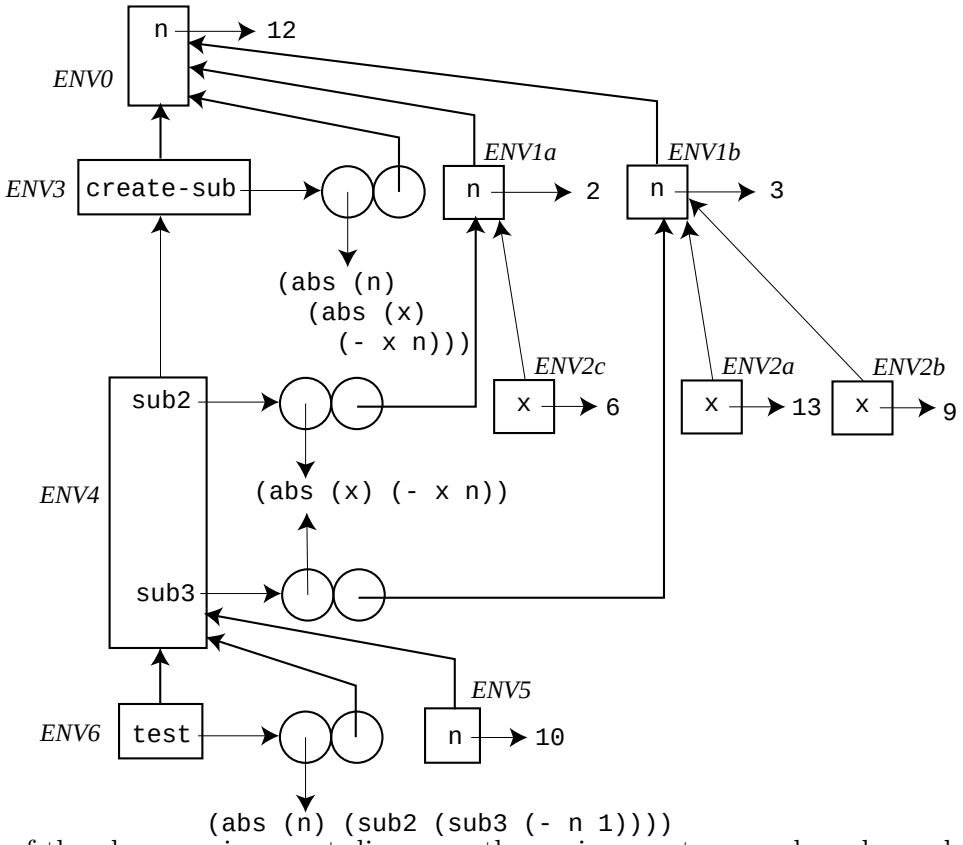


Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3. First, the subexpressions `add-a` and `(* 2 a)` must be evaluated in ENV_3; these evaluations yield the add-a closure and 26, respectively. Next, the closure is applied to 26. This creates a new frame ENV_1 binding `x` to 26; by 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.

3.4 Interpreter Implementation of Environment Model

Rules 1 and 2 of the previous section are easy to implement in an environment model interpreter. The implementation is shown in Figure 2. 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.

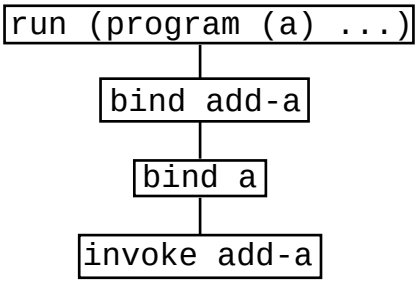
```
;; Implementation of ENV-EVAL using static scope
(define env-eval
  (lambda (exp env)
              .
              .
              .
    ;; Clause corresponding to rule 1
    ((abs? exp)
     (make-closure exp env)) ;; Remember environment of creation

    ;; Clause corresponding to rule 2
    ((funapp? exp)
     (let ((closure (env-eval (funapp-rator exp) env))
           (actuals (env-eval-list (funapp-rands exp) env)))
      (funapply closure actuals)))
              .
              .
              .
    ))

;; Auxiliary function used by clause for rule 2
(define funapply
  (lambda (closure actuals)
    (cond ((not (closure? closure))
           (throw 'funapply:application-of-non-closure closure))
          (not (= (length (closure-formals closure))
                  (length actuals))
               (throw 'funapply:formals-actuals-mismatch
                      list (closure-formals closure) actuals))
          (else
            (env-eval (closure-body closure)
                      (env-extend (closure-formals closure)
                                  actuals
                                  (closure-env closure) ;; env of creation
                                  )))
          )))
```

Figure 2: Essence of static scoping.

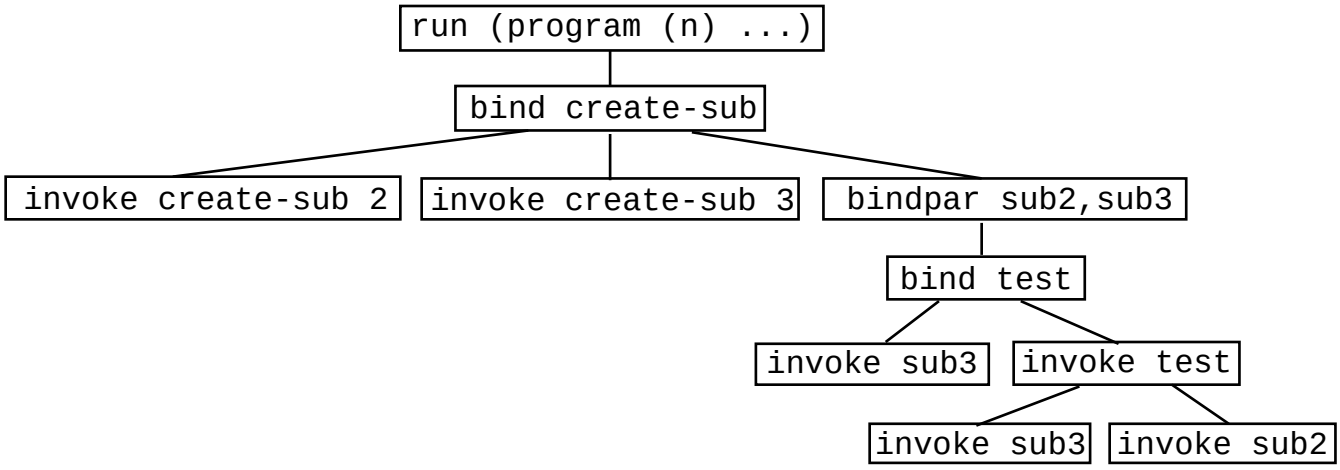
4 Dynamic Scoping

4.1 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:



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.

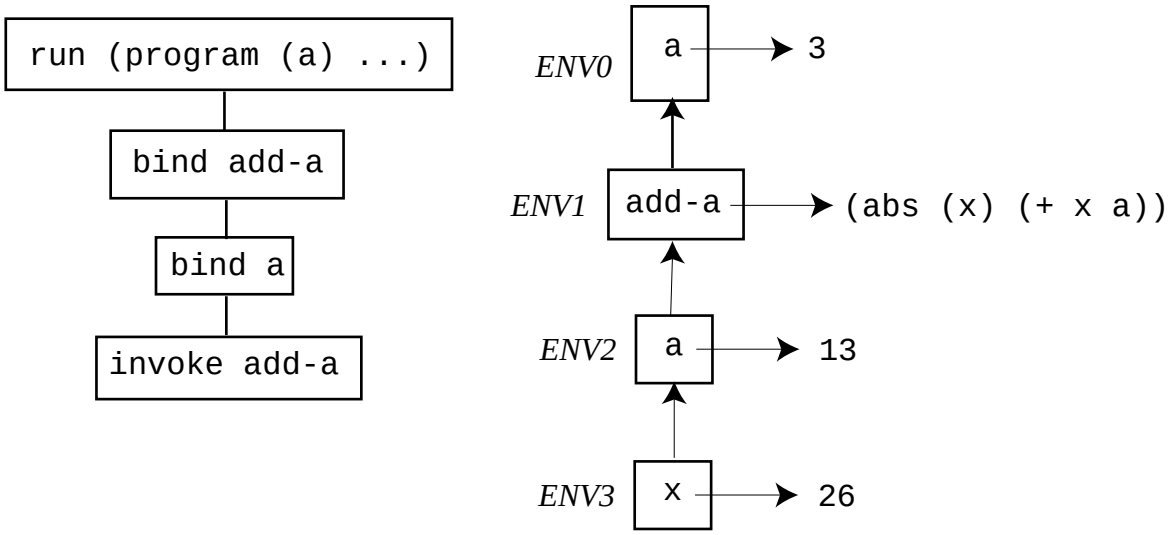
The key rules for dynamic scoping are as follows:

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 3 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.

4.2 Interpreter Implementation

The two rules of the dynamic scoping mechanism are easy to encode in the environment model. The implementation is shown in Figure 2. 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.

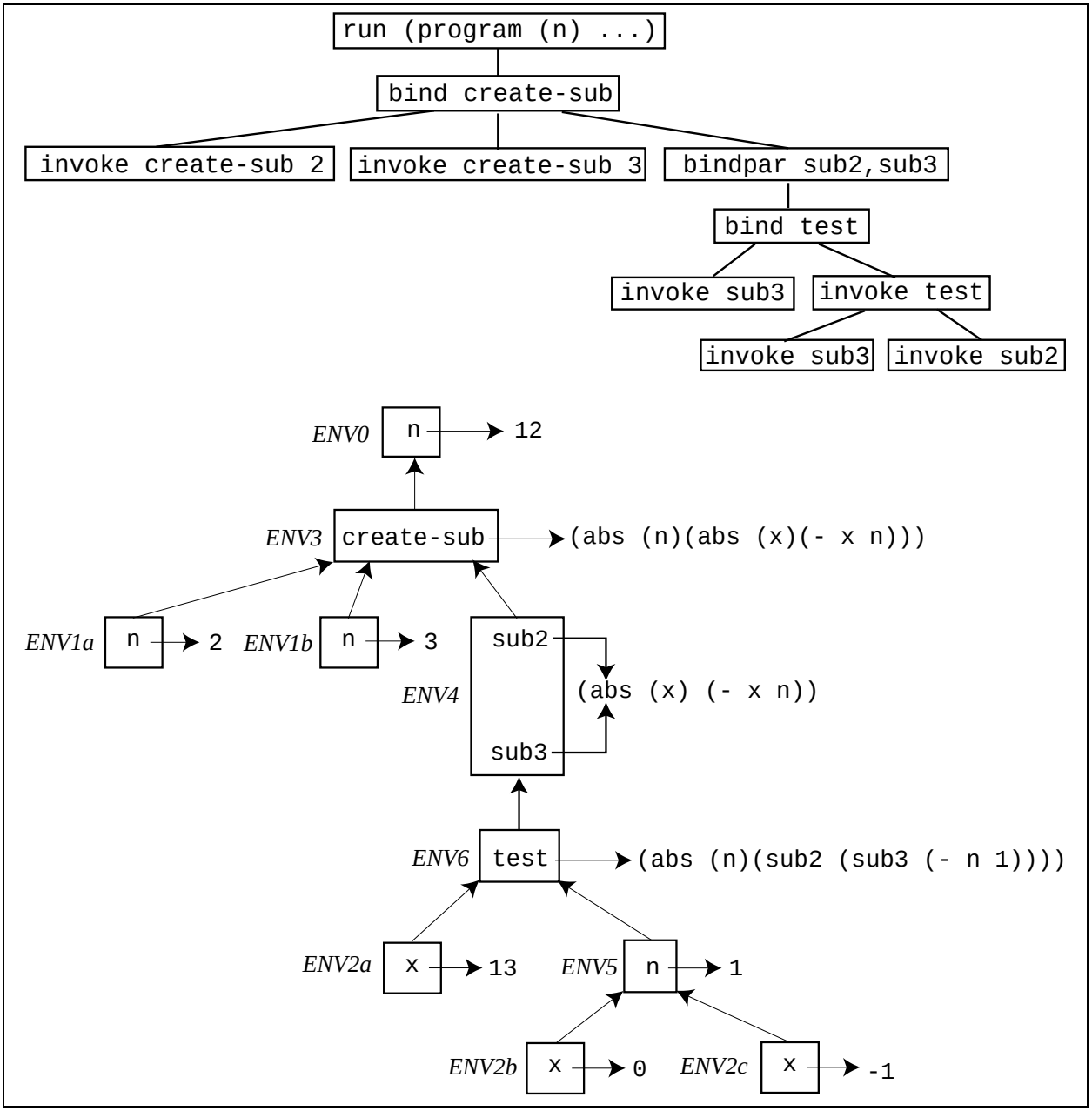


Figure 3: Invocation tree and environment diagram for the `create-sub` program run on 12.

```
;; Implementation of ENV-EVAL using dynamic scope
(define env-eval
  (lambda (exp env)
              .
              .
              .
    ;; Clause corresponding to rule 1
    ((abs? exp) exp) ; No need to create a closure in dynamic scope

    ;; Clause corresponding to rule 2
    ((funapp? exp)
      (let ((abst (env-eval (funapp-rator exp) env))
            (actuals (env-eval-list (funapp-rands exp) env)))
        (funapply abst actuals env))) ; Pass env of call
              .
              .
              .
    ))

;; Auxiliary function used by clause for rule 2
(define funapply
  (lambda (abst actuals dyn-env)
    (cond ((not (abs? abst))
           (throw 'funapply:application-of-non-abstraction abst))
          (not (= (length (abs-formals abst))
                  (length actuals))
               (throw 'funapply:formals-actuals-mismatch
                      list (abs-formals abst) actuals))
          (else
            (env-eval (abs-body abst)
                      (env-extend (abs-formals abst)
                                  actuals
                                  dyn-env))) ;; env of call
          )))
```

Figure 4: Essence of dynamic scoping.