CS251 Programming Languages **Handout # 26**
Prof. Lyn Turbak **April 3, 2001**
Wellesley College

Naming Issues

This handout summarizing naming issues (e.g., namespaces, scoping, block structure, environments, and closures) for the real and toy languages that we have studied so far in the course. It is required reading for completing Problem Set 5.

1 Declarations

Every programming language provides constructs that that introduce names for the kinds of entities that are manipulated by the language. Such constructs are known as declarations or binding constructs. Below are some of the languages we have studied, along with a list of (some of) their declaration constructs:

Toy Languages

- INTEX:
  - `program`: introduces parameters naming program inputs;
- BINDEX/IBEX:
  - `program`: introduces parameters naming program inputs;
  - `bind`: introduces a name for a calculated value;
  - `bindpar`, `bindseq`: introduce names for calculated values (desugar into `bind`).
- FOFL:
  - `fun`: introduces function names and function parameters;
  - `program`, `bind`, `bindpar`, `bindseq`: as in BINDEX/IBEX.
- FOBS:
  - `funrec`: introduces function names and function parameters;
  - `program`, `bind`, `bindpar`, `bindseq`, `fun`: as in FOFL (collections of `fun` desugar to `funrec`).
- HOFL:
  - `abs`: introduces function parameters;
  - `bindrec`: introduces recursively bound values;
  - `program`, `bind`, `bindpar`, `bindseq`, `fun`, `funrec`: as in FOBS (`funrec` desugars into `bindrec` and `abs`).

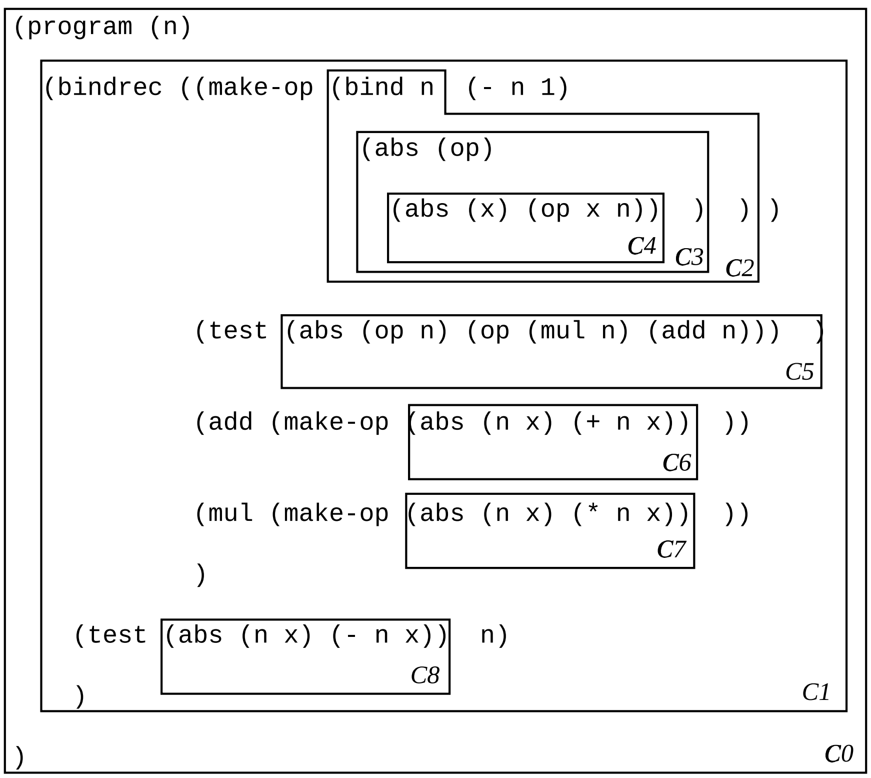
Real Languages

- Scheme:
  - `lambda`: introduces function parameters;
  - `let`: introduces names for calculated values (desugars into `lambda`);
  - `letrec`: introduces names for recursively defined values;
  - `define`: adds a name/value binding to the current environment (sequences of `define`s desugar into `letrec`).
- ML:
  - `fn`: introduces patterns for function parameters;
  - `val`: introduces patterns for calculated values ;
  - `fun`: introduces names and function parameter patterns for recursively defined functions (similar to a combination of `val` and `fn` except for recursive scope);
  - `case`: introduces patterns for summands of a discriminant;
  - `datatype`: introduces names of data type constructors;
  - `type`: introduces names that abbreviate types;
  - `exception`: introduces names for exception constructors;
  - `handle`: introduces patterns for the handled exception.
  - `structure`: introduces names for structures.
  - `signature`: instroduces names for signatures.
- Java:
  - `method`: introduces names for instance and class methods and their parameters (class methods are distinguished by the static keyword);
  - `class`: introduces names for classes;
  - instance variable declaration syntax (*type name*`=` *exp*, within a class declaration): introduces names for instance variables;
  - class variable declaration syntax (`static` *type name* `=` *exp*, within a class declaration): introduces names for class variables;
  - local variable declaration syntax (*type name* `=` *exp*, within a method body): introduces names for local variables.
  - `catch`: introduces a named for the caught exception.

2 Scope

Every declaration construct has a **binding occurrence** that introduces the declared name, and **reference occurrences** that refer to declared name. For example, in the Scheme abstraction `(lambda (x) (* x x))`, the first `x` is the binding occurrence, and the second and third `x`s are reference occurrences. Typically, the binding occurrence and reference occurrences have the same syntax; they are distinguished by their positions within the declaration construct. So in `lambda`, for instance, the parenthesized list of names following the `lambda` keyword are the binding occurrences, and the uses of these names in the body are reference occurrences.

Once declared, a name can usually only be used within a restricted part of the program. The region of a program in which it is possible to reference a declared name is called the **scope** of the declared name. In statically scoped languages (see Section 4) the scope of declared names can be shown via nested boxes called **lexical contours**. For example, the following diagram shows the lexical contours for a sample HOFL program:



Each contour shows the region of the program in which the names declared by the declaration in the contour can be used. For instance, the contour labeled C_3 shows the region of the program in which the variable `op` introduced by `(abs (op) ...)` can be used. The fact that the contour C_2 for the `(bind n (- n 1) ...)` expression includes the binding occurrence `n` and the body expression but not the definition expression `(- n 1)` indicates that the definition expression of a `bind` construct is not within the scope of the declared name. In contrast, the diagram indicates that the body and all definition expressions of a `bindrec` construct are within the scope of the declared names. This means that any definition expression of a `bindrec` can refer to the names declared for any of the definition expressions, including itself or later definitions. This makes it possible for the `test` definition to have a so-called **forward reference** to `add` and `mul`.

A name declared in an outer contour may be used within an inner contour unless the inner contour declares the same name as the outer contour. For example, the name `op` declared in C_3 may be used within C_4. However, the program parameter `n` declared in C_0 may not be used within contours C_2, C_5, C_6, C_7, or C_8, since all of these contours also declare a variable named `n`. The inner declarations of `n` are said to **shadow** the outer one, and the contours of the inner declarations are said to be **holes in the scope** of the outer declarations.

Lexical contours are especially helpful for reasoning about programs in which the same name is introduced by multiple declarations. In the above example, there are two logically distinct variables named `op`, four logically distinct variables named `x`, and 6 logically distinct variables named `n`.

In a statically scoped language, it is always possible to consistently rename binding occurrences and their corresponding reference occurrences in such a way that each binding occurrence has a unique name. For instance, performing consistent renaming on the sample program above can yield the following program, in which each potentially ambiguous variable name has been renamed using the index of its contour in the above diagram:

```
(program (n0)
  (bindrec ((make-op (bind n2 (- n0 1)
                       (abs (op3)
                         (abs (x4) (op3 x4 n2)))))
            (test (abs (op5 n5) (op5 (mul n5) (add n5))))
            (add (make-op (abs (n6 x6) (+ n6 x6))))
            (mul (make-op (abs (n7 x7) (* n7 x7))))
            )
    (test (abs (n8 x8) (- n8 x8)) n0)))
```

Consistent renaming that maintains program meaning is known as α**-renaming**. Alpha-renaming refers to any process of consistent renaming, not just renamings that make all binding occurrences unique. In the programming language literature, it is common to refer to α**-equivalence classes**, which are equivalence classes of expressions modulo α-renaming. This means that expressions that differ only in the naming of their variables are considered equivalent. For instance, the HOFL abstractions `(abs (a) (abs (b) (+ a b)))` and `(abs (x) (abs (y) (+ x y)))` are α-equivalent. Alpha-equivalence captures the notion that it is not the choice of names that matters, but rather the connectivity of reference occurrences and binding occurrences.

Even though names in some sense "do not matter", one must still pay close attention to particular name choices when α-renaming a program. For instance, suppose we want to rename the `b` in `(abs (a) (abs (b) (+ a b)))`. We can choose any name we want as the new name for `b` *except* `a`. The problem with renaming `b` to `a` is that in the resulting expression, `(abs (a) (abs (a) (+ a a)))`, the first `a` within `(+ a a)` no longer references the outer declaration of `a` but the inner one. We say that this reference occurrence of `a` has been **captured** by the inner declaration of `a`. In the presence of such variable capture, the resulting expression is not α-equivalent to the original.

In a given program phrase, a reference occurrence of a name for which there is no binding occurrence is called a **free variable**; otherwise it is said to be a **bound variable**. For instance, in the HOFL expression `(abs (y) (* x (+ y z)))`, `x` and `z` are free variables. In the corresponding Scheme expression `(lambda (y) (* x (+ y z)))`, `*` and `+` are free variables in addition to `x` and `z`. Note that some occurrences of a name in an expression can be free while others are bound. So in the HOFL expression `((abs (x) (+ x 1)) x)`, the first reference occurrence of `x` is bound while the second occurrence is free. Whether or not a variable is free in an expression depends on its context. For instance, `x` is a bound variable in `(abs (x) (+ x 1))`, but is a free variable in the body expresion `(+ x 1)`.

It's worth noting that all the naming concepts discussed in this section apply to general mathematics notational systems, not just to programming languages. For instance, the notions of binding occurrences, reference occurrences, scope, holes in scope, renaming, variable capture, and free variables carry over to logic (e.g., $\forall x.\exists y.P(x,y)$) calculus (e.g., $\int_0^1 x \cdot (\int_0^x y\, dy)\, dx$), and other mathematical notation (e.g. $\sum_{i=1}^{n} \prod_{j=1}^{k} i \cdot j$).

3 Namespaces

A programming language may have several different categories of names. Each such category is called a namespace. For example, Java has distinct namespaces for packages, classes, methods, instance variables, class variables, and method parameters/local variables.

In a language with multiple namespaces, the same name can simultaneously be used in different namespaces without any kind of naming conflict. For example, consider the following Java class declaration:

```
public class Circle {

  // Instance variable of a Circle object.
  public double radius;

  // Constructor method for creating Circle objects.
  public Circle (double r) {
    this.radius  = r;
  }

  // Instance  method for scaling Circles.
  public Circle scale (double factor) {
    return new Circle(factor * this.radius);
  }
}
```

It turns out that we can rename every one of the names appearing in the above program to `radius` (as shown below) and the class will have the same meaning!

```
public class radius {

  // Instance variable of a circle object.
  public double radius;

  // Constructor method for creating Circle objects.
  public radius (double  radius) {
    this.radius  =  radius;
  }

  // Instance  method for scaling Circles.
  public radius radius (double radius) {
    return new radius(radius * this.radius);
  }
}
```

Of course, in order to use the renamed class, we would need to change uses of the original class consistently. For instance, the expression `(new Circle(10)).scale(2).radius` would have to be renamed to `(new radius(10)).radius(2).radius`.

Although using the name radius to stand for entities in four different namespaces (class, instance variable, instance variable name, parameter name) would make the program very difficult for a human program to read, the Java compiler and Java bytecode interpreter treat the renamed program identically to the original.

Java has an unusually high number of namespaces. But many languages have at least two namespaces: one for functions, and one for variables. For instance, in this category are C, Pascal, and Common Lisp, as well as the toy languages FOFL and FOBS that we have studied. In contrast, many functional languages, such as Scheme, ML, and Haskell (as well as the toy FOFL language) have a single namespace for functions and variables. This is parsimonious with the first-classness of functions, which allows functions to be named like any other values.

As a somewhat silly example, consider the following working definition of a factorial function in FOBS:

```
(fun fact (fact)
  (if (= fact 0)
      1
      (* fact (fact (- fact 1)))))
```

In this example, there are two distinct entities named `fact`: the factorial function (in the function namespace) and the formal parameter of the factorial function (in the variable namespace). Because the namespaces are distinct, there is no confunsion between the entities. If the same experiment were tried in HOFL, Scheme, or ML, however, the function would encounter an error when applied to a number because all occurrences of `fact` in the body — including the one in the operator position — would refer to a number.

4 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?

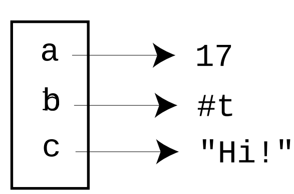
A **scoping mechanism** determines the binding occurrence in a program associated with a free variable reference within a function body. In languages with block structure and/or higher-order functions, it is common to encounter functions with free variables. Understanding the scoping mechanisms of such 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. First we introduce some conventions that are used to explain both scoping mechanisms. Then we explain static and dynamic scoping.

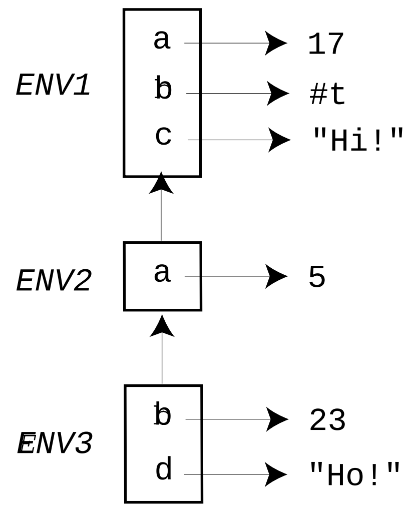
4.1 Environment Diagrams

Scoping mechanisms can be explained in the context of environment diagrams, which are a visual notation for the environment model of execution. We will use the following conventions in drawing environment diagrams.

- An **environment frame** represents an environment extension with bindings between names and values. Environment frames are depicted as boxes with bindings. For example, here is a frame with three bindings:



- An **environment** is represented as a linked chain of environment frames. For example, the following diagram shows three environments:



In a chain of frames, each environment frame except the last one points to its **parent environment**. The bindings in a frame shadow any bindings with the same names in its parent environment. Here are the results of looking up various names in the above environments:

```
env-lookup('a, ENV1) = 17
env-lookup('b, ENV1) = #t
env-lookup('c, ENV1) = "Hi!"
env-lookup('d, ENV1) = unbound

env-lookup('a, ENV2) = 5
env-lookup('b, ENV2) = #t
env-lookup('c, ENV2) = "Hi!"
env-lookup('d, ENV2) = unbound

env-lookup('a, ENV3) = 5
env-lookup('b, ENV3) = 23
env-lookup('c, ENV3) = "Hi!"
env-lookup('d, ENV3) = "Ho!"
```

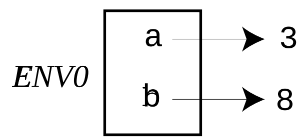
In the **environment model**, every expression is evaluated with respect to an environment. The environment determines the meaning of the free variables that appear within the expression. In many cases, the environment used for evaluating an expression is also used to evaluate its subexpresssions. For instance:

- 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*.

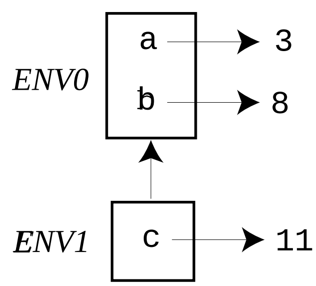
- To evaluate the primitive application `(primop` E_1 `...` E_n`)` in environment *ENV*, we must first evaluate the operand expressions E_1 through E_n in *ENV*. We then return the result of applying the primitive operator primop to the resulting operand values.

- 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.)

The evaluation of binding constructs involves evaluating some subexpressions in an extension of the given environment. For instance, consider the evaluation of `(bind c (+ a b) (div c 2))` in the following environment ENV_0:



The result of this expression is the result of evaluating the body `(div c 2)` in an environment ENV_1 that is the result of extending ENV_0 with a binding between `c` and the result of evaluating `(+ a b)` in ENV_0. Here is the environment ENV_1 resulting from the extension:



Evaluating `(div c 2)` in ENV_1 yields 5.

Here are the general rules for evaluating binding constructs in HOFL:

- 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 desugared into a sequence of nested `bind`s before being evaluated.

- The evaluation of a `bindrec` expression is a bit tricky to explain and is will be explained later in Section 5.

Finally, it is necessary to explain what it means to run a HOFL program. According to the environment model, the result of running a program on a given set of integer arguments is the result of evaluating the body of the program in an environment that binds the program parameters to the arguments. For example, running consider the following program:

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

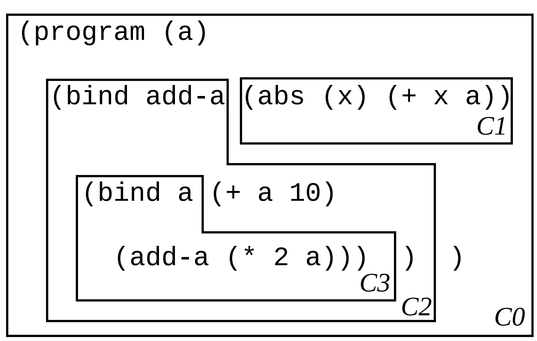
Running this program on the inputs 3 and 8 would give rise to the evaluation of the `bind` expression in the environment ENV_0 considered above. Note that the environment frame created to evaluate the body of a program has no parent environment. It effectively serves as the "global environment" mentioned in *SICP* 3.2.

4.2 Static Scoping: 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.

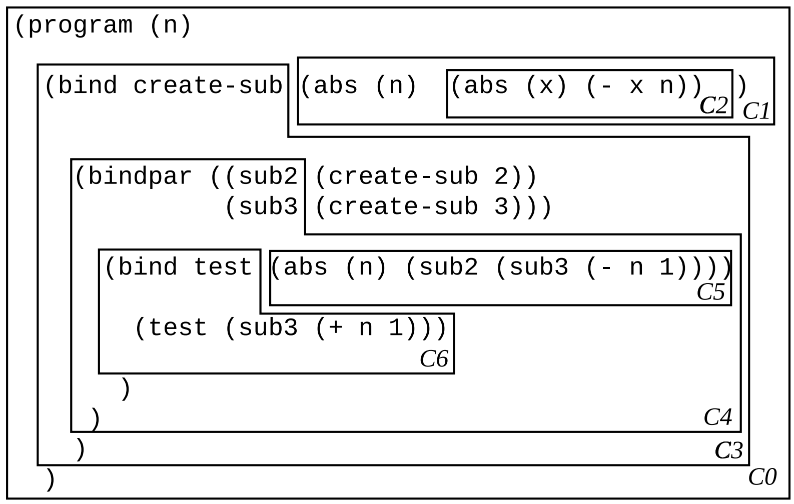
For example, below is the contour diagram associated with the `add-a` example. The reference to `a` in the expression `(+ x a)` lies within contour boxes C_1 and C_0. C_1 does not bind `a`, but C_0 does, so the `a` in `(+ x a)` refers to the *a* bound by `(program (a) ...)`. Similarly, it can be determined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



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) ...)`.

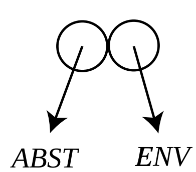
4.3 Static Scoping: Environment Model

We would like to be able to explain static scoping within the environment model of evaluation. 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?

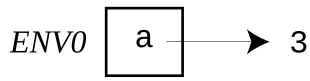
In the case of static scoping, answering these questions yields the following rules:

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:

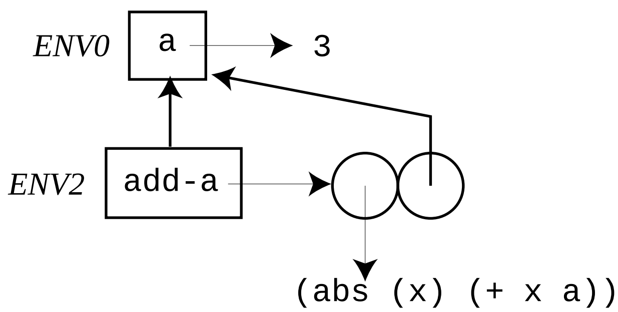


2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment remembered by the closure. That is, the new frame should extend the environment where the closure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

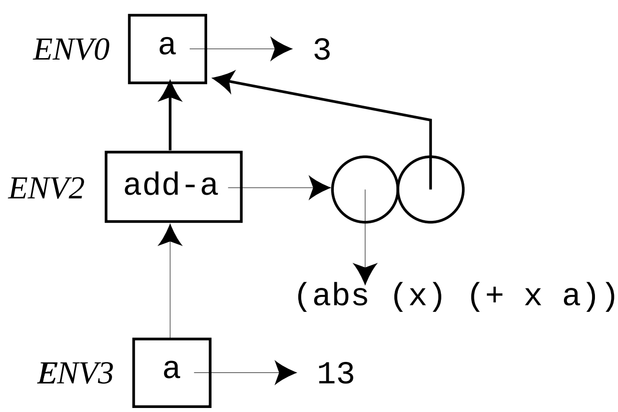
We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



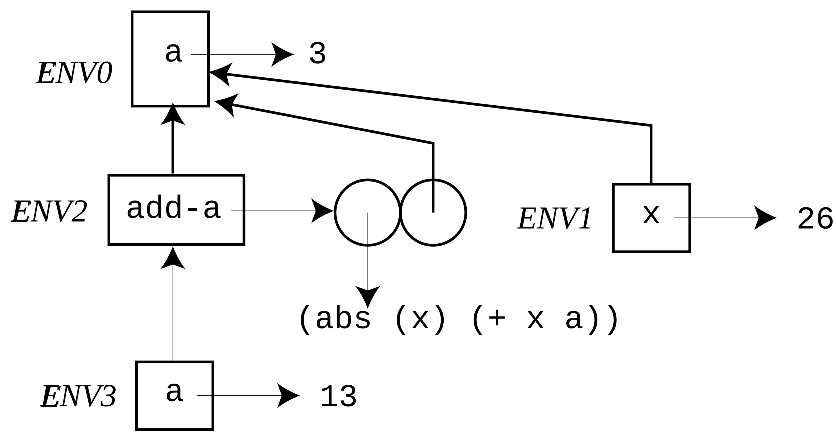
To evaluate the `(bind add-a ...)` expression, we must first evaluation the definition `(abs (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



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:

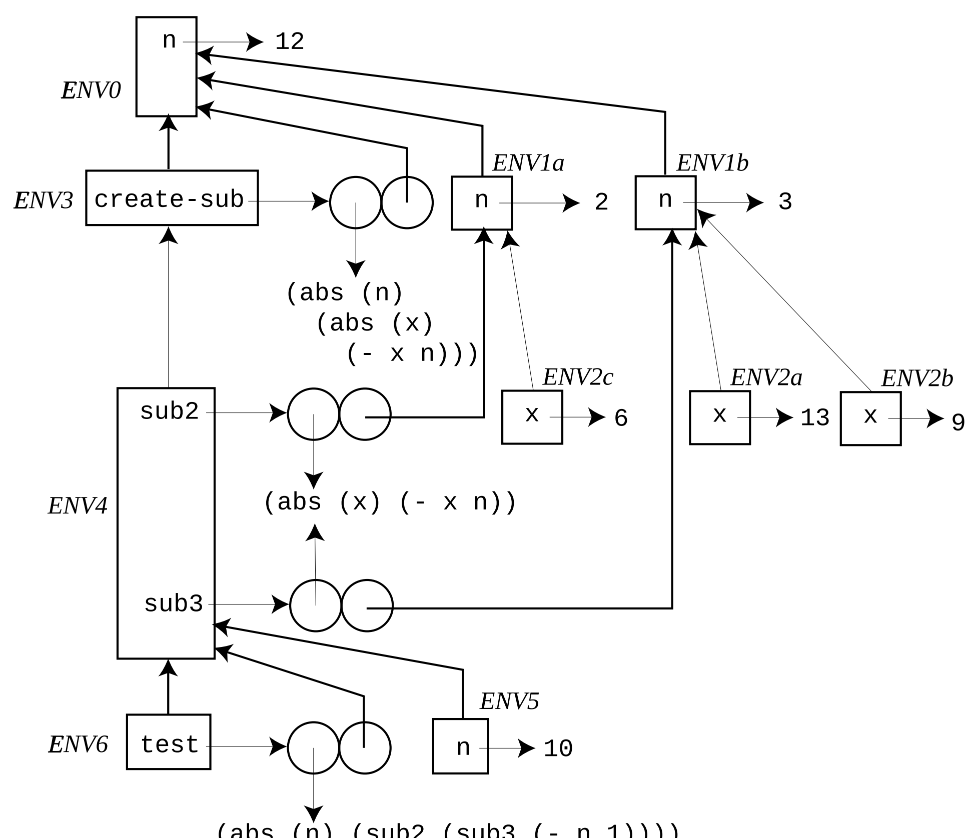


Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3. First, the subexpressions `add-a` and `(* 2 a)` must be evaluated in ENV_3; these evaluations yield the add-a closure and 26, respectively. Next, the closure is applied to 26. This creates a new frame ENV_1 binding `x` to 26; by 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 $ENV_{2c} \rightarrow ENV_{1a} \rightarrow ENV_0$, $ENV_{2a} \rightarrow ENV_{1b} \rightarrow ENV_0$, and $ENV_{2b} \rightarrow ENV_{1b} \rightarrow ENV_0$ model the contour nesting $C_2 \rightarrow C_1 \rightarrow C_0$.

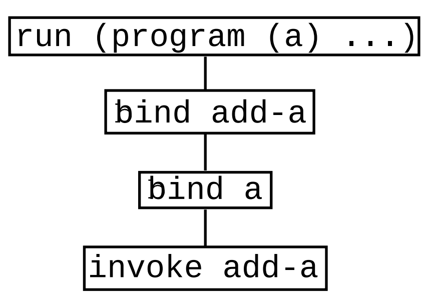
These correspondences are not coincidental, but by design. Since static scoping is defined by the contour diagrams, the environment model must somehow encode the nesting of contours. The environment component of closures is the mechanism by which this correspondence is achieved. The environment component 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.

4.4 Static Scoping: Implementation

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

4.5 Dynamic Scoping: 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:



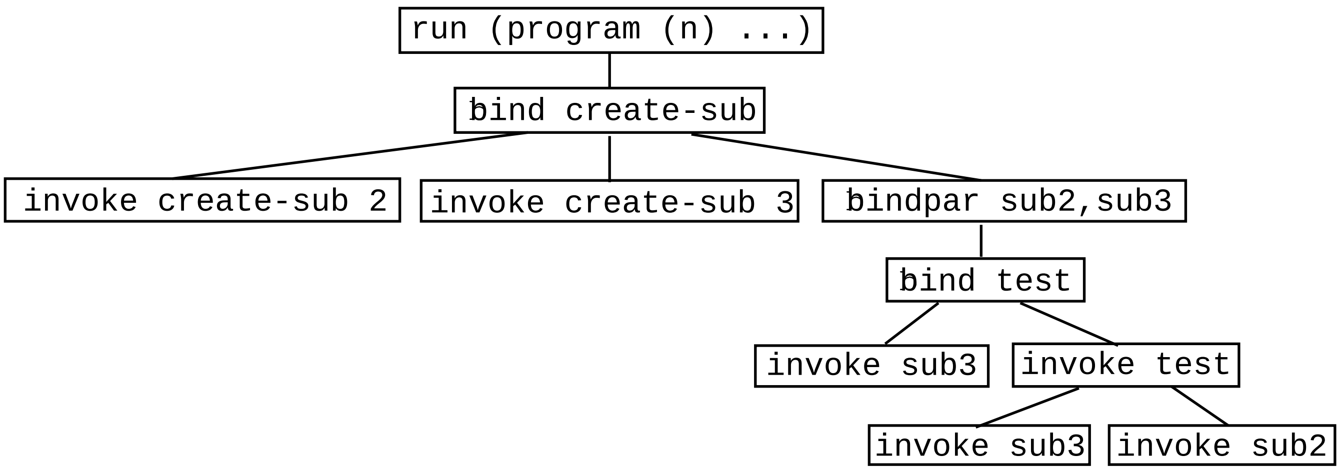
```
;; Implementation of ENV-EVAL using static scope
(define env-eval
  (lambda (exp env)
              .
              .
              .
    ;; Clause corresponding to rule 1
    ((abs? exp)
     (make-closure exp env)) ;; Remember environment of creation

    ;; 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 1: Essence of static scoping.

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



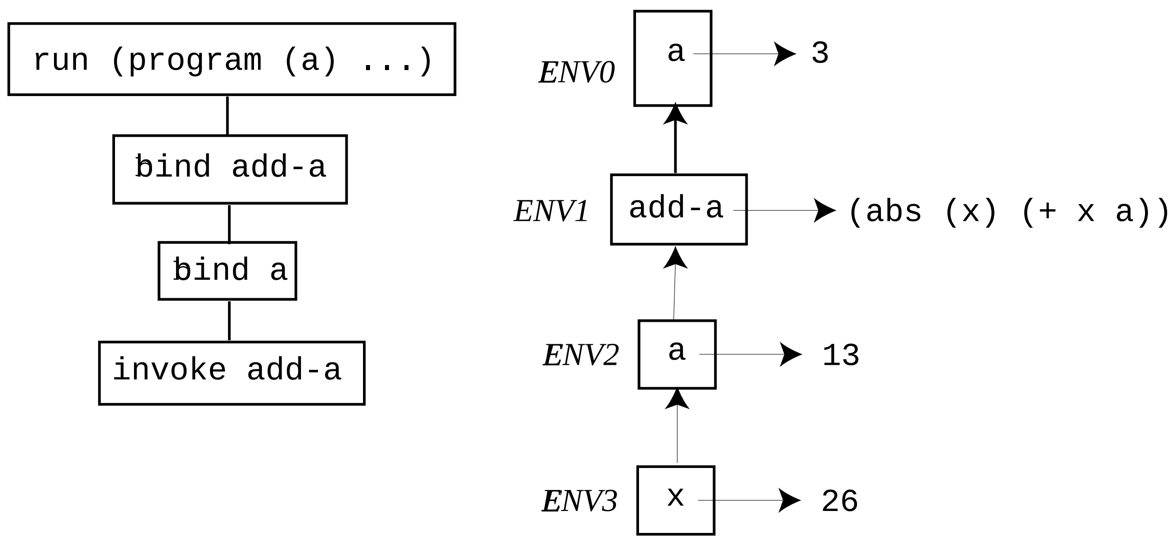
Note: in some cases (but not the above two), the shape of the invocation tree may depend on the values of the arguments at certain nodes, which in turn depends on the scoping mechanism. So the invocation tree cannot in general be drawn without fleshing out the details of the scoping mechanism.

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 2 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.

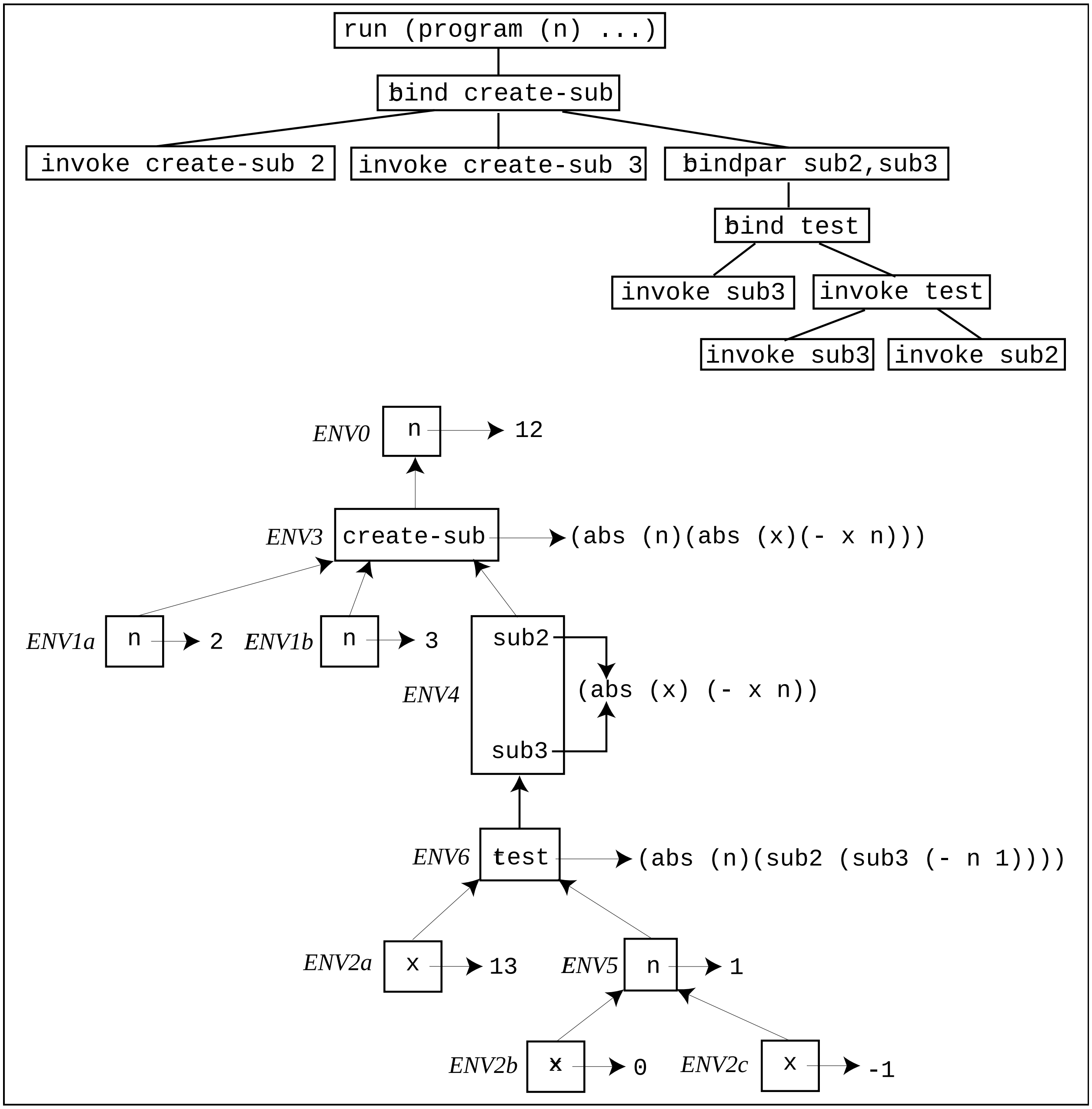


Figure 2: Invocation tree and environment diagram for the `create-sub` program run on 12.

4.6 Dynamic Scoping: Implementation

The two rules of the dynamic scoping mechanism are easy to encode in the environment model. The implementation is shown in Figure 1. 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.

```
;; 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)
    (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 3: Essence of dynamic scoping.

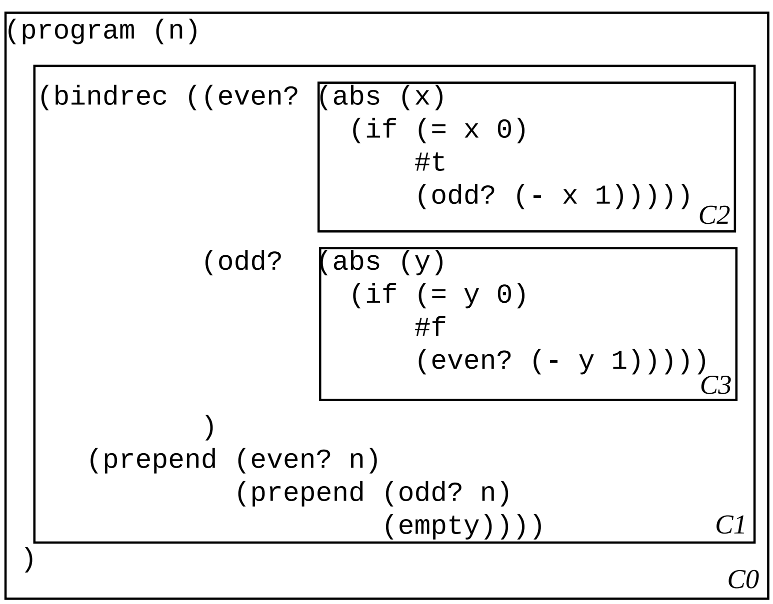
5 Recursive Bindings

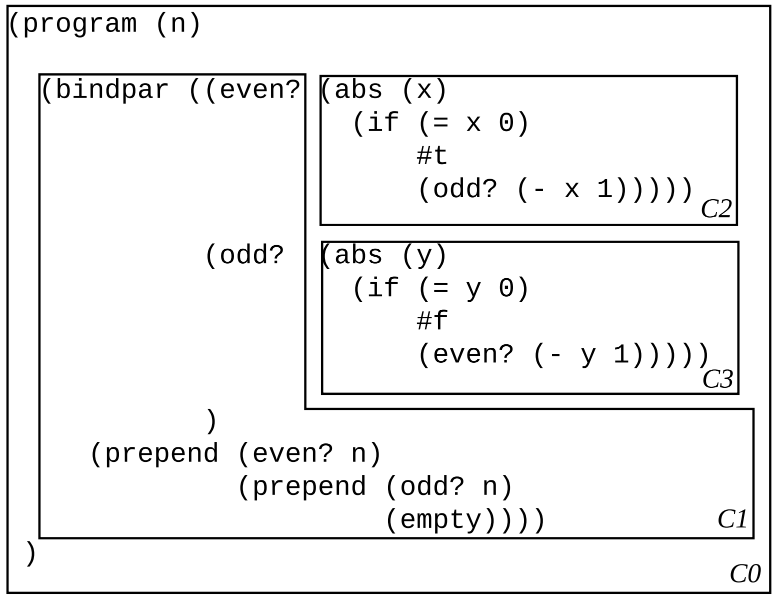
5.1 The `bindrec` Construct

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

```
(program (n)
  (bindrec ((even? (abs (n)
                     (if (= n 0)
                         true
                         (odd? (- n 1)))))
            (odd? (abs (n)
                     (if (= n 0)
                         false
                         (even? (- n 1)))))
            )
     (prepend (even? 5)
              (prepend (odd? 5)
                       (empty)))))
```

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 below:





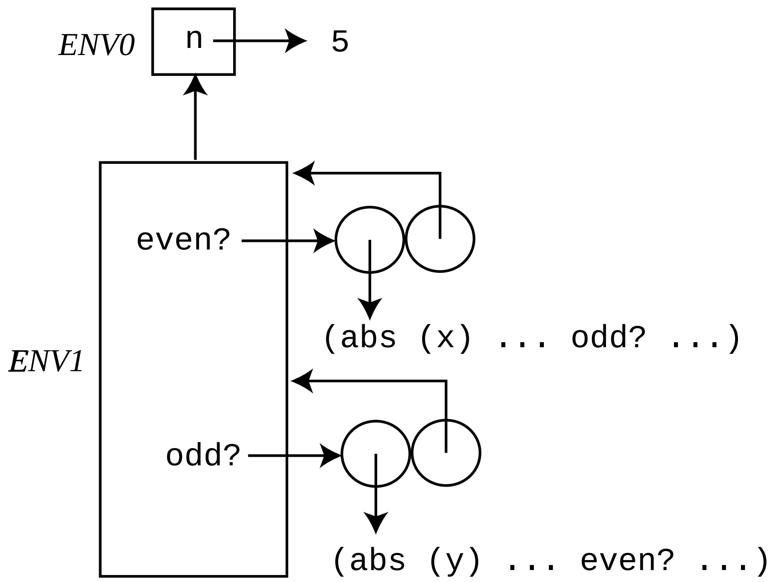
In the `bindrec` expresion, the reference occurrence of `odd?` within the `even?` abstraction has the binding name `odd?` as its binding occurrence; the case is similar for `even?`. However, when `bindrec` is changed to `bindpar` in this program, the names `odd?` and `even?` within the definitions become 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.

5.2 Evaluating `bindrec`

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 was called on the input 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`, we're asking to use the value `x` before we've determined it.

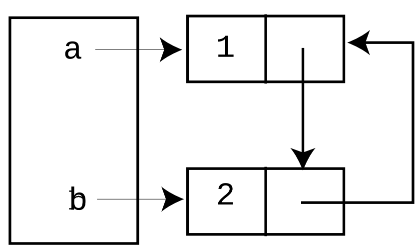
```
(bindrec ((x (+ x 1)))
  (* x 2))
```

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 (prepend 1 b))
          (b (prepend 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), the above definitions are very sensible, and stand for the following list structure:



However, call-by-value (a.k.a. strict or eager) languages (such as HOFL, Scheme, ML, 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 `(cons 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).

```
(program (n)
  (bindpar ((pair (abs (a b) (prepend a (prepend b (empty)))))
            (fst (abs (pair) (head pair)))
            (snd (abs (pair) (head (tail pair)))))
    (bindrec ((one-two (pair 1 (abs () two-one)))
              (two-one (pair 2 (abs () one-two)))
              (prefix (abs (num stream)
                        (if (= num 0)
                            (empty)
                            (prepend (fst stream)
                                     (prefix (- num 1)
                                             ((snd stream))))))
               )
      (prefix n one-two)))))
```

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

5.3 Implementing `bindrec`

Implementing the "knot-tying" aspect of the recursive bindings of bindrec within the `env-eval` function of the statically scoped HOFL interpreter proves to be rather tricky. We will consider a sequence of incorrect definitions for the `bindrec` clause on the path to developing some correct ones.

Here is a first attempt:

```
;; Broken Attempt 1
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (env-extend
     (bindrec-names exp)
     (map (lambda (defn)
            (env-eval defn ???))
          (bindrec-defns exp))
     env)))
```

There is a problem here: what should the environment *???* be? It shouldn't be `env` but the new environment that results from extending `env` with the recursive bindings. But the new environment has no name in the above clause.

A second attempt uses Scheme's `let` to name the result of `env-extend`:

```
;; Broken Attempt 2
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (let ((new-env (env-extend
                    (bindrec-names exp)
                    (map (lambda (defn)
                           (env-eval defn new-env))
                         (bindrec-defns exp))
                    env)))
     new-env)))
```

This attempt fails because, by the scoping rules of `let`, `new-env` is an unbound variable in `(env-extend ...)`.

A third attempt replaces `let` with `letrec`:

```
;; Broken Attempt 3
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (letrec ((new-env (env-extend
                       (bindrec-names exp)
                       (map (lambda (defn)
                              (env-eval defn new-env))
                            (bindrec-defns exp))
                       env)))
     new-env)))
```

The above clause attempts to use the knot-tying ability of Scheme's own recursive binding construct, `letrec`, to implement HOFL's recursive binding construct. Now the `new-env` within `(env-extend ...)` is indeed correctly scoped. Unfortunately, there is still a problem: because Scheme is a call-by-value language, we come face to face with the same sort of problem encountered in the recursive list example from above. That is, occurrence of `env-eval` within the `map` invocation requires that all its arguments be values before it is invoked. But its `new-env` argument is defined to be the result of a computation that depends on the result returned by this occurrence of `env-eval`. This leads to an irresolvable set of constraints: `env-eval` must return before it can be invoked!

We can fix the problem in the same way we fixed the recursive list problem: by using thunks to delay evaluation of the recursive bound variable. In particular, rather than storing the result of evaluating the definition in the environment, we can store in the environment a thunk for evaluating the definition:

```
;; Working Attempt 4
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (letrec ((new-env (env-extend
                       (bindrec-names exp)
                       (map (lambda (defn)
                              (lambda () ;; Introduce a thunk!
                                (env-eval defn new-env)))
                            (bindrec-defns exp))
                       env)))
      new-env)))
```

Once we do this, we must ensure (1) that *all* entities stored in the environments used by `env-eval` are thunks and (2) that whenever a thunk is looked up in the environment, it should be "dethunked" - i.e., applied to zero arguments to retrieve its value. This makes sense if you think in terms of types. Point (1) says that the type of environments is changing from (variable → value) to (variable → (unit → value)), where unit is the type of one element. Point (2) says that since the result of an environment lookup is now of type (unit → value) , it must be applied to zero arguments in order to get a value.

To implement point (1) we introduce the following auxiliary function:

```
(define map-delay
  (lambda (lst)
    (map (lambda (x) (lambda () x)))
         lst)))
```

We use `map-delay` within `env-run` and `funapply` as shown below:

```
(define env-run
  (lambda (pgm ints)
    (env-eval (desugar (program-body pgm))
              (env-extend (program-formals pgm)
                          (map-delay ints)
                          (env-empty)))))

(define funapply
  (lambda (closure actuals)
    ;; Error checking omitted
    (env-eval (closure-body closure)
              (env-extend (closure-formals closure)
                          (map-delay actuals)
                          (closure-env closure)))))
```

Point (2) is implemented by changing the variable reference clause of `env-eval` as follows:

```
((varref? exp)
 (let ((probe (env-lookup (varref-name exp) env)))
   (if (unbound? probe)
       (throw 'unbound-variable (varref-name exp))
       (dethunk probe)))) ; Force delayed computation
```

The `dethunk` function “forces” a thunk by applying it to zero arguments:

```
(define dethunk (lambda (thunk) (thunk)))
```

The above changes work fine, but they are inefficient. In particular, they require that a recursive definition expression be evaluated every time it is looked up in the environment.

It turns out that Scheme has a more efficient mechanism for delaying computations than thunks. The construct `(delay` E`)` delays the expression of E by returning a **promise**; if E_{prom} is an expression denoting a promise, its delayed computation can be forced via the application `(force` E_{prom}`)`. The promise "remembers" the value of its computation, so an attempt to force a promise the second time performs no computation but returns the previously computed value.

By replacing all instance of thunks and dethunking by delay and force in the code presented above, a more efficient implementation of recursive binding can be achieved. The highlights are shown in Figure 4.

```
(define env-run
  (lambda (pgm args)
    (env-eval (desugar (program-body pgm))
              (env-extend (program-formals pgm)
                          (map-delay args)
                          (env-empty)))))

(define env-eval
  (lambda (exp env)
    (cond             .
                      .
                      .
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe)
                 (throw 'unbound-variable (varref-name exp))
                 (force probe)))) ; *** PROBE is a promise; force it.
                      .
                      .
                      .
           ((bindrec? exp)
            (env-eval
              (bindrec-body exp)
              (letrec ((new-env (env-extend
                                  (bindrec-names exp)
                                  (map (lambda (defn)
                                         ; *** Evaluation of defns must be delayed
                                         (delay (env-eval defn new-env)))
                                       (bindrec-defns exp))
                                  env)))
                 new-env)))
                      .
                      .
                      .

            )))

(define funapply
  (lambda (closure actuals dyn-env)
    ;; Error checking omitted
    (env-eval (closure-body closure)
              (env-extend (closure-params closure)
                          (map-delay actuals) ; *** Bindings must be delayed
                          (closure-env closure)))))

(define map-delay
  (lambda (values)
    (map (lambda (val) (delay val)) values)))
```

Figure 4: Implementing `bindrec` with `delay` and `force`.