Scoping in HOFL

- *April 10* Added a new section on the handling of `bindrec` in the substitution model and made several minor edits.
- *April 14* (1) added a new section discussing dyanmic vs. static scope; (2) fixed bugs in version 4 of the `bindrec` evaluation rule in the environment model;

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** (Sec. 1) and **dynamic scoping** (Sec. 2) . 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 Sec. 3).

1 Static Scoping

1.1 Contour Model

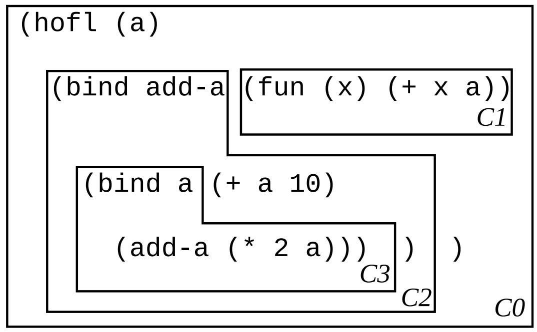
In **static scoping**, the meaning of every variable reference is determined by the lexical contour boxes introduced in Handout #30 on BINDEX. To determine the binding occurrence of any reference occurrence of a name, find the innermost contour enclosing the reference occurrence that binds the name. 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,

[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.

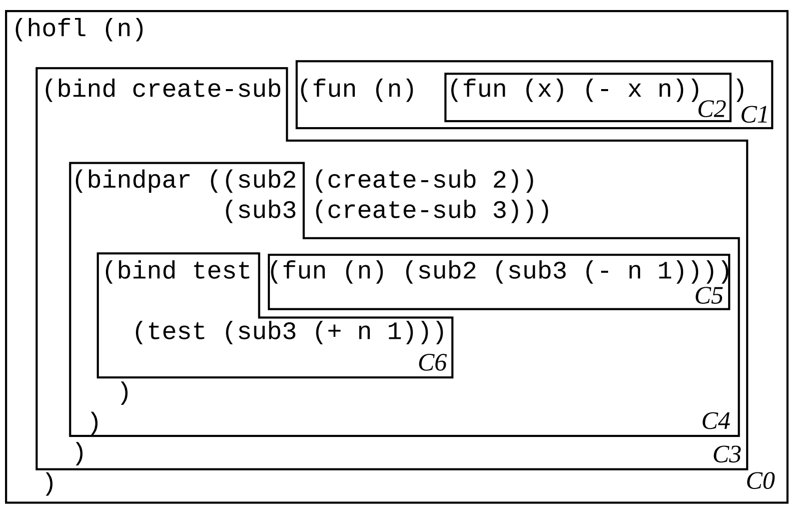
so the `a` in `(+ x a)` refers to the `a` bound by `(hofl (a) ... )`. Similarly, it can be determined that:

- 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 ... )`.



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) ... )`.

1.2 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 can formalize the HOFL substitution model by defining a substitution model evaluator in OCAML. Fig. 1 presents the abstract syntax and values used by the evaluator as well as the definition of substitution. The evaluator itself is presented in Fig. 2. The third component of a `Fun` value, an environment, is not used in the substitution model but plays a very important role in the environment model. The omitted `bindrec` case will be explained later.

1.3 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 Fig. 3 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:

- 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.

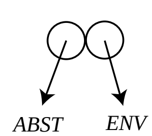
Most evaluation rules of the environment model are independent of the scoping mechanism. Such rules are shown in Fig. 4.

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

```
type var = string

type pgm = Pgm of var list * exp (* param names, body *)

and exp =
    Lit of valu (* integer, boolean, character, string, 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 *)
  | Abs of var * exp (* function abstraction *)
  | App of exp * exp (* function application *)
  | Bindrec of var list * exp list * exp (* recursive bindings *)

and valu =
    Int of int
  | Bool of bool
  | Char of char
  | String of string
  | Symbol of string
  | List of valu list
  | Fun of var * exp * valu Env.env (* formal, body, and environment *)
    (* the environment component is ignored in the substitution-model interpreter,
       but plays an important role in the environment-model interpreter. *)

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

let primopName (Primop(name,_)) = name
let primopFunction (Primop(_,fcn)) = fcn

(* val subst : exp -> exp Env.env -> exp *)
let rec subst exp env =
  match exp with
    Lit i -> exp
  | Var v -> (match Env.lookup v env with Some e -> e | None -> exp)
  | PrimApp(op,rands) -> PrimApp(op, map (flip subst env) rands)
  | If(tst,thn,els) -> If(subst tst env, subst thn env, subst els env)
  | Abs(fml,body) ->
      let fml' = fresh fml in Abs(fml', subst (rename1 fml fml' body) env)
  | App(rator,rand) -> App(subst rator env, subst rand env)
  | Bindrec(names,defns,body) ->
      let names' = map fresh names in
        Bindrec(names', map (flip subst env) (map (renameAll names names') defns),
                subst (renameAll names names' body) env)

(* val subst1 : exp -> var -> exp -> exp *)
and subst1 newexp name exp = subst exp (Env.make [name] [newexp])

(* val substAll: exp list -> var list -> exp -> exp *)
and substAll newexps names exp = subst exp (Env.make names newexps)

(* val rename1 : var -> var -> exp -> exp *)
and rename1 oldname newname exp = subst1 (Var newname) oldname exp

(* val renameAll : var list -> var list -> exp -> exp *)
and renameAll olds news exp = substAll (map (fun s -> Var s) news) olds exp
```

Figure 1: OCAML data types for the abstract syntax of HOFL.

```
(* val run : Hofl.pgm -> int list -> valu *)
let rec run (Pgm(fmls,body)) ints =
  let flen = length fmls
  and ilen = length ints
  in
    if flen = ilen then
      eval (substAll (map (fun i -> Lit (Int i)) ints) fmls body)
    else
      raise (EvalError ("Program expected " ^ (string_of_int flen)
                        ^ " arguments but got " ^ (string_of_int ilen)))

(* val eval : Hofl.exp -> valu *)
and eval exp =
  match exp with
    Lit v -> v
  | Var name -> raise (EvalError("Unbound variable: " ^ name))
  | PrimApp(op, rands) -> (primopFunction op) (map eval rands)
  | If(tst,thn,els) ->
     (match eval tst with
       Bool b -> if b then eval thn else eval els
     | v -> raise (EvalError ("Non-boolean test value "
                              ^ (valuToString v)
                              ^ " in if expression"))
     )
  | Abs(fml,body) -> Fun(fml,body,Env.empty) (* No env needed in subst. model *)
  | App(rator,rand) -> apply (eval rator) (eval rand)
  | Bindrec(names,defns,body) -> ... see discussion of bindrec ...

and apply fcn arg =
  match fcn with
    Fun(fml,body,_) -> eval (subst1 (Lit arg) fml body)
                       (* Lit converts any argument valu (including lists & functions)
                          into a literal for purposes of substitution *)
  | _ -> raise (EvalError ("Non-function rator in application: "
                            ^ (valuToString fcn)))
```

Figure 2: Substitution model evaluator in HOFL.

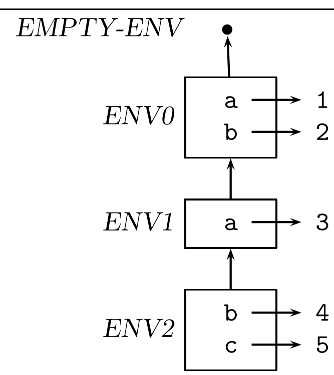


Figure 3: An example chain of environment frames.

Program Running Rule

- To run a Hofl program `(hofl (`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, 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.

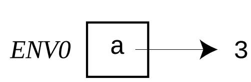
Although `bind`, `bindrec`, and `bindseq` can all be "desugared away", it is convenient to imagine that there are rules for evaluating these constructs directly:

- Evaluating `(bind` I_{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 I_{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.

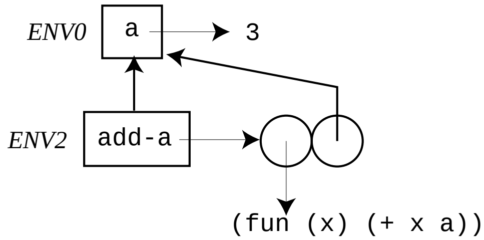
Figure 4: Environment model evaluation rules that are independent of the scoping mechanism.

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.

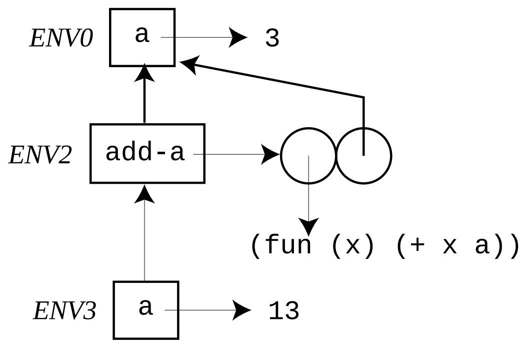
We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



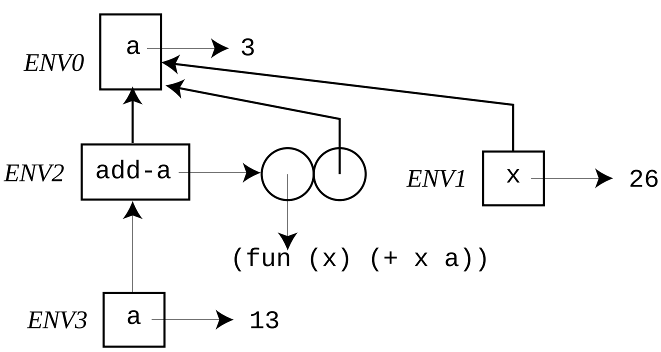
To evaluate the `(bind add-a ... )` expression, we first evaluate `(fun (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



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:

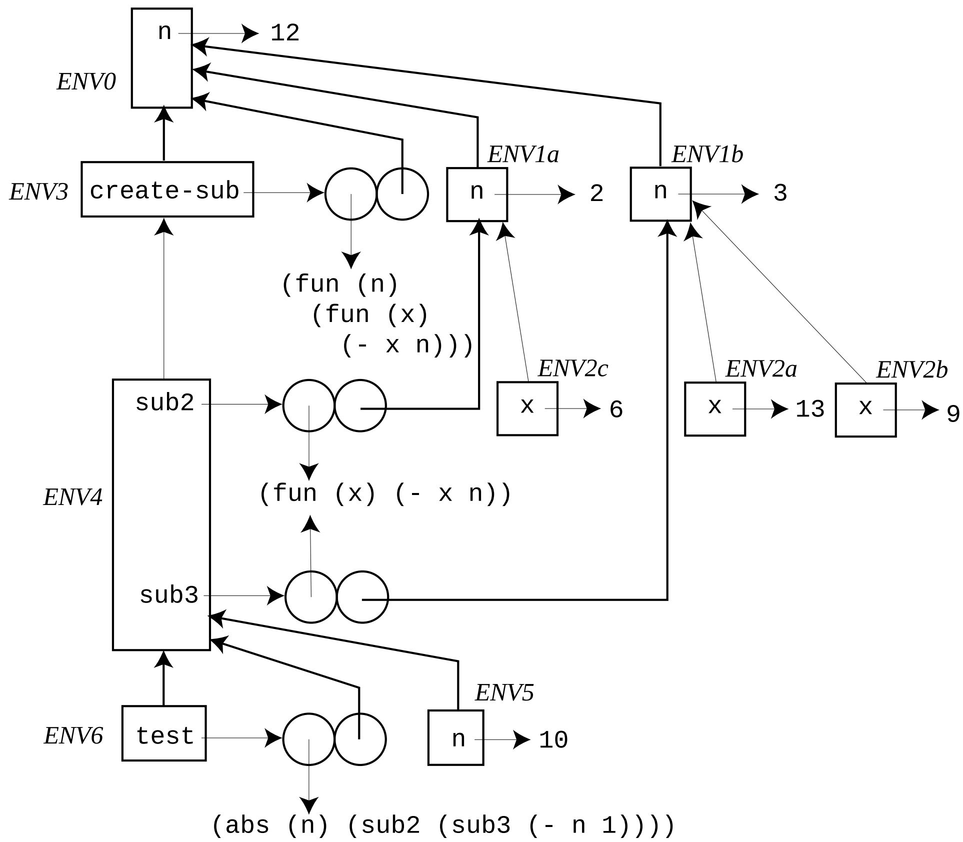


Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3. First, the subexpressions `add-a` and `(* 2 a)` must be evaluated in ENV_3; these evaluations yield the add-a closure and 26, respectively. Next, the closure is applied to 26. This creates a new frame ENV_1 binding `x` to 26; by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

1.4 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 5. 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.

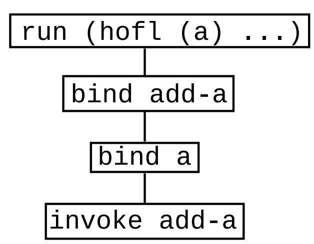
```
(* val eval : Hofl.exp -> valu Env.env -> valu *)
and eval exp env =
  match exp with
      ⋮
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> apply (eval rator env) (eval rand env)
      ⋮

and apply fcn arg =
  match fcn with
    Fun(fml,body,senv) -> eval body (Env.bind fml arg senv) (* extend static env *)
  | _ -> raise (EvalError ("Non-function rator in application: " ^ (valuToString fcn)))
```

Figure 5: Essence of static scoping in HOFL.

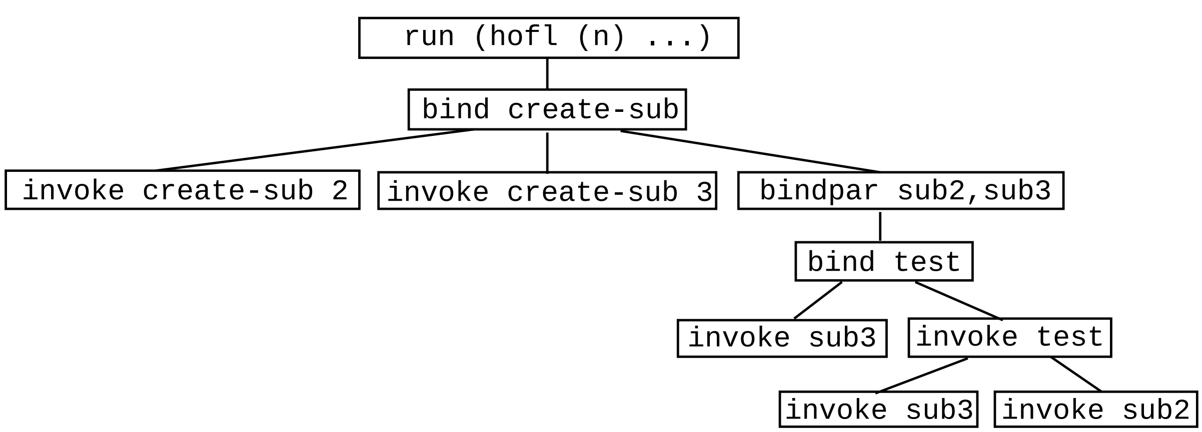
2 Dynamic Scoping

2.1 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:



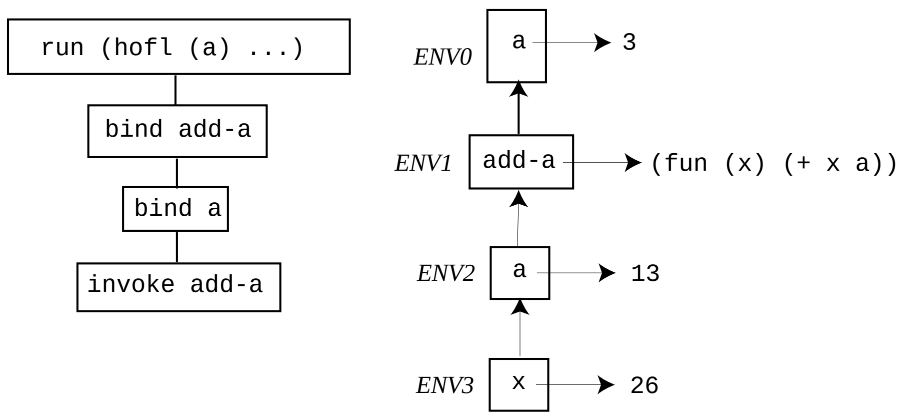
Note: in some cases (but not the above two), the shape of the invocation tree may depend on the values of the arguments at certain nodes, which in turn depends on the scoping mechanism. So the invocation tree cannot in general be drawn without fleshing out the details of the scoping mechanism.

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 6 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.

2.2 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 5. 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.

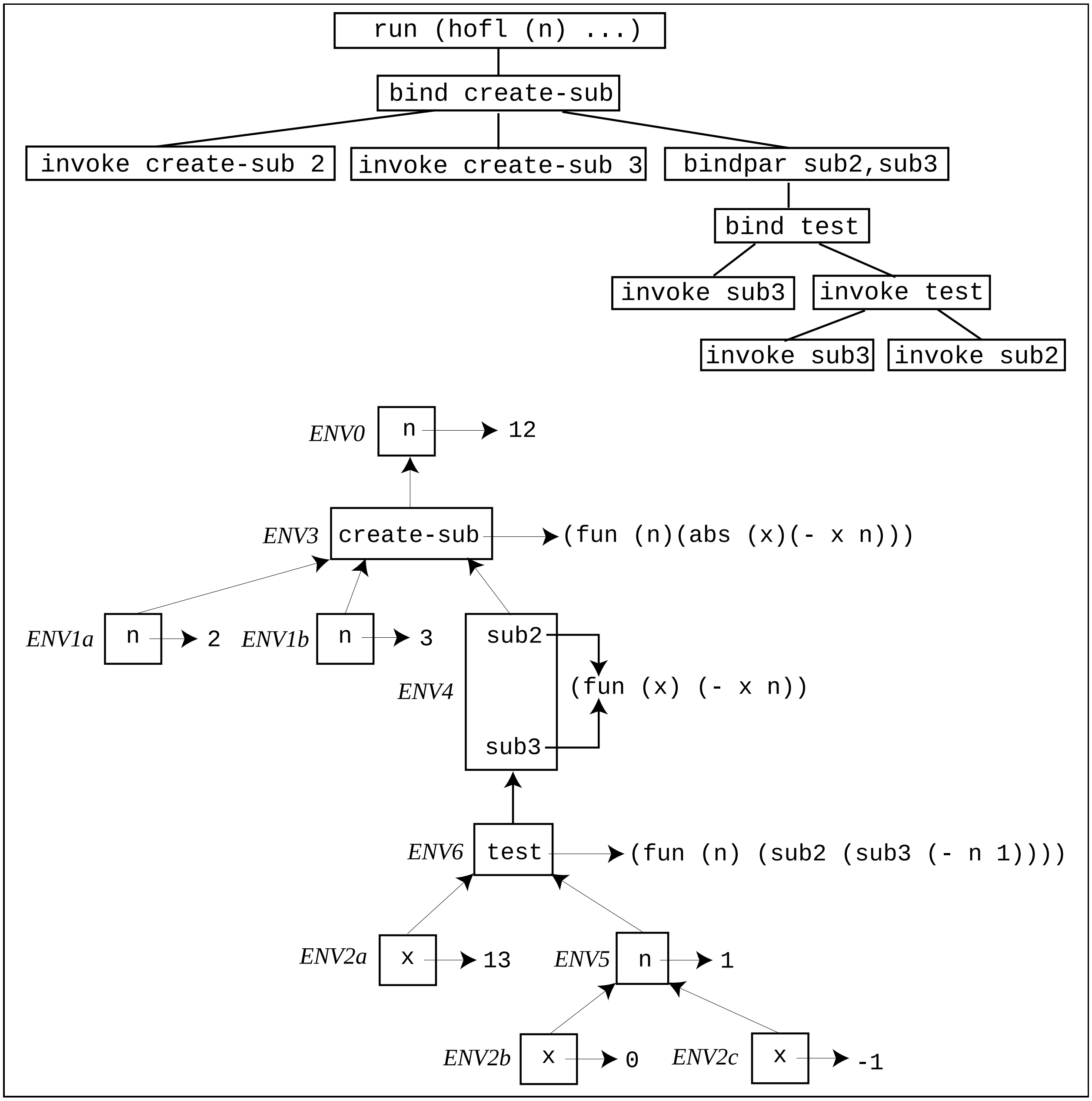


Figure 6: Invocation tree and environment diagram for the `create-sub` program run on 12.

2.3 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

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

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

Figure 7: Essence of dynamic scoping in HOFL.

language, the particular names chosen for variables in a function do not affect its behavior, so 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 function "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).

3 Recursive Bindings

3.1 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 Fig. 8. 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.

3.2 Substitution-Model Evaluation of bindrec

The evaluation rule for bindrec in the substitution-model interpreter is shown in Fig. 9. Each recursive definition in a bindrec is replaced by a copy of the definition in which each reference to a bindrec-bound name is replaced by an expression that wraps the name in a new bindrec with the same bindings. This has the effect of propagating the recursive nature of the bindrec to each reference to a bindrec-bound name.

To see how this works in practice, consider the following version of the even?/odd? program from above that has a simpler body:

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

Suppose we introduce the abbreviation E for the abstraction

```
(abs x
  (if (= x 0)
      #t
      (odd? (- x 1))))
```

and the abbrevation O for the abstraction

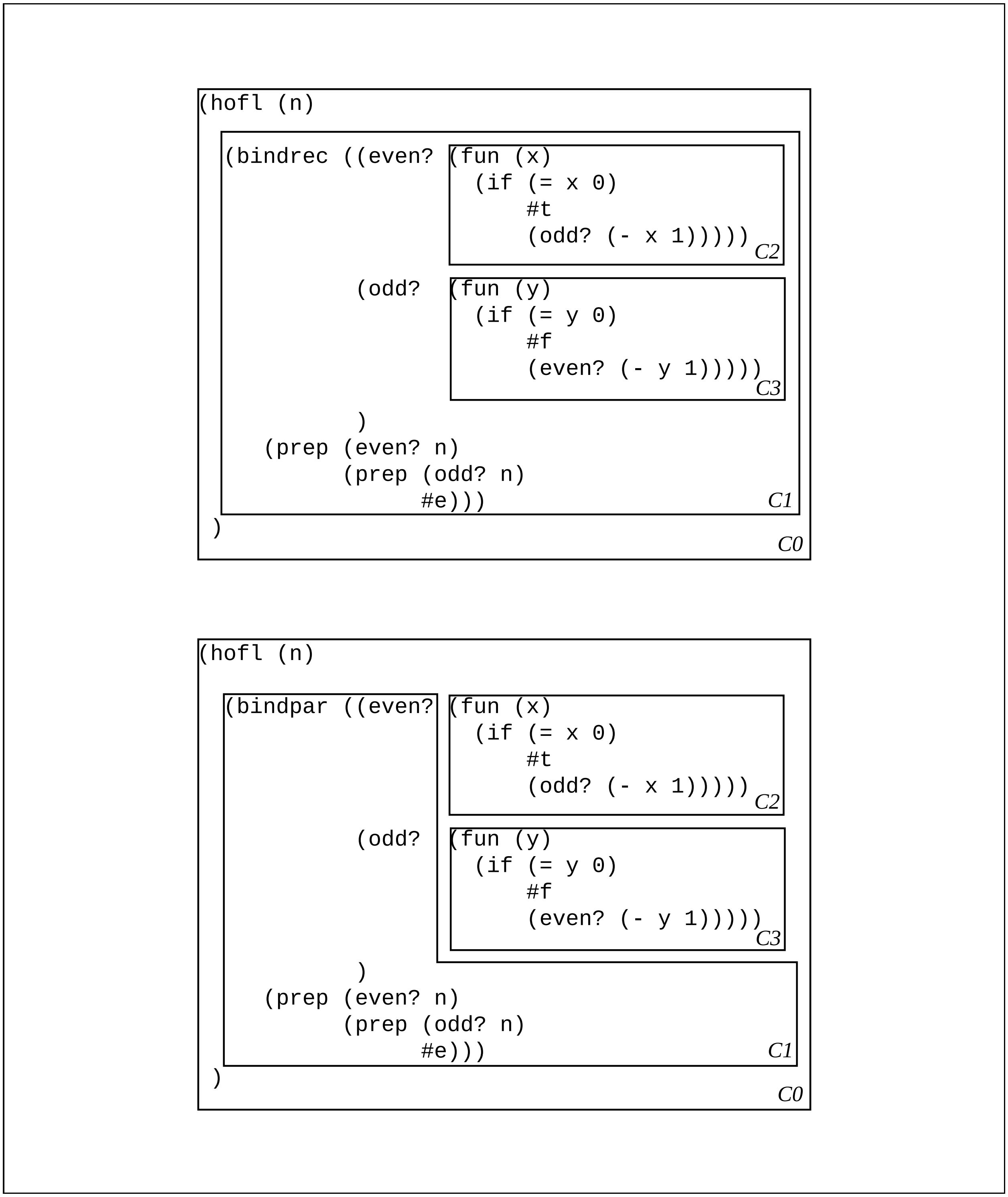


Figure 8: Lexical contours for versions of the `even?`/`odd?` program using `bindrec` and `bindpar`.

```
and eval exp =
  match exp with
      ⋮
  | Bindrec(names,defns,body) ->
      eval (substAll (map (fun defn -> Bindrec(names,defns,defn))
                          defns)
                     names
                     body)
```

Figure 9: Evaluation of bindrec in the substitution-model interpreter.

```
(abs y
  (if (= y 0)
      #f
      (even? (- y 1))))
```

Then Fig. 10 shows the substitution model evaluation of the program on the input 3. Note how the substitution that wraps each `bindrec`-bound name in a fresh `bindrec` allows the abstractions for the recursive functions to be unwound one level at a time, giving rise to the desired behavior for the recursive functions.

```
(hofl (n) (bindrec ((even? E) (odd? O)) (even? n))) run on [3]
; Here and below, assume a ''smart'' substitution that
; performs renaming only when variable capture is possible.

⇒ (bindrec ((even? E) (odd? O)) (even? 2))
⇒ ((abs (x) (if (= x 0) #t ((bindrec ((even? E) (odd? O)) odd?) (- x 1)))) 3)
⇒ (if (= 3 0) #t ((bindrec ((even? E) (odd? O)) odd?) (- 3 1)))
⇒ (if #f #t ((bindrec ((even? E) (odd? O)) odd?) (- 3 1)))

⇒ ((bindrec ((even? E) (odd? O)) odd?) (- 3 1))
⇒ ((abs (y) (if (= y 0) #f ((bindrec ((even? E) (odd? O)) even?) (- y 1)))) 2)
⇒ (if (= 2 0) #f ((bindrec ((even? E) (odd? O)) even?) (- 2 1)))
⇒ (if #f #f ((bindrec ((even? E) (odd? O)) even?) (- 2 1)))

⇒ ((bindrec ((even? E) (odd? O)) even?) (- 2 1))
⇒ ((abs (x) (if (= x 0) #t ((bindrec ((even? E) (odd? O)) odd?) (- x 1)))) 1)
⇒ (if (= 1 0) #t ((bindrec ((even? E) (odd? O)) odd?) (- 1 1)))
⇒ (if #f #t ((bindrec ((even? E) (odd? O)) odd?) (- 1 1)))

⇒ ((bindrec ((even? E) (odd? O)) odd?) (- 1 1))
⇒ ((abs (y) (if (= y 0) #f ((bindrec ((even? E) (odd? O)) even?) (- y 1)))) 0)
⇒ (if (= 0 0) #f ((bindrec ((even? E) (odd? O)) even?) (- 0 1)))
⇒ (if #t #f ((bindrec ((even? E) (odd? O)) even?) (- 0 1)))
⇒ #f
```

Figure 10: Example evaluation involving bindrec in the substitution-model interpreter.

3.3 Environment-Model Evaluation of `bindrec`

3.3.1 High-level Model

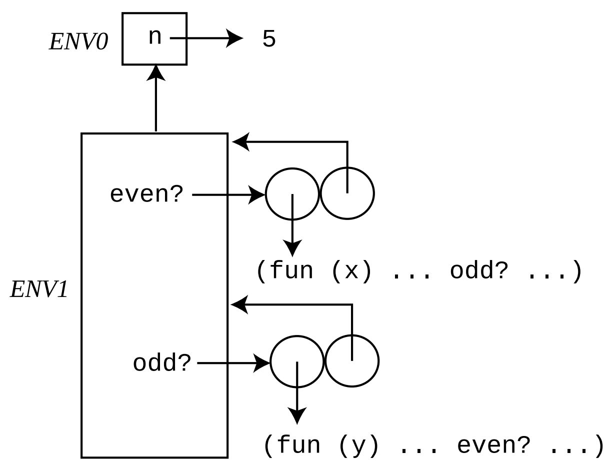
How is `bindrec` handled in the environment model? We do it in three stages:

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:

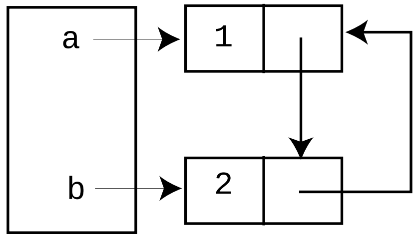
```
(bindrec ((x (+ x 1)))
  (* x 2))
```

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)`.

3.3.2 Implementing `bindrec`

Implementing the "knot-tying" aspect of the recursive bindings of bindrec within the `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(names,defns,body) ->
  eval body
       (Env.bindAll names
                    (map (fun defn -> eval defn ???)
                         defns)
                    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 Ocaml's `let` to name the result of `Env.bindAll`:

```
(* Broken Attempt 2 *)
| Bindrec(names,defns,body) ->
  eval body
       let newEnv = Env.bindAll names
                                (map (fun defn -> eval defn newEnv)
                                     defns)
                                env
       in newEnv
```

This attempt fails because, by the scoping rules of OCAML's `let` construct, `newEnv` is an unbound variable in the expression `Env.bindAll ...` .

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

```
(* Broken Attempt 3 *)
| Bindrec(names,defns,body) ->
  eval body
       let rec newEnv =
         Env.bindAll names
                     (map (fun defn -> eval defn newEnv)
                          defns)
                     env
       in newEnv
```

The above clause attempts to use the knot-tying ability of OCAML's own recursive binding construct, `let rec`, to implement HOFL's recursive binding construct. Now the `newEnv` within `Env.bindAll ...` is indeed correctly scoped. Unfortunately, there are still two problems:

1. *The* `let rec` *Syntax Restriction*: The OCAML `let rec` construct can only be used to define recursive functions that are specified by manifest abstractions. It cannot be used to define more general recursive values (such as recursive defined lists in the `one-two` example above).

2. *The Call-by-Value Knot-Tying Problem*: Even if OCAML *did* allow more general recursive values to be defined via `let rec`, because OCAML is a call-by-value language, we would still come face to face with the same sort of problem encountered in the recursive list example from above. That is, the `eval` within the functional argument to `map` requires that all its arguments be evaluated to values before it is invoked. But its `newEnv` argument is defined to be the result of a computation that depends on the result returned by invocations of this occurrence of `eval`. This leads to an irresolvable set of constraints: `eval` must return before it can be invoked!

We can fix the call-by-value knot-tying problem in the same way we fixed the recursive list problem above: 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:

```
(* Broken Attempt 4 *)
| Bindrec(names,defns,body) ->
  eval body
       let rec newEnv =
         Env.bindAll names
                     (map (fun defn -> (fun () -> eval defn newEnv))
                          defns)
                     env
       in newEnv
```

This evaluation rule for `bindrec` is still broken because of the syntactic restrictions on `let rec`.[2]

[2]In call-by-value languages supporting more flexible recursive binding constructs, such as Scheme's `letrec`, the

We'll see below that we can fix this problem as well. Once we do, we must also change the rest of the interpreter to ensure (1) that *all* entities stored in the environments used by `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 effectively changed from `var -> valu` to `var -> unit -> value`, where `unit` is the type of `()`. Point (2) says that since the result of an environment lookup is now a function of type `unit -> valu`, it must be applied to zero arguments in order to get a value. Fig. 11 shows how these changes can be made in the HOFL environment-model interpreter.

```
(* 1. The definition of closure values must change, since they
      have an environment component: *)
and valu =
      ⋮
  | Fun of var * exp * (unit -> valu) Env.env (* used to be valu Env.env *)

(* 2. The initial environment for evaluating the program body must be modified
      to thunk the argument integers. *)
let rec run (Pgm(fmls,body)) ints =
               ⋮
      eval body (Env.make fmls (map (fun i -> fun () -> Int i)
                                    (* used to be (fun i -> Int i) *)
                                    ints))

(* 3. In function application, the new environment frame must use thunks: *)
and apply fcn arg =
  match fcn with
    Fun(fml,body,env) ->
      eval body (Env.bind fml (fun () -> arg) (* used to be arg *) env)
  | _ -> raise (EvalError ("Non-function rator in application: " ^ (valuToString fcn)))

(* 4. When a variable reference is evaluated, it must dethunk the found thunk: *)
and eval exp env =
  match exp with
      ⋮
  | Var name ->
      (match Env.lookup name env with
         Some(thunk) -> thunk () (* dethunk the found thunk *)
       | None -> raise (EvalError("Unbound variable: " ^ name)))
      ⋮
```

Figure 11: Changing the HOFL environment-model interpreter so that environments that bind names to thunks rather than values.

To get around OCAML's syntactic restrictions on `let rec`, we (1) need a way to convert between environments and functions and (2) need to change the `newEnv` definition to be a function definition. The `ENV` signature in Fig. 12 shows an environment signature that includes functions (`fromFun` and `toFun`) for converting between lookup functions and environments. These functions would be challenging to implement for list-based environment representations, but are trivial in the function-based environment representation used in the `Env` structure in Fig. 12. Using these conversion functions, we define the following working version of the `bindrec` evaluation rule:

fourth attempt at the `bindrec` evaluation rule works without further modification.

```
(* Working (but complex) Attempt 5 that:
    (1) thunks environment values to perform call-by-value knot-tying and
    (2) converts between environments and lookup functions to
        get around OCAML's syntactic restrictions on LET REC. *)
| Bindrec(names,defns,body) ->
    eval body
      (let rec newEnvFcn =
                 fun name ->
                   (Env.toFun
                     (Env.bindAll names
                                   (map (fun defn ->
                                          fun () ->
                                            eval defn (Env.fromFun newEnvFcn)))
                                        defns)
                                  env)
                   name
           in Env.fromFun newEnvFcn)
```

This version defines a recursive lookup function `newEnvFcn` that takes a variable `name` and returns the result of looking it up in the recursive environment. The expression `Env.fromFun newEnvFcn` is used in the two spots where an environment is required. The enviroment resulting from `Env.bindAll` must be converted to a lookup function by `Env.toFun` so that it can be applied to the `name` argument. Conceptually, `fun name ->` E_{fcn} `name` is equivalent to E_{fcn} for any function-denoting expression E_{fcn}, but OCAML's `let rec` will not allow an arbitrary function-denoting expression for a definition; every such definition *must* be an abstraction.

Although the above approach "works", its technical acrobatics make it less elegant and less efficient than we'd like. A solution that is both more elegant and more efficient is possible if we utilize some other features of the environment signature and implementation in Fig. 12. This environment supports operations for binding both regular values (`bind` and `bindAll`) as well as thunked values (`bindThunk` and `bindAllThunks`) in such a way that the regular values may be looked up efficiently without any dethunking. Furthermore, it supports a abstract fixed point operator, `fix`, that internally uses OCAML's `let rec` construct to find the fixed point. This is possible because the implementation uses functions to represent environments, and `let rec` can be used to define recursive functions.

Using the extended environment module, we can implement `bindrec` as follows:

```
(* Final Working Version *)
| Bindrec(names,defns,body) ->
    eval body
         (Env.fix (fun e ->
                    (Env.bindAllThunks names
                                      (map (fun defn ->
                                             (fun () -> eval defn e))
                                           defns)
                                      env)))
```

This version does *not* require any other changes to the HOFL interpreter. For example, we would not make the changes in Fig. 11 with this version of the `bindrec` rule.

3.4 Fixed Points and the Y Operator

With all the complexity surrounding the implementation of `bindrec` in HOFL, it may be surprising that it is possible to define recursive functions in HOFL without `bindrec`! While `bindrec` is convenient for defining recursive functions, we will now see that it is not necessary. Higher-order functions have the power to express recursive computational processes by themselves.

```
module type ENV = sig
  type 'a env
  val empty: 'a env
  val bind : string -> 'a -> 'a env -> 'a env
  val bindAll : string list -> 'a list -> 'a env -> 'a env
  val make : string list -> 'a list -> 'a env
  val lookup : string -> 'a env -> 'a option
  val bindThunk : string -> (unit -> 'a) -> 'a env -> 'a env
  val bindAllThunks : string list -> (unit -> 'a) list -> 'a env -> 'a env
  val merge : 'a env -> 'a env -> 'a env
  val fix : ('a env -> 'a env) -> 'a env
  (* for converting between lookup functions and environments *)
  val fromFun : (string -> 'a option) -> 'a env
  val toFun : 'a env -> (string -> 'a option)
end

module Env : ENV = struct

  type 'a env = string -> 'a option

  let empty = fun s -> None

  let bind name valu env =
    fun s -> if s = name then Some valu else env s

  let bindAll names vals env = ListUtils.foldr2 bind env names vals

  let make names vals = bindAll names vals empty

  let lookup name env = env name

  (* val bindThunk : string -> (unit -> 'a) -> 'a env -> 'a env *)
  let bindThunk name valuThunk env =
    fun s -> if s = name then Some (valuThunk ()) else env s

  (* val bindAllThunks : string list -> (unit -> 'a) list -> 'a env -> 'a env *)
  let bindAllThunks names valThunks env =
    ListUtils.foldr2 bindThunk env names valThunks

  let merge env1 env2 =
    fun s -> (match env1 s with
                None -> env2 s
              | some -> some)

  (* val fix : ('a env -> 'a env) -> 'a env *)
  let fix gen = (* assume gen has type ('a env -> 'a env) -> 'a env *)
    (* define a recursive environment envfix, which is a function from
       a string name to a value of type 'a *)
    let rec envfix name = (gen envfix) name
     in envfix

  let fromFun f = f

  let toFun f = f

end
```

Figure 12: An environment signature and an function-based implementation of this signature.

A key step is converting a recursive definition for a function f to a so-called **generating function** g that maps functions to functions and has f as its **fixed point** — that is, $(gf) = f$. For example, here is a generating function (expressed in HOFL) for the factorial function:

```
(def fact-gen (fun (f)
                (fun (n)
                  (if (= n 0)
                      1
                      (* n (f (- n 1)))))))
```

You should convince yourself that the factorial function is the only fixed point of `fact-gen`.

Amazingly, we can define a recursionless function that automatically finds the fixed points of generating functions like `fact-gen`. It is called the **Y operator**. Here is a HOFL definition of the Y operator:

```
(def y (fun (g)
         ((fun (s) (fun (x) ((g (s s)) x)))
          (fun (s) (fun (x) ((g (s s)) x))))))
```

Using `y`, we can define the factorial function as

```
(def fact (y fact-gen))
```

Note that no recursion has been used!

To see how this work, let's use the abbreviation FG for

```
(fun (f)
  (fun (n)
    (if (= n 0)
        1
        (* n (f (- n 1))))))
```

and the abbreviation FS for

```
(fun (s) (fun (x) ((FG (s s)) x)))
```

Note that in the substitution model,

```
(FS FS) ⇒ (fun (x) ((FG (FS FS)) x))
```

Now we're ready to see how `fact` works in an example:

```
((y FG) 3)
⇒ ((FS FS) 3)
⇒ ((fun (x) ((FG (FS FS) x))) 3)
⇒ ((FG (FS FS) 3))
⇒ ((FG (FS FS) 3))
```

We conclude by showing that we can even define mutually recursive functions using `y` by using Church pairs to pair the functions:

```
(def church-pair (fun (x y) (fun (f) (f x y))))

(def church-fst (fun (p) (p (fun (x y) x))))

(def church-snd (fun (p) (p (fun (x y) y))))

(def even-odd-gen (fun (p)
                    (church-pair
                       (fun (x) ; even?
                        (if (= x 0)
                            #t
                            ((church-snd p) (- x 1))))
                      (fun (y) ; odd?
                        (if (= y 0)
                            #f
                            ((church-fst p) (- y 1)))))))

(def even? (church-fst (y even-odd-gen)))

(def odd? (church-snd (y even-odd-gen)))
```