CS251 Programming Languages **Handout # 15”**
Prof. Lyn Turbak **February 22, 2002**
Wellesley College

Local Binding (Revised, Final)

1 BINDEX

Studying the INTEX language is able to give us insight into the fundamental programming processes like evaluation and substitution. But INTEX is missing many important features of real programming languages. Our goal is to explore various programming language dimensions by incrementally extending INTEX with simple versions of various features, and then to explore the design space associated with such features.

The first feature we shall explore is the ability to name the results of intermediate computations. As an example of this, consider a new language, BINDEX that is a version of INTEX extended with a local binding construct of the form `(bind` I_{name} E_{defn} E_{body}`)`. Intuitively, a `bind` expression is evaluated as follows:

1. The definition expression E_{defn} is evaluated to a value V_{defn}.
2. The body expression E_{body} is evaluated in such a way that references to I_{name} denote V_{defn}.
3. The value of E_{body} is returned as the value of the `bind` expression.

Here is a simple example of a BINDEX program using `bind` to calculate the value $\frac{a^2+b^2}{a^2-b^2}$.

```
(program (a b)
  (bind a_sq (* a a)
    (bind b_sq (* b b)
      (bind numer (+ a_sq b_sq)
        (bind denom (- a_sq b_sq)
          (div numer denom))))))
```

Constructs like `bind` in BINDEX, `let` in Scheme, and *`type varname = exp;`* in Java are known as **local binding expressions** or **local variable declarations**. They introduce names for the result of definition expressions that can be used in some part of the program (in the case of `bind`, the body of the `bind` expression). Local binding constructs are used for three main purposes in a program:

1. Naming the result of evaluating a definition expression can avoid recalculating the value of that expression. This can avoid the expense of calculating an expression more than once. For instance, in the above example, naming the result of `(* a a)` means that the multiplication needs to be performed only once; it would need to be performed twice if the result were not named. In the case of a simple expression like `(* a a)`, it is not clear that any time is saved by avoiding recalculation. However, for expensive calculations, the time saved by avoiding recalculation can be considerable. In the context of recursion, naming the result of a recursive call can often change the asymptotic complexity of an algorithm.

2. In programs where values can have a time-varying state (think Java objects), it is essential to name values so that the same value can be referenced more than once. Since it is stateless, this is not important in BINDEX, but it will be important for later extensions to BINDEX that support stateful values.

3. Even when a result is not used more than once, naming the result of an intermediate expression can make a program easier to read. For instance, in the above example, `numer` and `denom` do not make the calculation any more efficient, but some programmers might find the code more readable than without the names. Such naming can be especially handy for breaking down deeply nested expressions into more manageable subexpressions.

2 General Naming Issues

Before we study how the `bind` construct is evaluated in the environment model and the substitution model, we first consider some general concepts and issues related to naming in programming languages and mathematics.

2.1 Declarations and Scope

Programming languages and mathematical languages almost always have constructs that introduce names for the kinds of entities that are manipulated by the language. Such constructs are known as **declarations** or **binding constructs**. Figure 1 shows some examples of declarations from programming and mathematics with which you are probably familiar.

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 many (but as we shall see later in the course, not all) languages, the scope of declared names can be shown via nested boxes called **lexical contours**. For example, draw the lexical contours for the following BINDEX program:

```
(program (x y)

  (* (bind a (* x y)

        (+ a y))

     (bind b (bind c (* 2 y)

                (+ 3 c))

        (div b x))))
```

Language	**Construct**	**Example**

Scheme	$\texttt{(lambda (}I_1 \ldots I_n\texttt{)}\ E_{body}\texttt{)}$	`(lambda (a b)`
		`  (lambda (c)`
		`    (* b (+ a c))))`

Scheme	$\texttt{(let ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}$	`(let ((x (+ 2 3))`
		`      (y (* 6 7)))`
		`  (let ((z (quotient y x)))`
		`    (* x (+ y z))))`

Scheme	$\texttt{(define}\ I_1\ E_{body}\texttt{)}$	`(define a (+ 2 3))`
		
		`(define square`
		`  (lambda (x) (* x x)))`
		
		`(define a-squared (square a))`

Java	$T_{\text{return-type}}\ I_{varname}\ \texttt{=}\ E_{defn}\texttt{;}$	`int x = 2 + 3;`
		`int y = 6 * 7;`
		`int z = y / x;`

INTEX	$\texttt{(program (}I_1 \ldots I_n\texttt{)}\ E_{body}\texttt{)}$	`(program (a b)`
		`  (div (+ a b) 2))`

BINDEX	$\texttt{(bind}\ I_{name}\ E_{defn}\ E_{body}\texttt{)}$	`(bind x (+ 2 3)`
		`  (bind y (* 6 7)`
		`    (bind z (div y x)`
		`      (* x (+ y z)))))`

Math	$\sum_{I_{var}=E_{lo}}^{E_{hi}} E_{body}, \quad \prod_{I_{var}=E_{lo}}^{E_{hi}} E_{body}$	$\sum_{i=1}^{10} \left(\prod_{j=1}^{i} (i+j)\right)$
Logic	$\forall I_{var}.E_{body}, \quad \exists I_{var}.E_{body}$	$\forall x.\exists y.(x+1) = y)$
Calculus	$\int_{E_{lo}}^{E_{hi}} E_{body}\, dI_{var}$	$\int_0^1 x \cdot (\int_0^x y\, dy)\, dx$

Figure 1: Examples of declarations in programming and mathematics.

Each contour shows the region of the program in which the names declared by the declaration in the contour can be used.

It is possible to replace all names in a program with arrows that point from the reference occurrences of a variable to its binding occurrence. Do this for the above expression:

2.2 Free Variables

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 BINDEX expression `(+ a b)`, `a` and `b` are free variables.
- in the BINDEX expression

  ```
  (bind b (* 2 3)
    (+ a b))
  ```

  `a` is a free variable, but `b` is bound.
- in the BINDEX expression

  ```
  (bind a (+ 1 7)
    (bind b (* 2 3)
      (+ a b))
  ```

  both `a` and `b` are bound.

  Note that some occurrences of a name in an expression can be free while other occurrence of the same name may be bound:

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

The **free variables** of a program phrase is the set of all variable names that occur free in that phrase. The following code shows how to calculate free variables in BINDEX:

```
(define free-vars-program
  (lambda (pgm)
    (set-difference (free-vars (program-body pgm))
                    (list->set (program-formals pgm)))))

;; Return the free vars of a BINDEX expression

(define free-vars
  (lambda (exp)
    (cond ((literal? exp)
           (set-empty))
          ((varref? exp)
           (set-singleton (varref-name exp)))
          ((binapp? exp)
           (set-union (free-vars (binapp-rand1 exp))
                      (free-vars (binapp-rand2 exp))))
          ((bind? exp)
           (set-union (free-vars (bind-defn exp))
                      (set-difference (free-vars (bind-body exp))
                                      (set-singleton (bind-name exp)))))
          (else (error "FREE-VARS: unrecognized expression -- " exp))
          )))
```

2.3 α-Renaming

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, the program

```
(program (x y)
  (* (bind a (* x y)
       (+ a y))
     (bind b (bind c (* 2 y)
               (+ 3 c))
       (div b x))))
```

can be consistently renamed to:

```
(program (x y)
  (+ (bind x (* x y)
       (+ a y))
     (bind y (bind x (* 2 y)
               (+ 3 x))
       (div y x))))
```

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 Scheme abstractions `(lambda (a) (abs (b) (+ a b)))` and `(lambda (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.

We can write a function `rename1` that renames occurrences of free variables in BINDEX expressions. Here are some examples of `rename1`:

```
> (rename1 'a 'b '(+ a c))
(+ b c)

> (rename1 'a 'b '(bind d (+ a c) (* a d)))
(bind d (+ b c) (* b d))

> (rename1 'a 'b '(bind a (+ a c) (* a d)))
(bind a (+ b c) (* a d))

> (rename1 'a 'b '(bind b (+ a b) (* a b)))
(bind b_1 (+ b b) (* b b_1))
```

The implementation of `rename1` is shown in figure 2. It is defined in terms of a more general `rename` function that renames a given expression according to an environment of renamings of names to other names. The `rename` function uses an auxiliary function `name-not-in` (whose implementation is not shown) that returns a subscripted version of a given name that is not in the list of names provided as a second argument. For example:

```
> (name-not-in 'a '(b c d))
a_1

> (name-not-in 'a '(a b c d))
a_1

> (name-not-in 'a '(b c a_1 d))
a_2

> (name-not-in 'a '(a_1 a_5 a_3 b a_2 c d))
a_4

> (name-not-in 'a '(a_1 a_5 a_3 b a_2 c d))
a_4

> (name-not-in 'a_1 '(a_1 a_5 a_3 b a_2 c d))
a_4

> (name-not-in 'a_4 '(a_1 a_5 a_3 b a_2 c d))
a_4
```

```
(define rename1
  (lambda (old new exp)
    (rename exp (make-env (list old) (list new)))))

(define rename*
  (lambda (olds news exp)
    (rename exp (make-env olds news))))

(define rename
  (lambda (exp env)
    (cond ((literal? exp) exp)
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe)
                 exp
                 (make-varref probe))))
          ((binapp? exp)
           (make-binapp (binapp-rator exp)
                        (rename (binapp-rand1 exp) env)
                        (rename (binapp-rand2 exp) env)))
          ((bind? exp)
           (let ((name (bind-name exp))
                 (defn (bind-defn exp))
                 (body (bind-body exp)))
             (cond ((member name (env-keys env))
                    (make-bind name
                               (rename defn env)
                               (rename body (env-remove (list name) env))))
                   ((member name (env-values env))
                    (let ((new-name (name-not-in name
                                                 (set-insert name
                                                             (free-vars body)))))
                      (make-bind new-name
                                 (rename defn env)
                                 (rename (rename1 name new-name body)
                                         env))))
                   (else
                    (make-bind name
                               (rename defn env)
                               (rename body env)))
                   )))
          (else (error "RENAME: unrecognized expression -- " exp))
          )))
```

Figure 2: The implementation of renaming in BINDEX.

3 Evaluating `bind`: Environment Model

In the environment model, it is straightforward to express the English-language description for evaluating `bind` expressions as a clause for handling `bind` within the definition of `env-eval`:

```
(define env-eval
  (lambda (exp env)
    (cond
      ⋮
      ((bind? exp)
       (env-eval
         (bind-body exp)   ; Return the result of evaluating the body
         (env-bind (bind-name exp) ; in an environment that binds the name
                   (env-eval (bind-defn exp) ; to the result of evaluating the defn
                             env)            ; in the current environment.
                   env))) ; The new binding extends the current environment.
      ⋮
      )))
```

Notes:

- Intuitively, environments flow from the root of an abstract syntax tree to the leaves. Their purpose is to transmit the value of each variable from the point where a value is bound to its binding occurrence to all variable references in the scope of the declaration.
- In a BINDEX program that has no unbound variables, the following invariant is observed: for any subexpression E of the program, when `(env-eval` E *env*`)` is called, all free variables of E are bound in the environment *env*.

```
(program (x y)
  (* (bind a (* x y)
       (+ a y))
     (bind b (bind c (* 2 y)
               (+ 3 c))
       (div b x))))

(program (x y)
  (+ (bind x (* x y)
       (+ a y))
     (bind y (bind x (* 2 y)
               (+ 3 x))
       (div y x))))
```

4 Evaluating `bind`: Substitution Model

Evaluating `bind` expressions in the substitution model is a little bit trickier than in the environment model. Extending the `subst-eval` function is straighforward:

```
(define subst-eval
  (lambda (exp)
    (cond
      ⋮
      ((bind? exp)
       (subst-eval
        (subst1 (subst-eval (bind-defn exp))
                (bind-name exp)
                (bind-body exp))))
      ⋮
    )))
```

Recall that `(subst1` I E_{subst} E_{body}`)` is an abbreviation for (`subst` (make-env (list I) (list E_{subst})) E_{body}).

The tricky part is extending `subst` to handle `bind`. Here is an "obvious" but **incorrect** definition of the `bind` clause for `subst`.

```
(define subst
  (lambda (env exp)
    (cond
      ⋮
      ((bind? env)
       (make-bind (bind-name exp)
                  (subst env (bind-defn exp))
                  (subst env (bind-body exp))))
      ⋮
    )))
```

To see why the above definition is wrong, consider what it will do in the following case:

```
(program (z)
  (bind a (* z z)
    (bind a (+ z z)
      (* 3 a))))
```

The problem is that the naive version of `subst` may "accidentally" replace a reference to the bound variable of a `bind` by the value of an enclosing variable with the same name. This problem can be fixed by removing any binding for the bound variable of a `bind` from the environment used by `subst` to process the body of the `bind`. Here is a **correct** version of `subst`:

```
(define subst
  (lambda (env exp)
    (cond
      ⋮
      ((bind? exp)
       (let ((name (bind-name exp))
             (defn (bind-defn exp))
             (body (bind-body exp)))
         (let ((new-env (env-remove (list name) env)))
           (make-bind name
                      (subst env defn)
                      (subst new-env body)))))
      ⋮
    )))
```

Here, `(env-remove` *`vars env`*`)` returns a new environment that has all the bindings in *env* except for those named by the variables in the list *vars*.

5 Multiple Bindings: `let` and `let*` in Scheme

Sometimes it is convenient to bind multiple names and values in a single construct. For example, Scheme's `let` is an example of such a construct:

```
(let ((a_sq (* a a))
      (b_sq (* b b)))
  (let ((numer (+ a_sq b_sq))
        (denom (- a_sq b_sq)))
    (quotient numer denom)))
```

In a construct that has multiple names and definition expressions, the question arises as to whether any of the definition expressions are in the scope of other names in the same construct. For example, what is the value of the following Scheme expression?

```
(let ((x 3)
      (a 4))
  (let ((a (* x 10))
        (b (+ x a)))
    (+ a b))))
```

In particular, the issue is whether the `a` in `(+ x a)` refers to the outer binding of `a` to `4`, or the inner binding of `a` to `(* x 10)`.

In Scheme, we can determine the answer by recalling that `let` is syntactic sugar for an application of a `lambda`, so the above is equivalent to:

```
((lambda (x a)
   ((lambda (a b) (+ a b))
    (* x 10)
    (+ x a)))
 3
 4)
```

In the desugaring, it is clear that the `a` in `(+ x a)` means the outer `a` (bound to `4`). So the result of the expression is as shown below:

```
(let ((x 3)  ; x = 3
      (a 4)) ; a = 4
  (let ((a (* x 10)) ; a = 30
        (b (+ x a))) ; b = 7
    (+ a b)))) ; result = 37
```

Scheme has another binding construct, `let*`, that has the same form as `let`, but in which each definition expression *is* in the scope of the previous names in the list of bindings. For example:

```
(let* ((x 3)  ; x = 3
       (a 4)) ; a = 4
  (let* ((a (* x 10)) ; a = 30
         (b (+ x a))) ; b = 33
    (+ a b)))) ; result = 63
```

The `let*` construct desugars into a sequence of nested `let`s. So the above example is equivalent to:

```
(let ((x 3))  ; x = 3
  (let ((a 4)) ; a = 4
    (let ((a (* x 10))) ; a = 30
      (let ((b (+ x a))) ; b = 33
        (+ a b))))) ; result = 63
```

6 Multiple Bindings: `bindpar` and `bindseq` in BINDEX

To explore multiple bindings in BINDEX, we consider extending BINDEX with two multiple binding constructs:

1. `(bindpar ((`I_{name_1} E_{defn_1}`)` ... `(`I_{name_n} E_{defn_n}`))` E_{body}`)` binds the names I_{name_1}...I_{name_n} to the values of the expressions E_{defn_1} ...E_{defn_n}, where these values are determined *in parallel* (i.e., like Scheme's `let`). All definition expressions are evaluated in the same environment in which the `bindpar` itself is evaluated. The result of the `bindpar` is the result of evaluating E_{body} in an environment that extends the current environment with bindings between all the names and the values of their respective definitions.

2. `(bindseq ((`I_{name_1} E_{defn_1}`)` ... `(`I_{name_n} E_{defn_n}`))` E_{body}`)` binds the names I_{name_1}...I_{name_n} to the values of the expressions E_{defn_1} ...E_{defn_n}, where these values are determined *sequentially* (i.e., like Scheme's `let*`). Each definition expression is evaluated in the environment of the `bindseq` extending with bindings for the names that appear in the bindings of the `bindseq` that precede it. The result of the `bindseq` is the result of evaluating E_{body} in an environment that extends the current environment with bindings between all the names and the values of their respective definitions.

We now study the changes we need to make to the BINDEX interpreter to handle `bindpar` and `bindseq`.

6.1 Free Variables

The free variables of `(bindpar ((`I_{name_1} E_{name_1}`) ... (`I_{name_n} E_{name_n}`))` E_{body}`)` are the free variables of all the E_{defn} expressions unioned with the the difference of the E_{body} expression and all the I_{name} identifiers. This calculation is expressed as a clause in the `free-vars` function as follows:

```
((bindpar? exp)
 (set-union (set-union-map free-vars (bindpar-defns exp))
            (set-difference (free-vars (bindpar-body exp))
                            (list->set (bindpar-names exp)))))
```

For `(bindseq ((`I_{name_1} E_{name_1}`) ... (`I_{name_n} E_{name_n}`))` E_{body}`)`, the calculation of free variables is more complex because the bound variable of a binding needs to be subtracted off from the free variables of the definitions of subsequent bindings. The following example suggests how to calculate the free variables of a `bindseq`.
The calculation in the above example can be expressed succinctly as a clause in `free-vars`:

```
((bindseq? exp)
 (foldr2 (lambda (name defn fvs)
           (set-union (free-vars defn)
                      (set-difference fvs (set-singleton name))))
         (free-vars (bindseq-body exp))
         (bindseq-names exp)
         (bindseq-defns exp)))
```

Just as `map2` is a generalization of `map` that is applied to two lists, `foldr2` is a generalization of `foldr` that accumulates results in a simultaneous right-to-left walk over two lists. For any ternary (three-argument) function *ternop* and initial value *init*,

```
(foldr ternop
       init
       (list a$_1$ a$_2$ ... a$_n$)
       (list b$_1$ b$_2$ ... b$_n$))
```

returns the result of the calculation

```
(ternop a$_1$ b$_1$
  (ternop a$_2$ b$_2$
    ...
    (ternop a$_n$ b$_n$ init) ...))
```

For instance,

```
(foldr (lambda (x y ans) (cons x (cons y ans)))
       '(1 2 3)
       '(4 5 6))
```

returns the list `(1 4 2 5 3 6)`. The `foldr2` function can be defined in terms of `foldr` and `zip` as follows:

```
(define foldr2
  (lambda (ternop init lst1 lst2)
    (foldr (lambda (duple result)
             (ternop (first duple) (second duple) result))
           init
           (zip lst1 lst2))))
```

At this time, it is natural to consider `foldl2`, a generalization of `foldl` that operates on two lists.

```
(foldl ternop
       init
       (list a_1 a_2 ... a_n)
       (list b_1 b_2 ... b_n))
```

returns the result of the calculation

```
(ternop a_n b_n
  ...
  (ternop a_2 b_2
    (ternop a_1 b_1 init) ...))
```

For instance,

```
(foldl (lambda (x y ans) (cons x (cons y ans)))
       '(1 2 3)
       '(4 5 6))
```

returns the list `(3 6 2 5 1 4)`. The `foldr2` function can be defined in terms of `foldr` and `zip` as follows:

```
(define foldl2
  (lambda (ternop init lst1 lst2)
    (foldl (lambda (duple result)
             (ternop (first duple) (second duple) result))
           init
           (zip lst1 lst2))))
```

6.2 Renaming

The renaming of a `bindpar` expression is a generalization of the renaming of a `bind` expression. Figure 3 shows the `rename` clause that handles `bindpar`.

Handling `bindseq` directly would be even trickier than handling `bindpar`, so we instead use a "quick-and-dirty" trick: we treat `bindseq` as if it is an instance of `bind` and a smaller `bindseq`. For example, we treat

```
((bindpar? exp)
 ;; This is a generalization of the clause for bind
 (let ((names (bindpar-names exp))
       (defns (bindpar-defns exp))
       (body (bindpar-body exp)))
   ;; Note: LET* in Scheme is like BINDSEQ in BINDEX
   (let* ((dont-renames (list-intersection names (env-keys env)))
          (new-env (env-remove dont-renames env))
          (body-fvs (free-vars body))
          (capturables
           (set-filter (lambda (x) (not (unbound? x)))
                       (set-map (lambda (fv) (env-lookup fv new-env))
                                body-fvs)))
          (no-goods (set-union capturables body-fvs))
          (capture-env
           (set-fold (lambda (nm env)
                       (env-bind nm (name-not-in nm no-goods) env))
                     (env-empty)
                     capturables))
          (new-names
           (map (lambda (nm)
                  (let ((probe (env-lookup nm capture-env)))
                    (if (unbound? probe)
                        nm
                        probe)))
                names))
          )
     (make-bindpar new-names
                   (map (lambda (d) (rename d env)) ; use *old* env here!
                        defns)
                   (rename (rename body capture-env)
                           new-env)))))
```

Figure 3: Clause of `rename` handling `bindpar`.

```
(bindseq ((a (- b c))
          (b (* a c))
          (c (+ a b)))
  (* a (+ b c)))
```

as if it were written

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

Since the `bindseq` nested inside the `bind` has fewer bindings than the original, this is the basis for an inductive definition of renaming on `bindseq`.

This approach to renaming `bindseq` is shown in figure 4. Treating `bindseq` as a composition of `bind` and a smaller `bindseq` effectively performs an "on the fly" desugaring of `bindseq` into nested `bind`s. Later, we shall see how this can be handled more modularly by a separate desugaring pass.

```
((bindseq? exp)
 ;; We take the "quick-and-dirty" approach of
 ;; rewriting BINDSEQ as a BIND around a BINDSEQ,
 ;; rewriting that, and then undoing the transformation.
 (let ((names (bindseq-names exp))
       (defns (bindseq-defns exp))
       (body (bindseq-body exp)))
   (if (null? names)
       (make-bindseq names defns (rename body env))
       (let* ((new-bind
               (make-bind (first (bindseq-names exp))
                          (first (bindseq-defns exp))
                          (make-bindseq (cdr (bindseq-names exp))
                                        (cdr (bindseq-defns exp))
                                        (bindseq-body exp))))
              (renamed-bind (rename new-bind env))
              (renamed-subbindseq (bind-body renamed-bind)))
         (make-bindseq (cons (bind-name renamed-bind)
                             (bindseq-names renamed-subbindseq))
                       (cons (bind-defn renamed-bind)
                             (bindseq-defns renamed-subbindseq))
                       (bindseq-body renamed-subbindseq))))))
```

Figure 4: Clause of `rename` handling `bindseq`.

6.3 Environment Model Interpreter

The `env-eval` function can be extended to handle `bindpar` and `bindseq` as shown in Figure 5. Note how `foldl2` figures prominently in determining the meaning of `bindseq`. How would the meaning of `bindseq` change if `foldl2` were replaced by `foldr2`?

```
(define env-eval
  (lambda (exp env)
    (cond
      ⋮
      ((bindpar? exp)
       (env-eval (bindpar-body exp)
                 (env-extend
                 (bindpar-names exp)
                 (map (lambda (defn) (env-eval defn env))
                        (bindpar-defns exp))
                env)))
      ((bindseq? exp)
       (env-eval (bindseq-body exp)
                (foldl2 (lambda (name defn e)
                          (env-bind name
                                    (env-eval defn e)
                                    e))
                        env
                        (bindseq-names exp)
                        (bindseq-defns exp))))
      ⋮
      )))
```

Figure 5: Clauses of `env-eval` handling `bindpar` and `bindseq`.

6.4 Substitution Model Interpreter

Changing to the substitution model interpreter to handle `bindpar` and `bindseq` are shown in figures 6 and 7. The clauses for `bindpar` are natural generalizations of the clauses for `bind`, while the clauses for `bindseq` perform the same sort of “on the fly” desugaring as in `rename`.

7 Call-by-name vs call-by-value

```
(define subst-eval
  (lambda (exp)
    (cond
      ⋮
      ((bindpar? exp)
       (subst-eval
        (subst* (map subst-eval (bindpar-defns exp))
                (bindpar-names exp)
                (bindpar-body exp))))
      ((bindseq? exp)
       ;; Rewrite BINDSEQ into a BIND that surrounds a
       ;; smaller BINDSEQ, and evaluate that.
       (if (null? (bindseq-names exp))
           (subst-eval (bindseq-body exp))
           (subst-eval
            (make-bind (first (bindseq-names exp))
                       (first (bindseq-defns exp))
                       (make-bindseq (cdr (bindseq-names exp))
                                     (cdr (bindseq-defns exp))
                                     (bindseq-body exp))))))
      ⋮
      )))
```

Figure 6: Clauses of `subst-eval` handling `bindpar` and `bindseq`.

```
(define subst
  (lambda (env exp)
    (cond
      ⋮
      ((bindpar? exp)
       (let ((names (bindpar-names exp))
             (defns (bindpar-defns exp))
             (body (bindpar-body exp)))
         (let ((new-env (env-remove (bindpar-names exp) env)))
           (make-bindpar names
                         (map (lambda (d) (subst env d)) defns)
                         (subst new-env body)))))
      ((bindseq? exp)
       ;; We take the "quick-and-dirty" approach of
       ;; rewriting BINDSEQ as a BIND around a BINDSEQ,
       ;; substituting into that, and then undoing the transformation.
       (let ((names (bindseq-names exp))
             (defns (bindseq-defns exp))
             (body (bindseq-body exp)))
         (if (null? names)
             (make-bindseq '() '() (subst env body))
             (let* ((new-bind
                     (make-bind (first (bindseq-names exp))
                                (first (bindseq-defns exp))
                                (make-bindseq (cdr (bindseq-names exp))
                                              (cdr (bindseq-defns exp))
                                              (bindseq-body exp))))
                    (substed-bind (subst env new-bind))
                    (substed-subbindseq (bind-body substed-bind)))
               (make-bindseq (cons (bind-name substed-bind)
                                   (bindseq-names substed-subbindseq))
                             (cons (bind-defn substed-bind)
                                   (bindseq-defns substed-subbindseq))
                             (bindseq-body substed-subbindseq))))))
      ⋮
      )))
```

Figure 7: Clauses of `subst` handling `bindpar` and `bindseq`.

```
;; Invoke a program on any number of integer values
(define subst-run-cbn
  (lambda (pgm ints)
    (literal-value
     (subst-eval-cbn
      (subst-cbn* (map make-literal ints)
                  (program-formals pgm)
                  (program-body pgm))))))

;; SUBST-EVAL takes an expression and returns another expression
;; (e.g., an integer literal node, *not* an integer) that is the
;; result of evaluting the expression.
(define subst-eval-cbn
  (lambda (exp)
    (cond ((literal? exp) exp)
          ((varref? exp)
           (throw 'unbound-variable (varref-name exp)))
          ((binapp? exp)
           ((binop-to-node-function (binapp-rator exp))
            (subst-eval-cbn (binapp-rand1 exp))
            (subst-eval-cbn (binapp-rand2 exp))))
          ((bind? exp)
           (subst-eval-cbn
            (subst-cbn1
             (bind-defn exp) ;; *** Unlike CBV, DEFN is not evaluated in CBN ***
             (bind-name exp)
             (bind-body exp))))
          ((bindpar? exp)
           (subst-eval-cbn
            (subst-cbn*
             (bindpar-defns exp) ;; *** Unlike CBV, DEFNS are not evaluated in CBN ***
             (bindpar-names exp)
             (bindpar-body exp))))
          ((bindseq? exp)
           ;; Rewrite BINDSEQ into a BIND that surrounds a
           ;; smaller BINDSEQ, and evaluate that.
           (if (null? (bindseq-names exp))
               (subst-eval-cbn (bindseq-body exp))
               (subst-eval-cbn
                (make-bind (first (bindseq-names exp))
                           (first (bindseq-defns exp))
                           (make-bindseq (cdr (bindseq-names exp))
                                         (cdr (bindseq-defns exp))
                                         (bindseq-body exp))))))
          ;; Note: would need extra clauses for BINDPAR and BINDSEQ
          ;; in the absence of recursion.
          (else (error "SUBST-EVAL-CBN: Unknown expression type -- " exp))
          )))
```

Figure 8: Call-by-name version of substitution model interpreter.

```
;; Auxiliary functions for substitution
(define subst-cbn1
  (lambda (new old exp)
    (subst-cbn (env-make (list old) (list new)) exp)))

(define subst-cbn*
  (lambda (news olds exp)
    (subst-cbn (env-make olds news) exp)))

;; (subst-cbn <env> <exp>) returns a copy of expression <exp>
;; in which every variable reference to a name bound in <env>
;; is replaced by the expression bound to the name. In BINDEX,
;; the entity bound to the name should be an initlit node,
;; *not* an integer.
(define subst-cbn
  (lambda (env exp)
    (cond ((literal? exp) exp)
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe) exp probe)))
          ((binapp? exp)
           (make-binapp (binapp-rator exp)
                        (subst-cbn env (binapp-rand1 exp))
                        (subst-cbn env (binapp-rand2 exp))))
          ((bind? exp)
           (let ((name (bind-name exp))
                 (defn (bind-defn exp))
                 (body (bind-body exp))
                 ;; BODY-FVS is the set of variables in the body
                 ;; that appear free outside the BIND
                 (body-fvs (set-difference (free-vars (bind-body exp))
                                           (set-singleton (bind-name exp)))))
             (let ((new-env (env-remove (list (bind-name exp)) env)))
               ;; CAPTURABLES is the set of vars that will be free in the
               ;; body of the copy of BIND returned by SUBST.
               (let ((capturables
                      (foldr set-union
                             (set-empty)
                             (map (lambda (fv)
                                    (let ((probe (env-lookup fv new-env)))
                                      (if (unbound? probe)
                                          (set-empty)
                                          (free-vars probe))))
                                  body-fvs))))
                 (if (set-member? name capturables)
                     (let ((new-name (name-not-in name capturables)))
                       (make-bind new-name
                                  (subst-cbn env defn)
                                  (subst-cbn new-env (rename1 name new-name body))))
                     (make-bind name
                                (subst-cbn env defn)
                                (subst-cbn new-env body)))))))
          ⋮
```

Figure 9: Call-by-name version of substitution, part 1.

```
⋮
((bindpar? exp)
 (let ((names (bindpar-names exp))
       (defns (bindpar-defns exp))
       (body (bindpar-body exp))
       ;; BODY-FVS is the set of variables in the body
       ;; that appear free outside the BINDPAR
       (body-fvs (set-difference (free-vars (bindpar-body exp))
                                 (list->set (bindpar-names exp)))))
   (let ((new-env (env-remove (bindpar-names exp) env)))
     ;; CAPTURABLES is the set of vars that will be free in the
     ;; body of the copy of BINDPAR returned by SUBST.
     (let* ((dont-renames (list-intersection names (env-keys env)))
            (new-env (env-remove dont-renames env))
            (capturables
             (foldr set-union
                    (set-empty)
                    (map (lambda (fv)
                           (let ((probe (env-lookup fv new-env)))
                             (if (unbound? probe)
                                 (set-empty)
                                 (free-vars probe))))
                         body-fvs)))
            (no-goods (set-union capturables body-fvs))
            (capture-env
             (set-fold (lambda (nm env)
                         (env-bind nm (name-not-in nm no-goods) env))
                       (env-empty)
                       capturables))
            (new-names
             (map (lambda (nm)
                    (let ((probe (env-lookup nm capture-env)))
                      (if (unbound? probe)
                          nm
                          probe)))
                  names))
            )
       (make-bindpar
        new-names
        (map (lambda (d) (subst-cbn env d)) defns)
        (subst-cbn new-env (rename body capture-env)))
       ))))
⋮
```

Figure 10: Call-by-name version of substitution, part 2.

```
((bindseq? exp)
 ;; We take the "quick-and-dirty" approach of
 ;; rewriting BINDSEQ as a BIND around a BINDSEQ,
 ;; substituting into that, and then undoing the transformation.
 (let ((names (bindseq-names exp))
       (defns (bindseq-defns exp))
       (body (bindseq-body exp)))
   (if (null? names)
       (make-bindseq '() '() (subst-cbn env body))
       (let* ((new-bind
               (make-bind (first (bindseq-names exp))
                          (first (bindseq-defns exp))
                          (make-bindseq (cdr (bindseq-names exp))
                                        (cdr (bindseq-defns exp))
                                        (bindseq-body exp))))
              (substed-bind (subst-cbn env new-bind))
              (substed-subbindseq (bind-body substed-bind)))
         (make-bindseq (cons (bind-name substed-bind)
                             (bindseq-names substed-subbindseq))
                       (cons (bind-defn substed-bind)
                             (bindseq-defns substed-subbindseq))
                       (bindseq-body substed-subbindseq))))))
(else (error "SUBST: Unrecognized expression -- " exp))
)))
```

Figure 11: Call-by-name version of substitution, part 3.

```
;;; ENV-EVAL-CBN.SCM
;;;
;;; Evaluation of BINDEX programs using the
;;; call-by-name environment model. In this
;;; model, environment maps names to *thunks*
;;; (zero-parameter functions) that return
;;; the value when called.

;; Invoke a program on any number of integer arguments
(define env-run-cbn
  (lambda (pgm ints)
    (env-eval-cbn (program-body pgm)
                  (env-make (program-formals pgm)
                            ;; Make thunk for each param
                            (map (lambda (n) (lambda () n))
                                 ints)))))

;; Evaluate EXP relative to the environment ENV.
;; The environment ENV is a collection of delayed substitutions.
(define env-eval-cbn
  (lambda (exp env)
    (cond ((literal? exp)
           (literal-value exp))
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe)
               (throw 'unbound-variable (varref-name exp))
               ;; Force the evaluation of thunk.
               (probe))))
          ((binapp? exp)
           ((binop-to-function (binapp-rator exp))
            (env-eval-cbn (binapp-rand1 exp) env)
            (env-eval-cbn (binapp-rand2 exp) env)))
          ((bind? exp)
           (env-eval-cbn (bind-body exp)
                         (env-bind (bind-name exp)
                                   ;; Bind name to thunk of evaluation
                                   (lambda () (env-eval-cbn (bind-defn exp) env))
                                   env)))
          ;; The clauses for BINDPAR and BINDSEQ are only needed in
          ;; the absence of desugaring
          ((bindpar? exp)
           (env-eval-cbn (bindpar-body exp)
                         (env-extend
                          (bindpar-names exp)
                          (map (lambda (defn)
                                 (lambda () (env-eval-cbn defn env)))
                               (bindpar-defns exp))
                          env)))
          ((bindseq? exp)
           (env-eval-cbn (bindseq-body exp)
                         (foldl2 (lambda (name defn e)
                                   (env-bind name
                                             (lambda () (env-eval-cbn defn e))
                                             e))
                                 env
                                 (bindseq-names exp)
                                 (bindseq-defns exp))))
          (else (error "ENV-EVAL: Unknown expression type -- " exp))
          )))
```