CS251 Programming Languages **Handout # 30**
Prof. Lyn Turbak **March 13, 2007**
Wellesley College

Bindex: An Introduction to Naming

1 Local Binding Expressions

Studying the INTEX language is able to give us insight into fundamental programming processes like interpretation and program analysis. 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 values. We shall do this in a new language, BINDEX, an extension to INTEX in which:

1. Program arguments are referenced by name rather than by position. For example, in BINDEX, the averaging program can be written as:

   ```
   (bindex (x y) (/ (+ x y) 2))
   ```

   The program inputs (`(x y)` in this case) are called the **formal parameters** of the program and each use of such a parameter in the program is called a **variable reference**.
2. There is a new kind of expression that has the form[1]

   `(bind` I_{name} E_{defn} E_{body}`)`

   Intuitively, a `bind` expression is evaluated as follows:

   (a) The **definition expression** E_{defn} is evaluated to an integer value V_{defn}.
   (b) The **body expression** E_{body} is evaluated in such a way that each variable reference I_{name} denotes V_{defn}.
   (c) The integer 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 $\frac{a^2+b^2}{a^2-b^2}$.

```
(bindex (a b)
  (bind a_sq (* a a)
    (bind b_sq (* b b)
      (bind numer (+ a_sq b_sq)
        (bind denom (- a_sq b_sq)
          (/ numer denom))))))
```

Constructs like `bind` in BINDEX, `let` in OCAML and in SCHEME, and *`type varname = exp;`* in JAVA and C are known as **local binding expressions** or **local variable declarations**. A local binding expression introduces a name for the result of a definition expression that can be used in some part of the program (in the case of `bind`, the body of the `bind` expression). Local binding expressions are used for three main purposes in a program:

1. Naming the result of evaluating a definition expression can avoid the cost of recalculating the value of that expression. 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

[1] We shall use (potentially subscripted) occurrences of the metavariable I to stand for names (a.k.a. identifiers) in a language and (potentially subscripted) occurrences of the metavariable E to stand for expressions.

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 BINDEX Syntax

The abstract syntax for BINDEX (Fig. 1) is the same as that for INTEX except for three changes:

1. The program form `Pgm` replaces the number of program arguments by a list of formal parameter names.
2. The argument reference form `Arg of int` is replaced by the variable reference form `Var of var`, where `var` is a synonym for `string`.
3. There is a new form, `Bind of var * exp * exp`, that represents the `bind` expression. The `var` component is the **bound variable**, while the two subexpressions, in order, are the **definition** (whose value is bound to the variable) and the **body** in which the bound variable may be used.

```
type var = string

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

and exp =
    Lit of int (* integer literal with value *)
  | Var of var (* variable reference *)
  | BinApp of binop * exp * exp (* binary operator application with rator, rands *)
  | Bind of var * exp * exp (* bind name to value of defn in body *)

and binop =
  | Add | Sub | Mul | Div | Rem (* binary arithmetic ops *)
```

Figure 1: Abstract syntax for BINDEX expressed in OCAML.

A `bind` expression may be used anywhere an expression is expected, including the rand positions of a binary application and the definition of another `bind` expression. For example, here is a sample program illustrating the flexibility of `bind` expressions (Fig. 2 shows its AST):

```
(bindex (x y)
  (+ (bind a (/ y x)
       (bind b (- a y)
         (* a b)))
     (bind c (bind d (+ x y) (* d y))
       (/ c x))))
```

The `let` expressions in OCAML and SCHEME have similar flexibility. But variable declarations in JAVA and C are much more restricted. In JAVA, variable declarations are statements, a syntactic phrase denoting an action that cannot occur inside an expression (which denotes a value). In C, variable declarations are even more restricted — they can only appear at the beginning of statement blocks (which are delimited by squiggly braces).

As in INTEX, we can define a `fold` function for BINDEX that expresses arbitrary divide/conquer/glue functions over BINDEX ASTs. The type of `fold` is

```
(* val fold : (int -> 'a) -> (* litfun *)
              (var -> 'a) -> (* varfun *)
              (binop -> 'a -> 'a -> 'a) (* appfun *)
              (var -> 'a -> 'a -> 'a) -> (* bindfun *)
              -> exp -> 'a *)
```

and its definition is:

```
let rec fold litfun varfun appfun bindfun exp =
  match exp with
    Lit i -> litfun i
  | Var s -> varfun s
  | BinApp(rator, rand1, rand2) ->
      appfun rator
             (fold litfun varfun appfun bindfun rand1)
             (fold litfun varfun appfun bindfun rand2)
  | Bind(name,defn,body) ->
      bindfun name
              (fold litfun varfun appfun bindfun defn)
              (fold litfun varfun appfun bindfun body)
```

However, when `fold` functions get this complex, we tend to use use explicit pattern matching with recursion to process such ASTs.

The `Bindex` module contains the definition of the BINDEX abstract syntax along with various utilities for manipulating the syntax. These include the following parsing and unparsing functions:

```
val sexpToPgm : Sexp.sexp -> Bindex.pgm
val sexpToExp : Sexp.sexp -> Bindex.exp
val stringToPgm : string -> Bindex.pgm
val stringToExp : string -> Bindex.exp
val pgmToSexp : Bindex.pgm -> Sexp.sexp
val expToSexp : Bindex.exp -> Sexp.sexp
val pgmToString : Bindex.pgm -> string
val expToString : Bindex.exp -> string
```

To treat BINDEX as an extension to INTEX, the `sexpToPgm` function parses a representation of an INTEX program with n parameters to a BINDEX programs by using `$1` ...`$`n as the parameter names and treating each argument reference as if it were the variable reference `$`i. For instance, an INTEX averaging program is parsed as if it were written

```
(bindex ($1 $2) (/ (+ $1 $2) 2))
```

This translation is sound as long as no local variable names have the form `$`i.

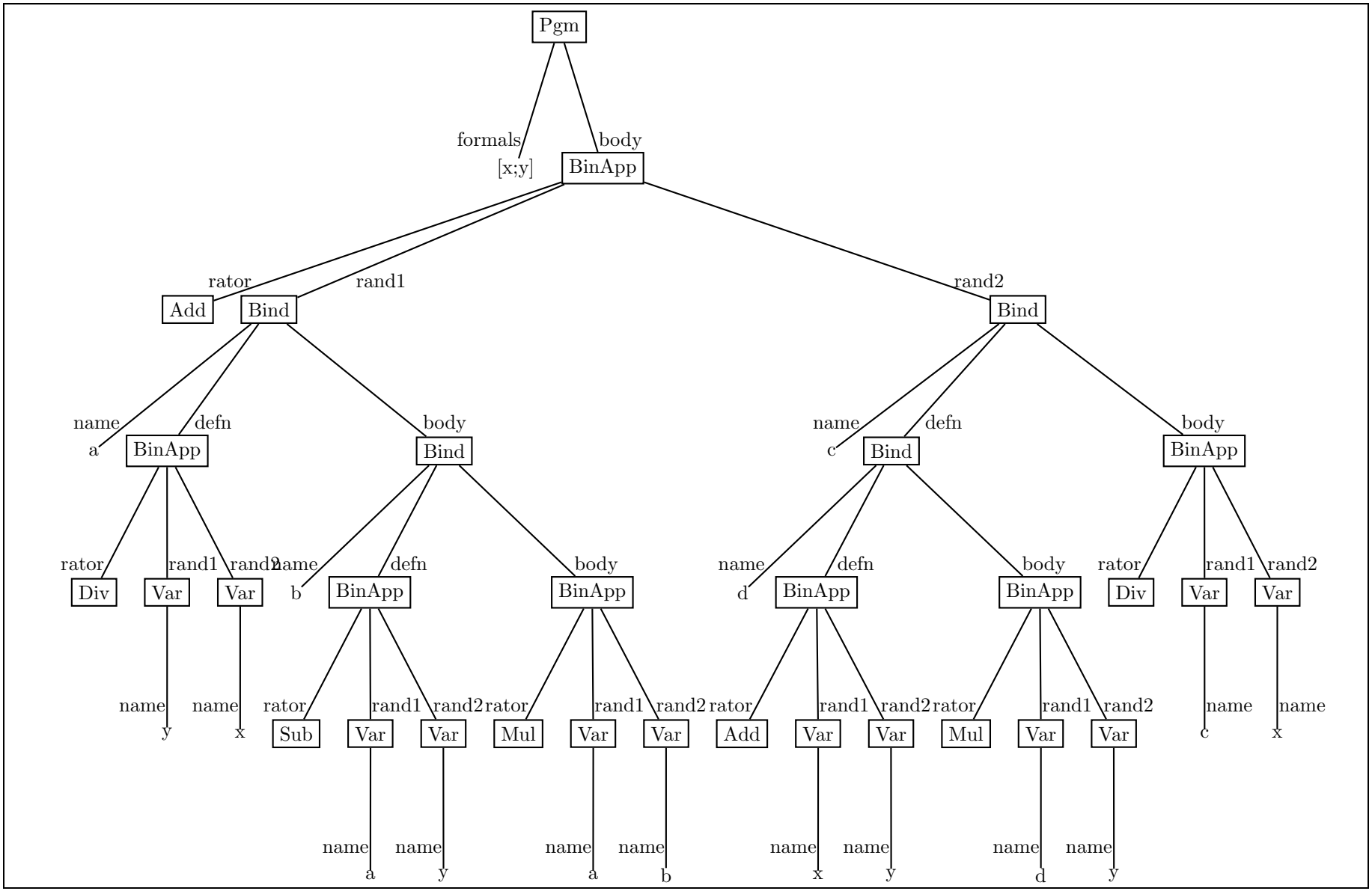


Figure 2: Abstract syntax trees for the sample BINDEX program.

3 General Naming Issues

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

3.1 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 3 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 OCAML abstraction `fun 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 `fun`, for instance, the name following the `fun` keyword is the the binding occurrence, and the uses of this name 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 so-called **statically scoped** languages, languages, the scope of declared names can be shown via nested boxes called **lexical contours**. For example, Fig. 4 shows the lexical contours for the sample BINDEX program. The scope of the formal parameter names `x` and `y` is the entire body of the program (contour C_0). The scope of a local name introduced by a `bind` encompasses the body of the `bind` expression but does *not* include the definition expression. This can be seen by the contours for `a` (C_1), `b` (C_1), `c` (C_3), and `c` (C_4).

When contours are nested, a name declared in an outer contour may be used within an inner contour unless the inner contour declares the same name as the outer contour. For example, the names which may be used in context C_2 are `x`, `y`, `a`, and `b`. However, suppose we rename `b` to `x` so that the first `bind` becomes

```
(bind a (/ y x)
  (bind x (- a y)
    (* a x)))
```

In this case, the `x` that is referenced in C_2 refers to the local name `x` declared in C_2, not the program parameter `x` declared in C_0. So the program parameter `x` declared in C_0 may be references everywhere in the program except in C_2. The inner declaration of `x` is said to **shadow** the outer one, and the contours of the inner declaration is said to be a **hole in the scope** of the outer declaration.

3.2 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** or **unbound 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 (- 8 1) (bind b (* 2 3) (* a b)))`, both `a` and `b` are bound.

Language	Construct	Example
OCAML	`fun` I `->` E_{body}	`fun x -> x * x`

OCAML	`let` I_1 `=` E_1 `and ... and` I_n `=` E_n `in` E_{body}	`let x = 2 + 3`
		`and y = 6 * 7 in`
		`  let z = y / x in`
		`  x * (y + z)`

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

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

SCHEME	`(define` I_1 E_{body}`)`	`(define a (+ 2 3))`
		
		`(define square`
		`  (lambda (x) (* x x)))`
		
		`(define a-squared (square a))`

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

BINDEX	`(bind` I_{name} E_{defn} E_{body}`)`	`(bind x (+ 2 3)`
		`  (bind y (* 6 7)`
		`  (bind z (/ 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 3: Examples of declarations in programming and mathematics.

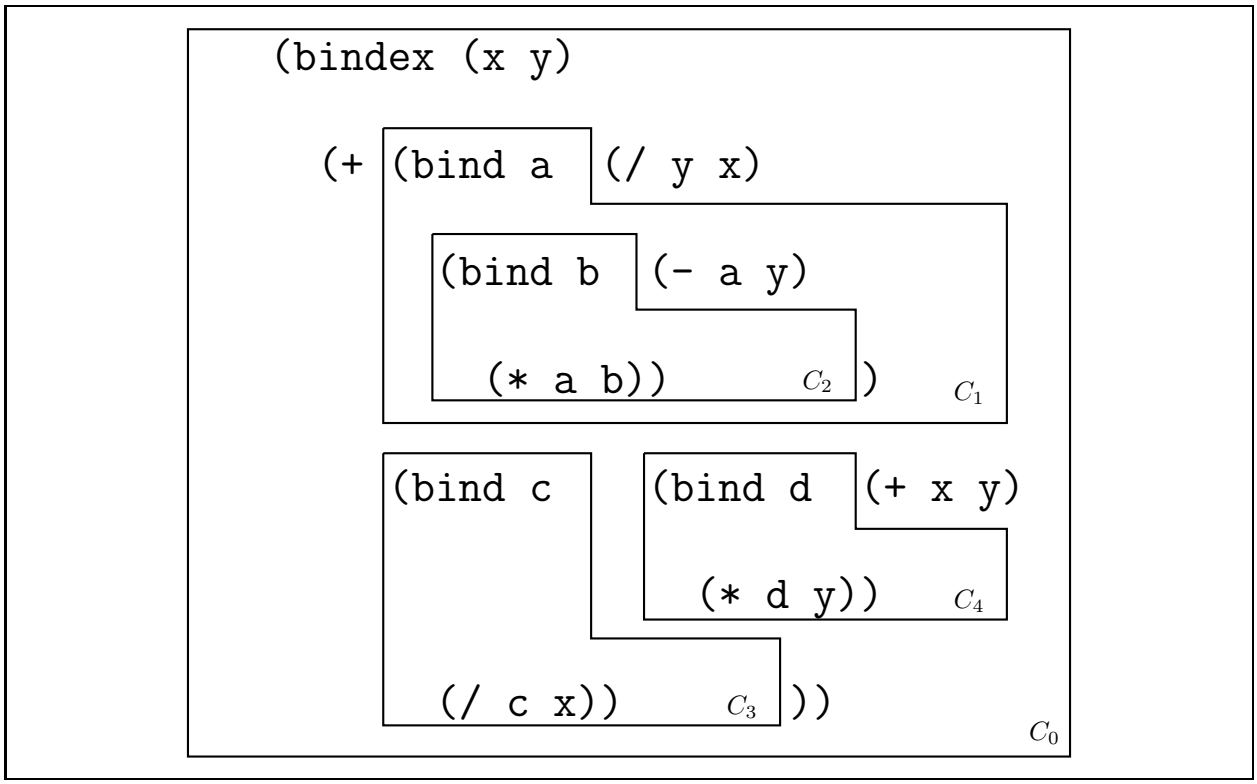


Figure 4: Contours for the sample BINDEX program.

Note that some occurrences of a name in an expression can be free while other occurrence of the same name may be bound. Consider the following expression, in which different reference occurrences of `a` and `b` are distinguished by subscript:

```
(bind a (- a1 b1)
  (bind b (* a2 b2)
    (+ a3 b3))
```

The variable references $\texttt{a}_1$, $\texttt{b}_1$, and $\texttt{b}_2$ are free while $\texttt{a}_2$, $\texttt{a}_3$, and $\texttt{b}_3$ are bound.

The **free variables** of a program phrase is the set of all variable names that occur free in that phrase. Fig. 5 shows functions in the `Bindex` module that calculate free variables in BINDEX. The functor invocation `Set.Make(String)` uses the standard OCAML set making functor to make a module for manipulating sets of strings. Here are some examples of the free variable functions:

```
# open Bindex;;
# setToList (freeVarsExp (stringToExp "(+ a (* b b))"));;
- : Bindex.S.elt list = ["a"; "b"]
# setToList (freeVarsExp (stringToExp "(bind b (* a c) (+ a (* b b)))"));;
- : Bindex.S.elt list = ["a"; "c"]
# setToList (freeVarsExp
             (stringToExp "(bind a (- b c) (bind b (* a c) (+ a (* b b))))"));;
- : Bindex.S.elt list = ["b"; "c"]
# setToList (freeVarsPgm
             (stringToPgm
              "(bindex (b c) (bind a (- b c) (bind b (* a c) (+ a (* b b)))))"));;
- : Bindex.S.elt list = []
```

```
module S = Set.Make(String) (* String Sets *)

(* val listToSet : S.elt list -> S.t *)
let listToSet strs = foldr S.add S.empty strs

(* val setToListList : S.t -> S.elt list *)
let setToList set = S.elements set

(* val freeVarsPgm : pgm -> S.t *)
(* Returns the free variables of a program *)
let rec freeVarsPgm (Pgm(fmls,body)) =
  S.diff (freeVarsExp body) (listToSet fmls)

(* val freeVarsExp : exp -> S.t *)
(* Returns the free variables of an expression *)
(* direct version *)
and freeVarsExp e =
  match e with
    Lit i -> S.empty
  | Var s -> S.singleton s
  | BinApp(_,r1,r2) -> S.union (freeVarsExp r1) (freeVarsExp r2)
  | Bind(name,defn,body) ->
      S.union (freeVarsExp defn)
              (S.diff (freeVarsExp body)
                      (S.singleton name))

(* val freeVarsExps : exp list -> S.t *)
(* Returns the free variables of a list of expressions *)
(* direct version *)
and freeVarsExps es =
  foldr S.union S.empty (map freeVarsExp es)

(* val varCheck : pgm -> unit *)
and varCheck pgm =
  let unbounds = freeVarsPgm pgm
   in if S.is_empty unbounds then
        () (* OK *)
      else
        raise (Unbound (setToList unbounds))
```

Figure 5: Functions for determining free variables in BINDEX.

A program phrase with no free variables is said to be **closed**. All BINDEX programs should be closed since a free variable in a program would have no meaning. The `varCheck` function in Fig. 5 statically checks a program to ensure that it is closed. If it is, it returns the unit value `()`. But if there are unbound variables, it raises an `Unbound` exception with the set of free variables. The `varCheck` function is the analog in BINDEX of INTEX's `argCheck` function.

3.3 α-Renaming

In a statically scoped language, it is always possible to consistently rename a binding occurrence and its corresponding reference occurrences without changing the meaning of a program. Consistent renaming that maintains program meaning is known as **α-renaming**. For example, in

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

we can rename `b` to `x`, `c` to `y`, and the bound `a` to `z` to yield

```
(bind x (* a a)
  (bind y (+ x a)
    (bind z (* x y)
      (/ (+ z x) (- z y)))))
```

Note that the free occurrences of `a` are not renamed.

There is one restriction on the renaming of bound variables. We cannot rename a bound variable to another variable that is free in its scope: this would cause **variable capture**. For example, we cannot rename `b` to `a` above, since the free reference to `a` in the expression `(+ b a)` within the body of `(bind b ...)` would be captured and become bound to the renamed binding occurrence of `b`.

When renaming a bound variable, it may be necessary to additionally rename other bound variables in its scope to avoid variable capture. For example, we can rename `c` to `a` in our example as long as we also rename the `bind`-bound `a` (say to `a.2`):

```
(bind b (* a a)
  (bind a (+ b a)
    (bind a.2 (* b a)
      (/ (+ a.2 a) (- a.2 b)))))
```

As a more complex of α-renaming, we reconsider the sample BINDEX program from earlier:

```
(bindex (x y)
  (+ (bind a (/ y x)
       (bind b (- a y)
         (* a b)))
     (bind c (bind d (+ x y) (* d y))
       (/ c x))))
```

It is possible to rename `a`, `b`, `c`, and `d` to one of `x` or `y` without changing the meaning of the program:

```
(bindex (x y)
  (+ (bind x (/ y x)
       (bind y (- x y)
         (* x y)))
     (bind y (bind x (+ x y) (* x y))
       (/ y x))))
```

The renamed program may be very difficult for human beings to understand, but because it has the same wiring diagram as the original program, it is indistinguishable from the original program for the purposes of program manipulation.

The `Bindex` module provides two renaming functions:

1. `val rename1 : var -> var -> exp -> exp`
   `rename1` *`oldName newName e`* returns a copy of the expression *e* in which all free occurrences of *oldName* have been renamed to *newName*.

2. `renameAll : var list -> var list -> exp -> exp`
   Assume that *oldNames* and *newNames* are string lists with the same length. Then the invocation `renameAll` *`oldNames newNames e`* returns a copy of the expression *e* in which all free occurrences of names in *oldNames* have been renamed to the corresponding name (by position) in *newNames*.

We will see in the next section how these functions are implemented. Note that both functions rename only the free occurrences of a variable in an expression. Program manipulation functions should not refer to the bound occurrences of variables by name because their behavior should not changed if the bound variables are α-renamed.

Below are some sample invocations of these functions:[2]

```
# rename1 "a" "b"
   (BinApp(Add, Var "a",
               Bind("b", BinApp(Mul, Var "a", Var "a"),
                    BinApp(Add, Var "a", Var "b"))));;
- : Bindex.exp =
BinApp (Add, Var "b",
            Bind ("b.10", BinApp (Mul, Var "b", Var "b"),
                  BinApp (Add, Var "b", Var "b.10")))
```

Note that the `bind`-bound occurrences of `b` have been automatically renamed (to `b.10`) to avoid variable capture involving the renamed `b`.

```
# rename1 "a" "z"
   (BinApp(Add, Var "a",
               Bind("b", BinApp(Mul, Var "a", Var "a"),
                    BinApp(Add, Var "a", Var "b"))));;
- : Bindex.exp =
BinApp (Add, Var "z",
            Bind ("b.11", BinApp (Mul, Var "z", Var "z"),
                  BinApp (Add, Var "z", Var "b.11")))
```

Here the `bind`-bound `b` has been renamed (to `b.11`) even though there is no threat of variable capture. This is an artifact of the way that `rename1` is implemented — it automatically renames all `bind`-bound names to avoid any possibility of variable capture, regardless of whether or not variable capture will actually occur.

```
# rename1 "a" "z"
   (BinApp(Add, Var "a",
               Bind("a", BinApp(Mul, Var "a", Var "a"),
                         BinApp(Add, Var "a", Var "a"))));;
- : Bindex.exp =
BinApp (Add, Var "z",
            Bind ("a.12", BinApp (Mul, Var "z", Var "z"),
                  BinApp (Add, Var "a.12", Var "a.12")))
```

This example shows that only the free occurrence of `a` are renamed to `z`. The bound occurrences of `a` are renamed to `a.12` as an artifact of the way `rename1` is implemented.

```
# renameAll ["a";"b"] ["b";"a"] (BinApp(Add, Var "a", Var "b"));;
- : Bindex.exp = BinApp (Add, Var "b", Var "a")
```

In this example, note that the renamings are performed simultaneously. This simultaneous renaming cannot be simulated by any ordering of two calls to `rename1`.

```
# renameAll ["a";"b"] ["b";"a"]
   (Bind("a", BinApp(Add, Var "a", Var "b"),
         Bind("b", BinApp(Sub, Var "a", Var "b"),
              BinApp(Mul, Var "a" , Var "b"))));;
- : Bindex.exp =
Bind ("a.13", BinApp (Add, Var "b", Var "a"),
      Bind ("b.15", BinApp (Sub, Var "a.13", Var "a"),
            BinApp (Mul, Var "a.13", Var "b.15")))
```

[2]Some of the results have been manually reformatted to make them easier to read.

We will use the `StringUtils.fresh` function to automatically generate "fresh" variable names as needed in our program manipulation functions. This function simply adds a dot followed by a unique number to the given variable name. It is assumed that the original user program does not contain such dotted variable names; they are only introduced by the program manipulation process. For example:

```
# StringUtils.fresh "foo";;
- : string = "foo.0"

# StringUtils.fresh "foo";;
- : string = "foo.1"

# StringUtils.fresh "bar";;
- : string = "bar.2"

# StringUtils.fresh "foo.0";;
- : string = "foo.3"
```

The final example shows that when `fresh` is called on names generated by `fresh`, only the "root" of the name (before the dot) is used. Thus, `fresh "foo.0"` yields `"foo.3"` rather than `"foo.0.3"`. This convention helps to make programs more readable.

We will say that a program is **uniquely named** if all logically distinct variables in the program have different names. Fig. 6 presents a functions `uniquifyPgm` and `uniquifyExp` that transform BINDEX programs and expressions, respectively, to uniquely named form. This is achieved by using `fresh` and `rename1` to give a unique name to every `bind`-bound name. The program formal parameters are *not* renamed; it is assumed that these are all distinct, and renaming the `bind`-bound names will make them all different from the formal parameters. For example:

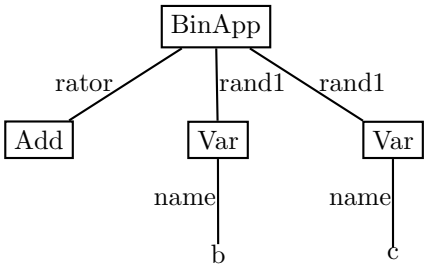
```
let rec uniquifyPgm pgm =
  match pgm with
    Pgm(args,body) -> Pgm(args,uniquifyExp body)

and uniquifyExp exp =
  match exp with
    Lit i -> exp
  | Var v -> exp
  | BinApp(op,r1,r2) -> BinApp(op, uniquifyExp r1, uniquifyExp r2)
  | Bind(name,defn,body) ->
      (* rename every bind-bound name to a fresh name *)
      let name' = StringUtils.fresh name in
        Bind(name', uniquifyExp defn, uniquifyExp (rename1 name name' body))
```

Figure 6: Functions that transform BINDEX programs and expressions to uniquely named form.

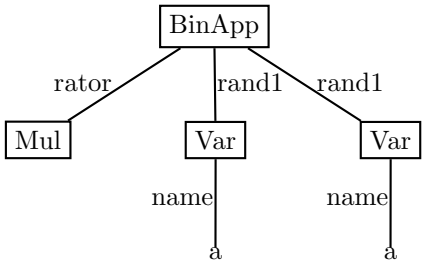
```
# print_string
   (pgmToString
    (uniquifyPgm
     (stringToPgm
      "(bindex (x y)
         (+ (bind x (/ y x)
              (bind y (- x y)
                (* x y)))
            (bind y (bind x (+ x y) (* x y))
              (/ y x))))")));;
(bindex (x y)
  (+ (bind x.5 (/ y x)
       (bind y.7 (- x.5 y) (* x.5 y.7)))
     (bind y.3 (bind x.4 (+ x y) (* x.4 y))
       (/ y.3 x))))- : unit = ()
```

3.4 Substitution

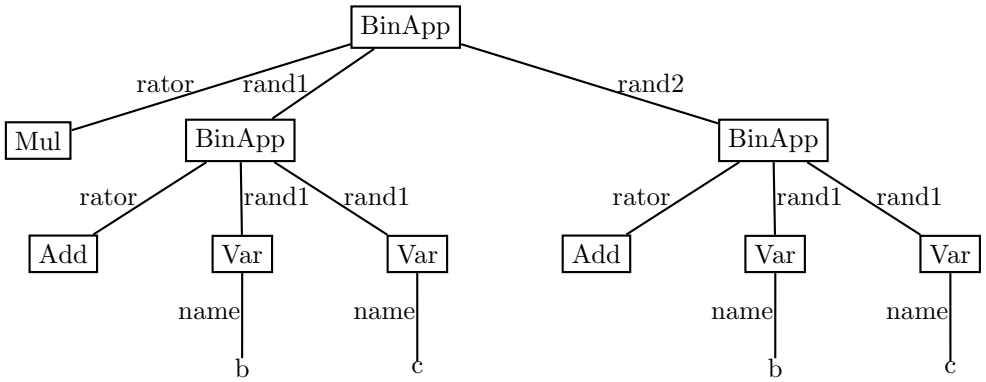
Renaming is a special case of a more general program form of program manipulation called **substitution**. In substitution, all free occurrences of a variable name are replaced by a given expression. For example, substituting `(+ b c)` for `a` in `(* a a)` yields `(* (+ b c) (+ b c))`. If we view expressions as trees, then substitution replaces certain leaves (those with the specified name) by new subtrees. For example, substituting the tree



for `a` in the tree



yields the tree



As in renaming, substitution must avoid variable capture to preserve the "naming wiring structure" of expressions. In particular:

- When substituting for a variable I, we will never perform substitutions on I within the scope of a `bind` expression that binds I. For example, substituting `(+ b c)` for `a` in `(bind a (* a a) (- a 3))` yields:[3]

  ```
  (bind a (* (+ b c) (+ b c)) (- a 3))
  ```

  Here the free occurrences of `a` have been replaced by `(+ b c)`, but the bound occurrences have not been changed.

- `bind`-bound variables may be renamed to avoid variable capture with free variables in the expression being substituted for the variable. For example, substituting `(+ b c)` for `a` in

  ```
  (+ (bind b (+ 1 a) (* a b))
     (bind c (* 2 a) (+ a c)))
  ```

  yields

  ```
  (+ (bind b.1 (+ 1 (+ b c)) (* (+ b c) b.1))
     (bind c.2 (* 2 (+ b c)) (+ (+ b c) c.2)))
  ```

  Here is is necessary to rename the `bind`-bound `b` and `c` to avoid capturing the free variables `b` and `c` in `(+ b c)`.

Substitution is an important tool for program manipulation. We have already encountered substitution in the context of the **substitution model** for the evaluation of OCAML programs, where function application was explained by substituting the argument values for the formal parameters in the function body. Later, we will present a similar substitution model for BINDEX evaluation.

We will now develop a substitution function for BINDEX that we will be able to use later for manipulating BINDEX programs. The substitution function will allow us to simultaneously substitute expressions for any number of variables. To specify the association between the variable names and the expressions to be substituted for them, we will use environments having the `ENV` signature in Fig. 7. This is similar to the `MENV` signature presented in the Modules handout (#23), except that `ENV` provides a `bindAll` function but does not provide a `merge` function.

Fig. 8 defines the following `subst` function:

val `subst: exp -> exp Env.env -> exp`
`subst` *exp env* returns a copy of the expression *exp* in which all free occurrences of names bound in the environment *env* have been replaced by their associated expressions. Bound variables in *exp* may be α-renamed in order to avoid variable capture.

The `subst` function works by performing a near-copy of the given BINDEX abstract syntax tree. There are two places where it does not perform an exact copy:

1. For a variable reference, if the variable name appears in the environment `env`, the variable reference is replaced by the expression bound to the variable name in `env`. Otherwise, the variable reference is copied.

2. For a `bind` expression, the bound variable of the `bind` is always α-renamed to a fresh variable before substitution is performed on its body. In many cases this does more renaming than is strictly necessary, but it is a simple way to avoid all variable capture problems and makes it unnecessary to check if the `bind`-bound name is bound in `env`. It is possible to write a

[3] In the BINDEX implementation of substitution, the bound `a` will be renamed to a fresh variable in this example.

```
module type ENV = sig
  type 'a env

  val empty: 'a env (* returns the empty env *)

  val bind : string -> 'a -> 'a env -> 'a env
  (* bind <name> <value> <env> returns a new env containing a binding
     of <name> to <value> in addition to all the bindings of <env>. *)

  val bindAll : string list -> 'a list -> 'a env -> 'a env
  (* bind <names> <values> <env> returns a new env containing bindings
     of <names> to <values> in addition to all the bindings of <env>. *)

  val make : string list -> 'a list -> 'a env
  (* make <names> <values> <env> returns a new env containing bindings
     of <names> to <values>. *)

  val lookup : string -> 'a env -> 'a option
  (* lookup <name> <env> returns Some <value> if <name> is bound to <value>
     in <env>; otherwise it returns None. *)
end
```

Figure 7: An environment signature.

cleverer version of `subst` that (1) renames variables only when absolutely necessary and (2) performs the renaming and substitution on the `bind` body in a single tree walk rather than two separate tree walks. This is left as an exercise for the reader.

Fig. 8 also defines two auxiliary substitution functions:

val `subst1: exp -> var -> exp -> exp`
`subst1` *exp var exp'* returns a copy of the expression *exp'* in which all free occurrences of *var* have been replaced by *exp*.

val `substAll: exp list -> var list -> exp -> exp`
Assume that *exps* and *vars* are equal-length lists of expressions and strings, respectively. Then `substAll` *exps vars exp* returns a copy of the expression *exp* in which all free occurrences of names in *vars* have been replaced by the corresponding expression in *exps*.

For example:

```
# let testSubst1 e v e' =
    print_string(expToString(subst1 (stringToExp e) v (stringToExp e')));;
val testSubst1 : string -> Bindex.var -> string -> unit = <fun>

# testSubst1 "(+ b c)" "a" "(bind a (* a a) (- a 3))";;
(bind a.17 (* (+ b c) (+ b c)) (- a.17 3))- : unit = ()

# testSubst1 "(+ b c)" "a" "(+ (bind b (+ 1 a) (* a b)) (bind c (* 2 a) (+ a c)))";;
(+ (bind b.19 (+ 1 (+ b c)) (* (+ b c) b.19))
   (bind c.18 (* 2 (+ b c)) (+ (+ b c) c.18))
   )- : unit = ()

# let testSubstAll es vs e' =
    print_string(expToString(substAll (List.map stringToExp es) vs (stringToExp e')));;
val testSubstAll : string list -> string list -> string -> unit = <fun>
```

```
(* 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)
  | BinApp(op,r1,r2) -> BinApp(op, subst r1 env, subst r2 env)
  | Bind(name,defn,body) ->
    (* Take the simple approach of renaming every name.
       With more work, we could avoid renaming unless absolutely necessary. *)
    let name' = StringUtils.fresh name in
      Bind(name', subst defn env, subst (rename1 name name' body) env)
      (* note: could be cleverer and do a single substitution/renaming *)

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

(* val substAll: exp list -> var list -> exp -> exp *)
(* subst <exps> <vars> <exp> substitutes <exps> for <vars> in <exp> *)
and substAll newexps names exp = subst exp (Env.make names newexps)

(* val rename1 : var -> var -> exp -> exp *)
(* rename <oldName> <newName> <exp> renames <oldName> to <newName> in <exp> *)
and rename1 oldname newname exp = subst1 (Var newname) oldname exp

(* val renameAll : string list -> var list -> exp -> exp *)
(* rename <oldNames> <newNames> <exp> renames <oldNames> to <newNames> in <exp> *)
and renameAll oldnames newnames exp =
  substAll (List.map (fun s -> Var s) newnames) oldnames exp
```

Figure 8: Substitution and renaming functions for BINDEX.

```
# testSubstAll ["(+ b c)"; "(* a b)"] ["a";"b"]
               "(+ (bind a (/ a b) (- a b)) (bind b (/ b a) (- b a)))";;
(+ (bind a.21 (/ (+ b c) (* a b)) (- a.21 (* a b)))
   (bind b.20 (/ (* a b) (+ b c)) (- b.20 (+ b c)))
   )- : unit = ()
```

Note that in Fig. 8, the `rename1` and `renameAll` functions introduced in Sec. 3.3 are easily defined in terms of `subst1` and `substAll`. The fact that these are defined in terms of the general `subst` function and that `rename1` is used in the definition of `subst` may seem unsettling at first. But note that the call to `rename1` in `subst` is called on a strictly smaller subexpression than the expression argument to `subst`. Since `subst` of the whole expression is being defined in terms of the value of `subst` on smaller subexpression, the recursion is well-defined.

4 A Substitution Model Interpreter for BINDEX

With a substitution function in hand, we can implement a substitution model interpreter for BINDEX (Fig. 9). This is similar to the INTEX interpreter except:

- Rather than passing the argument list as an argument to `eval`, the argument integers are substituted for the formal parameters using `substAll`. Note that the integers must be converted to expressions (via the `Lit` constructor) before the substitutions can take place.

```
module BindexSubstInterp = struct

  open Bindex
  open List

  exception EvalError of string

  (* val run : Bindex.pgm -> int list -> int *)
  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 i) ints) fmls body)
      else
        raise (EvalError ("Program expected " ^ (string_of_int flen)
                          ^ " arguments but got " ^ (string_of_int ilen)))

  (* val eval : Bindex.exp -> int *)
  and eval exp =
    match exp with
      Lit i -> i
    | Var name -> raise (EvalError("Unbound variable: " ^ name))
    | BinApp(rator,rand1,rand2) ->
        binApply rator (eval rand1) (eval rand2)
    | Bind(name,defn,body) ->
        eval (subst1 (Lit (eval defn)) name body)

  (* val binApply : Bindex.binop -> int -> int -> int *)
  and binApply op x y =
    match op with
    | Add -> x + y
    | Sub -> x - y
    | Mul -> x * y
    | Div -> if y = 0 then
               raise (EvalError ("Division by 0: "
                                 ^ (string_of_int x)))
             else
               x / y
    | Rem -> if y = 0 then
               raise (EvalError ("Remainder by 0: "
                                 ^ (string_of_int x)))
             else
               x mod y

  (* A function for running programs expressed as strings *)
  let runString pgmString args =
    run (sexpToPgm (Sexp.stringToSexp pgmString)) args

end
```

Figure 9: A substitution model BINDEX interpreter.

- The `eval` function does not need any argument other than the expression being evaluated, so it has type `exp -> int`.
- A `bind` node is evaluated by (1) first evaluating the definition expression to an integer and then (2) evaluating the result of substituting this integer for the `bind`-bound variable name in the body of the `bind` expression.
- How is a variable reference node evaluated? In the BINDEX substitution model, all variable references should be replaced by integer literals before they are encountered by `eval`. If a variable reference *is* encountered by `eval`, it must be the case that it is a free variable in the program — i.e., it is an unbound variable error.

5 An Environment Model Interpreter for BINDEX

An alternative interpretation strategy for BINDEX expressions is the **environment model**. In this strategy, substitutions of integers for variables are not performed eagerly but are delayed by remembering them in environments. The `eval` function is modified to accept a second argument that is an environment of all delayed substitutions, and the substitutions are only performed when a variable reference is reached.

An interpreter based on this strategy is presented in Fig. 10. Like the argument list in the INTEX interpreter, an initial environment associating the formal parameter names and the actual integer arguments is passed down the abstract syntax tree of the program body as part of evaluation. Unlike the INTEX argument list, however, the BINDEX environment is modified as it "flows" down the tree at every `bind` node, which adds a new binding to the environment used for the body of the `bind`.

```
module BindexEnvInterp = struct

  open Bindex
  open List

  exception EvalError of string

  (* val run : Bindex.pgm -> int list -> int *)
  let rec run (Pgm(fmls,body)) ints =
    let flen = length fmls
    and ilen = length ints
    in
      if flen = ilen then
        eval body (Env.make fmls ints)
      else
        raise (EvalError ("Program expected " ^ (string_of_int flen)
                          ^ " arguments but got " ^ (string_of_int ilen)))

  (* val eval : Bindex.exp -> int Env.env -> int *)
  and eval exp env =
    match exp with
      Lit i -> i
    | Var name ->
        (match Env.lookup name env with
           Some(i) -> i
         | None -> raise (EvalError("Unbound variable: " ^ name)))
    | Bind(name,defn,body) ->
        eval body (Env.bind name (eval defn env) env)
    | BinApp(rator,rand1,rand2) ->
        binApply rator (eval rand1 env) (eval rand2 env)

  (* val binApply : Bindex.binop -> int -> int -> int *)
  and binApply op x y =
    match op with
    | Add -> x + y
    | Sub -> x - y
    | Mul -> x * y
    | Div -> if y = 0 then
               raise (EvalError ("Division by 0: "
                                 ^ (string_of_int x)))
             else
               x / y
    | Rem -> if y = 0 then
               raise (EvalError ("Remainder by 0: "
                                 ^ (string_of_int x)))
             else
               x mod y

  (* A function for running programs expressed as strings *)
  let runString pgmString args =
    run (sexpToPgm (Sexp.stringToSexp pgmString)) args

end
```

Figure 10: A environment model BINDEX interpreter.