Bindex: Naming, Free Variables, and Environments

CS251 Programming Languages
Spring 2018, Lyn Turbak

Department of Computer Science
Wellesley College

Review: Scope and Lexical Contours

scope = area of program where declared name can be used.
Show scope in Racket via ***lexical contours*** in ***scope diagrams***.

```
(define add-n  (λ ( x ) (+  n  x ))  )
(define add-2n (λ ( y ) (add-n (add-n  y ))))
(define n 17)
(define f (λ ( z )
            (let {[ c (add-2n z ) ]
                  [ d (- z 3)     ]}
              (+ z (* c d )))     )    )
```


Review: Declarations vs. References

A **declaration** introduces an identifier (variable) into a scope.

A **reference** is a use of an identifier (variable) within a scope.

We can box declarations, circle references, and draw a line from each reference to its declaration. Dr. Racket does this for us (except it puts ovals around both declarations and references).

An identifier (variable) reference is **unbound** if there is no declaration to which it refers.

Review: Shadowing

An inner declaration of a name ***shadows*** uses of outer declarations of the same name.

```
(let {[x 2]}
  (- (let {[x (* x x)]}
       (+ x 3))   Can't refer to outer x here.
     x ))
```

Review: Alpha-renaming

Can consistently rename identifiers as long as it doesn't change the connections between uses and declarations.

```
(define (f w z)
  (* w
     (let {[c (add-2n z)]
           [d (- z 3)]}
       (+ z (* c d))))))
```

OK

```
(define (f c d)
  (* c
     (let {[b (add-2n d)]
           [c (- d 3)]}
       (+ d (* b c))))))
```

Not OK

```
(define (f x y)
  (* x
     (let {[x (add-2n y)]
           [y (- d y)]}
       (+ y (* x y))))))
```


Review: Scope, Free Variables, and Higher-order Functions

In a lexical contour, an identifier is a ***free variable*** if it is not defined by a declaration within that contour.

Scope diagrams are especially helpful for understanding the meaning of free variables in higher order functions.

```
(define (make-sub n )
  (λ ( x ) (- x n)) )

(define (map-scale factor ns )
  (map (λ ( num ) (* factor num )) ns) )
```


A New Mini-Language: Bindex

Bindex adds variable names to Intex in two ways:

1. The arguments of Bindex programs are expressed via variable names rather than positionally. E.g.:

```
(bindex (a b) (/ (+ a b) 2))

(bindex (a b c x) (+ (* a (* x x)) (+ (* b x) c)))
```

2. Bindex has a local naming construct `(bind I_defn E_defn E_body)` that behaves like Racket's `(let {[I_defn E_defn]} E_body)`

```
(bindex (p q)
  (bind sum (+ p q)
    (/ sum 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)))))
```

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

Can use `bind` in any expression position

Bindex REPL Interpreter in action

REPL = Read/Eval/Print Loop. Our goal is to see how this all works.

Try it out:
`~wx/sml/bindex/BindexEnvInterp.sml`

```
- BindexEnvInterp.repl();

bindex> (+ (/ 6 3) ( * 5 8))
42

bindex> (bind a (+ 1 2) (bind b ( * a 5) (- a b)))
~12

bindex> (#args (num 5) (p 10) (q 8))

bindex> ( * (- q num) p)
30

bindex> (#run (bindex (x y) (+ ( * x x) ( * y y))) 3 4)
25

bindex> (#run (bindex (a b) (bind sum (+ a b) (/ sum 2))) 5 15)
10

bindex> (#quit)
Moriturus te saluto!
val it = () : unit
```

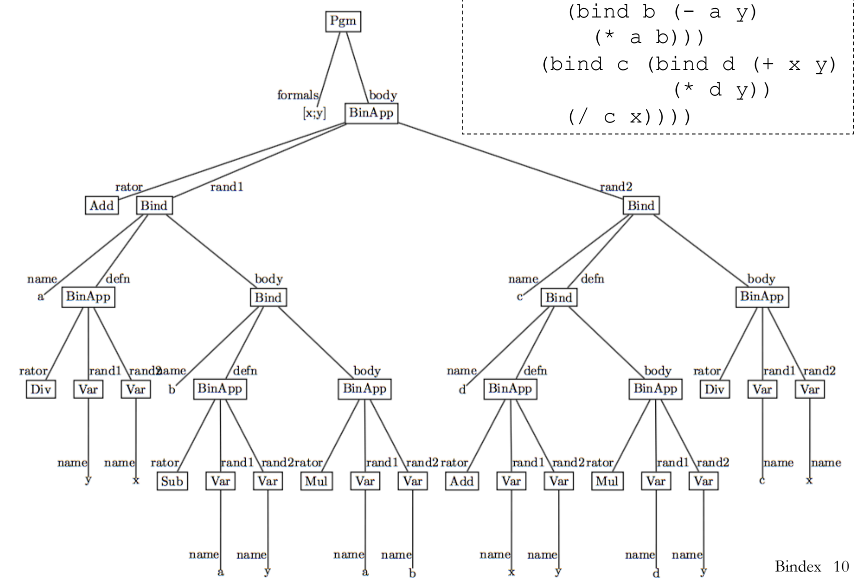
Bindex Abstract Syntax

```
type ident = string (* introduce ident as synonym for string *)

datatype pgm = Bindex of ident list * exp (* param names, body *)

and exp = Int of int (* integer literal with value *)
        | Var of ident(* variable reference *)
        | BinApp of binop * exp * exp
          (* binary application of rator to rand1 & rand2 *)
        | Bind of ident * exp * exp
          (* bind name to value of defn in body *)

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

val stringToExp : string -> exp
val stringToPgm : string -> pgm
val expToString : exp -> string
val pgmToString : pgm -> string
```

```
- Bindex.stringToPgm "(bindex (a b) (bind sum (+ a b) (/ sum 2)))"
val it =
  Bindex
    (["a","b"],
     Bind ("sum",BinApp (Add,Var "a",Var "b"),
                 BinApp (Div,Var "sum",Int 2))) : Bindex.pgm
```


Bindex AST example

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

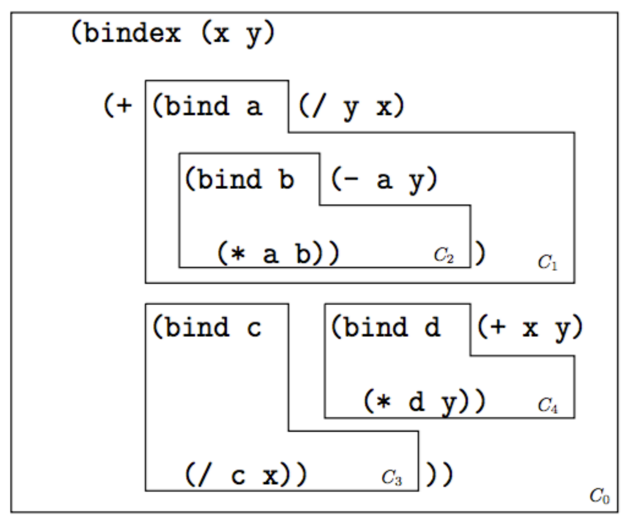


Calculating Free Variables in Bindex

it's your turn

Bindex Phrase P	Free Variables: FV(P)
L (integer literal)	
I (variable reference)	
(O_{rator} E_{rand1} E_{rand2})	
(bind I E_{defn} E_{body})	
(bindex (I_1 … I_n) E_{body})	

Bindex Lexical Contours and Free Variables



String sets
(similar to PS7 sets, but specialized to strings)

```
signature STRING_SET =
sig
  type t (* The type of a string set *)
  val empty : t
  val singleton : string -> t
  val isEmpty : t -> bool
  val size : t -> int
  val member : string -> t -> bool
  val insert : string -> t -> t
  val delete : string -> t -> t
  val union : t -> t -> t
  val intersection : t -> t -> t
  val difference : t -> t -> t
  val fromList : string list -> t
  val toList : t -> string list
  val toPred : t -> (string -> bool)
  val toString : t -> string
end

structure StringSetList :> STRING_SET = struct
  (* See ~wx/sml/utils/StringSet.sml for details *)
end
```


Bindex: Code for handling free variables

it's your turn

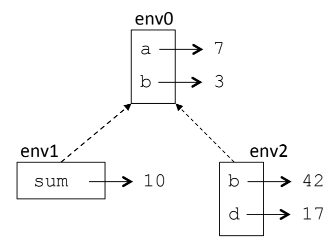
```
structure S = StringSetList

(* val freeVarsPgm : pgm -> S.t *)
(* Returns the free variables of a program *)
fun freeVarsPgm (Bindex(fmls,body)) =


(* val freeVarsExp : exp -> S.t *)
(* Returns the free variables of an expression *)
and freeVarsExp (Int i) =
  | freeVarsExp (Var name) =
  | freeVarsExp (BinApp(_,rand1,rand2)) =

  | freeVarsExp (Bind(name,defn,body)) =



(* val freeVarsExps : exp list -> S.t *)
(* Returns the free variables of a list of expressions *)
and freeVarsExps exps =
                                                        )

(* val varCheck : pgm -> bool *)
and varCheck pgm = S.isEmpty (freeVarsPgm pgm)
```


Environments bind names to values

```
signature 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 map: ('a -> 'a) -> 'a env -> 'a env
  val remove : string -> 'a env -> 'a env
  val removeAll : string list -> 'a env -> 'a env
  val merge : 'a env -> 'a env -> 'a env
end

structure Env :> ENV = struct
  (* See ~wx/sml/utils/Env.sml for details *)
end
```

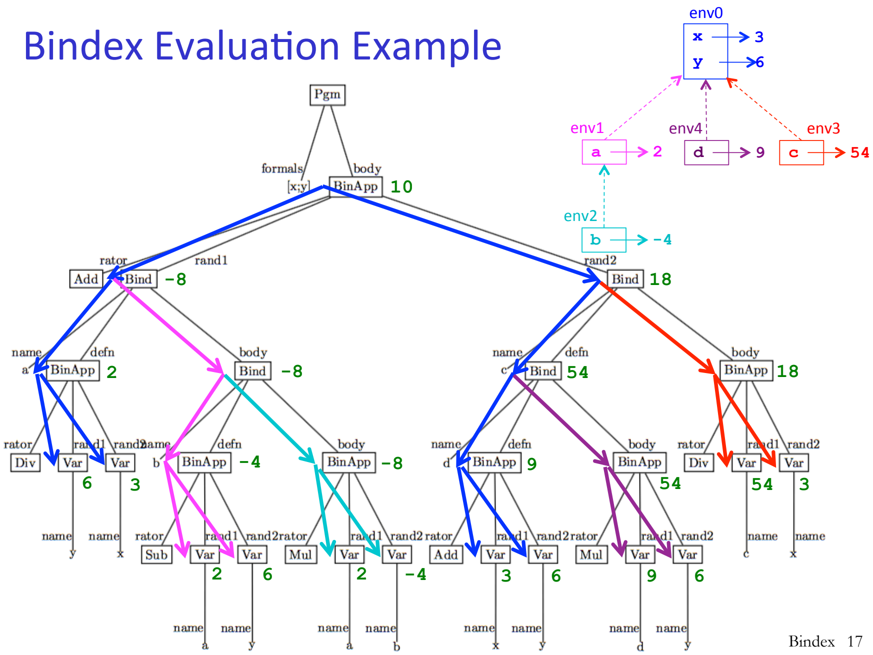

Environment Examples

```
- val env0 = Env.make ["a", "b"] [7, 3]
val env0 = - : int Env.env

- Env.lookup "a" env0;
val it = SOME 7 : int option

- Env.lookup "b" env0;
val it = SOME 3 : int option

- Env.lookup "c" env0;
val it = NONE : int option

- val env1 = Env.bind "sum" 10 env0;
val env1 = - : int Env.env

- Env.lookup "sum" env1;
val it = SOME 10 : int option

- Env.lookup "sum" env0;
val it = NONE : int option

- Env.lookup "a" env1;
val it = SOME 7 : int option

- val env2 =
    Env.bindAll ["b", "d"] [42, 17] env0;
val env2 = - : int Env.env
```




```
- Env.lookup "d" env2;
val it = SOME 17 : int option

- Env.lookup "b" env2;
val it = SOME 42 : int option

-  Env.lookup "a" env2;
val it = SOME 7 : int option
```

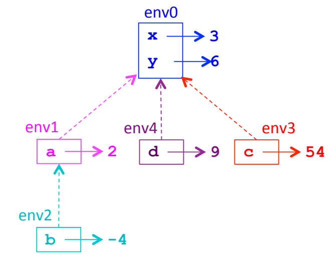
Bindex Evaluation Example



Environments follow contours!

- For each contour C_i, there is a corresponding environment env_i that binds the variables in C_i
- If C_k is nested directly inside of C_j, environment frame env_k has frame env_j as its parent

```
(bindex (x y)

  (+ (bind a (/ y x)

       (bind b (- a y)

          (* a b))      C2 )   C1

     (bind c          (bind d (+ x y)

                        (* d y))    C4

       (/ c x))     C3  ))     C0
```



Bindex Interpreter

it's your turn

```
open Bindex
exception EvalError of string

(* val run : Bindex.pgm -> int list -> int *)
fun run (Bindex(fmls,body)) ints =
  let val flen = length fmls
  val ilen = length ints
  in if flen = ilen then

     else
       raise (EvalError ("Program expected " ^ (Int.toString flen)
                         ^ " arguments but got " ^ (Int.toString ilen)))
  end

(* val eval : Bindex.exp -> int Env.env -> int *)
and eval (Int i) env =
  | eval (Var name) env =



  | eval (BinApp(rator,rand1,rand2)) env =

  | eval (Bind(name,defn,body)) env =


(* val binopToFun : Bindex.binop -> (int * int) -> int *)
(* This is unchanged from the Intex interpreter *)
```


BindexEnvInterp examples

```
- eval (stringToExp "(/ y x)") env0;
val it = 2 : int

- val env1 = Env.bind "a" 2 env0;
val env1 = - : int Env.env

- eval (stringToExp "(- a y)") env1;
val it = ~4 : int

- val env2 = Env.bind "b" ~4 env1;
val env2 = - : int Env.env

- eval (stringToExp "(* a b)") env2;
val it = ~8 : int
```

```
- eval (stringToExp "(+ x y)") env0;
val it = 9 : int

- val env4 = Env.bind "d" 9 env0;
val env4 = - : int Env.env

- eval (stringToExp "(* d y)") env4;
val it = 54 : int

- val env3 = Env.bind "c" 54 env0;
val env3 = - : int Env.env

- eval (stringToExp "(/ c x)") env3;
val it = 18 : int
```

```
- eval (stringToExp "(bind a (/ y x) (bind b (- a y) (* a b)))") env0;
val it = ~8 : int

- eval (stringToExp "(bind c (bind d (+ x y) (* d y)) (/ c x))") env0;
val it = 18 : int

- runFile "scope.bdx" [3,6];
val it = 10 : int

- run (stringToPgm "(bindex (a b) (bind sum (+ a b) (/ sum 2)))") [7,3];
val it = 5 : int
```

Extending Bindex: Sigmex = Bindex + `sigma`

(sigma I_{var} E_{lo} E_{hi} E_{body}**)**
Assume that I_{var} is a variable name, E_{lo} and E_{hi} are expressions denoting integers that are not in the scope of I_{var}, and E_{body} is an expression that is in the scope of var. Returns the sum of E_{body} evaluated at all values of the index variable I_{var} ranging from the integer value of E_{lo} up to the integer value of E_{hi}, inclusive. This sum would be expressed in traditional mathematical summation notation as:

$$\sum_{I_{var}=E_{lo}}^{E_{hi}} E_{body}$$

If the value of E_{lo} is greater than that of E_{hi}, the sum is 0.

Sigmex: `sigma` examples

Mathematical Notation	BINDEX Notation	Value
$\sum_{i=3}^{7} i$	`(sigma i 3 7 i)`	$3+4+5+6+7=25$

$\sum_{j=1+2}^{2*3} j^2$	`(sigma j (+ 1 2)`
	`(* 2 3)`
	`(* j j))`
		$3^2+4^2+5^2+6^2=86$

$\sum_{j=5}^{1} j^2$	`(sigma j 5 1 (* j j))`	0

$\sum_{i=2}^{5}\sum_{j=i}^{4} i\cdot j$	`(sigma i 2 5`
	`(sigma j i 4`
	`(* i j)))`
		$2\cdot2+2\cdot3+2\cdot4+3\cdot3+3\cdot4+4\cdot4=55$

$\sum_{i=\sum_{k=1}^{3} k^2}^{\sum_{j=1}^{5} j} i$	`(sigma i (sigma k 1 3`
	`(* k k))`
	`(sigma j 1 5 j)`
	`i)`
		$\sum_{i=(1^2+2^2+3^2)}^{1+2+3+4+5} = \sum_{i=14}^{15} = 14+15 = 29$

Sigmex: Parsing/unparsing `sigma` expression from/to S-expressions

```
datatype pgm = Sigmex of ident list * exp (* param names, body *)
     and exp = … Int, Var, BinApp, Bind from Bindex …
           | Sigma of ident * exp * exp * exp (* E_lo, E_hi, E_body *)
```

```
(* val sexpToExp : Sexp.sexp -> exp *)
and sexpToExp (Sexp.Int i) = Int i
 | … other clauses for Bindex …
 | sexpToExp (Seq [Sym "bind", Sym name, defnx, bodyx]) =
      Bind (name, sexpToExp defnx, sexpToExp bodyx)
 (* Figure out parsing of sigma below by analogy with bind above *)
 |
```

```
(* val expToSexp : exp -> Sexp.sexp *)
and expToSexp (Int i) = Sexp.Int I
  | … other clauses for Bindex …
  | expToSexp (Bind(name, defn, body)) =
      Seq [Sym "bind", Sym name, expToSexp defn, expToSexp body]
  (* Figure out unparsing of sigma below by analogy with bind above *)
  |
```


Sigmex: free vars of `sigma` expression

Free variable rule:

Bindex Phrase P	Free Variables: FV(P)
`(sigma I E_lo E_hi E_body)`	

Expressing `sigma` free variable rule in Sigmex program:

```
datatype pgm = Sigmex of var list * exp (* param names, body *)
     and exp = … Int, Var, BinApp, Bind from Bindex …
             | Sigma of var * exp * exp * exp (* E_lo, E_hi, E_body *)
```

```
(* val freeVarsExp : exp -> S.t *)
and freeVarsExp (Int i) = S.empty
 | … other clauses for Bindex …
 | freeVarsExp (Bind(name,defn,body)) =
     S.union (freeVarsExp defn)
             (S.difference (freeVarsExp body) (S.singleton name))
 | freeVarsExp (Sigma(name,lo,hi,body)) =
```

Sigmex: `sigma` evaluation

it's your turn

How should the following `sigma` **expression be evaluated in an environment** ***env1*** = a ⟼ 2, b ⟼ 3 **?**

```
(sigma j (+ a 1) (* a b) (+ a (* b j)))
```


Sigmex: `sigma` evaluation clause

it's your turn

```
datatype pgm = Sigmex of var list * exp (* param names, body *)
     and exp = … Int, Var, BinApp, Bind from Bindex …
             | Sigma of var * exp * exp * exp (* E_lo, E_hi, E_body *)
```

```
(* val eval : Sigmex.exp -> int Env.env -> int *)
and eval … other clauses from bindex …
| eval (Bind(name,defn,body)) env =
    eval body (Env.bind name (eval defn env) env)
| eval (Sigma(name, lo, hi, body)) env =
```