

Bindex: Naming, Free Variables, and Environments

SOLUTIONS



CS251 Programming Languages
Fall 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 ) ) ) ) )
```

Bindex 2

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.

Bindex 3

Review: Shadowing

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

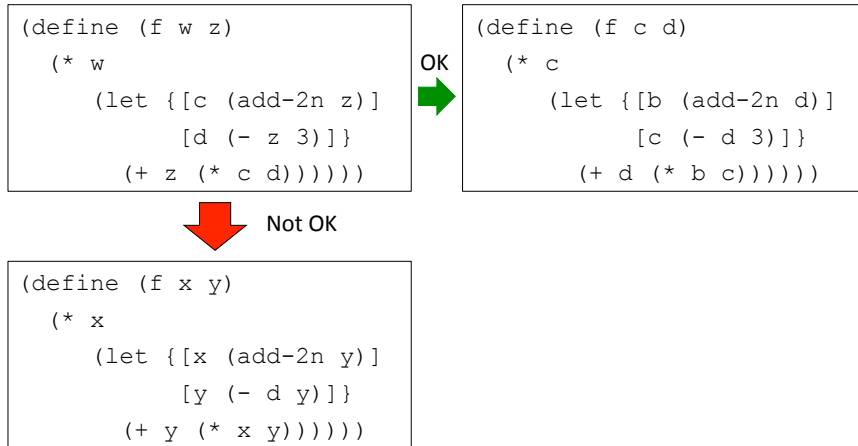
```
(let {[x 2]}  
  (- (let {[x (* x x)]}  
      (+ x 3) )  
    x ) )
```

Can't refer to outer x here.

Bindex 4

Review: Alpha-renaming

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

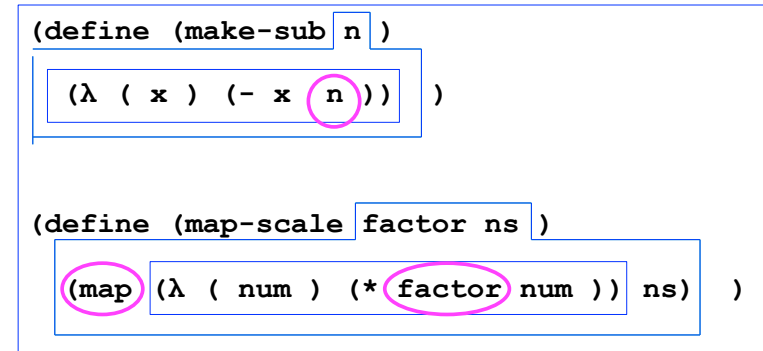


Bindex 5

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.



Bindex 6

A New Mini-Language: Bindex

Bindex adds variable names to Intex in two ways:

- 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)))
  
```

- 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 7

Bindex REPL Interpreter in action

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

```

- 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 8

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 13

Bindex: Code for handling free variables

SOLUTIONS



```
structure S = StringSetList

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

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

(* val freeVarsExps : exp list -> S.t *)
(* Returns the free variables of a list of expressions *)
and freeVarsExps exps =
  foldr (fn (s1,s2) => S.union s1 s2) S.empty (map freeVarsExp exps)

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

Bindex 14

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
```

Bindex 15

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

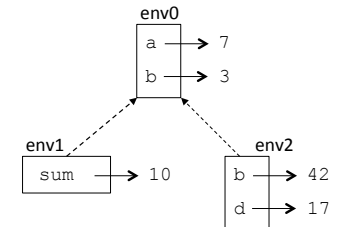
- val env1 = Env.bind "sum" 10 env0;
val env1 = - : int Env.env

- 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 16

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.

Bindex 21

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)))	$2 \cdot 2 + 2 \cdot 3 + 2 \cdot 4 + 3 \cdot 3 + 3 \cdot 4 + 4 \cdot 4 = 55$
$\sum_{i=\sum_{k=1}^5 k^2}^5 j$	(sigma i (sigma k 1 3 (* k k)) (sigma j 1 5 j) i)	$\sum_{i=(1^2+2^2+3^2)}^{1+2+3+4+5} = \sum_{i=14}^{15} = 14+15 = 29$

Bindex 22

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



```
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 *)
| sexpToExp (Seq [Sym "sigma", Sym name, lox, hix, bodyx]) =
  Sigma (name, sexpToExp lox, sexpToExp hix, sexpToExp bodyx)
```

```
(* 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 *)
| expToSexp (Sigma (name, lo, hi, body)) =
  Seq [Sym "sigma", Sym name, expToSexp lo,
  expToSexp hi, expToSexp body]
```

Bindex 23

Sigmex: free vars of sigma expression

SOLUTIONS



Free variable rule:

Bindex Phrase P	Free Variables: FV(P)
(sigma I E_lo E_hi E_body)	$FV(E_{lo}) \cup FV(E_{hi}) \cup (FV(E_{body}) - \{I\})$

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)) =
  S.union (freeVarsExp lo)
  (S.union (freeVarsExp hi)
  (S.difference (freeVarsExp body)
  (S.singleton name)))
```

Bindex 24

Sigmex: sigma evaluation SOLUTIONS



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))) # env1  
=>* (sigma j 3 6 (+ a (* b j)) # env1
```

Sum up the following:

```
(+ a (* b j)) # j ↦ 3, env1  
(+ a (* b j)) # j ↦ 4, env1  
(+ a (* b j)) # j ↦ 5, env1  
(+ a (* b j)) # j ↦ 6, env1
```

```
=>* (+ 11 (+ 14 (+ 17 (+ 20 0)))) # env1  
=>* 62 # env1
```

Bindex 25

Sigmex: sigma evaluation clause SOLUTIONS



```
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 =  
  let val vlo = eval lo env  
      val vhi = eval hi env  
      val ints = Utils.range vlo (vhi + 1)  
      val vals =  
        List.map (fn i => eval body (Env.bind name i env))  
                ints  
      in List.foldr op+ 0 vals (* could use foldl instead *)  
      end
```

Bindex 26