Part I

Curly with Arithmetic and Functions

```
; An EXP is either
;   - `NUMBER
;   - `SYMBOL
;   - `{+ EXP EXP}
;   - `{* EXP EXP}
;   - `{SYMBOL EXP}
```

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {<Symbol> <Exp>}
```

Curly with Arithmetic and Functions

```
; An EXP is either
;   - `NUMBER
;   - `SYMBOL
;   - `{+ EXP EXP}
;   - `{* EXP EXP}
;   - `{SYMBOL EXP}
```

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {<Symbol> <Exp>}
```

Backus Naur Form
(***BNF***)

Curly with Arithmetic and Functions

```
<Exp>  ::=  <Number>
        |   <Symbol>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   {<Symbol> <Exp>}
```

```
(define-type Exp
  (numE [n : Number])
  (idE [s : Symbol])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp])
  (appE [s : Symbol] [arg : Exp]))
```

Curly with Local Definitions

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]}     NEW
            <Exp>}
```

```
{let {[x {+ 1 2}]}
  {+ x x}}                 ⇒    6
```

Curly with Local Definitions

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {<Symbol> <Exp>}
         |  {let {[<Symbol> <Exp>]}     NEW
              <Exp>}
```

```
{+ {let {[x {+ 1 2}]}
     {+ x x}}
   1}                       ⇒   7
```

Curly with Local Definitions

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {<Symbol> <Exp>}
         |  {let {[<Symbol> <Exp>]}    NEW
              <Exp>}
```

```
{+ {let {[x {+ 1 2}]}
     {+ x x}}
   {let {[x {- 4 3}]}
     {+ x x}}}            ⇒   8
```

Curly with Local Definitions

```
<Exp>  ::=  <Number>
        |   <Symbol>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   {<Symbol> <Exp>}
        |   {let {[<Symbol> <Exp>]}     NEW
              <Exp>}
```

```
{+ {let {[x {+ 1 2}]}
     {+ x x}}
   {let {[y {- 4 3}]}
     {+ y y}}}              ⇒   8
```

Curly with Local Definitions

```
<Exp> ::= <Number>
        |  <Symbol>
        |  {+ <Exp> <Exp>}
        |  {* <Exp> <Exp>}
        |  {<Symbol> <Exp>}
        |  {let {[<Symbol> <Exp>]}      NEW
             <Exp>}
```

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 3}]}
    {+ x x}}}              ⇒   2
```

Curly with Local Definitions

```
<Exp> ::= <Number>
        |  <Symbol>
        |  {+ <Exp> <Exp>}
        |  {* <Exp> <Exp>}
        |  {<Symbol> <Exp>}
        |  {let {[<Symbol> <Exp>]}     NEW
             <Exp>}
```

```
{let {[x {+ 1 2}]}
  {let {[y {- 4 3}]}
    {+ x x}}}              ⇒   6
```

Curly with Local Definitions

```
<Exp>  ::=  <Number>
        |   <Symbol>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   {<Symbol> <Exp>}
        |   {let {[<Symbol> <Exp>]}   NEW
              <Exp>}
```

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 x}]}
    {+ x x}}}           ⇒   2
```

Curly with Local Definitions

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]}     NEW
            <Exp>}
```

Curly with Local Definitions

```
<Exp>  ::=  <Number>
        |   <Symbol>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   {<Symbol> <Exp>}
        |   {let {[<Symbol> <Exp>]}     NEW
              <Exp>}
```

```
(define-type Exp
  (numE [n : Number])
  (idE [s : Symbol])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp])
  (appE [s : Symbol] [arg : Exp])
  (letE [n : Symbol] [rhs : Exp]
        [body : Exp]))
```

Part 2

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;        EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
                                        (second
                                         (s-exp->list s))
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
`{[SYMBOL EXP]}
```

```
(second
 (s-exp->list s))
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;        EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(first
 (s-exp->list (second
                (s-exp->list s))))
```

```
`{[SYMBOL EXP]}
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
`[SYMBOL EXP]
```

```
(first
 (s-exp->list (second
               (s-exp->list s))))
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(let ([bs (s-exp->list (first
                         (s-exp->list (second
                                       (s-exp->list s)))))])




                                        )
```

`[SYMBOL EXP]

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;        EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(let ([bs (s-exp->list (first
                        (s-exp->list (second
                                      (s-exp->list s)))))])
    (list `SYMBOL `EXP)

                                                    )
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]}
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(let ([bs (s-exp->list (first
                        (s-exp->list (second
             `SYMBOL                  (s-exp->list s)))))])
                       (first bs)
               (second bs)
        `EXP   (third (s-exp->list s))  )
        `EXP
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(let ([bs (s-exp->list (first
                         (s-exp->list (second
                                       (s-exp->list s)))))])
 SYMBOL
       (s-exp->symbol (first bs))
             (second bs)
  `EXP       (third (s-exp->list s))   )
  `EXP
```

Parsing `let`

```
; An EXP is either ...
; - `{let {[SYMBOL EXP]})
;         EXP}
```

```
(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
```

```
(let ([bs (s-exp->list (first
                         (s-exp->list (second
                                        (s-exp->list s)))))])
  (letE (s-exp->symbol (first bs))
        (parse (second bs))
        (parse (third (s-exp->list s)))))
```

Part 3

Substitution

```
(interp (parse `{let {[x 8]}
                  {+ x x}})
        ....)

⇒  (interp (subst (parse `8)
                  'x
                  (parse `{+ x x}))
           ....)
```

Sustitutions and Local Binding

```
; 10 for x in  {let {[y 17]} x} ⇒ {let {[y 17]} 10}
(test (subst (numE 10) 'x (letE 'y (numE 17) (idE 'x)))
      (letE 'y (numE 17) (numE 10)))

; 10 for x in  {let {[y x]} y} ⇒ {let {[y 10]} y}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'y)))
      (letE 'y (numE 10) (idE 'y)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))

; 10 for x in  {let {[x x]} x} ⇒ {let {[x 10]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'x) (idE 'x)))
      (letE 'x (numE 10) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     ....]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE ....)]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE n
           ....)]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE n
           (subst what for rhs)
           ....)]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE n
           (subst what for rhs)
           (if (symbol=? n for)
               ....
               ....))]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE n
           (subst what for rhs)
           (if (symbol=? n for)
               body
               ....))]))
```

```
; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
(test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
      (letE 'y (numE 10) (numE 10)))

; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
      (letE 'x (idE 'y) (idE 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : Exp] [for : Symbol] [in : Exp])
  (type-case Exp in
    ....
    [(letE n rhs body)
     (letE n
           (subst what for rhs)
           (if (symbol=? n for)
               body
               (subst what for body)))]))


 ; 10 for x in  {let {[y x]} x} ⇒ {let {[y 10]} 10}
 (test (subst (numE 10) 'x (letE 'y (idE 'x) (idE 'x)))
       (letE 'y (numE 10) (numE 10)))

 ; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
 (test (subst (numE 10) 'x (letE 'x (idE 'y) (idE 'x)))
       (letE 'x (idE 'y) (idE 'x)))
```

Part 4

Cost of Substitution

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}})
```

$\Rightarrow$

```
(interp {let {[y 2]}
          {+ 100 {+ 99 {+ 98 ... {+ y 1}}}}})
```

$\Rightarrow$

```
(interp {+ 100 {+ 99 {+ 98 ... {+ 2 1}}}})
```

With **n** variables, evaluation will take O(**n**^{2}) time!

Deferring Substitution

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}})
```

```
⇒
```

x = 1

```
(interp {let {[y 2]}
          {+ 100 {+ 99 {+ 98 ... {+ y x}}}}})
```

```
⇒
```

y = 2   x = 1

```
(interp {+ 100 {+ 99 {+ 98 ... {+ y x}}}})
```

```
⇒ ... ⇒
```

y = 2   x = 1

```
(interp y)
```

Deferring Substitution with the Same Identifier

```
(interp {let {[x 1]}
          {let {[x 2]}
            x}})
```

⇒

`x = 1`

```
(interp {let {[x 2]}
          x})
```

⇒

`x = 2   x = 1`

```
(interp x)
```

Always add to start, then always check from start

Part 5

Representing Deferred Substitution: Environments

Change

```
interp : (Exp (Listof Func-Defn) -> Number)
```

to

```
interp : (Exp Env (Listof Func-Defn) -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

```
mt-env
```

Representing Deferred Substitution: Environments

Change

```
interp : (Exp (Listof Func-Defn) -> Number)
```

to

```
interp : (Exp Env (Listof Func-Defn) -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

```
x = 1     (extend-env (bind 'x 1)
                      mt-env)
```

Representing Deferred Substitution: Environments

Change

```
interp : (Exp (Listof Func-Defn) -> Number)
```

to

```
interp : (Exp Env (Listof Func-Defn) -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

```
y = 2  x = 1
```

```
(extend-env (bind 'y 2)
            (extend-env (bind 'x 1)
                        mt-env))
```

Environments

```
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))

(define-type-alias Env (Listof Binding))

(define mt-env empty)
(define extend-env cons)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  ....)
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty ....]
    [(cons b rst-env) ....]))
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....]))
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....
                                                  b

                                  (lookup n rst-env)   ]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....
                          (symbol=? n (bind-name b))

                               (lookup n rst-env)   ]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) (cond
                        [(symbol=? n (bind-name b))
                         (bind-val b)]
                        [else (lookup n rst-env)])]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Part 6

Interp with Environments

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}}}
        mt-env)

⇒ (interp {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}
          (extend-env (bind 'x 1) mt-env))

⇒ (interp {+ 100 {+ 99 {+ 98 ... {+ y x}}}}
          (extend-env (bind 'y 2)
                      (extend-env (bind 'x 1)
                                  mt-env)))
⇒ ...
⇒ (interp y (extend-env (bind 'y 2)
                        (extend-env (bind 'x 1)
                                    mt-env)))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) ...]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) ...]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) ...
                                ...
                                 ...     (interp rhs env defs)
                                ...
                        ...          ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                     ...)]
    [(letE n rhs body) ...
                                ...
                                 (bind n (interp rhs env defs))
                                ...
                  ...                ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) ...
                                 (extend-env
                                  (bind n (interp rhs env defs))
                                  env)
                          ...           ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp                    terp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...)]
                 n rhs body) (interp body
                                     (extend-env
                                      (bind n (interp rhs env defs))
                                      env)
                                     defs)]))
```

```
{define {f x}
  {* x x}}
```

```
{f {+ 1 2}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp                  terp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...
                            ...
                    ...      ...
                            ...    (interp arg env defs)

                                  )]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

```
{define {f x}
  {* x x}}
```

```
{f {+ 1 2}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp                 terp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    ...
                            ...
                    ...       (bind (fd-arg fd)
                            ...     (interp arg env defs))

                                         )]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

```
{define {f x}
  {* x x}}
```

```
{f {+ 1 2}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp                    terp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    (interp (fd-body fd)
                            ...
                             (bind (fd-arg fd)
                            ...    (interp arg env defs))

                            defs))]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

```
{define {f x}
  {* x x}}
```

```
{f {+ 1 2}}
```

Function Calls

```
{define {bad x} {+ x y}}

(interp {let {[y 2]}
          {bad 10}} )
```

$\Rightarrow$

```
                 y = 2
(interp {bad 10} )
```

$\Rightarrow$

```
               ...
(interp {+ x y} )
```

Function Calls

```
{define {bad x} {+ x y}}

(interp {let {[y 2]}
          {bad 10}} )

⇒
                      y = 2
(interp {bad 10} )

⇒
                     x = 10
(interp {+ x y} )
```

Interpreting function body starts with only one substitution

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    (interp (fd-body fd)
                            ...
                             (bind (fd-arg fd)
                            ...    (interp arg env defs))

                            defs))]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env] [defs : (Listof Func-Defn)])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env defs) (interp r env defs))]
    [(multE l r) (* (interp l env defs) (interp r env defs))]
    [(appE s arg) (local [(define fd (get-fundef s defs))]
                    (interp (fd-body fd)
                            (extend-env
                             (bind (fd-arg fd)
                                   (interp arg env defs))
                             mt-env)
                            defs))]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

Part 7

Binding Terminology

binding — where an identifier gets its meaning

```
{let {[x 5]} ....}
{define {f x} ....}
```

bound — refers to a binding

```
{let {[x 5]} .... x ....}
{define {f x} .... x ....}
```

free — does not have a binding

```
{let {[x 5]} .... y ....}
{define {f x} .... y ....}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

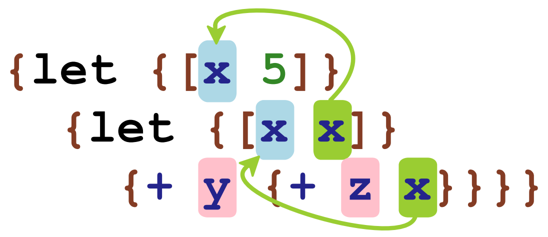
Free and Bound

```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound



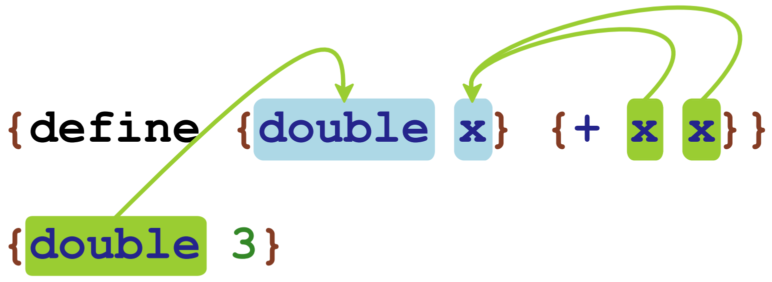
```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{define {double x} {+ x x}}

{double 3}
```

Free and Bound



```
{define {double x} {+ x x}}
{double 3}
```