Part I

Moe with Arithmetic and Functions

```
// An EXP is either
//   - 'NUMBER'
//   - 'SYMBOL'
//   - 'EXP + EXP'
//   - 'EXP * EXP'
//   - 'SYMBOL(EXP)'
```

```
<Exp> ::= <Int>
        | <Symbol>
        | <Exp> + <Exp>
        | <Exp> * <Exp>
        | <Symbol>(<Exp>)
```

Moe with Arithmetic and Functions

```
// An EXP is either
//   - 'NUMBER'
//   - 'SYMBOL'
//   - 'EXP + EXP'
//   - 'EXP * EXP'
//   - 'SYMBOL(EXP)'
```

```
<Exp>  ::=  <Int>
         |  <Symbol>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>(<Exp>)
```

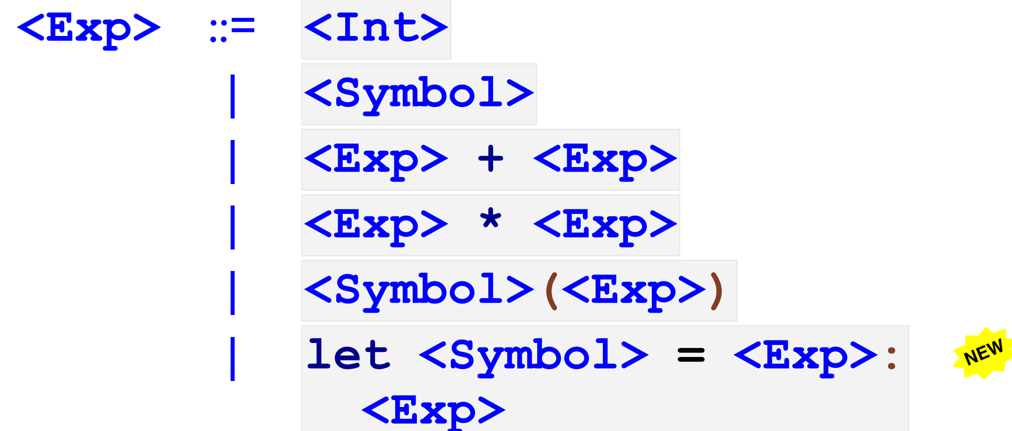
Backus Naur Form
(***BNF***)

Moe with Arithmetic and Functions

```
<Exp>  ::=  <Int>
         |  <Symbol>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>(<Exp>)
```

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp, r :: Exp)
| multE(l :: Exp, r :: Exp)
| appE(s :: Symbol, arg :: Exp)
```

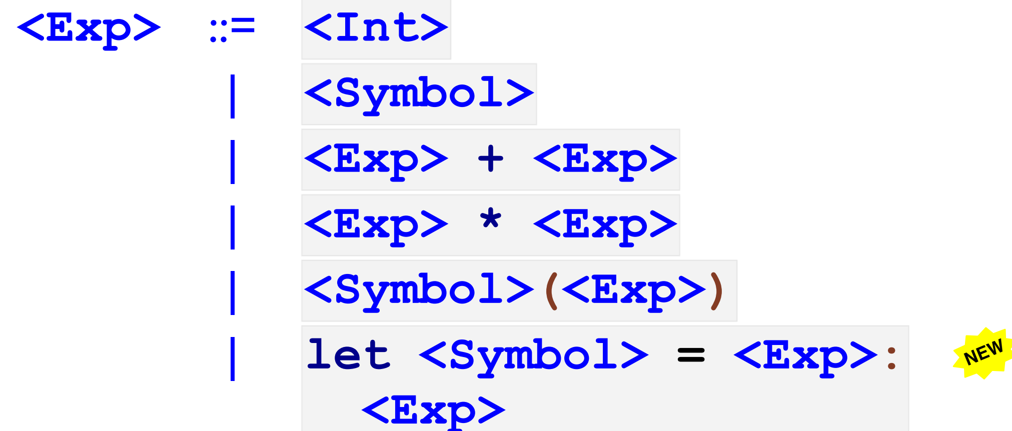
Moe with Local Definitions




```
let x = 1 + 2:
  x + x
```

⇒ 6

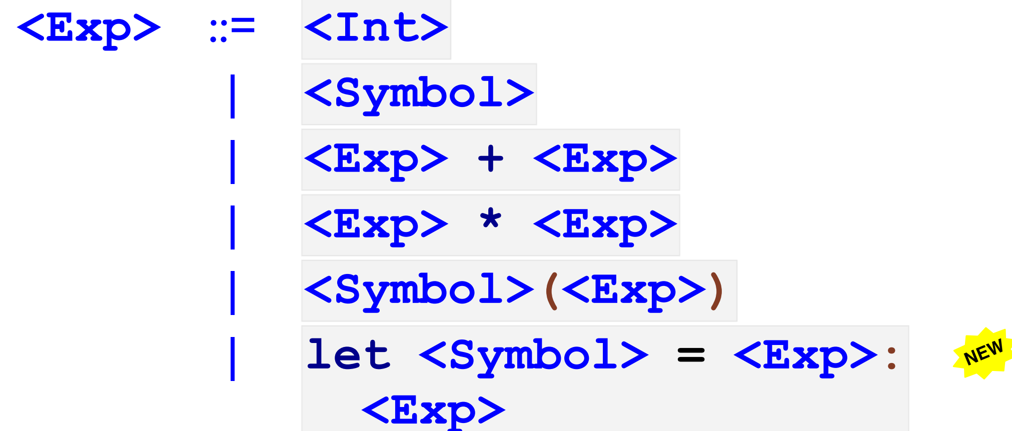
Moe with Local Definitions




```
(let x = 1 + 2:
   x + x)
  + 1
```

⇒ 7

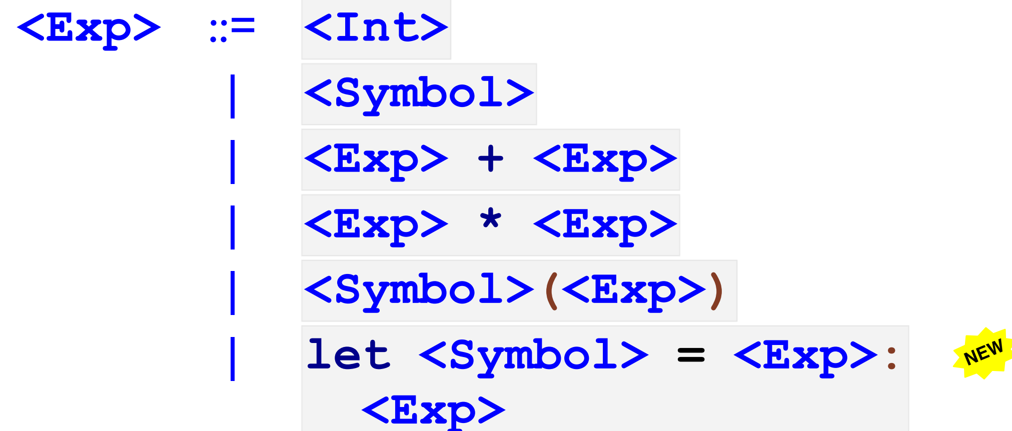
Moe with Local Definitions




```
(let x = 1 + 2:
   x + x)
  + (let x = 4 + -3:
       x + x)
```

$\Longrightarrow$ 8

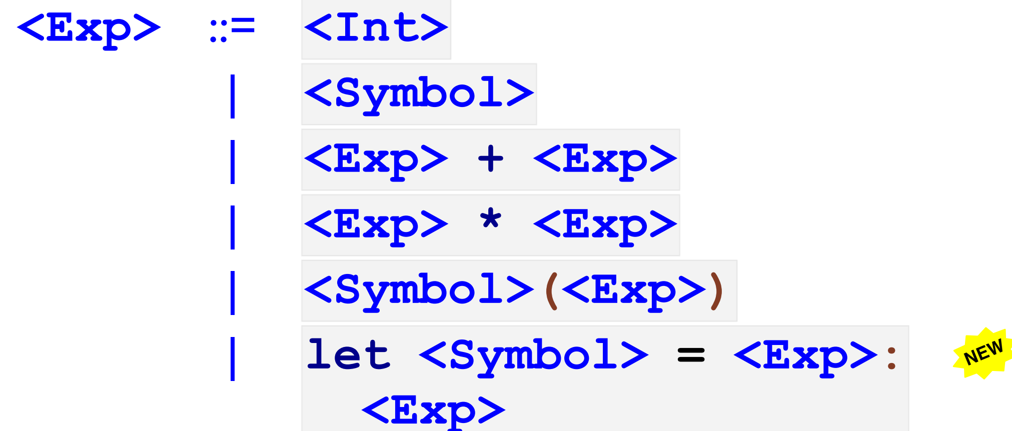
Moe with Local Definitions




```
(let x = 1 + 2:
   x + x)
  + (let y = 4 + -3:
       y + y)
```

$\Rightarrow$ 8

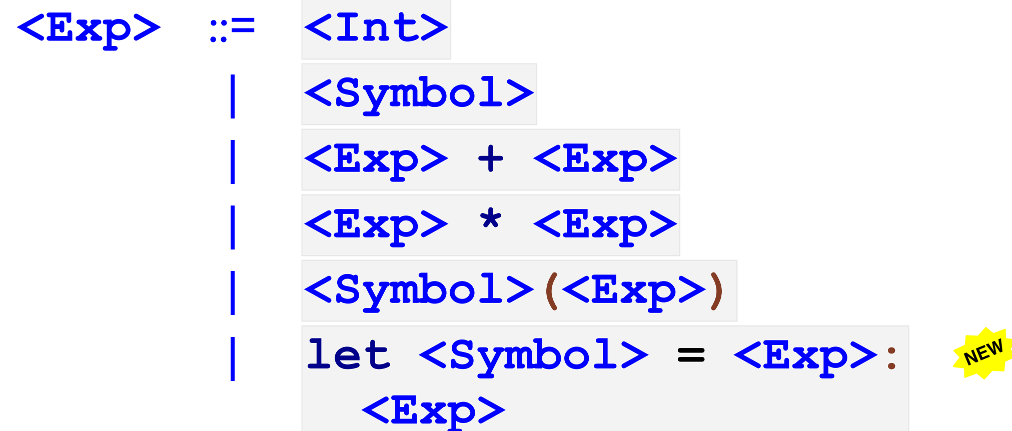
Moe with Local Definitions




```
let x = 1 + 2:
  let x = 4 + -3:
    x + x
```

$\Rightarrow$ 2

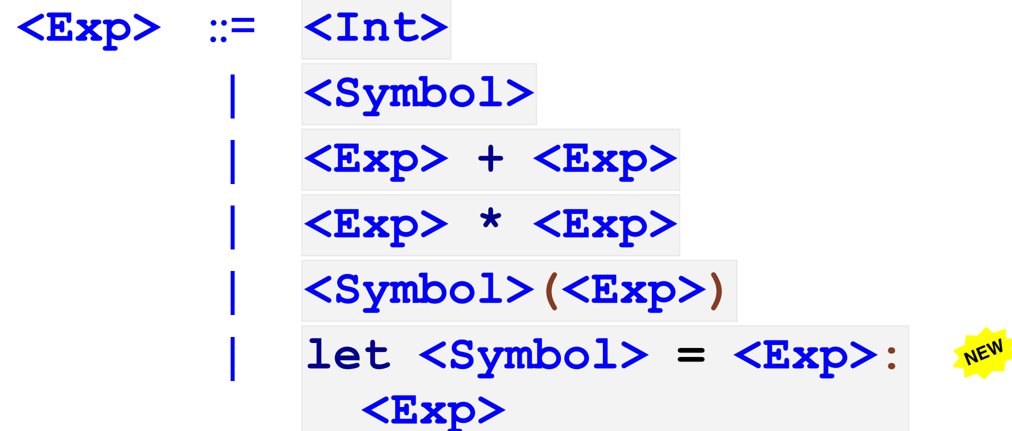
Moe with Local Definitions




```
let x = 1 + 2:
  let y = 4 + -3:
    x + x
```

$\Rightarrow$ 6

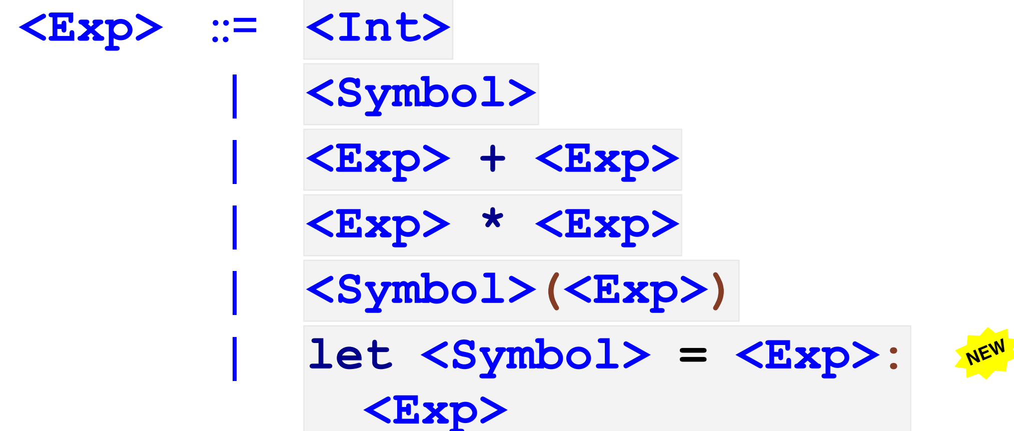
Moe with Local Definitions



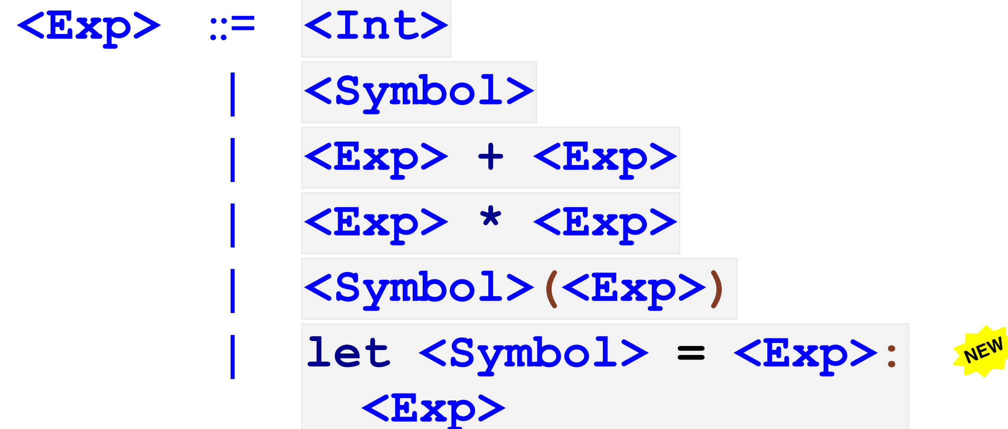

```
let x = 1 + 2:
  let x = x + -4:
    x + x
```

$\Rightarrow$ -2

Moe with Local Definitions



Moe with Local Definitions




```
type Exp
| intE(n :: Int)
| 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'

match s
| 'let $id = $rhs: $body':

                  ....

| ....
```

Parsing `let`

```
// An EXP is either ...
// - 'let SYMBOL = EXP:
//       EXP'

match s
| 'let $id = $rhs: $body':
                                  id
                 rhs
                 body
| ....
```

Parsing `let`

```
// An EXP is either ...
// - 'let SYMBOL = EXP:
//        EXP'

match s
| 'let $id = $rhs: $body':
    letE(syntax_to_symbol(id),
         parse(rhs),
         parse(body))
| ....
```

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
check: subst(intE(10), #'x, letE(#'y, intE(17), idE(#'x)))
       ~is letE(#'y, intE(17), intE(10))

// 10 for x in  let y = x: y ⇒ let y = 10: y
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'y)))
       ~is letE(#'y, intE(10), idE(#'y))

// 10 for x in  let x = y: x ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))

// 10 for x in  let x = x: x ⇒ let x = 10: x
check: subst(intE(10), #'x, letE(#'x, idE(#'x), idE(#'x)))
       ~is letE(#'x, intE(10), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      ....
```

```
// 10 for x in  let y = x: x  ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(....)
```

```
// 10 for x in  let y = x: x  ⇒  let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒  let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(n,
           ....)
```

```
// 10 for x in  let y = x: x  ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(n,
           subst(what, for, rhs),
           ....)
```

```
// 10 for x in  let y = x: x  ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(n,
           subst(what, for, rhs),
           if n == for
           | ....
           | ....)
```

```
// 10 for x in let y = x: x ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in let x = y: x ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(n,
           subst(what, for, rhs),
           if n == for
           | body
           | ....)
```

```
// 10 for x in  let y = x: x  ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Sustitutions and Local Binding

```
fun subst(what :: Exp, for :: Symbol, in :: Exp):
  match in
  | ....
  | letE(n, rhs, body):
      letE(n,
           subst(what, for, rhs),
           if n == for
           | body
           | subst(what, for, body))
```

```
// 10 for x in  let y = x: x  ⇒ let y = 10: 10
check: subst(intE(10), #'x, letE(#'y, idE(#'x), idE(#'x)))
       ~is letE(#'y, intE(10), intE(10))

// 10 for x in  let x = y: x  ⇒ let x = y: x
check: subst(intE(10), #'x, letE(#'x, idE(#'y), idE(#'x)))
       ~is letE(#'x, idE(#'y), idE(#'x))
```

Part 3a

Notation

```
1 + 2
```

Notation

```
1 + 2
```

=

```
parse('1 + 2')
```

```
interp( 1 + 2 )
```

Notation

```
1 + 2
```

=

```
parse('1 + 2')
```

```
interp( 1 + 2 )
```

=

```
interp(parse('1 + 2'))
```

Part 4

Cost of Substitution

```
interp(let x = 1:
         let y = 2:
           100 + 99 + 98 + ... + y + x)
```

⇒

```
interp(let y = 2:
         100 + 99 + 98 + ... + y + 1)
```

⇒

```
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 )
```

⇒

Environment: x = 1

```
interp( let y = 2:
          100 + 99 + 98 + ... + y + x )
```

⇒

Environment: y = 2   x = 1

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

⇒ .... ⇒

Environment: 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(FunDef)) -> Int
```

to

```
interp :: (Exp, Env, Listof(FunDef)) -> Int
```

```
mt_env :: Env
extend_env :: (Binding, Env) -> Env
bind :: (Symbol, Int) -> Binding
lookup :: (Symbol, Env) -> Int
```

Representing Deferred Substitution: Environments

Change

```
interp :: (Exp, Listof(FunDef)) -> Int
```

to

```
interp :: (Exp, Env, Listof(FunDef)) -> Int
```

```
mt_env :: Env
extend_env :: (Binding, Env) -> Env
bind :: (Symbol, Int) -> Binding
lookup :: (Symbol, Env) -> Int
```

Representing Deferred Substitution: Environments

Change

```
interp :: (Exp, Listof(FunDef)) -> Int
```

to

```
interp :: (Exp, Env, Listof(FunDef)) -> Int
```

```
mt_env :: Env
extend_env :: (Binding, Env) -> Env
bind :: (Symbol, Int) -> Binding
lookup :: (Symbol, Env) -> Int
```

y = 2 x = 1

```
extend_env(bind(#'y, 2),
           extend_env(bind(#'x, 1),
                      mt_env))
```

Environments

```
type Binding
| bind(name :: Symbol,
       val :: Int)

type Env = Listof(Binding)

def mt_env = []
def extend_env = cons
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  ....
```

```
check: lookup(#'x, mt_env)
         ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
         ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
         ~is 2
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  match env
  | []: ....
  | cons(b, rst_env): ....
```

```
check: lookup(#'x, mt_env)
       ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
       ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
       ~is 2
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): ....
```

```
check: lookup(#'x, mt_env)
       ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
       ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
       ~is 2
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): ....
                                           b

                                  lookup(n, rst_env)


check: lookup(#'x, mt_env)
       ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
       ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
       ~is 2
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): ....
                        n == bind.name(b)

                               lookup(n, rst_env)


check: lookup(#'x, mt_env)
       ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
       ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
       ~is 2
```

Environment Lookup

```
fun lookup(n :: Symbol, env :: Env) :: Int:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): cond
                      | n == bind.name(b):
                          bind.val(b)
                      | ~else: lookup(n, rst_env)


check: lookup(#'x, mt_env)
       ~raises "free variable"
check: lookup(#'x, extend_env(bind(#'x, 1), mt_env))
       ~is 1
check: lookup(#'x, extend_env(bind(#'y, 1),
                              extend_env(bind(#'x, 2), mt_env)))
       ~is 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

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun fd: get_fundef(s, defs)
                    ....
  | letE(n, rhs, body): ....
```

Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun fd: get_fundef(s, defs)
                    ....
  | letE(n, rhs, body): ....
```

Interp with Environments

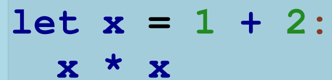
```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(n): n
  | idE(s): lookup(s, env)
  | plusE(l, r): interp(l, env, defs) + interp(r, env, defs)
  | multE(l, r): interp(l, env, defs) * interp(r, env, defs)
  | appE(s, arg): block:
                    fun fd: get_fundef(s, defs)
                    ....
  | letE(n, rhs, body): ....
```

```
let x = 1 + 2:
  x * x
```

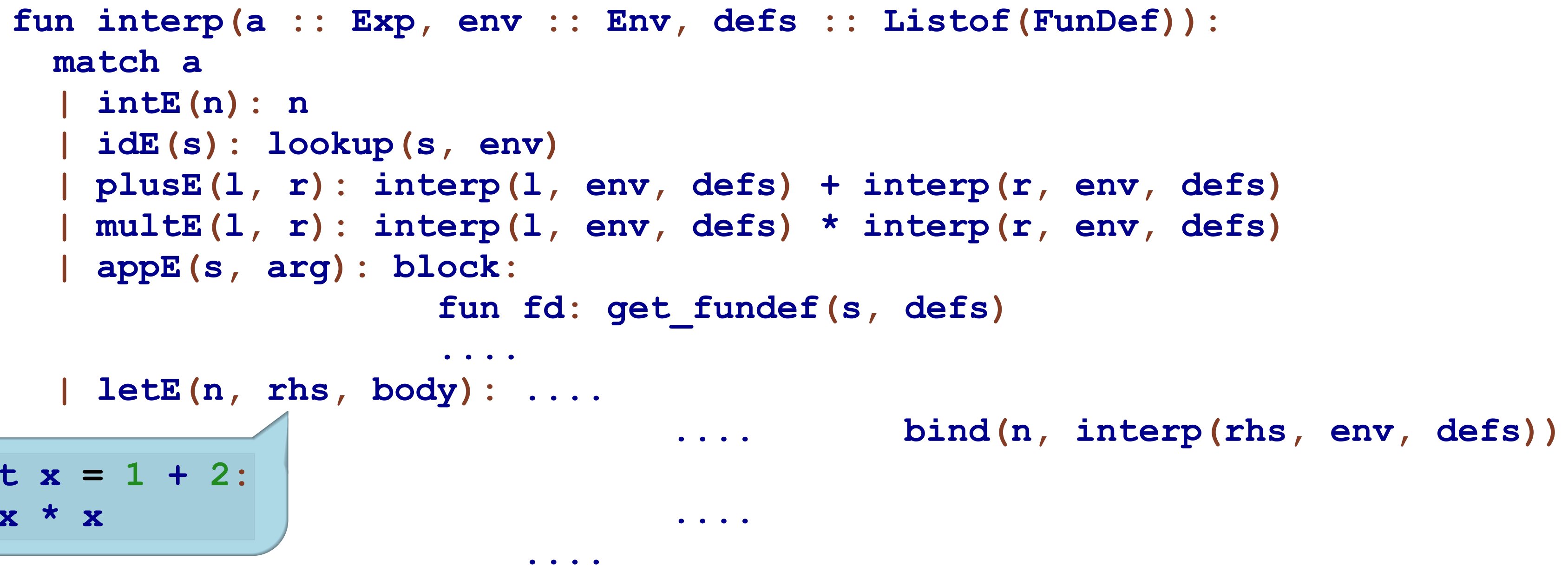
Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun fd: get_fundef(s, defs)
                    ....
  | letE(n, rhs, body): ....
                             ....        ....    interp(rhs, env, defs)

                             ....
                        ....
```



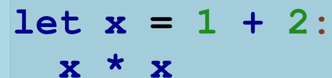
Interp with Environments



Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun fd: get_fundef(s, defs)
                    ....
  | letE(n, rhs, body): ....
                           extend_env(bind(n, interp(rhs, env, defs)),
                                      env)

                        ....
```



Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun 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

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(n): n
  | idE(s): lookup(s, env)
  | plusE(l, r): in            defs) + interp(r, env, defs)
  | multE(l, r): in            defs) * interp(r, env, defs)
  | appE(s, arg): block:
                   fun fd: get_fundef(s, defs)
                   ....
  | letE(n, rhs, body): interp(body,
                               extend_env(bind(n, interp(rhs, env, defs)),
                                          env),
                               defs)
```

```
fun f(x):
  x * x
```

```
f(1 + 2)
```

Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(n): n
  | idE(s): lookup(s, env)
  | plusE(l, r): in          defs) + interp(r, env, defs)
  | multE(l, r): in          defs) * interp(r, env, defs)
  | appE(s, arg): block:
                  fun 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)
```

```
fun f(x):
  x * x
```

```
f(1 + 2)
```

Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun 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)
```

```
fun f(x):
  x * x
```

```
f(1 + 2)
```

Interp with Environments

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(n): n
  | idE(s): lookup(s, env)
  | plusE(l, r): in    defs) + interp(r, env, defs)
  | multE(l, r): in    defs) * interp(r, env, defs)
  | appE(s, arg): block:
                   fun 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)
```

```
fun f(x):
  x * x
```

```
f(1 + 2)
```

Function Calls

```
fun bad(x): x + y
```

```
interp( let y = 2:
          bad(10) )
```

⇒

```
interp( bad(10) )
```

y = 2

⇒

```
interp( x + y )
```

...

Function Calls

```
fun 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

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun 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

```
fun interp(a :: Exp, env :: Env, defs :: Listof(FunDef)):
  match a
  | intE(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): block:
                    fun 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: ....
```

```
fun f(x): ....
```

bound — refers to a binding

```
let x = 5: .... x ....
```

```
fun f(x): .... x ....
```

free — does not have a binding

```
let x = 5: .... y ....
```

```
fun 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
```

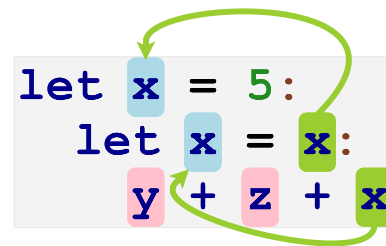
Free and Bound

```
let x = 5:
  let x = x:
    y + z + x
```

Free and Bound

```
let x = 5:
    let x = x:
        y + z + x
```

Free and Bound



Free and Bound

```
fun double(x): x + x

double(3)
```

Free and Bound