UMass Boston Computer Science

CS450 High Level Languages

Variables, Environments, and Scoping

Thursday, April 17, 2025

Logistics

- HW 10 out
  - due: Tues 4/22 11am EST

Last Time

The "CS450" Programming Lang!

Programmer writes:

```
;; A Program is one of:
;; - Atom
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
;; - `(~= ,Program ,Program)
;; - `(iffy ,Program ,Program ,Program)
```

```
;; An Atom is one of:
;; - Number
;; - 450Bool
;; - String
```

```
;; A 450Bool is either:
;; - '450true
;; - '450false
```

Last Time

The "CS450" Programming Lang!

Programmer writes:

Next Feature: **Variables?**

```
;; A Program is one of:
;; - Atom
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
;; - `(~= ,Program ,Program)
;; - `(iffy ,Program ,Program ,Program)
```

parse

```
;; An AST is one of:
;; - …
;; - (mk-add AST AST)
;; - …


;; …
(struct add [lft rgt])
;; …
```

"eval450"

```
;; A Result is one of:
;; - Number
;; - Boolean
;; - String
;; - NaN
```

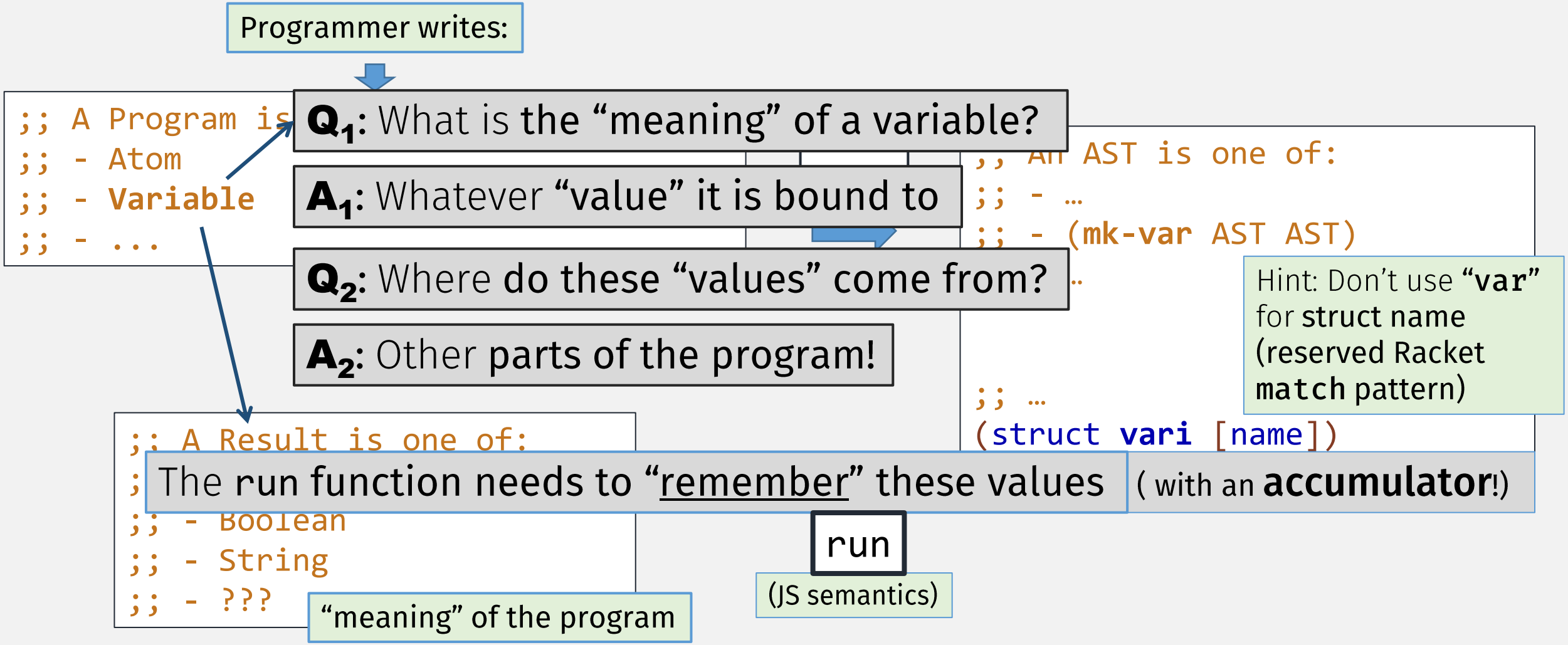
"meaning" of the program

run

(JS semantics)

(don't need for HW10)

Adding Variables



Guess Who's Back!

Design Recipe For Accumulator Functions

When a function needs **"extra information":**

1. *Specify* **accumulator:**
   - Name
   - Signature
   - Invariant
2. *Define* **internal "helper" fn** with **extra accumulator arg**

(Helper fn does not need extra description, statement, or examples, if they are the same …)

3. *Call* **"helper" fn , with initial accumulator value, from original fn**

run, with an accumulator

```
;; run: AST -> Result
;; Computes result of running a CS450 Lang program AST
(define (run p)
  ;; accumulator acc : Environment
  ;; invariant: Contains variable values … currently in-scope
  (define (run/acc p acc)
    (match p
      [(num n) n]
      [(add x y) (450+ (run/acc x) (run/acc y))]))
 (run/acc p  ???  ))
```

Environments

- A **data structure** that **“associates” two things** (`var`, `val`) **together**
  - E.g., **maps, hashes,** etc
  - For simplicity, let’s use **list-of-pairs**

```
;; An Environment is one of:
;; - empty
;; - (cons (list Var Result) Environment)
```

```
;; interpretation: a runtime environment for
;; (i.e., gives meaning to) cs450-lang variables
```

```
;; if there are duplicates,
;; vars at front of list shadow those in back
```

Environments

- A **data structure** that **“associates” two things** (`var`, `val`) **together**
  - E.g., **`maps`, `hashes`,** etc
  - For simplicity, let’s use **list-of-pairs**

```
;; An Environment is one of:
;; - empty
;; - (cons (list Var Result) Environment)
```

- Needed operations:
  - `env-add    : Env Var Result -> Env`
  - `env-lookup : Env Var -> Result`

Environments

```
;; An Environment is one of:
;; - empty
;; - (cons (list Var Result) Environment)
```

```
;; interpretation: a runtime environment
;; gives meaning to cs450lang variables
```

```
;; for duplicates, vars at front of
;; list shadow those in back
```

- Needed operations:
  - `env-add    : Env Var Result -> Env`
  - `env-lookup : Env Var -> Result`

Think about examples where this happens!

`env-add` examples

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
```

```
(check-equal? (env-add '() 'x 1)
              '((x 1)) )  ; add to empty
```

```
(check-equal? (env-add '((x 1)) 'y 2)
              '((y 2) (x 1)) ) ; add new var
```

```
(check-equal? (env-add '((x 1)) 'x 3)
              '((x 3) (x 1)) ) ; add shadowed var
```

```
;; for duplicates, vars at front of
;; list shadow those in back
```

Env template

2 cases

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

2 cases

```
(define (env-fn env … )
  (cond
   [(empty? env) … ]
   [else
    (match-let
        ([(cons (list x result) rest-env) env])
      … x … result … (env-fn rest-env … ) … ]))
```

2nd case extracts components of compound data

Env template

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
(define (env-fn env … )
  (cond
   [(empty? env) … ]
   [else
    (match-let
        ([(cons (list x result) rest-env) env])
      ([`((,x ,result) . ,rest-env) env])
      … x … result … (env-fn rest-env … ) … ]))
```

2 cases

cons pattern

Quasiquote pattern

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cond
   [(empty? env) … ]
   [else
    (match-let
        ([(cons (list x result) rest-env) env])
        ([`((,x ,result) . ,rest-env) env])
      … x … result …(env-add rest-env … ) … ]))
```

Start with template

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cond
   [(empty? env) ...]
   [else   … ]))
```

Examples

```
(check-equal? (env-add '() 'x 1)
              '((x 1)) ) ; add to empty
```

Base case – empty env

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cond
   [(empty? env) (cons (list new-x new-res) env)]
   [else   … ]))
```

Base case – empty env

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cond
   [(empty? env) (cons (list new-x new-res) env)]
   [else                ...]))
```

recursive case?
(non-empty env)

Examples

```
(check-equal? (env-add '((x 1)) 'y 2)
              '((y 2) (x 1)) ) ; add new var
```

```
(check-equal? (env-add '((x 1)) 'x 3)
              '((x 3) (x 1)) ) ; add shadowed var
```

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cond
   [(empty? env) (cons (list new-x new-res) env)]
   [else         (cons (list new-x new-res) env)]))
```

recursive case?
(non-empty env)

Sometimes you start with template … but don't use it!

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; env-add: Env Var Result -> Env
(define (env-add env new-x new-res)
  (cons (list new-x new-res) env))
```

Collapse similar cases

Sometimes you start with template … but don’t use it!

env-lookup examples

```
;; env-lookup: Env Var -> Result
```

```
(check-equal? (env-lookup '((y 2) (x 1)) 'x)
              1) ; no dup
```

```
(check-equal? (env-lookup '((x 2) (x 1)) 'x)
              2) ; duplicate
```

```
(check-equal? (env-lookup '() 'x)
              UNDEFINED-ERROR ) ; not found!
```

```
;; A Result is one of:
;; - Number
;; …
;; - UNDEFINED-ERROR
```

An "error" is a valid program "Result"!

… for now, just represent with special `Result` value

NOTE: we don't want Racket exception because this is a "CS450 Lang error" … Racket program runs fine!

env-lookup

```
;; env-lookup: Env Var -> Result
(define (env-lookup env target-x)
  (cond
   [(empty? env) … ]
   [else
    (match-let
        ([`((,x ,res) . ,rest-env) env])
      … x … res … (env-lookup rest-env … ) … ]))
```

TEMPLATE!

env-lookup: empty (error) case

```
;; env-lookup: Env Var -> Result
(define (env-lookup env target-x)
  (cond
   [(empty? env) UNDEFINED-ERROR]
   [else
    … ]))
```

env-lookup: non-empty case

```
;; env-lookup: Env Var -> Result
(define (env-lookup env target-x)
  (cond
   [(empty? env) UNDEFINED-ERROR]
   [else                                  Extract the pieces
    (match-let
        ([`((,x ,res) . ,rest-env) env])
      … x … res … (env-lookup rest-env … ) … ]))
```

env-lookup: non-empty case

```
;; env-lookup: Env Var -> Result
(define (env-lookup env target-x)
  (cond
   [(empty? env) UNDEFINED-ERROR]
   [else
    (match-let
        ([`((,x ,res) . ,rest-env) env])
      (if (var=? x target-x)
          res
          … (env-lookup rest-env … ) … ]))
```

Found target-x

env-lookup: non-empty case

```
;; env-lookup: Env Var -> Result
(define (env-lookup env target-x)
  (cond
   [(empty? env) UNDEFINED-ERROR]
   [else
    (match-let
        ([`((,x ,res) . ,rest-env) env])
      (if (var=? x target-x)
          res
          (env-lookup rest-env target-x)))]))
```

Else, recursive call with remaining env

run, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
     [(num n) n]
     [(add x y) (450+ (run/env x) (run/env y))]))
 (run/env p  ???  ))
```

run, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) … (env-add env x (run/env e env)) …]
      …   ))
 (run/env p  ???  ))
```

run, with an Environment accumulator

TODO:
- When are variables “added” to environment
- What is initial environment?

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind ??? body) … (env-add env x ??? e env)) …]
      …  ))
 (run/env p ???  ))
```

Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - ?????
```

Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - ...
```

(like "`let`" in other langs)

Interlude: What is a "binding"?

mdn web docs

"identifer" = name

"value" = "result"

In programming, a **binding** is an association of an identifier with a value. Not all bindings are variables — for example, function parameters and the binding created by the `catch (e)` block are not "variables" in the strict sense. In addition, some bindings are implicitly created by the language — for example, `this` and `new.target` in JavaScript.

A binding is mutable if it can be re-assigned, and immutable otherwise; this does *not* mean that the value it holds is immutable.

A binding is often associated with a scope. Some languages allow re-creating bindings (also called redeclaring) within the same scope, while others don't; in JavaScript, whether bindings can be redeclared depends on the construct used to create the binding.

Mutation (e.g., `set!`) not allowed in this class (so far)

`https://developer.mozilla.org/en-US/docs/Glossary/Binding`

Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - ...
```

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - …

;; …
(struct vari [name])
(struct bind [var expr body])
;; …
```

run

???

Interlude: What is a “binding”?

mdn web docs _

In programming, a **binding** is an association of an identifier with a value. Not all bindings are variables — for example, function parameters and the binding created by the `catch (e)` block are not "variables" in the strict sense. In addition, some bindings are implicitly created by the language — for example, `this` and `new.target` in JavaScript.

A binding is mutable if it can be re-assigned, and immutable otherwise; this does *not* mean that the value it holds is immutable.

A binding is often associated with a scope. Some languages allow re-creating bindings (also called redeclaring) within the same scope, while others don't; in JavaScript, whether bindings can be redeclared depends on the construct used to create the binding.

`https://developer.mozilla.org/en-US/docs/Glossary/Binding`

Bind scoping examples

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - ...
```

bind obeys "lexical" or "static" scoping

Generally accepted to be "best choice" for programming language design
(bc it's determined only by program syntax)

Different Kinds of Scope

(Perl)

- **Lexical (Static)** Scope
  - Variable value determined by **syntactic** code location

```
$a = 0;
sub foo {
  return $a;
}
```

```
sub staticScope {
  my $a = 1; # lexical (static)
  return foo();
}
```

```
print staticScope(); # 0 (from the saved global frame)
```

- **Dynamic** Scope
  - Variable value determined by **runtime** code location
  - Discouraged: violates "separation of concerns" principal

```
$b = 0;
sub bar {
  return $b;
}
```

```
sub dynamicScope {
  local $b = 1;
  return bar();
}
```

```
print dynamicScope(); # 1 (from the caller's frame)
```

Other Kinds of Scope

- JS “function scope”
  - `var` declarations
    - follow lexical scope inside functions
    - but <u>not</u> other blocks! (weird?)
  - `let` declarations
    - follow lexical scope inside functions
    - and <u>all</u> other blocks!

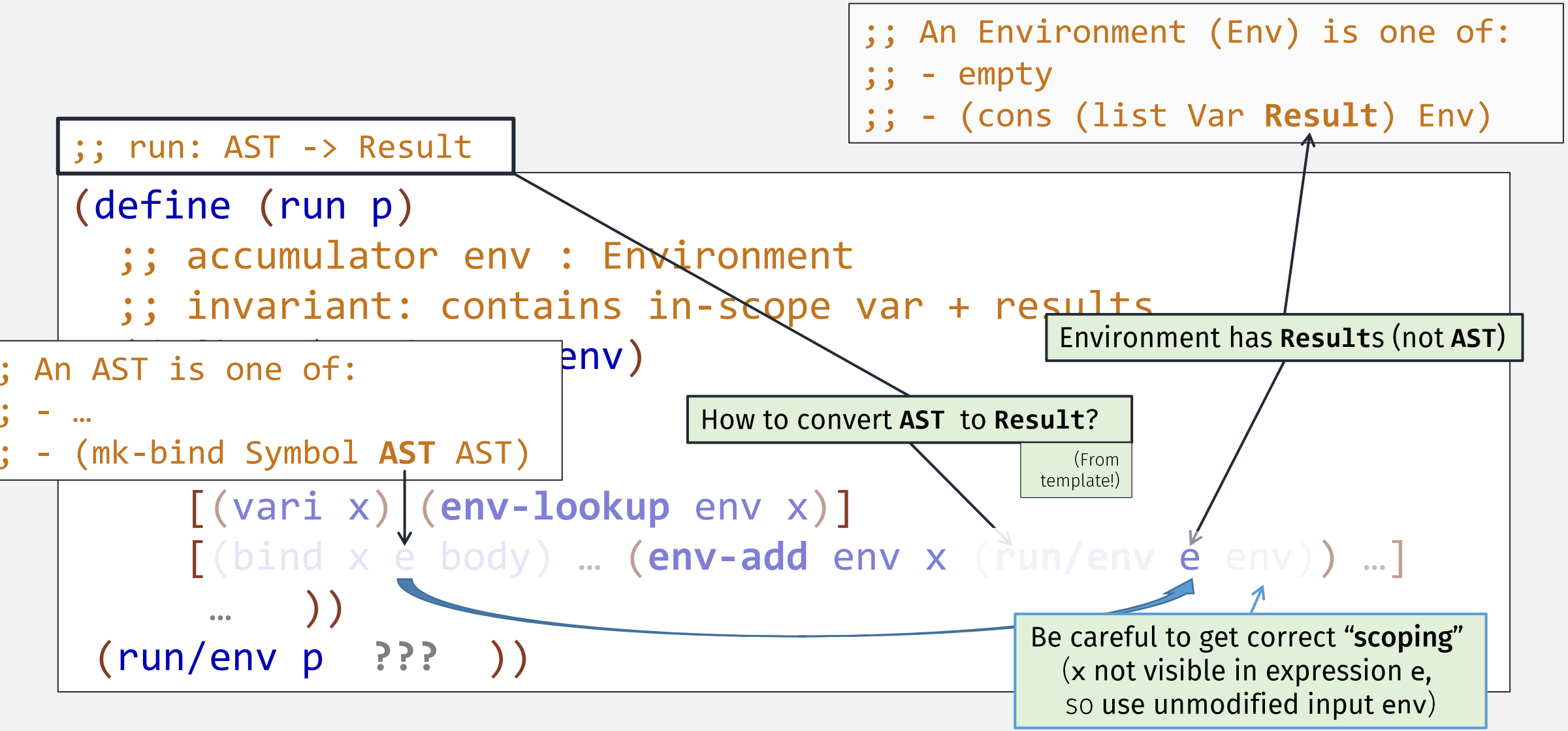
```
{
  var x = 2;
}
// x CAN be used here
```

Introduced in ES6 (2015) to fix var weirdness

```
{
  let x = 2;
}
// x can NOT be used here
```

- Global scope
  - Variables in-scope everywhere
  - Added to “initial environment” before program runs

run, with **bind**



`run`, with **`bind`**

`run` must produce **`Result`**

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env e env)
    (type-case AST e
      [(vari x) (env-lookup env x)]
      [(bind x e body) ??? (env-add env x (run/env e env)) …]
      …  ))
  (run/env p  ???  ))
```

```
; An AST is one of:
; - …
; - (mk-bind Symbol AST AST)
```

run, with **bind**

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
     [(vari x) (env-lookup env x)]
     [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …   ))
 (run/env p  ???  ))
```

(From template!)

run body with new env containing **x**

Initial Environment?

TODO:
- ~~When are variables "added" to environment~~
- What is initial environment? `empty` (for now)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …    ))
 (run/env p  ???  ))
```

`empty` ??? (for now)

Initial Environment

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

These don't need to be separate constructs

Put these into "initial" environment

Initial Environment

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

Put these into “initial” environment

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
(define INIT-ENV
`((+ ,450+)
  (× ,450*)))
```

+ variable

Maps to our “450+” function

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - (Racket) Function
```

For Program: +

Initial Environment

How do users call these functions???

```
(define INIT-ENV `((+ ,450+) (× ,450*)))
```

```
(define (run p)

  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(vari x) (lookup env x)]
     [(bind x e body) (run/e body (env-add env x (run/e e env)))]
      … ))
 (run/e p INIT-ENV   ))
```

Function Application in CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(fncall ,Program . ,List<Program>)
```

function

arguments

“rest” arg

Specifies arbitrary number of args

Function Application in CS450 Lang: Examples

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(fncall ,Program . ,List<Program>)
```

function

arguments

```
(fncall + 1 2)
```

Programmers shouldn’t need to write the explicit “fncall”

Function Application in CS450 Lang: Examples

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(,Program . ,List<Program>)
```

```
(+ 1 2)
```

Function call case (must be last, why?)

Must be careful when parsing this (not until HW 11!)

Function Application in CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(,Program . ,List<Program>)
```

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-call AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct call [fn args])
```

“Running” Function Calls

TEMPLATE: extract pieces of compound data

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-call AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct call [fn args])
```

```
(define (run p)


  (define (run/e p env)
    (match p
           …
      [(call fn args) (apply
                       (run/e fn env)
                       (map (curryr run/e env) args))]

           …
      ))
  (run/e p INIT-ENV))
```

“Running” Function Calls

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-call AST List<AST>)
```

```
(define (run p)

  (define (run/e p env)
    (match p
      …
      [(call fn args) (apply
                       (run/e fn env)
                       (map (curry ??? run/e env) args))]

      …
      ))
  (run/e p INIT-ENV))
```

TEMPLATE: recursive calls

“run” args before calling function – “call by value”

“Running” Function Calls

How do we actually run the function?

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - (Racket) Function
```

```
(define (run p)


  (define (run/e p env)
    (match p
          …
      [(call fn args) (apply ???
                             (run/e fn env)
                             (map (curryr run/e env) args))]

          …
      ))
  (run/e p INIT-ENV))
```

(this only “works” for now)

Function Application in CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(,Program . ,List<Program>)
```

Function call case (must be last)

This doesn't let users define their own functions!

Next Feature: **Lambdas?**

In-class Coding 4/17: `bind` + "call" examples

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(,Program . ,List<Program>)
```

Come up with some of your own!

```
(check-equal?
  (eval450 '(bind [x 10] x))
  10 ) ; no shadow
```

```
(check-equal?
  (eval450 '(bind [x 10] (bind [x 20] x))
  20 ) ; shadow
```

```
(check-equal?
  (eval450
    '(bind [x 10]
       (+ (bind [x 20]
            x)
          x)) ; 2nd x outof scope here
  30 )
```

```
(check-equal?
  (eval450
   '(bind [x 10]
      (bind [x (+ x 20)] ; x = 10 here
        x))) ; x = 30 here
   30 )
```