

UMass Boston Computer Science
CS450 High Level Languages (section 2)

Variables and Environments in CS450 Lang

Monday, November 18, 2024



Logistics

- HW 10 in
 - ~~due: Mon 11/18 12pm noon EST~~
- HW 11 out
 - due: Mon 11/25 12pm noon EST
- HW 12
 - out: Mon 12/25 12pm noon EST
 - due: Wed 12/4 12pm noon EST



Introducing: The “CS450” Programming Lang!

Programmer writes:

Next Feature: Variables?

;; A **450LangExpr** (Expr) is one of:
;; - Number
;; - String
;; - (list '+ Expr Expr)
;; - (list '- Expr Expr)

parse

;; An **AST** is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)

“eval450”

;; A **Result** is one of:
;; - Number
;; - String
;; - NaN

“meaning” of the program

run

(JS semantics)

(struct num [val])
(struct str [val])
(struct add [lft rgt])
(struct sub [lft rgt])

Adding Variables

;; A **Variable (Var)** is a Symbol

;; A 450LangExpr (Expr) is one of:
;; - Number
;; - String
;; - **Variable**
;; - (list '+ Expr Expr)
;; - (list '- Expr Expr)

parse

Q₁: What is the “meaning” of a variable?

A₁: Whatever “value” it is bound to

Q₂: Where do these “values” come from?

A₂: Other parts of the program!

;; A Result is
;; - Number
;; - String
;; - NaN

run

;; An AST is one of:
(num Number)
(str String)
(vari Symbol)
(add AST AST)
(sub AST AST)

(struct num [val])
(struct str [val])
(struct **vari** [name])
(struct add [lft rgt])

The run function needs to “remember” these values (with an **accumulator!**)

“meaning” of the program

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)
```

- Needed operations:

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

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

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

Think about examples where this happens!

env-add examples

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

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

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 ... (env-fn rest-env ... ) ... ]))
```

2 cases

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

Quasiquote pattern

cons "rest" 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 ... ) ... ])))
```

```
;; 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 ... ]))
```



```
;; 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)]))
```

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))
```

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
```

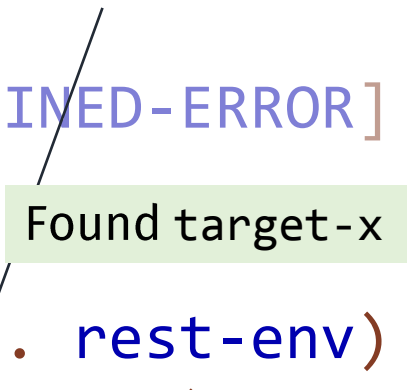
```
(define (env-lookup env target-x)
  (cond
    [(empty? env) UNDEFINED-ERROR]
    [else
     (match-let
       ([`((,x ,res) . rest-env) env])
       ... x ... res ... (env-lookup rest-env ... ) ... ]))
```

Extract the pieces

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 ... ) ... ]))
```



The diagram illustrates the execution of the `env-lookup` function in the non-empty case. A green box labeled "Found target-x" has two arrows pointing to the code. One arrow points to the `x` in the match-let binding `[`((,x ,res) . rest-env) env]`, indicating that `x` is the variable found in the environment. The other arrow points to the `x` in the `var=? x target-x` check, indicating that the function is verifying that the found variable matches the target.

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 accumulator)

```
;; run: AST -> Result
```

```
(define (run p)
```

```
;; accumulator env : Environment
```

```
;; invariant: has var+result pairs that are in-scope
```

```
(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 ??? ))
```

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

Environment has **Results** (not AST)

How to convert **AST** to **Result**?

Be careful to get correct “**scoping**”
(x not visible in expression e,
so use unmodified input env)

Bind scoping examples

```
;; A 450LangExpr (Expr) is one of:  
;; - Atom  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)
```

This is called “lexical” or “static” scoping

Generally accepted to be “best choice”
for programming language design
(it’s determined only by program syntax)

We will use this for “CS450 Lang”

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

Variable reference

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

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

Variable references

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

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)
```

(eval450-hook) needed “dynamic scope”

Other Kinds of Scope

- JS “function scope”

- var declarations
 - follow lexical scope inside functions
 - but not other blocks! (weird?)
- let declarations
 - follow lexical scope inside functions
 - and all 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 an Environment

```
;; run: AST -> Result
```

```
(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 ??? ))
```

(From
template!)

3. run body with that new environment

2. add variable x to environment

(From
template!)

1. Compute Result that
variable x represents



Initial Environment

```
(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 ??? ))
```

Initial Environment

```
;; A 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (list '+ Expr Expr)  
;; - (list '- Expr Expr)
```

These don't need to be separate constructs

Put these into "initial" environment

Initial Environment

```
;; A 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (list '+ Expr Expr)  
;; - (list '- Expr Expr)
```

Put these into “initial” environment

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

+ variable

Maps to our
“450+” function

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

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

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 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (list 'fncall Expr . List<Expr>)
```

function

arguments

“rest” arg

Specifies arbitrary number of args

Function Application in CS450 Lang: Examples

```
;; A 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (list 'fncall Expr . List<Expr>)
```

function

arguments

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

Programmers shouldn't need to write the explicit "fncall"

Function Application in CS450 Lang: Examples

```
;; A 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (cons Expr List<Expr>)
```

(+ 1 2)

No longer need “rest” arg (why?)

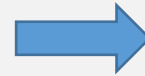
Function call case (must be last, why?)

Must be careful when parsing this (HW 11!)

Function Application in CS450 Lang

```
;; A 450LangExpr (Expr) is one of:  
;; - Number  
;; - String  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (cons Expr List<Expr>)
```

parse



```
;; An AST is one of:  
;; - ...  
;; - (vari Symbol)  
;; - (bind Symbol AST AST)  
;; - (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

```
(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))
```

```
;; An AST (AST) is one of:
;; - ...
;; - (vari Symbol)
;; - (bind Symbol AST AST)
;; - (call AST List<AST>)
```

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

“Running” Function Calls

```
(define (run p)
```

```
  (define (run/e p env)
    (match p
```

TEMPLATE: recursive calls

```
      ...
      [(call fn args) (apply
                        (run/e fn env)
                        (map (curry ??? run/e env) args))])
    ...
  ))
```

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

```
;; An AST (AST) is one of:
;; - ...
;; - (vari Symbol)
;; - (bind Symbol AST AST)
;; - (call AST List<AST>)
```

“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))])
    ...
```

(this only “works” for now)

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

Function Application in CS450 Lang

```
;; A 450LangExpr (Expr) is one of:  
;; - Atom  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (cons Expr List<Expr>)
```

Function call case (must be last)

This doesn't let users define their own functions!

Next Feature: **Lambdas?**

- Repo: [cs450f24/in-class-11-18](#)
- File: `bind-examples-<your last name>.rkt`

In-class Coding 11/18: `bind` + “`call`” examples

```
;; A 450LangExpr (Expr) is one of:  
;; - Atom  
;; - Variable  
;; - (list 'bind [Var Expr] Expr)  
;; - (cons Expr List<Expr>)
```

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 out of scope here  
  30 )
```

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