Interpreting Recursion, with Mutation!

Monday, December 4, 2023
Logistics

• HW 8 in
  • due: Sun 12/3 11:59 pm EST
  • due: Mon 12/4 11:59 am EST

• HW 9 out
  • due: Sun 12/10 11:59 pm EST
"bind" in "CS450js" Lang

;; A Variable (Var) is a Symbol

;; A 450jsExpr (Expr) is one of:
;; ...
;; - Var
;; - (list 'bind [Var Expr] Expr)
;; ...

Reference a variable binding
Create new variable binding
new binding is not in-scope here
new binding is in-scope (can be referenced) here
bind examples

;; A 450jsExpr (Expr) is one of:
;; ...
;; - Var
;; - (list `bind [Var Expr] Expr)
;; ...

(check-equal? (eval450 `(bind [x (+ x 20)] x))
  UNDEFINED-ERROR)
bind examples, with functions

;; A 450jsExpr (Expr) is one of:
;; ...
;; - Var
;; - (list (bind [Var Expr] Expr)
;;      (list 'fn List<Var> Expr)
;;      (cons Expr List<Expr>))
;; ...

(check-equal?
 (eval450
  '(bind [f (fn (x) (+ x 4))] (f 6))
  10)

f not in-scope here
(function can’t be recursive!)
"bind/rec" in "CS450js" Lang

```plaintext
;; A 450jsExpr (Expr) is one of:
;; ...
;; - (list 'bind/rec [Var Expr] Expr)
;; ...
```

- new binding is *in-scope* (can be referenced) here
- Create new variable binding
- new binding is also in-scope here!
Racket recursive function examples

\[
\text{(define (fac n)} \quad (\text{if} \ (= \ n \ 0) \quad 1 \quad (* \ n \ (\text{fac} \ (- \ n \ 1)))))
\]

\[
(\text{fac} \ 5); \Rightarrow \ 120
\]

\[
(\text{Racket})
\]

\[
\text{(letrec} \quad ([\text{fac}] \quad (\lambda \ (n)} \quad (\text{if} \ (= \ n \ 0) \quad 1 \quad (* \ n \ (\text{fac} \ (- \ n \ 1)))))) \quad \text{[Racket]}
\]

\[
(\text{(letrec} \quad ([\text{fac}] \quad (\lambda \ (n)} \quad (\text{if} \ (= \ n \ 0) \quad 1 \quad (* \ n \ (\text{fac} \ (- \ n \ 1))))]) \quad \text{(Racket)}
\]

\[
(\text{fac} \ 5); \Rightarrow \ 120
\]

Recursive call
bind/rec examples

;; A 450jsExpr (Expr) is one of:
;; ...
;; - (list 'bind/rec [Var Expr] Expr)
;; - (list 'iffy Expr Expr Expr)
;; ...

JS “truthy” if(hw7)

(letrec
  ([fac
     (λ (n)
       (if (= n 0)
         1
         (* n (fac (- n 1))))])
   (fac 5)) ; => 120

RACKET

 Equivalent to ...

(bind/rec
  [fac
   (fn (n)
     (iffy n
       (* n (fac (- n 1)))
       1))]
   (fac 5)) ; => 120

CS450JSLANG

Zero is “truthy” false (hw7)
Running **bind/rec** programs

;; A 450jsExpr (Expr) is one of:
;; ...
;; - (list `bind/rec [Var Expr] Expr)
;; ...

```plaintext
parse

;; A 450jsAST (AST) is one of:
;; ...
;; - (recb Symbol AST AST)
;; ...
(struct recb [var expr body])
```

```plaintext
run

;; A 450jsResult (Result) is a:
;; - ...
```
Running `bind/rec` programs

```plaintext
;; run: AST -> Result
;; Computes result of running CS450js AST
```

```
;; A 450jsAST (AST) is one of:
;; ... 
;; - (recb Symbol AST AST)
;; ... 
;; (struct recb [var expr body])
```

```
run
```

```
;; A 450jsResult (Result) is a:
;; - ...
```
Running **bind/rec**

;;; run: AST -> Result

```scheme
(define (run p)
  (define (run/e p env)
    (match p
      ...
      [(recb x e body) ?? x ?? e ?? body )]
    (run/e p ??? )))
```

;;; A 450jsAST (AST) is one of:
;;; ...
;;; - (recb Symbol AST AST)
;;; ...
;;; (struct recb [var expr body])

**TEMPLATE : extract pieces**
Running \texttt{bind/rec}

\begin{verbatim}
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p
      ...
      [(recb x e body) x (run/e e) (run/e body)]
      ...
    ))
  (run/e p))

;; A 450jsAST (AST) is one of:
;; ...
;; - (recb Symbol AST AST)
;; ...
;; (struct recb [var expr body])
\end{verbatim}

\texttt{TEMPLATE: recursive call}
Running **bind/rec**, using environment

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      ...
      [(reb x e body) ?? x ?? (run/e e ??) ?? (run/e body ??) ]
      ... ))
  (run/e p INIT-ENV ))
```

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

;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
Running \textbf{bind/rec}, using environment

\begin{align*}
\text{;; run: AST} & \rightarrow \text{Result} \\
\text{(define (run p)} & \\
\text{;; accumulator env : Environment} & \\
\text{(define (run/e p env)} & \\
\text{ (match p} & \\
\text{...} & \\
\text{ [(recb x e body)} & \\
\text{ (define env/x (env-add env x (run/e e env))} & \\
\text{ (run/e body env/x)]} & \\
\text{ ... ))} & \\
\text{(run/e p INIT-ENV ))} & 
\end{align*}
Running `bind/rec`, using environment

```
;; run: AST -> Result

(define (run p)
  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      ...
      [(recb x e body)
       (define env/x (env-add env x (run/e e
         (run/e body env/x))]
       ...))
    (run/e p INIT-ENV)

(fac 5)) ; => 120
```

This is circular! (no base case)

PROBLEM:

> `x` should be in-scope here too!
Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined

```
(define x 3)
(display x) ; 3
(set! x 5); mutate x
(display x) ; 5
```
Interlude: Mutation

• **Mutating** a variable means to change its value after it is defined

• **Mutation** should be **rarely used**, only in appropriate situations
Interlude: Mutation

• **Mutating** a variable means to change its value after it is defined

• **Mutation** should be *rarely used*, only in appropriate situations

---


Joshua Bloch, Google's chief Java architect, is a former Distinguished Engineer at Sun Microsystems, where he led the design and implementation of numerous Java platform features, including JDK 5.0 language enhancements and the award-winning Java Collections Framework.

---

Immutability makes code easier to read and understand.

Item 15 tells you to keep the state space of each object as simple as possible. If an object is immutable, it can be in only one state, and you win big. You never have to worry about what state the object is in, and you can share it freely, with no need for synchronization. If you can't make an object immutable, at least minimize the amount of mutation that is possible. This makes it easier to use the object correctly.
Interlude: Mutation

- **Mutating** a variable means to change its value after it is defined.

- **Mutation** should be *rarely used*, only in appropriate situations.

Because:

- It **makes code more difficult to read**
  - (just like inheritance and dynamic scope)

- It violates “Separation of concerns”

```
(define x 3)
(do-something x); mutate x??
(display x); ???
```
Interlude: Mutation

• **Mutating** a variable means to change its value after it is defined

• **Mutation** should be rarely used

When is using **mutation** ok:

• Performance
  • Typically **not using high-level languages!** (OS, AAA game i.e., not this class!)
  • Beware of **pre-mature optimization!**

• **Shared state** (in distributed programs)
  • Beware of **race conditions and deadlock!**

• **Circular data structures** (e.g., circular lists)
Running `bind/rec`, recursive environment items

```scheme
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p ... 
      [(brec x e body)
       (define env/x (env-add env x (run/e e env)))]
      ...
    ))
  (run/env body env/x))

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

```
???
This is **circular**! (no base case)
```

```
PROBLEM:
  x should be in-scope here too!
```

Compute body with x in-scope
Running bind/rec, recursive environment items

;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p ...
      [(brec x e body)
       (define placeholder (box CIRCULAR-ERROR)
        (define env/x (env-add env x placeholder))
        (run/e p env/x)]]
    ...
    )))
  (run/e p INIT-ENV ))

;; A 450jsResult is a:
;; - Number
;; - FunctionResult
;; - 450jsErrorResult

;; A 450jsErrorResult is a:
;; - UNDEFINED-ERROR
;; - ARITY-ERROR
;; - CIRCULAR-ERROR

Creates mutable box
Makes mutation explicit
Running **bind/rec**, recursive environment items

```scheme
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p
      ...
      [(brec x e body)
        (define placeholder (box CIRCULAR-ERROR))
        (define env/x (env-add env x placeholder))
        ...
      ]
      (run/env body env/x)]
  ...
)(run/e p INIT-ENV))
```

;; An Environment (OLD) (Env) is one of:
;; - empty
;; - (cons (list Var 450jsResult) Env)

; (how would env-add and env-lookup need to change?)

;; An Environment (Env) is a: List<EnvVal>

;; An EnvVal is one of:
;; - 450jsResult
;; - Box<450jsResult>

;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var 450jsResult) Env)

;; An Environment (Env) is a: List<EnvVal>

;; An EnvVal is one of:
;; - 450jsResult
;; - Box<450jsResult>

;; An Environment (OLD) (Env) is one of:
;; - empty
;; - (cons (list Var 450jsResult) Env)
Running **bind/rec**, recursive environment items

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

(define (run p)
  (define (run/e p env)
    (match p ...
    
      [(brec x e body)
        (define placeholder (box CIRCULAR-ERROR))
        (define env/x (env-add env x placeholder))
        (define x-result (run/env e env/x))
        (run/env body env/x) ...]
      (run/e p INIT-ENV ))))

(bind/rec [f f] f)
; => CIRCULAR-ERROR
```

Non-function, circular recursive references (no base case) produce error results!
Running **bind/rec**, recursive environment items

```scheme
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p ...
      [(brec x e body)
        (define placeholder (box CIRCULAR-ERROR))
        (define env/x (env-add env x placeholder))
        (define x-result (run/env e env/x)
          (set-box! placeholder x-result)
          (run/env body env/x)]
      ...))
  (run/e p INIT-ENV))
```

- Explicitly mutate mutable box
- Close the (circular data structure) loop, with **mutation!**
Running \textbf{bind/rec}, recursive environment items

\begin{verbatim}
;; run: AST -> Result
(define (run p)
  (define (run/e p env)
    (match p ...
      [(brec x e body)
       (define placeholder (box CIRCULAR-ERROR)
        (define env/x (env-add env x placeholder)
        (define x-result (run/env e env/x)
        (set-box! placeholder x-result)
        (run/env body env/x))
      ...))]
    (run/e p INIT-ENV )))

(bind/rec
 [fac
  (fn (n)
   (iffy n
    (* n (fac (- n 1)))
    1))]
 (fac 5)) ; => 120
\end{verbatim}
HW 9 Preview: Recursion!

Use “CS450 JS LANG”! ... to write recursive programs:

- `fac` (factorial)
- `filt` (filter)
- `qsort` (functional quicksort)
- `gcd`
- `sierpinski` (fractal)

(Extra primitives will be added to INIT-ENV, ask if you need more)

- Look it up if you don’t know any of these
  - Using any resources, e.g., ChatGPT, Co-pilot, is allowed
  - (still can’t submit else’s hw, obv)
Recursion review

- Most recursion is structural (comes from data definitions)!

```scheme
(define (lst-fn lst)
  (cond
   [(empty? lst) ...]
   [else ... (first lst) ... (lst-fn (rest lst)) ...]))

;; A List<X> is
;; - empty
;; - (cons X List<X>)
```
A Different Kind of Recursion!

• Not all recursion is structural *(comes from data definitions)*!

(define (lst-fn lst)
  (cond
    [(empty? lst) ...]
    [else ... (first lst) ... (lst-fn (rest lst)) ...])))
A Different Kind of Recursion!

• Not all recursion is structural (comes from data definitions)!

```scheme
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid’s algorithm
;; termination argument:
;; m is halved (at least) every iteration (via modulo fn)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))))
```

What template is this following??
A Different Kind of Recursion!

- **Non-structural recursion** (doesn’t come from data definitions is called **generative recursion**
- no template, but requires **Termination Argument**
  - Explains why the function terminates!

```scheme
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid’s algorithm
;; termination argument:
;; m is halved (at least) every iteration (via modulo fn)
(define (gcd n m)
  (if (= m 0)
    n
    (gcd m (modulo n m))))
```

Recursive call must be on “smaller” version of the problem
HW 9 Preview: Recursion!

Use “CS450JS LANG”! ... to write recursive programs:

• fac (factorial)
• filt (filter)
• qsort (functional quicksort)
• gcd
• sierpinski (fractal)

(Extra primitives will be added to INIT-ENV, ask if you need more)

• Look it up if you don’t know any of these
  • Using any resources, e.g., ChatGPT, Co-pilot, is allowed
  • (still can’t submit else’s hw, obv)
In-class Coding 12/4: recursion

Use "CS450JS LANG"! ... to write recursive programs:

- **fac** (factorial)
- **filt** (filter)
- **qsort** (functional quicksort)
- **gcd**
- **sierpinski** (fractal)

• Look it up if you don’t know any of these
  • Using any resources, e.g., ChatGPT, Co-pilot, is allowed
  • (still can’t submit else’s hw, obv)

(Extra primitives will be added to INIT-ENV, ask if you need more)
No More Quizzes!

but push your in-class work to:
Repo: cs450f23/lecture25-inclass