CS339: Abstractions and Paradigms for Programming

Metacircular Evaluator (Cont.)

Manas Thakur
CSE, IIT Bombay

Autumn 2024

Exploring design choices: Lexical vs Dynamic Scoping

Lexical vs Dynamic Scoping

- What's the value returned by g?
  - Lexical scoping: 0
  - Dynamic scoping: 1

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

- Lexical scoping: Free variables are bound in the environment in which the procedure was defined (closure).
  - Examples: Pascal, Ada, Scheme, Haskell, C and family.
- Dynamic scoping: Free variables are bound the most recently assigned value during program execution.
  - Examples: Original Lisp, Bash, LaTeX.

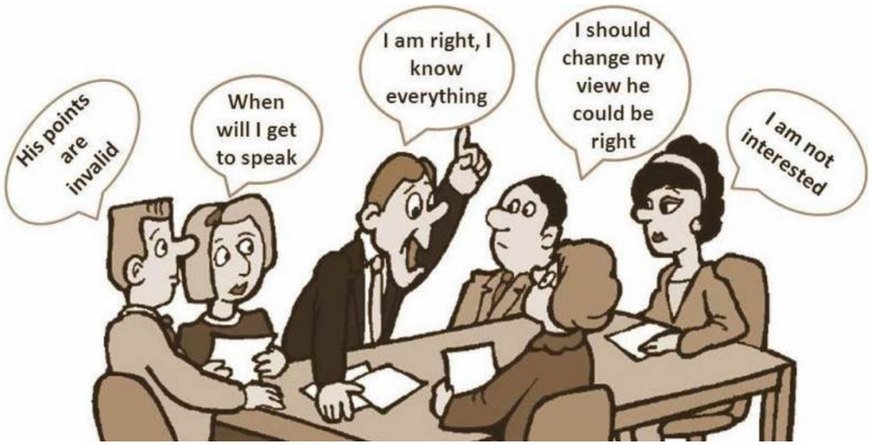
Lexical vs Dynamic Scoping (Cont.)

- What's the value returned by g?
  - Lexical scoping: 1
  - Dynamic scoping: 2
- Which language is this?
- **Homework:** Find out which scoping is used in R.
  - Read (3 blog posts!):

```
x <- 1
f <- function(a) x + a
g <- function() {
     x <- 2
     f(0)
}
g()
```

Discussion

- Which one — lexical or dynamic scoping — is more natural?
- Which one should be easier to implement?
- Original Lisp had dynamic scoping!
- Then why switch to lexical scoping in Scheme?



Flashback

- Recall the general sum procedure from the days we learnt about higher order functions:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

- Usage:

```
(define (sum-cubes a b)
  (define (cube x) (expt x 3))
  (sum cube a inc b))
```

- One more (sum of nth powers):

```
(define (sum-powers a b n)
  (define (nth-power x) (expt x n))
  (sum nth-power a inc b))
```

Notice the binding

Why not Dynamic Scoping?

Problem with Dynamic Scoping

- Say the programmer renamed b in the procedure sum to n:

```
(define (sum term a next n)
  (if (> a n)
      0
      (+ (term a)
         (sum term (next a) next n))))

(define (sum-powers a b n)
  (define (nth-power x) (expt x n))
  (sum nth-power a inc b))
```

- Whoops! Problem *irukke*.

n got bound to b

Then why Dynamic Scoping?

Classwork

- Write an analogous procedure to multiply the nth powers.

```
(define (product-powers a b n)
  (define (nth-power x) (expt x n))
  (product nth-power a inc b))
```

- Can we abstract out nth-power?
  - Lexical scoping: No.
  - Dynamic scoping: Yes!

```
(define (sum-powers a b n)
  (define (nth-power x) (expt x n))
  (sum nth-power a inc b))
```

```
(define (sum-powers a b n)
  (sum nth-power a inc b))

(define (product-powers a b n)
  (product nth-power a inc b))

(define (nth-power x)
  (expt x n))
```

Now comes fun!

Make Scheme Dynamically Scoped

- Remarkably simple: Just extend the correct environment!

```
(define eval
  (lambda (exp env)
    (cond ((number? exp) exp)
          ...
          ((pair? exp) (apply (eval (car exp) env) (evallist (cdr exp) env)))
          (else (error "unknown expression type")))))

(define (apply
  (lambda (proc args)
    (cond ((primitive-op? proc) (apply-prim-op proc args))
          ((compound-op? proc) (eval (caddr proc) (extend-environ (cadr proc) args (cadddr proc)))))))
```

```
(define eval
  (lambda (exp env)
    (cond ((number? exp) exp)
          ...
          ((pair? exp) (apply (eval (car exp) env) (evallist (cdr exp) env) env))
          (else (error "unknown expression type")))))

(define (apply
  (lambda (proc args env)
    (cond ((primitive-op? proc) (apply-prim-op proc args))
          ((compound-op? proc) (eval (caddr proc) (extend-environ (cadr proc) args env)))))))
```

In fact, we won't need to bind the environment while creating procedure objects!

Scoping in Action

Insight of the Week (Semester!)

- How do we abstract out nth-power with lexical scoping?

THE LAST

```
(define make-exp
  (lambda (n)
    (lambda (x) (expt x n))))
```

By being good in *computing science :-)*

```
(define (sum-powers a b n)
  (sum (make-exp n) a inc b))
```

```
(define (product-powers a b n)
  (product (make-exp n) a inc b))
```

- What's the powerful feature that allowed you to solve this problem?
  - Ability to return functions as values.
  - In general, granting **first-class citizenship to functions**!