Exam 2 Topics

Topics

- Scheme (Lectures 12 and 13, plus chapters)
  - S-expression syntax
  - Lists and recursion
  - Shallow and deep recursion
  - Equality
  - Higher-order functions
  - **map, foldl,** and **foldr**
  - Programming with **map, foldl,** and **foldr**
  - Tail recursion

Topics

- Scheme (Lecture 14, plus chapters)
  - Binding with **let, let*, letrec**
  - Scoping in Scheme
  - Closures and closure bindings

Topics

- Scoping, revisited (Lecture 14, plus chapters)
  - Static scoping
    - Reference environment
    - Functions as third-class values vs.
    - Functions as first-class values
  - Dynamic scoping
    - With shallow binding
    - With deep binding

Topics

(define (fun x) S-expr)
(define fun (lambda (x) S-expr))

- Lambda calculus (Lectures 15, 16, and 17)
  - Syntax and semantics
  - Free and bound variables
  - Substitution
  - Rules of the Lambda calculus: Alpha-conversion and Beta-reduction
  - Normal forms
  - Reduction strategies: Normal order and Applicative order
  - Fix-point combinator and recursion

Quiz 5

Question 1. (2pts) Scheme's scoping discipline is

Select one:

- static scoping
- dynamic scoping

Question 2. (2pts) Scheme's typing discipline is

(cdr e)

Select one:

- static typing
- dynamic typing

Quiz 5

Questions 3 and 4 refer to the following Scheme function:

```
(define (fun a b)
  (cond ((= a b) a)
        ((> a b) (fun (- a b) b))
        (else (fun a (- b a)))))
```

Question 3. (2pts) What does `fun` compute? Note: you may assume that the arguments are positive integers.

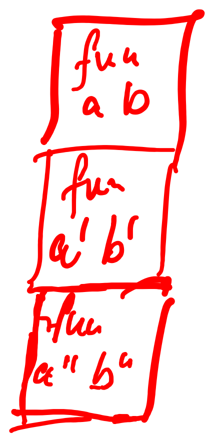
gcd

Question 4. (2pts) `fun` is tail-recursive.

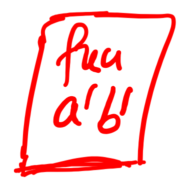
Select one:

- ◯ true
- ◯ false

standard:



optimised:



A single frame on stack.

Quiz 5

Question 5. (2pts) Function `atomcount`, defined below, attempts to count the number of atoms nested in a list. Recall predicate `atom?` that we wrote in class — it returns true if given an object that is *not a pair* (i.e., an object we cannot take the `car` or the `cdr` of).

```
(define (atomcount lis)
  (cond ((atom? lis) 1)
        ((null? lis) 0)
        (else (+ (atomcount (car lis)) (atomcount (cdr lis))))))
```

`(atomcount '(1 (2 (3))))` yields

6

Scoping with First-Class Functions

- Functions as first-class values
- Static scoping is more involved
  - Function value may outlive static referencing environment!
  - Therefore, we need "immortal" closure bindings

Scoping with First-Class Functions

- Dynamic scoping is more involved
  - Shallow binding vs. deep binding

- Dynamic scoping with shallow binding
  - Reference environment for function/routine is not created until the function is called
    - As a result, all non-local references are resolved using the most-recent-frame-on-stack rule

Scoping with First-Class Functions

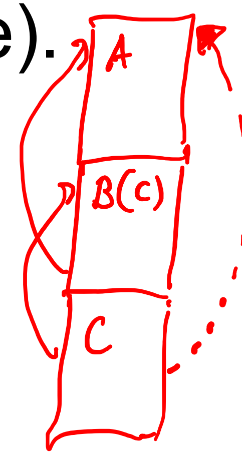
- Dynamic scoping with deep binding
  - When a function/routine is passed as an argument, the code that passes the function/routine has a particular reference environment (the current one!) in mind. It passes this reference environment along with the function value (it passes a closure).

A

B(C)

C = { function value, A's ref. env.

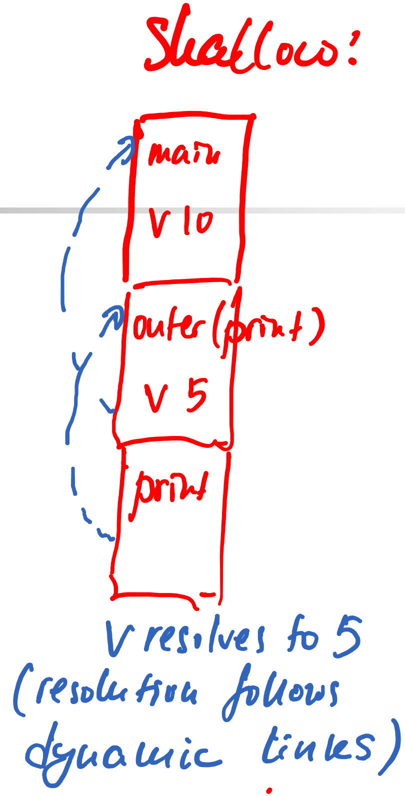
C is a closure. Includes function value + A's reference environment.

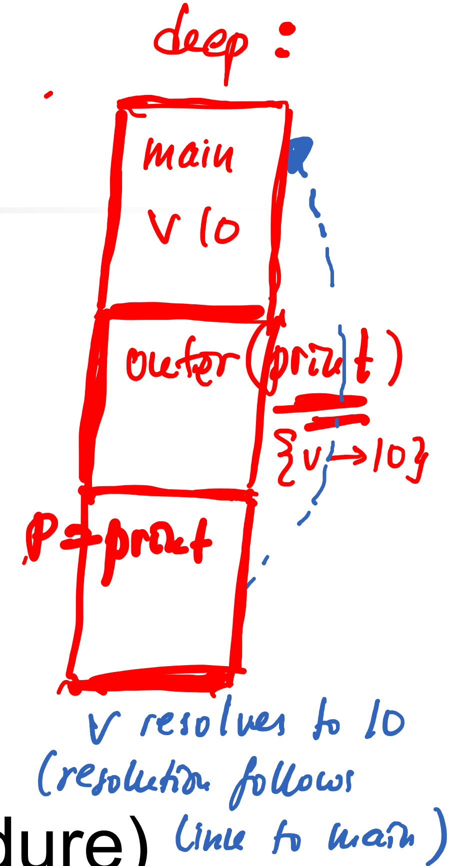


resolution of non-local refs in C uses link to A.

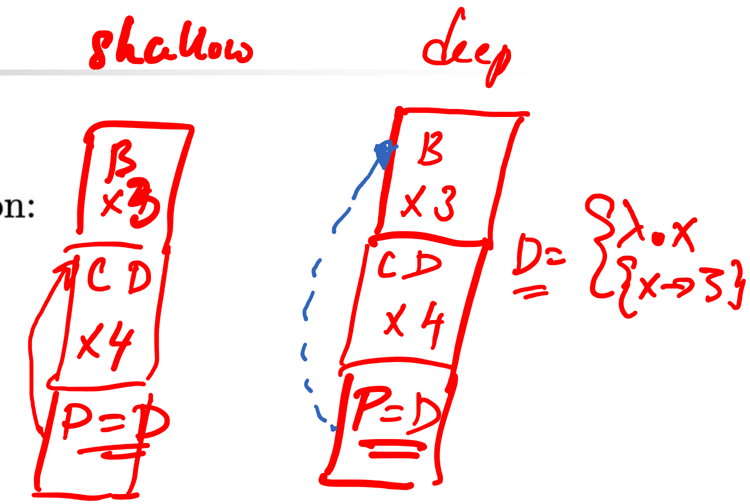
Example

```
v : integer := 10
people : database

print_routine (p : person)
  if p.age > v
    write_person(p)

other_routine (db : database, P : procedure)
    v : integer := 5
    foreach record r in db
        P(r)

other_routine(people, print_routine) /* call in main */
```





Quiz 6



Questions 1-3 below refer to the following Scheme function:

```
(define A
    (lambda ()
        (let* ((x 2)
               (C (lambda (P) (let ((x 4)) (P) )))
               (D (lambda () x))
               (B (lambda () (let ((x 3)) (C D)))))
          (B))))
```

Question 1. (1pts) What gets printed when we call the function in the interpreter?

2

Question 2. (2pts) What would get printed if Scheme used *dynamic scoping with shallow binding*?

4

Question 3. (2pts) What would get printed if Scheme used *dynamic scoping with deep binding*?

3

Quiz 6

Question 4. (2pts) Willy Wazoo wrote a function `f`. What does it do?

```
(define (f lis)
  (foldl (lambda (x y) (if (and (number? x) (> x 5)) (cons x y) y)) lis '()))
```

partial result

next element

y y y x x

(append x (cons y '()))

Select one:

- ○ selects all numerical elements in lis greater than 5
- ○ selects all numerical elements in lis less or equal than 5
- ○ neither; Willy's function has a bug

Quiz 6

Question 5. (1pt) Consider the lambda term $(\lambda v.v)\ ((\lambda v.v)\ (\lambda x.(\lambda x.x)\ z))$. There are this many reducible expressions in this term:

3

E_1 E_2

Question 6. (1pt) Is lambda term $x\ ((\lambda y.y)\ z)$ in Weak Head Normal Form (WHNF)?

YES

λx. (λy.y) z x is in WHNF

λx. (λy.y) z x is not in HNF

Question 7. (1pt) Is term $x\ ((\lambda y.y)\ z)$ in Normal Form (NF)?

NO

Unification (simplified)

- **Unify**: tries to unify terms τ_1 and τ_2 and binds variables to values if $\tau_1 = \tau_2$ if unification is successful

def **Unify**(τ_1,τ_2) =

This is the occurs check!

case (τ_1,τ_2)

(τ_1,**V**$_2$) -> ***success*** with binding [τ_1/**V**$_2$] if **V**$_2$ does not occur in τ_1; ***fail*** otherwise

(**V**$_1$,τ_2) -> ***success*** with binding [τ_2/**V**$_1$] if **V**$_1$ does not occur in τ_2; ***fail*** otherwise

(**c**$_1$,**c**$_2$) -> ***success*** if (eq? **c**$_1$ **c**$_2$); ***fail*** otherwise

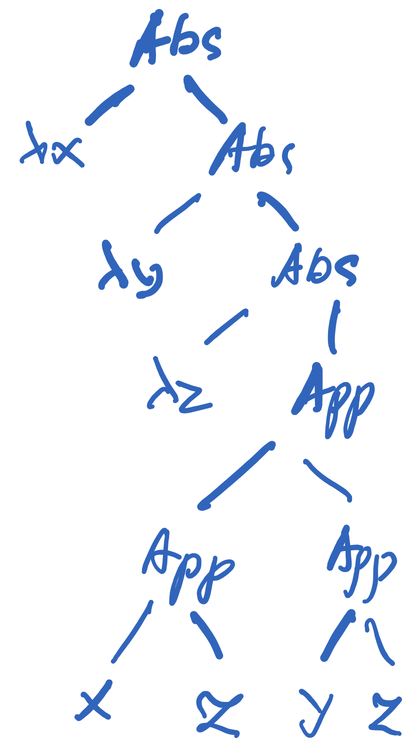
(**p**(τ_{11},τ_{12}), **p**(τ_{21},τ_{22})) -> **Unify**(τ_{11},τ_{21}); **Unify**(τ_{12},τ_{22})

otherwise = ***fail***

Questions?

Abbrev. $\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z)$

$S = \lambda xyz.\ x\ z\ (y\ z) \qquad I = \lambda x.x$



$S\ I\ I\ I = ((S\ I)\ I)\ I$ Applicative order

$(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ I\ I\ I \to_\beta$

$(\lambda y.\lambda z.\ I\ z\ (y\ z))\ I\ I =$

$(\lambda y.\lambda z.\ (\lambda x.x)\ z\ (y\ z))\ I\ I \to_\beta$

$(\lambda y.\lambda z.\ z\ (y\ z))\ I\ I \to_\beta$

$(\lambda z.\ z\ (I\ z))\ I =$

$(\lambda z.\ z\ ((\lambda x.x)\ z))\ I \to_\beta$

$(\lambda z.z\ z)\ I \to_\beta I\ I = (\lambda x.x)\ I \to_\beta I = \lambda x.x$

The End

Normal order:

$$(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ I\ I\ I \rightarrow$$

$$(\lambda y.\lambda z.\ I\ z\ (y\ z))\ I\ I \rightarrow$$

$$(\lambda z.\ I\ z\ (I\ z))\ I \rightarrow$$

$$I\ I\ (I\ I) =$$

$$(\lambda x.x)\ I\ (I\ I) \rightarrow$$

$$I\ (I\ I) =$$

$$(\lambda x.x)\ (I\ I) \rightarrow$$

$$I\ I = (\lambda x.x)\ I \rightarrow I$$