CS 421 Lecture 17 – Functional programming

- Using fold_right and fold_left
- Expression evaluation
  - Substitution model
  - Scope of definitions
- "Simple" examples
- Combinator programming

fold_right

`fold_right f [x1;x2;...xn] z`
`= f x1 (f x2 (...(f xn z)...))`
`fold_right : (α->β->β)->(α list)->β->β`

Use `fold_right` to remove all negative elements from a list:

`fold_right` (fun x y → if x<0 then y else x::y) lis []

fold_left (corrected def)

```
fold_left : (α->β->α)-> α -> β list -> α
fold_left f z [x1;x2;...xn]
    = f(... (f (f z x1) x2)...) xn
```

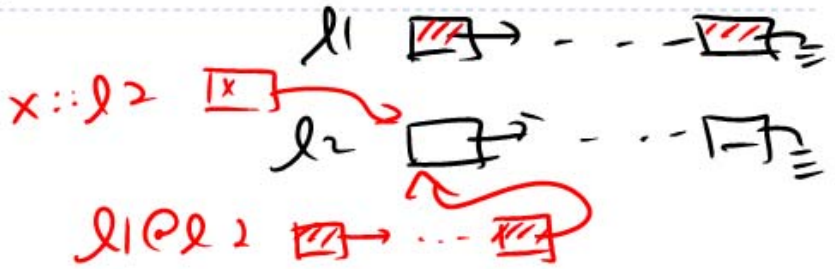
Use `fold_left` to compute the length of lis

`fold_left` (fun x y → 1 + x) 0 lis

Use `fold_left` to compute map f lis

`fold_left` (fun x y → x @ [f y]) [] lis

l1 @ l2



Defining higher-order functions

```
let rec fold_right f lis z =
  if lis = [] then z
  else f (hd lis)
    (fold_right f (tl lis) z)
```

Define `fold_left`:

let rec fold_left f z lis

=

Evaluation of expressions

Use substitution model – in function calls, substitute actual parameter for formal parameter in body of function.

- No expressions with free variables evaluated
- Expressions: constants, function definitions (fun x -> e), application of built-in functions, if, application of user-defined functions
- let expressions syntactic sugar for function applic; top-level definitions implicitly in let
- Will handle recursive functions after break; also will discuss closure model after break

Evaluation of expressions

Evaluate expression without free variables:

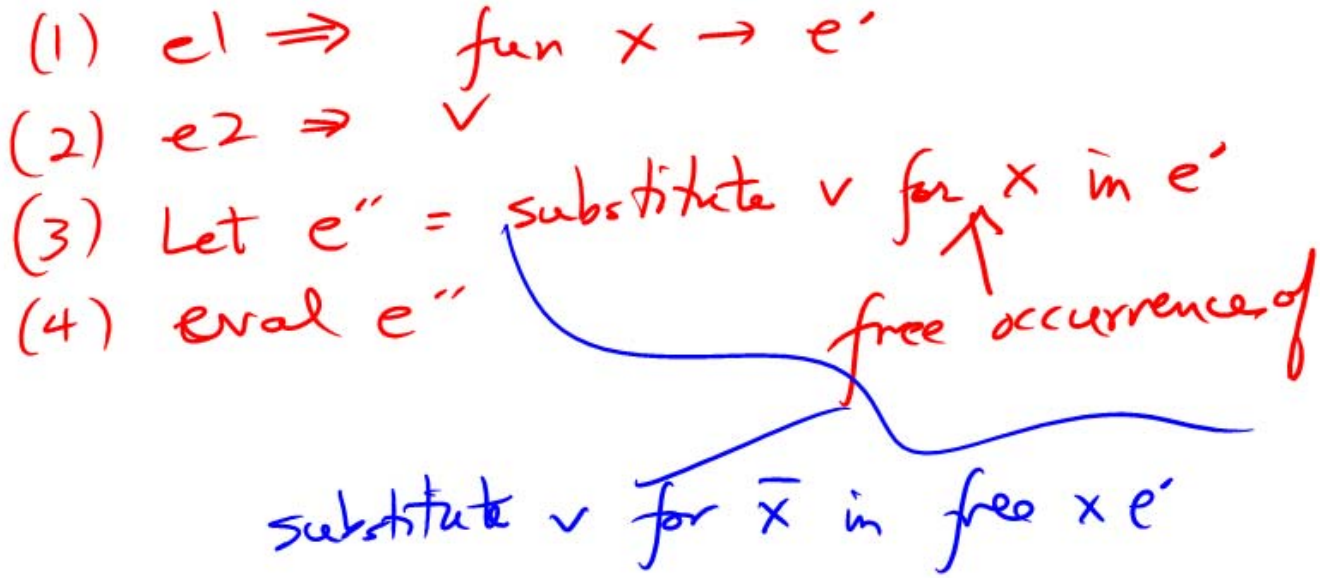
- Constant n (int, bool, string, list, ..) $\Rightarrow$ n
- Abstraction fun x -> e $\Rightarrow$ fun $x \to e$

- Application of built-in operator: e1 + e2
  (1) e1 $\Rightarrow n_1$
  (2) e2 $\Rightarrow n_2$
  (3) return $n_1 + n_2$

  similarly for other built-in op's, eg. :: , ^ , ...
- if e1 then e2 else e3
  (1) e1 $\to$ v
  (2) if v is "true" eval e2
  o.w. eval e3

Evaluation of expressions (cont.)

- Application of user-defined function: e1 e2



Example of evaluation

(Function application associates to the left)

```
((fun x -> fun y -> x+y) 1) 2
```

e1 e2

(1) Eval e1: (fun x → fun y → x+y) 1 ⇒ fun x → fun y → x+y

⇒ fun y → 1+y

(2) 2 ⇒ 2

(3) Subst 2 for y in 1+y, giving 1+2

(4) Eval 1+2

⇒ 3

Example of evaluation

```
((fun x -> fun y -> x y) (fun y -> y 4)) (fun z -> z+1)
```

e_1 e_2

(1) Eval e1 : (fun x → ..)(fun y → ..)

eval → fun x → fun y → x y

eval → fun y → y 4

subst : fun y → (fun y → y 4) y

(2) Eval e2 : fun z → z + 1 = v

(3) Subst v for y in (fun y → y 4) y,

giving (fun y → y 4)(fun z → z+1) e''

(4) Eval e''

(1)

(2)

(3) subst (fun z → z+1) 4

(4) eval

Free variables

free y ((fun y → y 4) y) = (fun y → y 4) $\bar{y}$

In rule for applications, substitute v for *free occurrences* of x in e'. Need to define "free occurrence."

Def. Free occurrences of x in e are those marked with an overbar after applying free to x and e:

free x e = match e with

n (a constant) → n

x → $\bar{x}$

y → $\bar{y}$

$e_1 + e_2$ → (free x e_1) + (free x e_2)
(similarly for if $e_1 \cdots$, e_1 e_2)

fun x → e' → fun x → e'

fun y → e' → fun y → free x e'

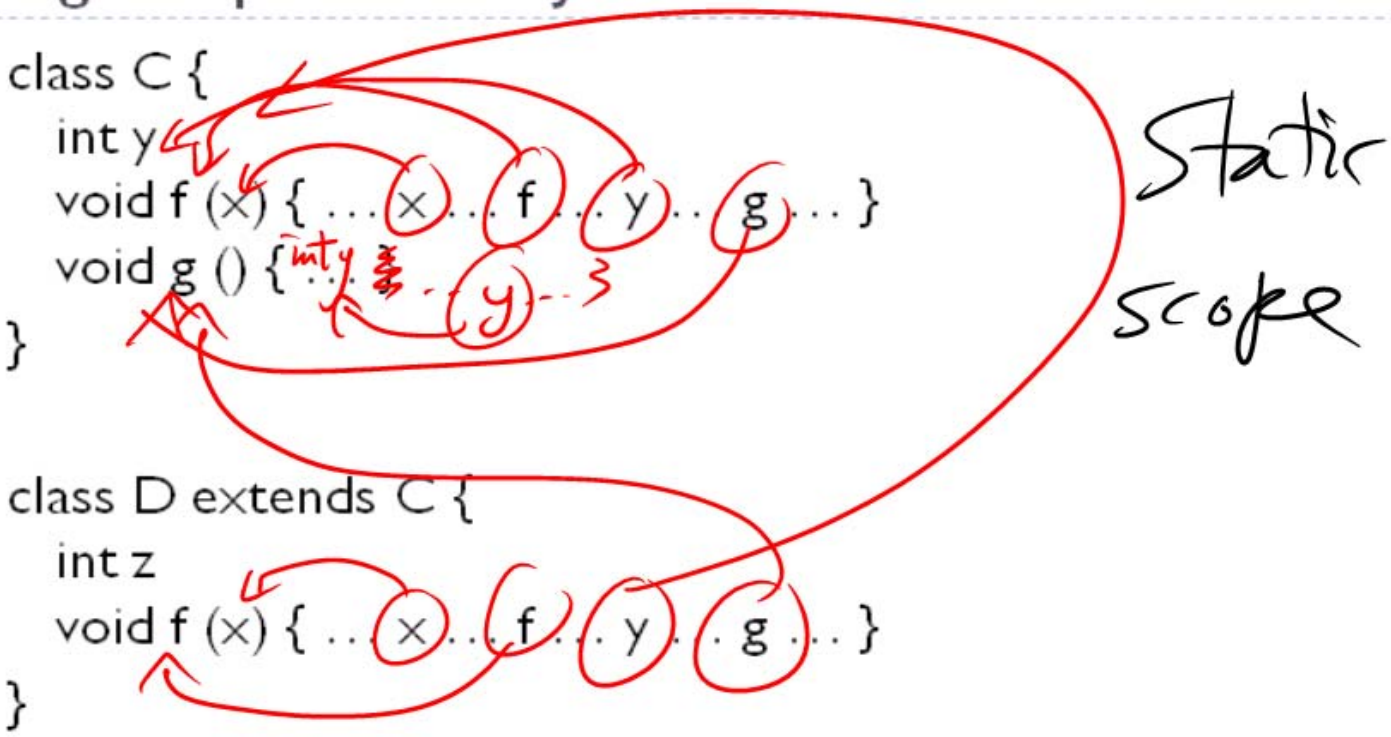
Example of free occurrences

(fun x -> fun y -> x y) (fun y -> y 4) (fun z -> z+1)

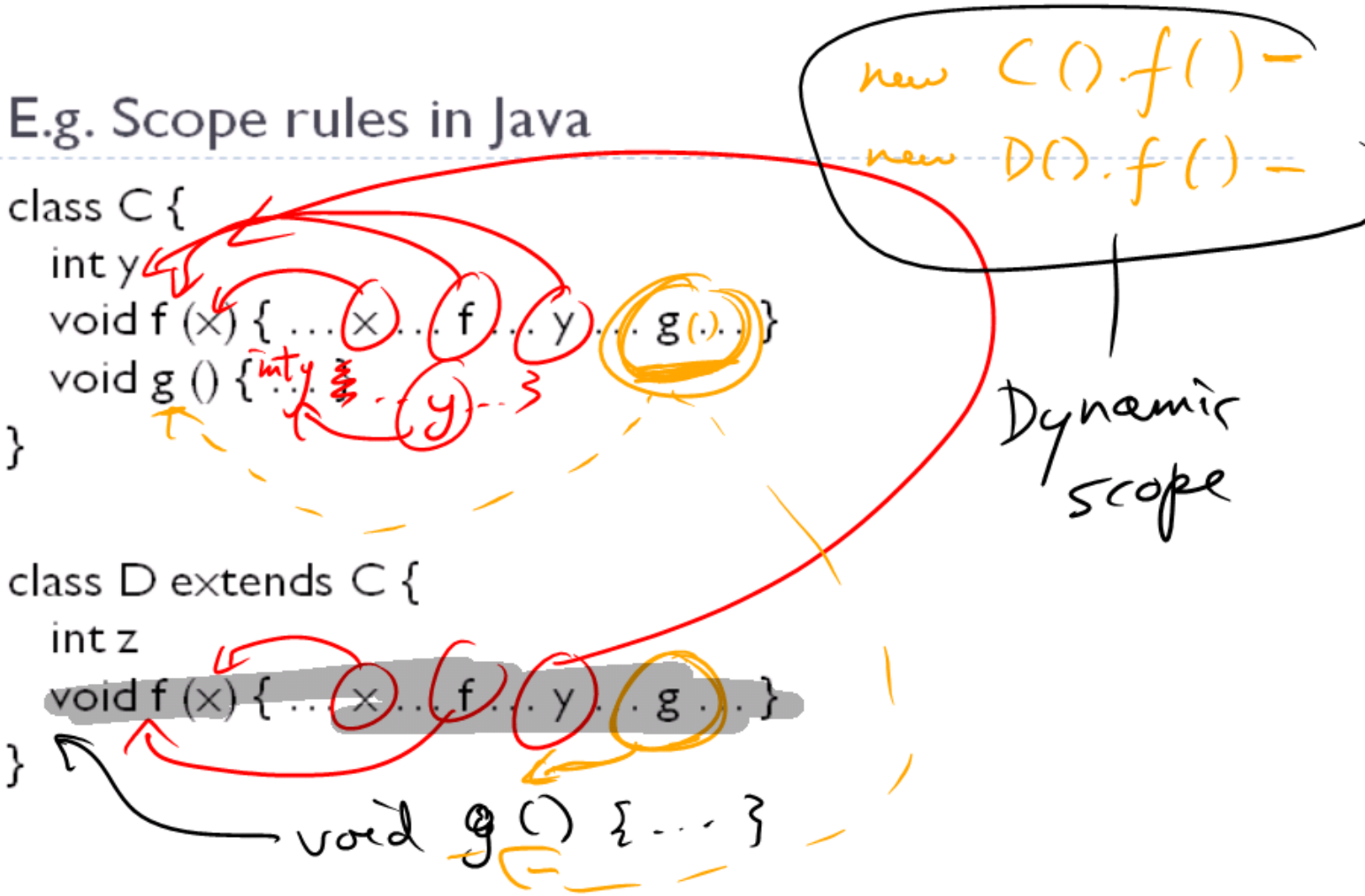
Scope rules

- Programs introduce names via "declarations", then refer to those names in "uses." A given name can be introduced in more than one declaration, but every use corresponds to a particular declaration. The question is: which one?
- The *scope of a declaration* of a name x is the parts of the program in which a use of x refers to this declaration
- A use of a name is *in the scope of a declaration* if that use is in the scope of that declaration
- N.B. the scope of a declaration can have holes, where the declaration is covered up by another declaration of the same name.

E.g. Scope rules in Java



E.g. Scope rules in Java



E.g. Scope rules in OCaml - Only static scope

1.
```
let x = 2
in let f = fun x -> x+x
   in f x
```
2.
```
let x = 2
in let y = x
   in let f z = let x=3 in y+z
      in f x
```
3.
```
let x = 2
in let add = fun x -> fun y -> x+y
   in let addx = add x
      in let x = 3 in  addx 1
```

Scope rules in OCaml

Scope rules are implied by expression evaluation rules.

Declarations are just function definitions fun x ->e

Scope of this declaration of x is exactly the free occurrences of x in e.

(Put differently, a use of a variable x is in the scope of the closest enclosing function definition for which x is the formal parameter.)

This is called *static scope*, or *lexical scope*, because the declaration corresponding to any use is known statically (before run time).

The scope rule of Lisp

- In Lisp, the declaration associated with a use of a variable x is determined as follows: at run-time, the most recent function application that has x as formal parameter (and which is still on the stack) gives the declaration of x.
- Lisp vs. Ocaml:

```
let h f = let x = 3 in f x
let f x = let g y = x + y in h g
f 5 => ?
```

"Simple" examples - Currying

- Can define two-argument functions in two ways:
  - Curried: let f x y = … x … y …
    (or, let f = fun x y -> … x … y …
    or, let f = fun x -> fun y -> … x … y …)
  - Uncurried: let f (x,y) = … x … y …
    (or, let f = fun (x,y) -> … x … y …
    or, let f = fun p -> … (fst p) … (snd p) …)

Sometimes want to use the "same" function both ways.

"Simple" examples - Currying

- Can use higher-order function to turn curried function to uncurried form, and vice versa

```
let curry f = fun x -> fun y -> f (x, y)
let uncurry g = fun (x, y) -> g x y
```

curry : $(\alpha * \beta \to \gamma) \to (\alpha \to \beta \to \gamma)$

uncurry : $(\alpha \to \beta \to \gamma) \to (\alpha * \beta \to \gamma)$

$f \equiv$ uncurry (curry f)

"Simple" examples – reversing arguments

Given f: $\alpha \rightarrow \beta \rightarrow \gamma$, produce f_R: $\beta \rightarrow \alpha \rightarrow \gamma$, s.t.

$f_R \times y = f\ y\ x$

```
let reverse f = fun x -> fun y -> f y x
```

eg. `reverse (-) 3 4 = 1`

"Simple" examples – applying function twice

Given f: $\alpha \rightarrow \alpha \rightarrow \alpha$, want ff: $\alpha \rightarrow \alpha \rightarrow \alpha$ such that
ff x = f (f x)

```
fun double f = fun x -> f (f x)

(double incr) 5 = 7
```

Combinator-style programming

Can write complex programs by defining a library of higher-order functions and applying them to one another (and to first-order or built-in functions).

Advantage: easy of creating programs – programs are just expressions

Example: build a parser by writing "parser combinators".

Parser combinators

Define a parser to be a function from token list -> (token list) option.

Idea is to define functions that build parsers, rather than building parsers "by hand."

E.g. Parser to recognize a single token:

```
let token s = fun cl -> if cl=[] then None
                        else if s=hd cl then Some (tl cl)
                        else None;;
let parsex = token 'x';;
parsex ['x'];;
parsex['a'];;
```

Parser combinators

"Combinators" to combine parsers into larger parsers:

```
let (++) p q = fun cl -> match p cl with None -> None
                                | Some cl' -> q cl';;

let parsexy = token 'x' ++ token 'y'
parsexy ['x', 'y']
parsexy ['x', 'z']

let (||) p q = fun cl ->  match p cl with None -> q cl
                                | Some cl' -> Some cl';;
let parsexyorz = parsexy || token 'z'
parsexyorz['x', 'y']
parsexyorz ['z']
```

Parser combinators

Put this together to define parser for grammar:

```
A -> aB | b
B -> cB | A
```

```
let rec parseA cl = ((token 'a' ++ parseB) || token 'b') cl
    and parseB cl = ((token 'c' ++ parseB) || parseA) cl;;

parseA ['a';'c';'c';'a';'b'];
```