Functional Programming

Lecture 3: Higher order functions

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Last lecture

- Evaluation strategies
- Program vs. data
- Debugging
- Lambda abstraction

```
(lambda (arg1 … argN) <expr>)
```

- Let, let*, append, merge-sort
- Home assignment 1

Higher order functions

Functions taking other functions as arguments or returning functions as the result

- Capture and reuse common patterns
- Create fundamentally new concepts
- The reason why functional programs are compact

Order of data

- Order 0

  Non function data

- Order 1

  Functions with domain and range of order 0

- Order 2

  Functions with domain and range of order 1

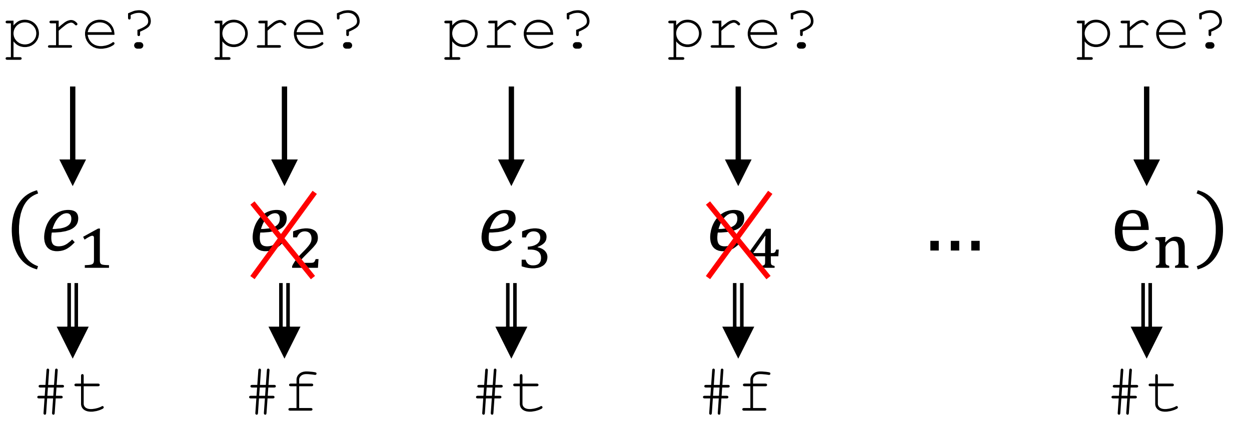
- Order k

  Functions with domain and range of order k-1

Filter

```
(filter pre? list)
```

In the previous lecture



pre? pre? pre? pre? pre?

$(e_1 \quad \cancel{e_2} \quad e_3 \quad \cancel{e_4} \quad \ldots \quad e_n)$

#t #f #t #f #t

$(e_1\ e_3 \ldots e_n)$

Apply

Applies a function to the arguments

```
(apply proc arg1 ... rest-args)
```

Example:

```
(apply + 1 2 3 '(4 5))
```

Apply

```
(define (my-apply1 f args)
  (define (quote-all list)
    (cond ((null? list) '())
          (#t (cons
               `(quote ,(car list))
                (quote-all (cdr list))))
              )
          )
  (eval (cons f (quote-all args))))
```

Variable number of arguments

```
(define (fn arg1 arg2 . args-list) <body>)
```

After calling, the remaining arguments are in `args-list`.

```
(lambda args-list <body>)
```

Append

```
(define (my-append . args)
    (cond
      ((null? args) args)
      (#t (append2 (car args)
                   (apply my-append
                          (cdr args))))
    )
)
```

Apply

```
(define (my-apply f . args)
  (define (appendlast list)
    (cond ((null? (cdr list)) (car list))
          (#t (cons
                 (car list)
                 (appendlast (cdr list))))))
  (my-apply1 f (appendlast args)))
```

Compose

```
(compose f g)
```

Arguments are functions

Returns a function

```
(define (compose1 f g)
  (lambda args
    (f (apply g args))))
```

Inc each / dec each

```
(define (incall list)
  (cond ((null? list) '())
        (#t (cons (+ (car list) 1)
                  (incall (cdr list))))))

(define (decall2 list)
  (cond ((null? list) '())
        (#t (cons (- (car list) 2)
                  (decall2 (cdr list))))))
```

Map

```
(define (incall list)
  (cond ((null? list) '())
        (#t (cons (+ (car list) 1)
                  (incall (cdr list))))))

(define (map1 f list)
  (cond ((null? list) '())
        (#t (cons (f (car list))
                  (map1 f (cdr list))))))
```

Map

Calls `proc` of `N` arguments on all elements of the list and returns the result as a list

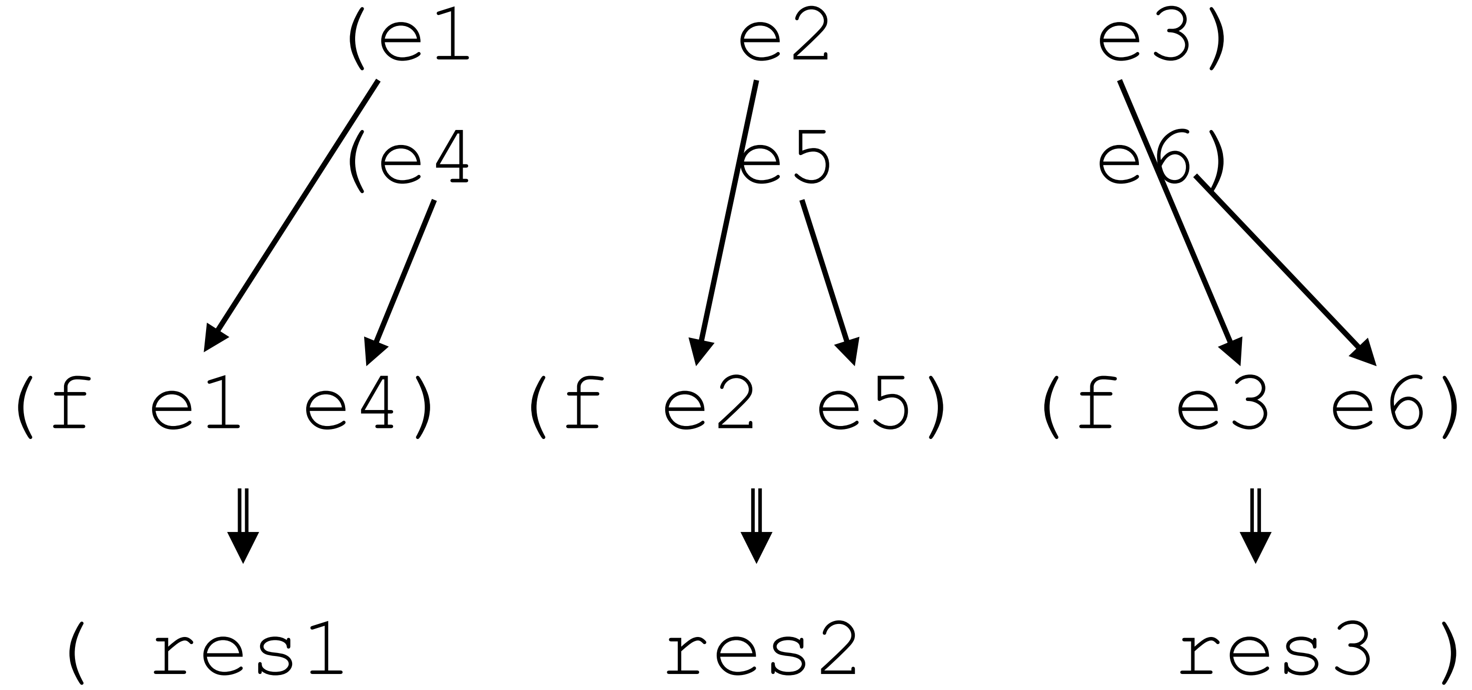
```
(map proc list1 list2 ... listN)
```

Example:

```
(map + '(1 2 3) '(4 5 6))
```

Map

```
(map f '(e1 e2 e3) '(e4 e5 e6))
```



```
        (e1        e2        e3)
        (e4        e5        e6)

(f e1 e4)  (f e2 e5)  (f e3 e6)

   ( res1       res2       res3 )
```

Map

```
(define (my-map proc . args)
  (cond ((null? (car args)) '())
        (#t (cons
              (apply proc (map1 car args))
              (apply my-map
                    (cons
                      proc
                      (map1 cdr args)))))))
```

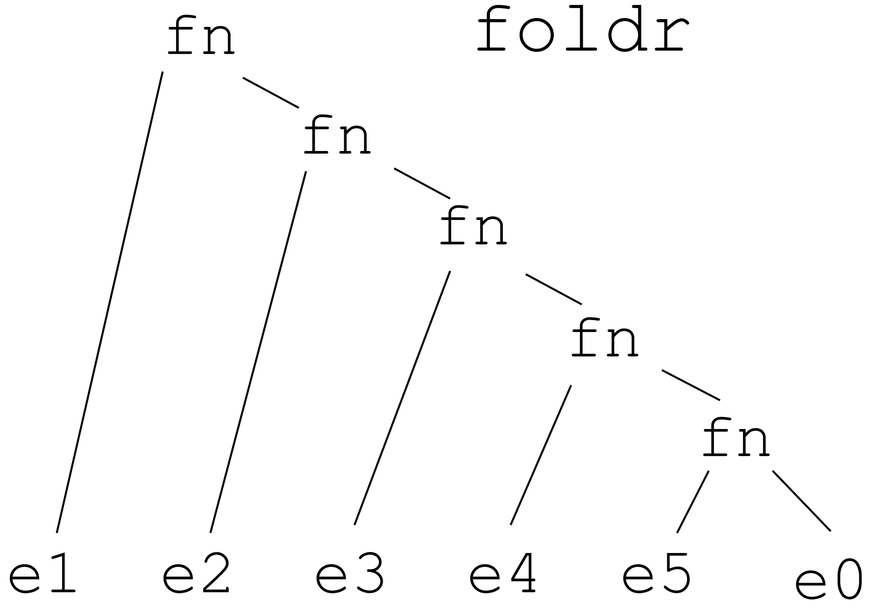
Min / sum

```
(define (min-all list)
  (cond ((null? (cdr list)) (car list))
        (#t (min (car list) (min-all (cdr list))))))

(define (sum-all list)
  (cond ((null? (cdr list)) (car list))
        (#t (+ (car list) (sum-all (cdr list))))))

(define (reduce f list)
  (cond ((null? (cdr list)) (car list))
        (#t (f (car list) (reduce f (cdr list))))))
```

Reduce

Often called `foldr` and `foldl` in scheme

```
(foldr fn e0 '(e1 e2 e3 e4 e5))
```

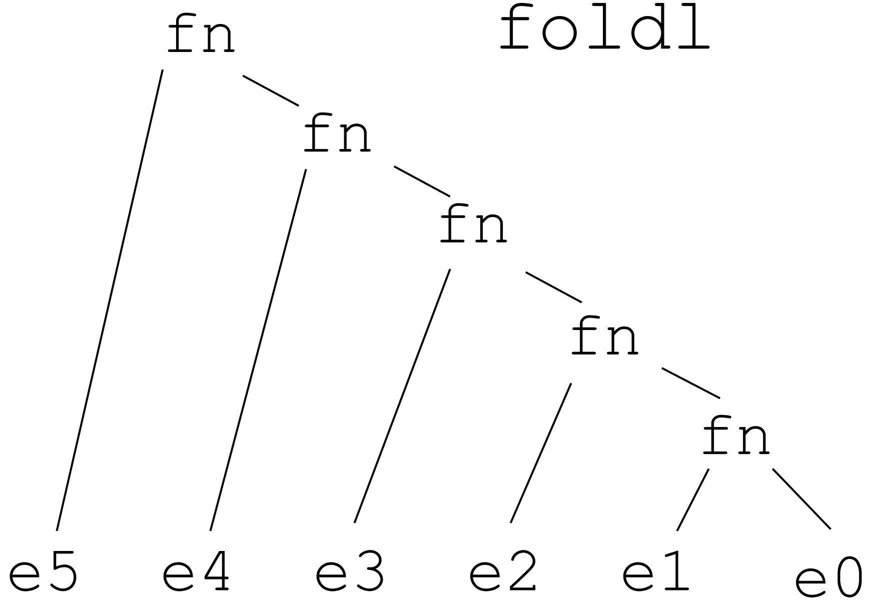


Foldr

```
(define (fold-right f a list)
  (cond ((null? list) a)
        (#t (f (car list)
               (fold-right f a (cdr list))))
))
```

Foldl

```
(foldl fn e0 '(e1 e2 e3 e4 e5))
```

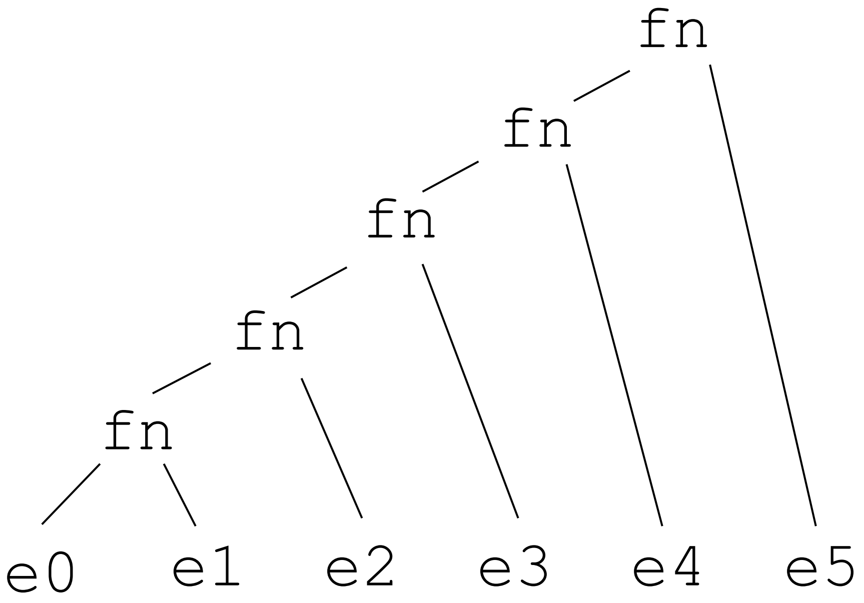


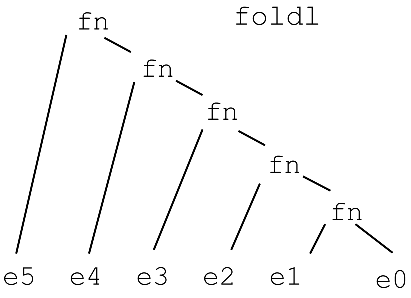
Foldl

```
(define (fold-right f a list)
  (cond ((null? list) a)
        (#t (f (car list)
               (fold-right f a (cdr list))))
))

(define (fold-left f a list)
  (cond ((null? list) a)
        (#t (fold-left
               f
               (f (car list) a)
               (cdr list)))))
```

Swap

```
(define (swapargs  f)
  (lambda (x y) (f y x)))

(foldl (swapargs fn) e0 '(e1 e2 e3 e4 e5)
```





Every / some

```
(every pred list1 … listN)
```

```
(define (every1 pred list)
  (cond ((null? list) #t)
        (#t (and
              (pred (car list))
              (every1 pred (cdr list)))))
```

```
(define (some1 f list)
  (not (every1 (lambda (x) (not (f x)))  list)))
```

Derivative

$$Df(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

```
(define (deriv f)
  (lambda (x)
    (/ (- (f (+ x epsilon)) (f x))
       epsilon)))
```

Combining higher order functions

- add-only-numbers
- some
- flatten
- L2 norm
- filter
- length

Summary

- Higher order functions take functions as arguments or return functions
- Used to capture/reuse common patterns
- Create fundamentally new concepts
- Filter, apply, map, fold, swap