Functional Programming

Lecture 3: Higher order functions

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague
viliam.lisy@fel.cvut.cz
xhorcik@fel.cvut.cz

Last lecture

- Evaluation strategies
- Debugging
- Lambda abstraction

```
(lambda (arg1 … argN) <expr>)
```

- Let, let*, append, quicksort
- 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

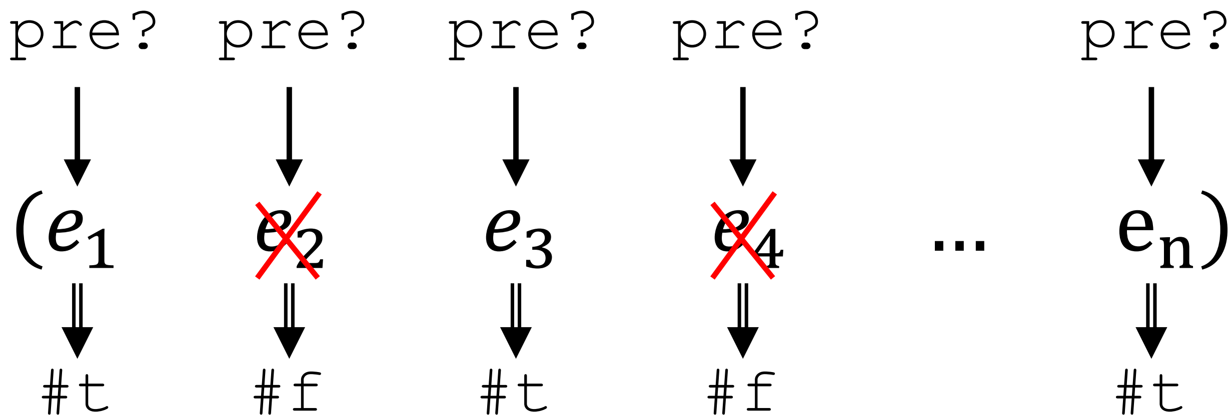
  Functions with domain and range of order 1
- Order k

  Functions with domain and range of order k-1

Filter

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

In the previous lecture



$(e_1\ e_3 \ldots e_n)$

Variable number of arguments

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

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

Corresponding lambda term

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

Alternatively

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

Apply

Applies a function to the arguments

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

Example:

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

Append

```
;;; Append arbitrary list of lists
(define (my-append . args)
  (cond
    ((null? args) args)
    (else
     (append2 (car args)
              (apply my-append (cdr args)))
     )
    )
  )
```

Apply

```
(define (my-apply1 f args)
  (define (quote-all lst)
    (cond
      ((null? lst) '())
      (else (cons
              `(quote ,(car lst))
              (quote-all (cdr lst))))
      )
    )
  (eval (cons f (quote-all args)))
  )
```

Apply

```
(define (my-apply f . args)
  (define (append-last lst)
    (cond
      ((null? (cdr lst)) (car lst))
      (else (cons (car lst)
                  (append-last (cdr lst)))
            )
      )
    )
  (my-apply1 f (append-last 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 (map1 f lst)
  (cond
    ((null? lst) '())
    (else
     (cons (f (car lst))
           (map1 f (cdr lst)))
     )
    )
  )
```

Map

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

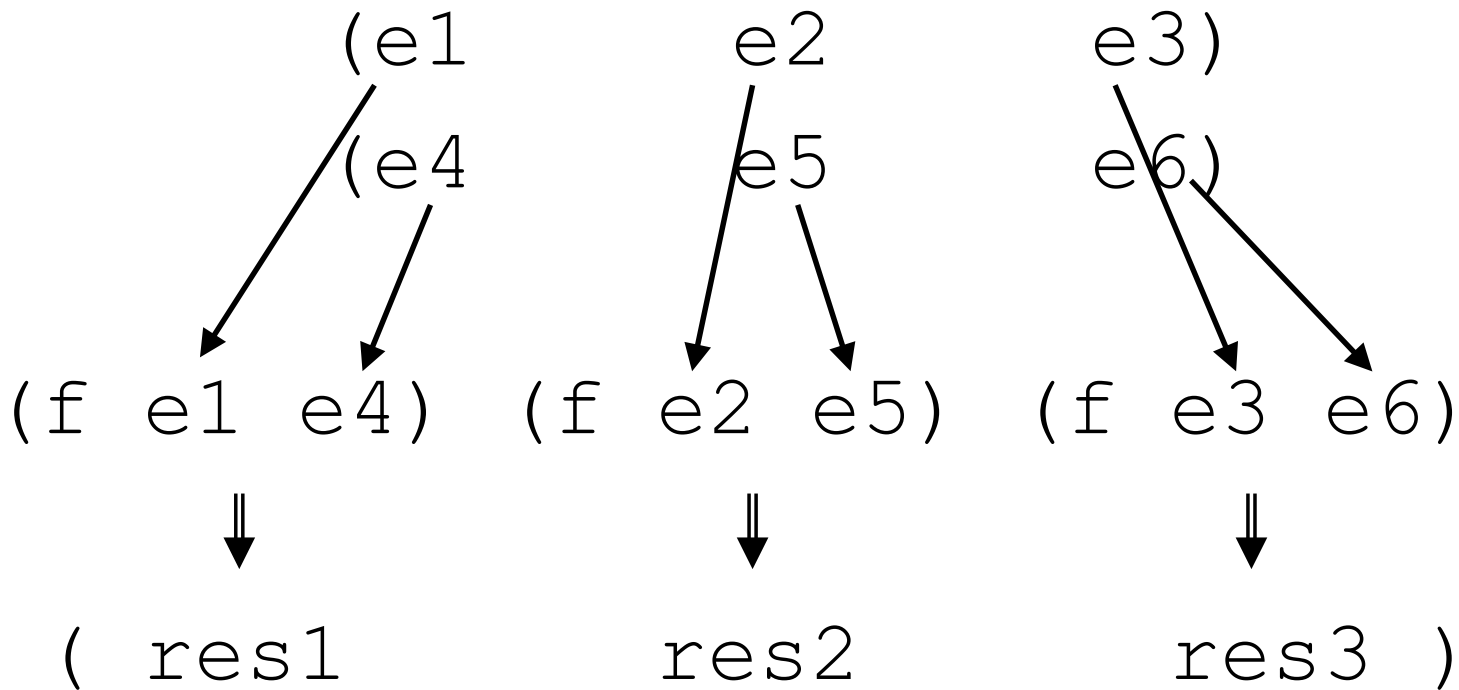
```
(map proc list1 list2 ... listN)
```

Example:

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

Map

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



Map

```
(define (my-map f . args)
  (cond
    ((null? (car args)) '())
    (else
     (cons
      (apply f (map1 car args))
      (apply my-map (cons f (map1 cdr args)))
      )
     )
    )
  )
```

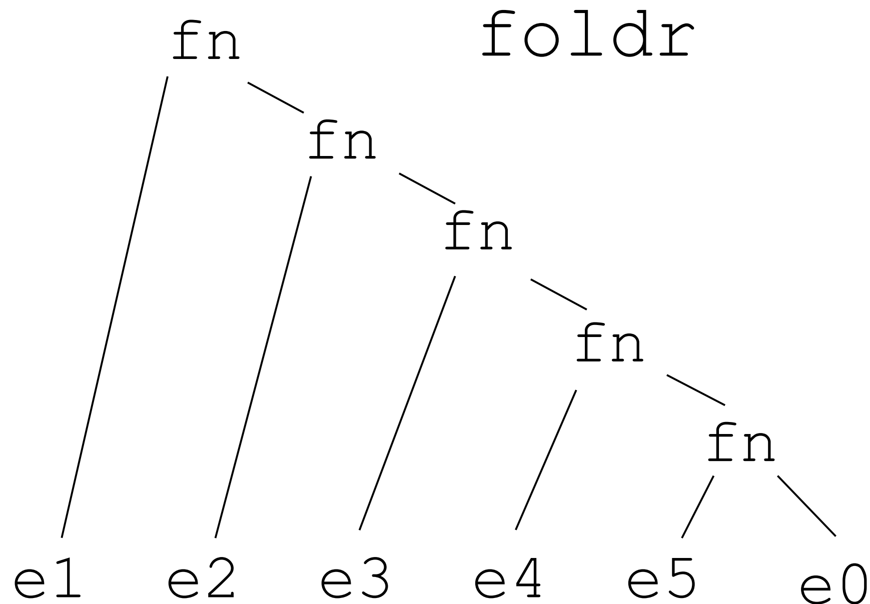
Min / sum

```
(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))
```

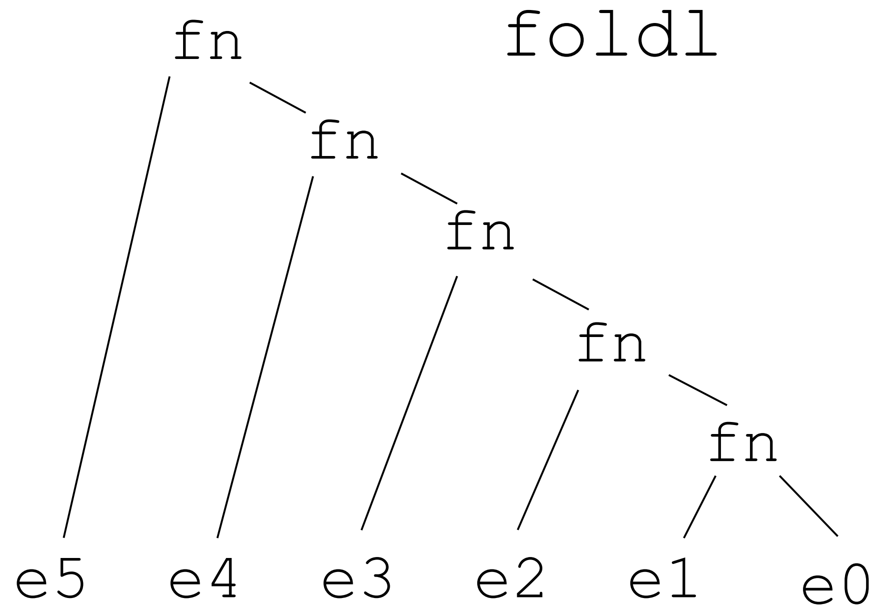


Foldr

```
(define (my-foldr f a lst)
  (cond
    ((null? lst) a)
    (else
     (f (car lst)
        (my-foldr f a (cdr lst))
        )
     )
    )
  )
```

Foldl

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

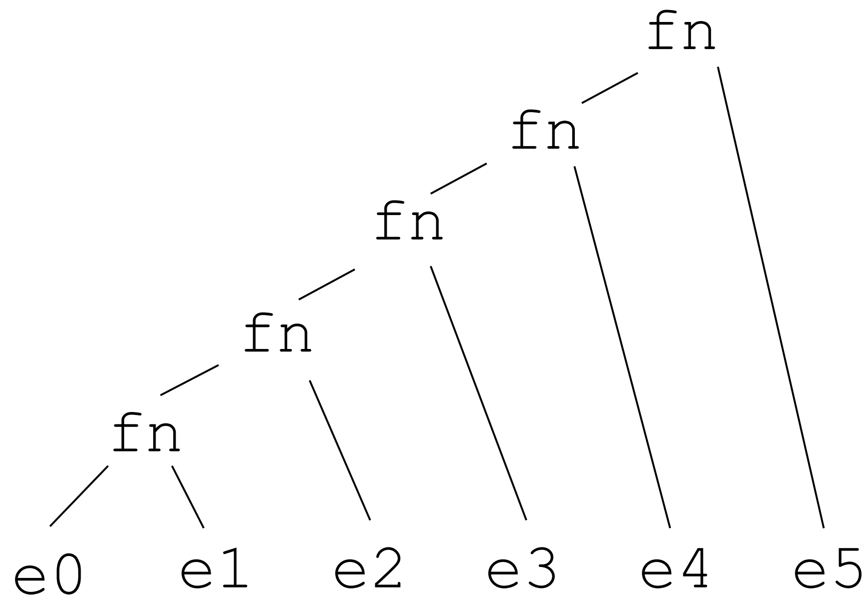


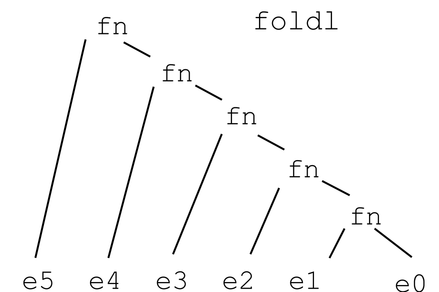
Foldl

```
(define (my-foldr f a lst)
  (cond
    ((null? lst) a)
    (else (f (car lst) (my-foldr f a (cdr lst)))
     )
    )
  )

(define (my-foldl f a lst)
  (cond
    ((null? lst) a)
    (else (my-foldl f (f (car lst) a) (cdr lst)))
    )
  )
```

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 lst)
  (cond
    ((null? lst) #t)
    (else (and (pred (car lst))
               (every1 pred (cdr lst))))
    )
  )

(define (some1 pred lst)
  (not
   (every1 (lambda (x) (not (pred x))) lst)))
```

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