CSCI-344
Programming Language Concepts (Section 3)

Lecture 9

Higher-order Functions for Lists

Instructor: Hossein Hojjat

September 19, 2016

Where we are

Done:

- Functions as first-class citizens
- Closures, Currying

This session:

- Higher-order functions for Lists

List Operation Example

Square every element of a list. For example, `'(1 2 3)` ⇒ `'(1 4 9)`

```
(define sqr (n) (* n n))
(define list-sqr (xs)
  (if (null? xs) '()
   (cons (sqr (car xs)) (list-sqr (cdr xs)))))
```

List Operation Example

Square every element of a list. For example, `'(1 2 3)` ⇒ `'(1 4 9)`

```
(define sqr (n) (* n n))
(define list-sqr (xs)
  (if (null? xs) '()
   (cons (sqr (car xs)) (list-sqr (cdr xs)))))
```

Cube every element of a list. For example, `'(1 2 3)` ⇒ `'(1 8 27)`

```
(define cube (n) (* n (* n n)))
(define list-cube (xs)
  (if (null? xs) '()
   (cons (cube (car xs)) (list-cube (cdr xs)))))
```

List Operation Example

Square every element of a list. For example, `'(1 2 3)` ⇒ `'(1 4 9)`

```
(define sqr (n) (* n n))
(define list-sqr (xs)
  (if (null? xs) '()
   (cons (sqr (car xs)) (list-sqr (cdr xs)))))
```

Cube every element of a list. For example, `'(1 2 3)` ⇒ `'(1 8 27)`

```
(define cube (n) (* n (* n n)))
(define list-cube (xs)
  (if (null? xs) '()
   (cons (cube (car xs)) (list-cube (cdr xs)))))
```

List Map

- `map`: Applies a function `f` to every element of a list `xs`

```
(define map (f xs)
  (if (null? xs) '()
    (cons (f (car xs)) (map f (cdr xs)))
  )
)

(define list-sqr (xs)
  (map (lambda (x) (* x x)) xs))

(define list-cube (xs)
  (map (lambda (x) (* x (* x x))) xs))
```

Example

What is the value of `l` after evaluating this code?

```
(define odd? (n) (= 1 (mod n 2)))
(val l (map odd? '(2 4 2 1 3)))
```

Example

What is the value of `l` after evaluating this code?

```
(define odd? (n) (= 1 (mod n 2)))
(val l (map odd? '(2 4 2 1 3)))
```

Answer: `(#f #f #f #t #t)`

Higher-order List Operations

- Higher-order list functions (e.g. `map`, `filter`, and `reduce`) capture common list-manipulation patterns
- They normally exist in standard libraries of functional languages such as Scheme
- Using those operations is usually simpler than writing recursive functions from scratch for performing list manipulation tasks

filter

- `filter` applies a predicate (boolean function) to every element of a list
- Only elements on which the predicate returns true are returned

```
   (filter (lambda (n) (< n 5)) '(4 5 7 2 10))
 = '(4 2)
```

filter

- `filter` applies a predicate (boolean function) to every element of a list
- Only elements on which the predicate returns true are returned

```
  (filter (lambda (n) (< n 5)) '(4 5 7 2 10))
= '(4 2)
```

```
(define filter (p? xs)
  (if (null? xs)
      '()
      (if (p? (car xs))
        (cons (car xs) (filter p? (cdr xs)))
        (filter p? (cdr xs)))))
```

exists?

- Applies a predicate to each element of list
- If predicate is true for an element of list returns true, otherwise returns false

```
(define exists? (p? xs)
  (if (null? xs) #f
   (if (p? (car xs)) #t
     (exists? p? (cdr xs)))))
```

exists?

- Applies a predicate to each element of list
- If predicate is true for an element of list returns true, otherwise returns false

```
(define exists? (p? xs)
  (if (null? xs) #f
   (if (p? (car xs)) #t
     (exists? p? (cdr xs)))))
```

Examples

- `(exists? pair? '(1 2 3 4))`

exists?

- Applies a predicate to each element of list
- If predicate is true for an element of list returns true, otherwise returns false

```
(define exists? (p? xs)
  (if (null? xs) #f
   (if (p? (car xs)) #t
     (exists? p? (cdr xs)))))
```

Examples

- `(exists? pair? '(1 2 3 4))` `#f`
- `(exists? pair? '(1 (2) 3 4))`

exists?

- Applies a predicate to each element of list
- If predicate is true for an element of list returns true, otherwise returns false

```scheme
(define exists? (p? xs)
  (if (null? xs) #f
   (if (p? (car xs)) #t
     (exists? p? (cdr xs)))))
```

Examples

- `(exists? pair? '(1 2 3 4))` #f
- `(exists? pair? '(1 (2) 3 4))` #t

all?

- Applies a predicate to each element of list
- If predicate is true for every element of list returns true, otherwise returns false

```
(define all? (p? xs)
  (if (null? xs)
    #t
    (if (p? (car xs))
       (all? p? (cdr xs))
       #f)))
```

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

```
'(1 2 3 4)
```

$$\Downarrow$$

$$1 + 2 + 3 + 4$$

$$\Downarrow$$

$$10$$

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

`'(1 2 3 4)`

$$\Downarrow$$

$$1 + 2 + 3 + 4$$

$$\Downarrow$$

$$10$$

- What if the list has a single element?
  - For example, reduce a list `'(3)` with the operator +
- Need an initial value: take an explicit initial value
- Reduce operation with an initial value is called **fold** operation

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

`'(1 2 3 4)`

$$\Downarrow$$

$$1 + 2 + 3 + 4$$

$$\Downarrow$$

$$10$$

- What if the list has a single element?
  - For example, reduce a list `'(3)` with the operator `+`
- Need an initial value: take an explicit initial value
- Reduce operation with an initial value is called **fold** operation

Examples:

- fold `'(1 2 3 4)` with `0` and `+`:

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

`'(1 2 3 4)`

$$\Downarrow$$

$$1 + 2 + 3 + 4$$

$$\Downarrow$$

$$10$$

- What if the list has a single element?
  - For example, reduce a list `'(3)` with the operator `+`
- Need an initial value: take an explicit initial value
- Reduce operation with an initial value is called **fold** operation

Examples:

- fold `'(1 2 3 4)` with `0` and `+`: $1 + 2 + 3 + 4 + 0 = 10$
- fold `'()` with `5` and `+`:

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

`'(1 2 3 4)`

$$\Downarrow$$

$$1 + 2 + 3 + 4$$

$$\Downarrow$$

$$10$$

- What if the list has a single element?
  - For example, reduce a list `'(3)` with the operator `+`
- Need an initial value: take an explicit initial value
- Reduce operation with an initial value is called **fold** operation

Examples:

- fold `'(1 2 3 4)` with `0` and `+`: $1 + 2 + 3 + 4 + 0 = 10$
- fold `'()` with `5` and `+`: 5
- fold `'(1 2)` with `3` and `*`:

Reducing a List

Idea: Reduce a list of elements to a single value by combining elements

`'(1 2 3 4)`

$$\Downarrow$$

$$1+2+3+4$$

$$\Downarrow$$

$$10$$

- What if the list has a single element?
  - For example, reduce a list `'(3)` with the operator `+`
- Need an initial value: take an explicit initial value
- Reduce operation with an initial value is called **fold** operation

Examples:

- fold `'(1 2 3 4)` with `0` and `+`: $1+2+3+4+0=10$
- fold `'()` with `5` and `+`: 5
- fold `'(1 2)` with `3` and `*`: $1\times 2\times 3=6$

Folding a List

- Order of combining elements does not matter in case of associative operators (like + and $\star$)
- But for non-associative operators (like subtraction) it can matter
- `foldr` groups the binary operations right-associatively,
  `foldl` groups the binary operations left-associatively.

```
(foldr f b (list x1 x2 ··· xn)) = (f x1 (f x2 ··· (f xn b)))
(foldl f b (list x1 x2 ··· xn)) = (f xn ··· (f x2 (f x1 b))···)
```

$$(\texttt{foldr}\ f\ b\ (\texttt{list}\ x_1\ x_2\ \cdots\ x_n)) = (f\ x_1\ (f\ x_2\ \cdots\ (f\ x_n\ b)))$$

$$(\texttt{foldl}\ f\ b\ (\texttt{list}\ x_1\ x_2\ \cdots\ x_n)) = (f\ x_n\ \cdots\ (f\ x_2\ (f\ x_1\ b))\cdots)$$

Folding a List

- Order of combining elements does not matter in case of associative operators (like + and $*$)
- But for non-associative operators (like subtraction) it can matter
- `foldr` groups the binary operations right-associatively, `foldl` groups the binary operations left-associatively.

```
(foldr f b (list x₁ x₂ ⋯ xₙ)) = (f x₁ (f x₂ ⋯ (f xₙ b)))
(foldl f b (list x₁ x₂ ⋯ xₙ)) = (f xₙ ⋯ (f x₂ (f x₁ b))⋯)
```

Example:

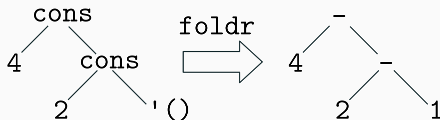
- Folding `'(4 2)` with base `1` and operation `-`
  - `foldr :` $4 - (2 - 1) = 3$
  - `foldl :` $2 - (4 - 1) = -1$

foldr vs. foldl

- We can view a fold on lists as the following steps

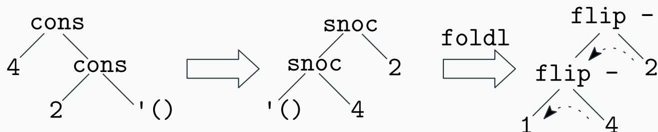
foldr

1. Replace the `nil` at the end of the list with the initial value,
2. Replace each cons with the combination function



foldl

1. Represent the list using `snoc` operation (instead of `cons`)
2. Replace the `nil` at the end of the list with the initial value,
3. Replace each `sonc` with the `flip` of combination function



Folding a List: Examples

- `(foldr mod 5 '(8 3))`
- `(foldl mod 5 '(8 3))`
- `(foldl / 2 '(4 16))`
- `(foldl cons '() '(1 2 3 4 5))`

Folding a List: Examples

- `(foldr mod 5 '(8 3))` 2
- `(foldl mod 5 '(8 3))`
- `(foldl / 2 '(4 16))`
- `(foldl cons '() '(1 2 3 4 5))`

Folding a List: Examples

- `(foldr mod 5 '(8 3))` 2
- `(foldl mod 5 '(8 3))` 0
- `(foldl / 2 '(4 16))`
- `(foldl cons '() '(1 2 3 4 5))`

Folding a List: Examples

- (**foldr** mod 5 '(8 3)) 2
- (**foldl** mod 5 '(8 3)) 0
- (**foldl** / 2 '(4 16)) 8
- (**foldl** cons '() '(1 2 3 4 5))

Folding a List: Examples

- `(foldr mod 5 '(8 3))` 2
- `(foldl mod 5 '(8 3))` 0
- `(foldl / 2 '(4 16))` 8
- `(foldl cons '() '(1 2 3 4 5))` `'(5 4 3 2 1)`

foldr and foldl

```
(define foldr (op b xs)
    (if (null? xs)
        b
        (op (car xs) (foldr op b (cdr xs)))))
```

```
(define foldl (op b xs)
    (if (null? xs)
        b
        (foldl op (op (car xs) b) (cdr xs))))
```

foldr and foldl

```
(define foldr (op b xs)
   (if (null? xs)
      b
      (op (car xs) (foldr op b (cdr xs)))))
```

```
(define foldl (op b xs)
   (if (null? xs)
      b
      (foldl op (op (car xs) b) (cdr xs))))
```

- `foldl` is tail recursive:
  recursive call is the last thing executed by the function

Tail Recursion

- Compilers can optimize **tail-recursive** functions
- Since the recursive call is the last statement, there is nothing left to do in the current function
- Saving the current function stack frame is of no use
- Current stack frame can be replaced by the stack frame of the callee

Applications of Map/Reduce

- In 2003 two clever people at Google noticed that large-scale computations could be viewed as map and reduce (fold) operations
- Hadoop Map/Reduce is a popular software framework for distributed processing of large data

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately."

[Dean and Ghemawat, 2008]

Summary

This Lecture

- Higher-order functions on lists
- `map`, `exists?`, `all?`, `foldl`, `foldr`

Next Lecture

- Continuation, μScheme Operational Semantics