Lecture 17: Map, Reduce, Filter

Dave Naumann

Department of Computer Science
Stevens Institute of Technology

CS 135 Discrete Structures Spring 2015

MapReduce

How to compute on massive datasets, e.g., to trawl web request logs (Google) or search DNS caches and network traffic logs for attacks (NSA)?

Simple data model: sets of $(key, value)$ pairs (relations, not functions).

Simple programming model:

- *map* one pair to another (or to a set of them)
- *reduce* a set of pairs to a single value (or nothing)

The model facilitates sophisticated implementation using distributed, replicated data and parallelism.

Search for "MapReduce", "Hadoop", or read a bit of the research paper "MapReduce: Simplified Data Processing on Large Clusters".

MapReduce

How to compute on massive datasets, e.g., to trawl web request logs (Google) or search DNS caches and network traffic logs for attacks (NSA)?

Simple data model: sets of $(key, value)$ pairs (relations, not functions).

Simple programming model:

- *map* one pair to another (or to a set of them)
- *reduce* a set of pairs to a single value (or nothing)

The model facilitates sophisticated implementation using distributed, replicated data and parallelism.

Search for "MapReduce", "Hadoop", or read a bit of the research paper "MapReduce: Simplified Data Processing on Large Clusters".

MapReduce

How to compute on massive datasets, e.g., to trawl web request logs (Google) or search DNS caches and network traffic logs for attacks (NSA)?

Simple data model: sets of $(key, value)$ pairs (relations, not functions).

Simple programming model:

- *map* one pair to another (or to a set of them)
- *reduce* a set of pairs to a single value (or nothing)

The model facilitates sophisticated implementation using distributed, replicated data and parallelism.

Search for "MapReduce", "Hadoop", or read a bit of the research paper "MapReduce: Simplified Data Processing on Large Clusters".

Review: lambda expressions

```
(define e (fib 50))
(define (f x) (* 27 x)) –which is shorthand for:
(define f (lambda (x) (* 27 x)))
```

The expression "(fib 50)" evaluates to a number.

The expression "(lambda (x) (* 27 x))" evaluates to a procedure.

Anonymous function, applied to argument:

```
((lambda (x) (* 2 x)) 5) = 10
```

Why? Let's try it:

```
  ((lambda (x) (* 2 x)) 5)
= (* 2 5) execute procedure by instantiating its parameters
= 10
```

Review: lambda expressions

```
(define e (fib 50))
(define (f x) (* 27 x)) –which is shorthand for:
(define f (lambda (x) (* 27 x)))
```

The expression "(fib 50)" evaluates to a number.
The expression "(lambda (x) (* 27 x))" evaluates to a procedure.

Anonymous function, applied to argument:

```
((lambda (x) (* 2 x)) 5) = 10
```

Why? Let's try it:

```
  ((lambda (x) (* 2 x)) 5)
= (* 2 5) execute procedure by instantiating its parameters
= 10
```

Review: lambda expressions

```
(define e (fib 50))
(define (f x) (* 27 x)) –which is shorthand for:
(define f (lambda (x) (* 27 x)))
```

The expression "(fib 50)" evaluates to a number.
The expression "(lambda (x) (* 27 x))" evaluates to a procedure.

Anonymous function, applied to argument:

```
((lambda (x) (* 2 x)) 5) = 10
```

Why? Let's try it:

```
  ((lambda (x) (* 2 x)) 5)
= (* 2 5) execute procedure by instantiating its parameters
= 10
```

Review: evaluation in Scheme

```
(define f (lambda (x y) (+ (* 2 x) y)))
```

```
   (f 3 (+ 1 5))
= ((lambda (x y) (+ (* 2 x) y)) 3 (+ 1 5)) evaluate operator
= ((lambda (x y) (+ (* 2 x) y)) 3 6) evaluate operands
= (+ (* 2 3) 6) instantiate function
= (+ 6 6) evaluate operands
= 12
```

Applying a lambda to its arguments: yields the body of the lambda expression, with the arguments substituted in place of the parameters.

A common idiom

Add 27 to every element of *lon*:

```
(define (add27all lon)
  (cond [(null? lon) '()]
        [else (cons (+ 27 (car lon)) (add27all (cdr lon)))]))
```

```
(define (add27all lon)
  (map (lambda (x) (+ 27 x)) lon))
```

Map (apply to each)

Function as arguments to another function (see lect18.ss):

```
(define (map fn lst)
  (cond [(null? lst) '()]
        [else (cons (fn (car lst)) (map fn (cdr lst)))]))
```

Building a lists of (x, f(x)) pairs (recall lab 3).

```
(define (graph f xs)
  (map (lambda (x) (list x (f x))) xs))
```

Map (apply to each)

Function as arguments to another function (see lect18.ss):

```
(define (map fn lst)
  (cond [(null? lst) '()]
        [else (cons (fn (car lst)) (map fn (cdr lst)))]))
```

Building a lists of (x, f(x)) pairs (recall lab 3).

```
(define (graph f xs)
  (map (lambda (x) (list x (f x))) xs))
```

Apply

What if f takes two arguments and you want to map it over a list of pairs? `(+ '(1 1))` versus `(apply + '(1 1))`

General law: `(apply f (list x y z)) = (f x y z)`

Both map and apply are built-in functions of Racket.

Apply

What if f takes two arguments and you want to map it over a list of pairs? `(+ '(1 1))` versus `(apply + '(1 1))`

General law: `(apply f (list x y z))` = `(f x y z)`

Both map and apply are built-in functions of Racket.

Apply

What if f takes two arguments and you want to map it over a list of pairs? `(+ '(1 1))` versus `(apply + '(1 1))`

General law: `(apply f (list x y z))` = `(f x y z)`

Both map and apply are built-in functions of Racket.

Another example

```
(define (negs lon) ; Assume lon is a list of numbers.
        ; Returns a list of booleans that indicate whether the
        ; corresponding input value is negative or not.
        ; For example (negs '(7 -2 0)) = '(#f #t #f)
  (cond [(null? lon) '()]
        [(< (car lon) 0) (cons #t (negs (cdr lon)))]
        [else (cons #f (negs (cdr lon)))]))
```

Can be written succinctly like this:

```
(define (negs lon)
    (map (lambda (n) (cond [(< n 0) #t] [else #f])) lon))
```

Exercise: filter

```
(define (filter lst pred?)
...)
```

Assume lst is a list of things and pred? is a function from things to boolean. Return the sub-set of lst consisting of things that satisfy pred?.

For example (filter '(1 2 3 4 5) even?) should be '(2 4)

Reduce (a collection to a single value)

Another idiom:

```
(define (sum lst)
  (cond ((null? lst) 0)
        (else (+ (car lst) (sum (cdr lst))))))
```

(reduce + '(1 2 3) 0) = (+ 1 (+ 2 (+ 3 0)))

Or in math notation: $reduce(+, (1, 2, 3)) = (1 + (2 + (3 + 0)))$

```
(define (reduce fn lst base)
  ; Assume fn is two-parameter function
  (cond [(null? lst) base]
        [else (fn (car lst) (reduce fn (cdr lst) base))]))
```

Reduce (a collection to a single value)

Another idiom:

```
(define (sum lst)
  (cond ((null? lst) 0)
        (else (+ (car lst) (sum (cdr lst))))))
```

(reduce + '(1 2 3) 0) = (+ 1 (+ 2 (+ 3 0)))

Or in math notation: $reduce(+, (1, 2, 3)) = (1 + (2 + (3 + 0)))$

```
(define (reduce fn lst base)
  ; Assume fn is two-parameter function
  (cond [(null? lst) base]
        [else (fn (car lst) (reduce fn (cdr lst) base))]))
```

Reduce (a collection to a single value)

Another idiom:

```
(define (sum lst)
  (cond ((null? lst) 0)
        (else (+ (car lst) (sum (cdr lst))))))
```

(reduce + '(1 2 3) 0) = (+ 1 (+ 2 (+ 3 0)))

Or in math notation: $reduce(+, (1, 2, 3)) = (1 + (2 + (3 + 0)))$

```
(define (reduce fn lst base)
  ; Assume fn is two-parameter function
  (cond [(null? lst) base]
        [else (fn (car lst) (reduce fn (cdr lst) base))]))
```

Examples using reduce

```
(define (reduce fn lst base)
  ; Assume fn is two-parameter function
  (cond [(null? lst) base]
        [else (fn (car lst) (reduce fn (cdr lst) base))]))
```

$\Sigma_{i=0}^{N} f(i)$ is (big-sigma f (list 0 1 ... N)) where

```
(define (big-sigma fn lst)
  (reduce (lambda (x y) (+ (fn x) y)) lst 0))
```

Example: (big-sigma (lambda (x) (* x x)) '(1 2 3) 0)

$= $ (reduce (lambda (x y) (+ (* x x) y) '(1 2 3) 0)

$= 1^2 + (2^2 + (3^2 + 0))$

Exercise

```
(define (length lon)
  (cond [(null? lon) 0]
        [else (+ 1 (length (cdr lon)))]))
```

Express this using reduce.