Functional Programming in Scheme and Lisp

Overview

- In a functional programming language, functions are first class objects.
- You can create them, put them in data structures, compose them, specialize them, apply them to arguments, etc.
- We'll look at how functional programming things are done in Lisp

eval

- Remember: Lisp code is just an s-expression
- You can call Lisp's evaluation process with the eval function.
  - (define s (list 'cadr '(one two three)))
  - s
  - (CADR '(ONE TWO THREE))

  > (eval s)

  TWO

  > (eval (list 'cdr (car '((quote (a . b)) c))))

  B

Apply

- *Apply* takes a function and a list of arguments for it, and returns the result of applying the function to the arguments:

```
> (apply  +  '(1  2  3))
6
```

- It can be given any number of arguments, so long as the last is a list:

```
> (apply  +  1  2  '(3  4  5))
15
```

- A simple version of apply could be written as

```
(define (apply f list) (eval (cons f list)))
```

Lambda

- The *define* special form creates a function and gives it a name.
- However, functions don't have to have names, and we don't need *define* to define them.
- We can refer to functions literally by using a *lambda expression*.

Lambda expression

- A *lambda expression* is a list containing the symbol *lambda*, followed by a list of *parameters*, followed by a *body* of zero or more expressions:

```
> (define f (lambda (x) (+ x 2)))
> f
#<proceedure:f>
> (f 100)
102
```

Lambda expression

- A lambda expression is a special form
- When evaluated, it creates a function and returns a reference to it
- The function does not have a name
- a lambda expression can be the first element of a function call:

```
> ( (lambda  (x)  (+  x  100))  1)
101
```

- Other languages like python and javascript have adopted the idea

define vs. define

```
(define (add2 x)
     (+ x 2) )

(define add2
   (lambda (x) (+ x 2)))

(define add2 #f)
(set! add2
    (lambda (x) (+ x 2)))
```

- The define special form comes in two varieties
- The three expressions to the right are entirely equivalent
- The first define form is just more familiar and convenient when defining a function

Mapping functions

- Common Lisp and Scheme provides several mapping functions
- ***map*** *(mapcar in Lisp)* is the most frequently used.
- It takes a function and one or more lists, and returns the result of applying the function to elements taken from each list, until one of the lists runs out:

```
> (map abs '(3 -4 2 -5 -6))
(3 4 2 5 6)
> (map cons '(a b c) '(1 2 3))
((a . 1) (b . 2) (c . 3))
 > (map (lambda  (x)  (+  x  10))  ‘(1  2  3))
(11  12  13)
> (map  list  ‘(a  b  c)  ‘(1  2  3  4))
map: all lists must have same size; arguments
  were: #<procedure:list> (1 2) (a b c)
```

Defining map

- Defining a simple “one argument” version of map is easy

```
(define (map1 func list)
   (if (null? list)
       null
       (cons (func (first list))
             (map1 func (rest list)))))
```

Define Lisp's Every and Some

- *every* and *some* take a predicate and one or more sequences
- When given just one sequence, they test whether the elements satisfy the predicate:
  - (every odd? '(1 3 5))
  - #t
  - (some even? '(1 2 3))
  - #t
- If given >1 sequences, the predicate takes as many args as there are sequences and args are drawn one at a time from them:
  - (every > '(1 3 5) '(0 2 4))
  - #t

every

```
(define (every f list)
   ;; note the use of the and function
   (if (null? list)
       #t
       (and (f (first list))
            (every f (rest list)))))
```

some

```
(define (some f list)
  (if (null? list)
      #f
      (or (f (first list))
          (some f (rest list)))))
```

Will this work?

```
(define (some f list)
    (not (every (lambda (x) (not (f x)))
                list)))
```

filter

(filter <f> <list>) returns a list of the elements of <list> which satisfy the predicate <f>

```
> (filter odd? '(0 1 2 3 4 5))
(1 3 5)
> (filter (lambda (x) (> x 98.6))
    '(101.1 98.6 98.1 99.4 102.2))
(101.1 99.4 102.2)
```

Example: filter

```
(define (myfilter list func)
  ;; returns a list of elements of list where function is true
  (cond ((null? list) nil)
        ((func (first list))
         (cons (first list)
               (myfilter (rest list) func)))
        (#t (myfilter (rest list) func))))
```

```
> (myfilter '(1 2 3 4 5 6 7) even?)
(2 4 6)
```

Example: filter

- Define *integers* as a function that returns a list of integers between a min and max

```
(define (integers min max)
  (if (> min max)
      empty
      (cons min (integers (add1 min) max))))
```

- And prime? as a predicate that is true of prime numbers and false otherwise

```
> (filter prime? (integers 2 20) )
(2 3 5 7 11 13 17 19)
```

Here's another pattern

- We often want to do something like sum the elements of a sequence

```
(define (sum-list l)
    (if (null? l)
        0
        (+ (first l) (sum-list (rest l)))))
```

- And other times we want their product

```
(define (multiply-list l)
    (if (null? l)
        1
        (* (first l) (multiply-list (rest l)))))
```

Here's another pattern

- We often want to do something like sum the elements of a sequence

```
(define (sum-list l)
    (if (null? l)
        0
        (+ (first l) (sum-list (rest l)))))
```

- And other times we want their product

```
(define (multiply-list l)
    (if (null? l)
        1
        (* (first l) (multiply-list (rest l)))))
```

Example: reduce

- Reduce takes (i) a function, (ii) a final value, and (iii) a list
- Reduce(+ 0 (v1 v2 v3 … vn)) is just

  V1 + V2 + V3 + … Vn +0
- In Scheme/Lisp notation:

  > (reduce + 0 ‘(1 2 3 4 5))

  15

  (reduce * 1 ‘(1 2 3 4 5))

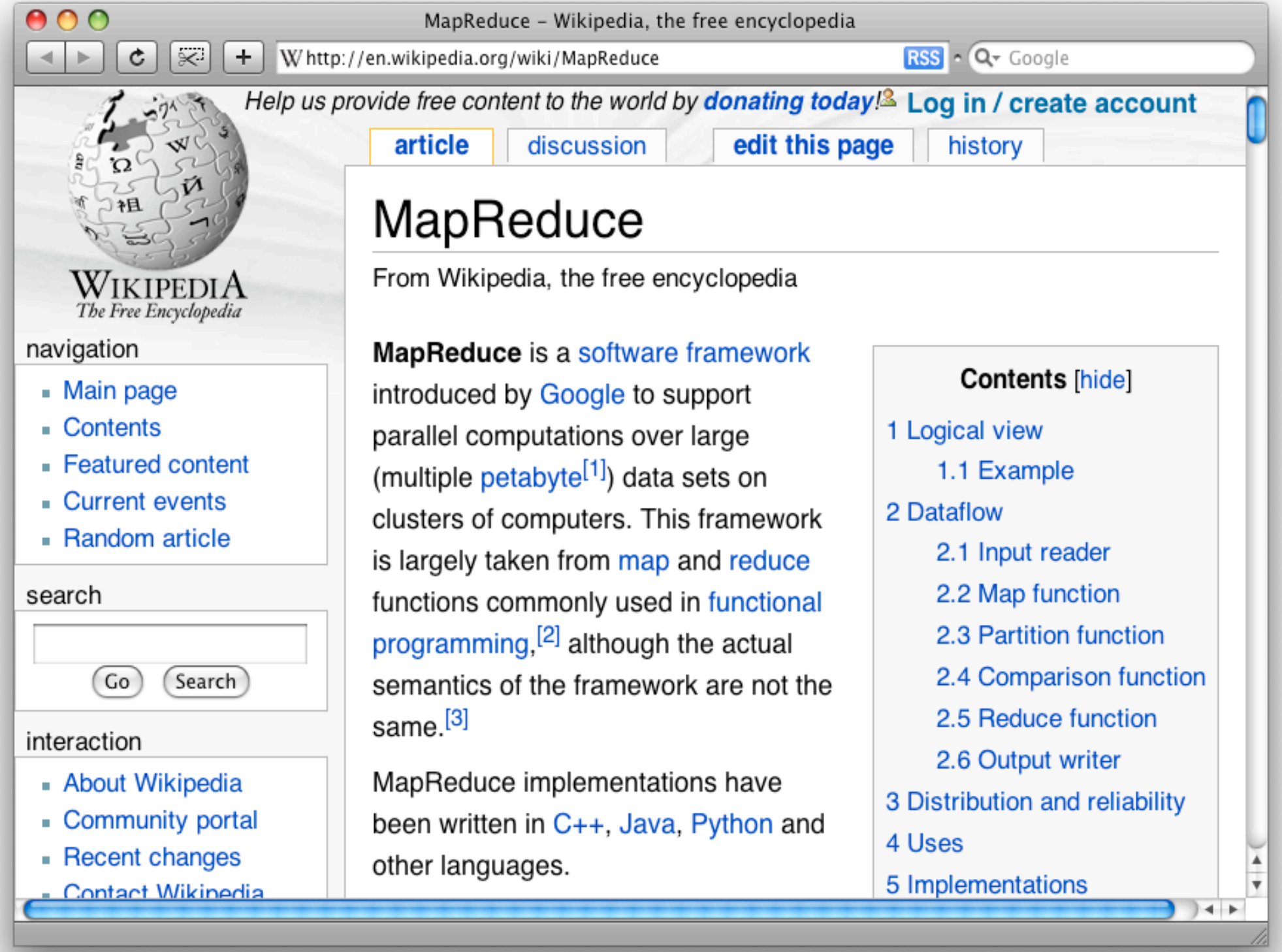
  120

Example: reduce

```
(define (reduce function final list)
  (if (null? list)
      final
      (function
          (first list)
          (reduce function final (rest list)))))
```

Using reduce

```
(define (sum-list list)
   ;; returns the sum of the list elements
   (reduce + 0 list))
(define (mul-list list)
   ;; returns the sum of the list elements
   (reduce * 1 list))
(define (copy-list list)
   ;; copies the top level of a list
   (reduce cons '() list))
(define (append-list list)
   ;; appends all of the sublists in a list
   (reduce append '() list))
```


MapReduce - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/MapReduce
RSS
Google
Help us provide free content to the world by donating today!
Log in / create account
article
discussion
edit this page
history
WIKIPEDIA
The Free Encyclopedia
navigation
Main page
Contents
Featured content
Current events
Random article
search
Go
Search
interaction
About Wikipedia
Community portal
Recent changes
Contact Wikipedia
MapReduce
From Wikipedia, the free encyclopedia
MapReduce is a software framework introduced by Google to support parallel computations over large (multiple petabyte[1]) data sets on clusters of computers. This framework is largely taken from map and reduce functions commonly used in functional programming,[2] although the actual semantics of the framework are not the same.[3]
MapReduce implementations have been written in C++, Java, Python and other languages.
Contents [hide]
1 Logical view
1.1 Example
2 Dataflow
2.1 Input reader
2.2 Map function
2.3 Partition function
2.4 Comparison function
2.5 Reduce function
2.6 Output writer
3 Distribution and reliability
4 Uses
5 Implementations

Composing functions

```
> compose
#<procedure:compose>
> (define (square x) (* x x))
> (define (double x) (* x 2))
> (square (double 10))
400
> (double (square 10))
200
> (define sd (compose square double))
> (sd 10)
400
> ((compose double square) 10)
200
```

Here's how to define it

```
(define (my-compose f1 f2)
  (lambda (x) (f1 (f2 x))))
```

Variables, free and bound

- In this function, to what does the variable *GOOGOL* refer?

```
(define (big-number? x)
    ;; returns true if x is a really big number
    (> x GOOGOL))
```

- The **scope** of the variable X is just the body of the function for which it's a parameter.

Here, GOOGOL is a global variable

```
> (define GOOGOL (expt 10 100))
> GOOGOL
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
> (define (big-number? x) (> x GOOGOL))
> (big-number? (add1 (expt 10 100)))
#t
```

Which X is accessed at the end?

```
> (define GOOGOL (expt 10 100))
> GOOGOL
100000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000
    000000000000000
> (define x -1)
> (define (big-number? x) (> x GOOGOL))
> (big-number? (add1 (expt 10 100)))
#t
```

Variables, free and bound

- In the body of this function, we say that the variable (or symbol) X is **bound** and GOOGOL is **free**.

```
(define (big-number? x)
  ; returns true if X is a really big number
  (> X GOOGOL))
```

- If it has a value, it has to be bound somewhere else

The let form creates local variables

```
> (let [(pi 3.1415)
        (e 2.7168)]
      (big-number? (expt pi e)))
#f
```

Note: square brackets are line parens, but only match other square brackets. They are there to help you cope with paren fatigue.

- The general form is (let <varlist> . <body>)
- It creates a local environment, binding the variables to their initial values, and evaluates the expressions in <body>

Let creates a block of expressions

```
(if (> a b)
    (let (  )
         (printf "a is bigger than b.~n")
         (printf "b is smaller than a.~n")
          #t)
    #f)
```

Let is just syntactic sugar for lambda

```
(let [(pi 3.1415) (e 2.7168)]
   (big-number? (expt pi e)))

((lambda (pi e) (big-number? (expt pi e)))
  3.1415
  2.7168)
```

and this is how we did it back before ~1973

Let is just syntactic sugar for lambda

What happens here:

```
(define x 2)
(let [ (x 10) (xx (* x 2)) ]
    (printf "x is ~s and xx is ~s.~n" x xx))
```

Let is just syntactic sugar for lambda

What happens here:

```
(define x 2)
( (lambda (x xx) (printf "x is ~s and xx is ~s.~n" x xx))
  10
  (* 2 x))
```

Let is just syntactic sugar for lambda

What happens here:

```
(define x 2)

(define (f000034 x xx)
  (printf "x is ~s and xx is ~s.~n" x xx))
(f000034 10 (* 2 x))
```

let and let*

- The let special form evaluates all of the initial value expressions, and then creates a new environment in which the local variables are bound to them, “in parallel”
- The let* form does is sequentially
- let* expands to a series of nested lets
  - (let* [(x 100)(xx (* 2 x))] (foo x xx))
  - (let [(x 100)]
    (let [(xx (* 2 x))]
    (foo x xx)))

What happens here?

```
> (define X 10)
> (let [(X (* X X))]
        (printf "X is ~s.~n" X)
        (set! X 1000)
        (printf "X is ~s.~n" X)
        -1 )
???
> X
???
```

What happens here?

```
> (define X 10)
```

-
```
(let [(X (* X X))]
      (printf “X is ~s\n” X)
      (set! X 1000)
      (printf “X is ~s\n” X)
      -1 )
```

```
X is 100
X is 1000
-1
> X
10
```

What happens here?

```
> (define GOOGOL (expt 10 100))
> (define (big-number? x) (> x GOOGOL))
> (let [(GOOGOL (expt 10 101))]
    (big-number? (add1 (expt 10 100))))
```

???

What happens here?

```
> (define GOOGOL (expt 10 100))
> (define (big-number? x) (> x GOOGOL))
> (let [(GOOGOL (expt 10 101))]
    (big-number? (add1 (expt 10 100))))
#t
```

- The free variable GOOGOL is looked up in the environment in which the big-number? Function was defined!

functions

- Note that a simple notion of a function can give us the machinery for
  - Creating a block of code with a sequence of expressions to be evaluated in order
  - Creating a block of code with one or more local variables
- Functional programming language is to use functions to provide other familiar constructs (e.g., objects)
- And also constructs that are unfamiliar

Dynamic vs. Static Scoping

- Programming languages either use dynamic or static (aka lexical) scoping
- In a statically scoped language, free variables in functions are looked up in the environment in which the function is defined
- In a dynamically scoped language, free variables are looked up in the environment in which the function is called

Closures

- Lisp is a **lexically scoped** language.
- Free variables referenced in a function those are looked up in the environment in which the function is defined.

  Free variables are those a function (or block) doesn't create scope for.
- A **closure** is a function that remembers the environment in which it was created
- An **environment** is just a collection of variable bindings and their values.

Closure example

```
> (define (make-counter)
    (let ((count 0))  (lambda () (set! count (add1 count)))))
> (define c1 (make-counter))
> (define c2 (make-counter))
> (c1)
1
> (c1)
2
> (c1)
3
> (c2)
???
```

A fancier make-counter

Write a fancier make-counter function that takes an optional argument that specifies the increment

```
> (define by1 (make-counter))
> (define by2 (make-counter 2))
> (define decrement (make-counter -1))
> (by2)
2
(by2)
4
```

Optional arguments in Scheme

```
(define (make-counter . args)
  ;; args is bound to a list of the actual arguments passed to the
     function
  (let [(count 0)
        (inc (if (null? args) 1 (first args)))]
    (lambda ( ) (set! count (+ count inc)))))
```

Keyword arguments in Scheme

- Scheme, like Lisp, also has a way to define functions that take *keyword arguments*
  - (make-counter)
  - (make-counter :initial 100)
  - (make-counter :increment -1)
  - (make-counter :initial 10 :increment -2)
- Different Scheme dialects have introduced different ways to mix positional arguments, optional arguments, default values, keyword argument, etc.

Closure tricks

We can write several functions that are closed in the same environment, which can then provide a private communication channel

```
(define foo #f)
(define bar #f)

(let ((secret-msg "none"))
  (set! foo
    (lambda (msg)
      (set! secret-msg msg)))
  (set! bar
    (lambda () secret-msg)))

(display (bar)) ; prints "none"
(newline)
(foo ”attack at dawn")
(display (bar)) ; prints ”attack at dawn"
```