Functional Programming in Lisp

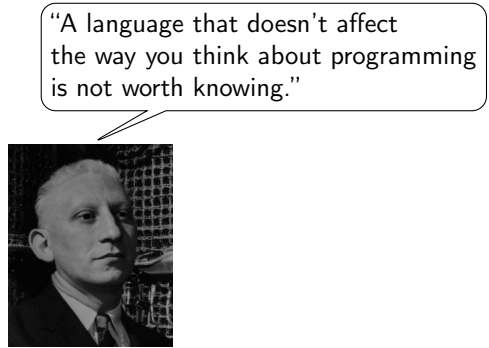
Dr. Neil T. Dantam

CSCI-561, Colorado School of Mines

Fall 2020

Alan Perlis on Programming Languages

https://doi.org/10.1145%2F947955.1083808



Alan J. Perlis (1922-1990)
First Turing Award Recipient, 1966

MINES

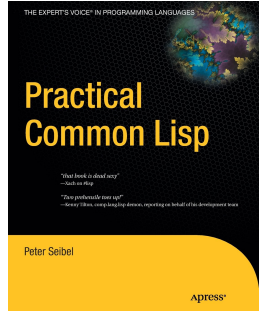
Eric Raymond and Paul Graham on Lisp

Lisp is worth learning for a different reason—
the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.
–Eric Raymond
`http://www.catb.org/esr/faqs/hacker-howto.html`

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one.
(This is probably what Eric Raymond meant about Lisp making you a better programmer.)
–Paul Graham
`http://www.paulgraham.com/avg.html`

Peter Siebel on Lisp, “Blub,” and “Omega”

https://youtu.be/4NO83wZVT0A?t=18m26s



Introduction

Functional Programming Features

- **Persistence:** variables and data structures are immutable (constant)
- **Recursion:** construct algorithms as recursive functions (vs. loops)
- **First-class functions:** can be passed to and returned from other functions

Outcomes

- Review/understand concepts of functional programming
- Implement Lisp programs in functional style

Trade-offs of Functional Programming

Pros

- Easy (comparatively) to reason about (prove) correctness
- Compact (fewer LoC)
- Immutable structures shared between modules, threads, etc.

Cons

- Different way of thinking about programs (also a pro!)
- Sometimes less (constant-factor) efficient

Well-aligned with the algorithms and proofs in this course.

Outline

Recursion

First-Class Functions

Higher-order Functions

Outline

Definition: Recursion

Review

Definition (Recursion)

A function or other object defined in terms of itself.

Base Case: Terminating condition

Recursive Case: Reduction towards the base case

Example (Factorial)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$

Function fact(n)

1 **if** $0 = n$ **then return** 1 ;
2 **else return** $n * \texttt{fact}(n-1)$;

Example: Factorial

"Math"

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$

Pseudocode

Procedure fact(x)

```
1 if 0 = x then /* Base Case */
2 |   return 1;
3 else /* Recursive Case     */
4 |   return x * fact(x − 1);
```

Common Lisp

```
(defun fact (n)
  (if (= n 0)
      1
      (* n
         (fact (- n 1)))))
```

Exercise: Recursive Fibonacci Sequence

$$(1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55, \ldots)$$

$$\mathrm{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

Exercise: Recursive Fibonacci Sequence

continued

Pseudocode

Procedure fib(x)

1

Common Lisp

Example: Recursive Accumulate in Lisp

Pseudocode

Function accum(S)

```
1 if S then // Recursive Case
2 |  return car(S) + accum(cdr(S));
3 else // Base Case
4 |  return 0;
```

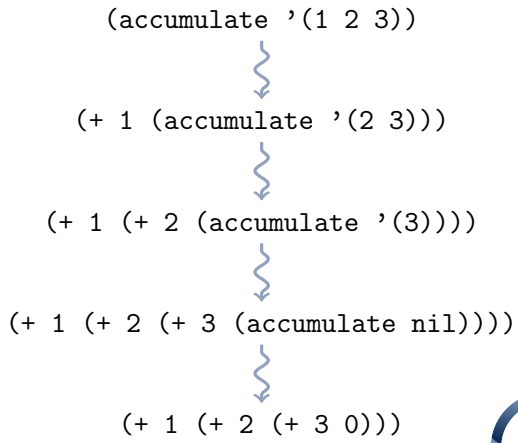
Recursive Accumulate in Lisp

```
(defun accum (list)
  (if list
      ;; recursive case
      (+ (car list)
         (accum (cdr list)))
      ;; base case
      0))
```

Example: Recursive Accumulate Execution Trace

Recursive Accumulate in Lisp

```
(defun accum (list)
  (if list
      ;; recursive case
      (+ (car list)
         (accum (cdr list)))
      ;; base case
      0))
```



Example: Alternate Recursive Accumulate

Accumulate

```
(defun accum (list)
  (if list
      ;; recursive case
      (+ (car list)
         (accum (cdr list)))
      ;; base case
      0))
```

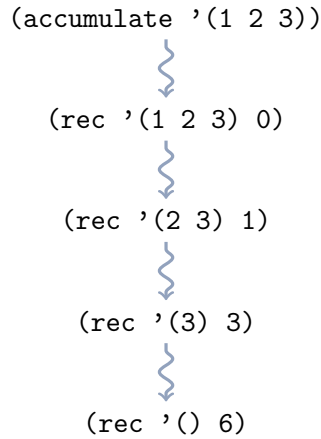
Alternate Accumulate

```
(defun accum (list)
  (labels ((rec (list accum)
             (if list
                 ;; recursive case
                 (rec (cdr list)
                      (+ (car list)
                         accum))
                 ;; base case
                 accum)))
    (rec list 0)))
```

Example: Alternate Accumulate Execution Trace

Alternate Accumulate

```
(defun accum (list)
  (labels ((rec (list accum)
             (if list
                 ;; recursive case
                 (rec (cdr list)
                      (+ (car list)
                         accum))
                 ;; base case
                 accum)))
    (rec list 0)))
```



Exercise: Recursive Reverse

$$(a_0\ a_1\ \ldots\ a_{n-1}\ a_n) \overset{\text{reverse}}{\rightsquigarrow} (a_n\ a_{n-1}\ \ldots\ a_1\ a_0)$$

Pseudocode

Procedure reverse(L)

1

Common Lisp

DESTRUCTURING-BIND

DESTRUCTURING-BIND

DESTRUCTURING-BIND bind variables to corresponding values draw from a list.

Example

```
(destructuring-bind (a b c)
    '(1 2 3)
  (list c b a))
```

Output

```
(3 2 1)
```

Example

```
(destructuring-bind (op e1 e2)
    '(+ 1 2)
  (+ e1 e2))
```

Output

```
3
```

Example: `destructuring-bind`

Return S-Expression as infix string

Code

```
(defun arith-string (e)
  "Return s-exp E as an infix string."
  (if (listp e)
      (destructuring-bind (op e1 e2)
          e
        (format nil "(~A) ~A (~A)"
                (arith-string e1)
                op
                (arith-string e2)))
      (format nil "~A" e)))
```

Exercise: `destructuring-bind`

Evaluate Arithmetic S-Expression

Code

Outline

First-class functions

Definition: First-class functions

A programming language has **first-class functions** when it treats functions like any other variable or object. First-class functions can be:

- Bind variables to the function
- Passed as arguments to other functions
- Returned as the result of other functions

Function Closure

Definition (Function Closure)

A function closure or lexical closure is a function and an associated set of variable definitions. Etymology: from "closed expression."

Example (C Function Pointer)

```c
/* Definition */
struct context {
    int val;
};

int adder(struct context *cx, int x) {
    return cx->a + x;
}

/* Usage */
struct context c;
c.val = 1;
int y = adder(c,2);
```

Example (Java Class)

```java
// Definition
class Adder {
    public int a;
    public Adder(int a_) {
        a = a_;
    }
    public int call(int x) {
        return x+a;
    }
}

// Usage
Adder A = new Adder(1);
int y = A.call(2);
```

Closures in Lisp: Local Functions

LABELS

Defines local functions and executes body using those local functions:

```
(labels ((FUNCTION-NAME VARIABLES FUNCTION-BODY)...)  LABELS-BODY)
```

Example

```
(let ((a 1))
  (labels ((adder (x)
             (+ x a)))
    (adder 2)))
```

Output

3

Closures in Lisp: Anonymous Functions

LAMBDA

Defines an anonymous function:

```
(lambda VARIABLES FUNCTION-BODY)
```

FUNCALL

Apply a function to the provided arguments:

```
(funcall FUNCTION ARGUMENTS...)
```

Example

```
(let ((a 1))
  (funcall (lambda (x)
             (+ x a))
           2))
```

Output

3

Value and Function Namespaces

Value Namespace

- Records values
- Local: `let`, `let*`
- Global: `defparameter`

Function Namespace

- Records function definitions
- Local: `labels`, `flet`
- Global: `defun`

Example

```
(defun foo (x) (+ 1 x))

(let ((foo 10))
  (print foo)          ; => 10
  (print (foo 1))      ; => 2
  (print (foo foo)))   ; => 11
```

Output

```
10
2
11
```

function and funcall

FUNCTION

Returns the functional value of a name:

(function NAME)

⇝ The function bound to name

Example

- (function +)
- (#' +)
- (defun foo (x) (+ 1 x))
  #'foo
- (labels ((foo (x) (+ 1 x)))
    #'foo)

FUNCALL

Apply a function to the provided arguments:

(funcall FUNCTION ARGS...)

⇝ Return value of FUNCTION called on ARGS....

Example

- (funcall (lambda (x)
    (+ 1 x))
  1)
  ⇝ 2
- (funcall #'+ 1 2) ⇝ 3

Outline

Higher-order functions

Definition: Higher-order function

A function that takes another function as an argument or returns another function as its result.

Example (Passing)

```
Function f(g,a)
1 return g(42, a);
```

Example (Returning)

```
Function f(a)
1 function g(b) is
2     return a + b;
3 return g;
```

Counterexample

```
Function f(a,b)
1 return a + b;
```

Example: Sorting

Example (`sorted()` in Python)

```
sorted(["a", "ccc", "bb"], key=len)
# => ['a', 'bb', 'ccc']
```

Example (`qsort()` in libc)

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*cmp)(const void *a, const void *b));
void qsort_r(void *base, size_t nmemb, size_t size,
             int (*cmp)(const void *a, const void *b,
                        void *cx),
             void *cx);
```

Example: Sorting in Lisp

Example (Predicate)

```
(sort '("a" "ccc" "bb")
      (lambda (a b)
        (< (length a)
           (length b))))

;; => "a" "bb" "ccc"
```

Example (Predicate and Key)

```
(sort '("a" "ccc" "bb")
      #'< :key #'length)

;; => "a" "bb" "ccc"
```

Exercise: Sorting in Lisp

Call `SORT` to sort the following list in numerically descending order:

```
(14 63 8 11 51 24 71 89 17 42)
```

Sort

Common Higher-order Functions

map transform elements of a collection

fold combine elements of a collection

Map function

Definition (map)

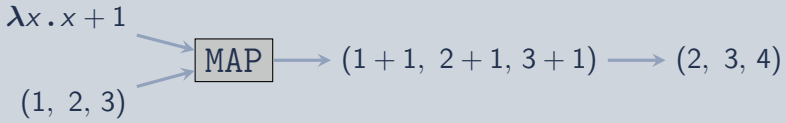
Apply a function to every member of an input sequence, and collect the results into the output sequence.

$$\texttt{map} \quad : \quad \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \quad \times \quad \underbrace{\mathbb{X}^n}_{\text{input sequence}} \quad \mapsto \quad \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

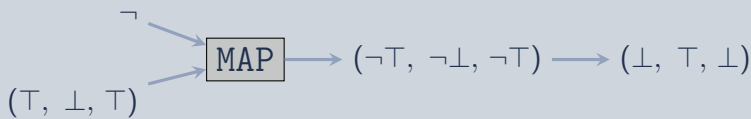
Illustration

Example: Map

Increment every element by one



Not (¬) every element



Algorithm: Map function on lists

Functional Map

Function map(f,s)

```
1 if empty(s) then /* s is empty                    */
2 |   return NIL
3 else /* s has members                             */
4 |   return cons(f(first(s)), map(f, rest(s)));
```

Imperative Map

Procedure map(f,s)

1 $n \leftarrow \texttt{length}(s)$;
2 $Y \leftarrow \texttt{make-sequence}(n)$;
3 $i \leftarrow 0$;
4 **while** $i < n$ **do**
5 $\quad Y[i] \leftarrow f(s[i])$;
6 $\quad i \leftarrow i + 1$;
7 **return** Y;

Example: Map

Example (Illustration)

Example (Common Lisp)

```
(map 'list                    ; result type
     (lambda (x) (+ 1 x))     ; function
     (list 1 2 3))            ; sequence
;; RESULT: (2 3 4)
```

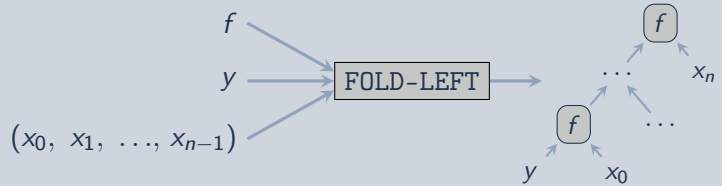
Fold-left

Definition (fold-left)

Apply a binary function to each member of a sequence and the prior result, starting from the left.

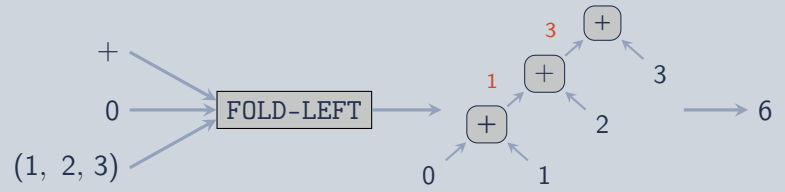
$$\text{fold-left} : \underbrace{(\mathbb{Y} \times \mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

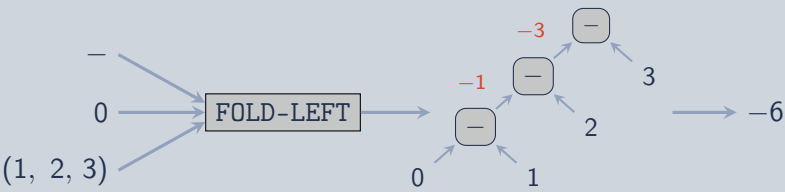


Example: Fold-left

Example (Addition)



Example (Subtraction)



Algorithm: Fold-left

Functional

Function fold-left(f, y, X)

```
1 if empty(X) then return y;
2 else
3     let y' ← f(y, first(X)) in
4         return fold-left(f, y', rest(X));
```

Imperative

Procedure fold-left(f, y, X)

```
1 i ← 0;
2 while i < |X| do
3     y ← f(y, X_i);
4     i ← i + 1;
5 return y;
```

Example: Fold-left in Lisp

Example (Addition)

```
(reduce #'+
        '(1 2 3)
        :initial-value 0)
;;; => (+ (+ (+ 0 1) 2) 3)
;;; => 6
```

Example (Subtraction)

```
(reduce #'-
        '(1 2 3)
        :initial-value 0)
;;; => (- (- (- 0 1) 2) 3)
;;; => -6
```

Example: Fold-Left Accumulate

Pseudocode

Procedure accum-fold(L)

1 **function** $h(a, x)$ **is**
2 $\quad$ **return** $a + x$;
3 **return fold-left**$(h, 0, L)$;

Lisp

```
(defun accum-fold (list)
  (reduce (lambda (accum x)
            (+ x accum))
          list
          :initial-value 0))
```

Exercise: Fold-Left Reverse

$$(a_0\ a_1\ \ldots\ a_{n-1}\ a_n) \overset{\text{reverse}}{\rightsquigarrow} (a_n\ a_{n-1}\ \ldots\ a_1\ a_0)$$

Pseudocode

Procedure reverse(L)

1

Common Lisp

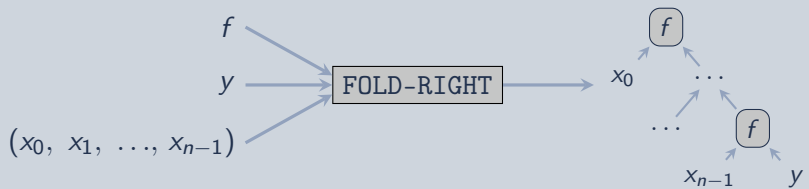
Fold-right

Definition (fold-right)

Apply a binary function to each member of a sequence and the prior result, starting from the right.

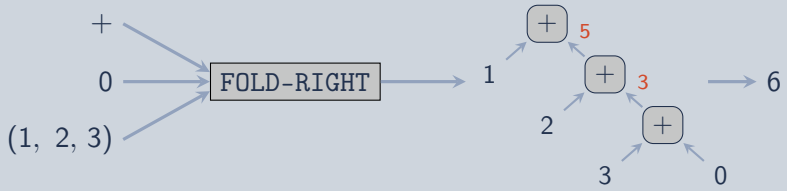
$$\text{fold-right} : \underbrace{(\mathbb{X} \times \mathbb{Y} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

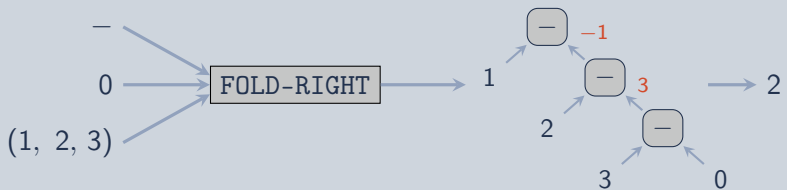


Example: Fold-right

Example (Addition)



Example (Subtraction)



Algorithm: Fold-right

Recursive

Function fold-right(f,y,X)

```
1 if empty(X) then return y;
2 else
3     let y' ← fold-right(f, y, rest(X)) in
4         return f(first(X), y');
```

Procedural

Procedure fold-right(f,y,X)

```
1 i ← |X| − 1;
2 while i ≥ 0 do
3     y ← f(X_i, y) ;
4     i ← i − 1 ;
5 return y;
```

Example: Fold-right in Lisp

Example (Addition)

```
(reduce #'+
        '(1 2 3)
        :initial-value 0
        :from-end t)
;;; => (+ 1 (+ 2 (+ 3 0)))
;;; => 6
```

Example (Subtraction)

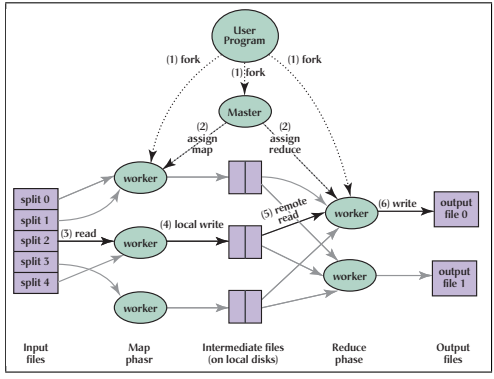
```
(reduce #'-
        '(1 2 3)
        :initial-value 0
        :from-end t)
;;; => (- 1 (- 2 (- 3 0)))
;;; => 2
```

Application: MapReduce

Function MapReduce(f,g,X)

1 $Y \leftarrow$ parallel-map(f, X);
2 **return** reduce(g, Y);

- Idea:
  - (parallel) map
  - (serial) reduce/fold
- Provides scalability, fault-tolerance
- Implementations:
  - Google MapReduce
  - Apache Hadoop



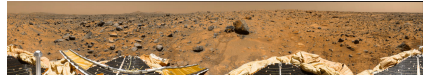
Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM. 2008.

Summary

- **Functional Style:**
  - Avoid side-effects (assignment), often recursive
  - Aligns with inductive proofs
- **S-Expressions:**
  - Abstract representation of mathematical expressions
  - Thinking about expression structure, not syntax
- **Homoiconicity:**
  - Same structure for code and data
  - Process code just like any other data structure
- **Applications in symbolic reasoning:**
  - Computer Algebra Systems (CAS)
  - (Robot) Planning and Scheduling

Maxima
CAS

TI-89
(Derive CAS)



Mars Pathfinder planner

Learn to think and program functionally and symbolically.

MINES

Dijkstra on Lisp

"LISP has jokingly been described as 'the most intelligent way to misuse a computer'. I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in **thinking previously impossible thoughts**." [emphasis added]

https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

MINES