LISP

Moritz Heidkamp
moritz@twoticketsplease.de

August 20, 2011
Greenspun’s Tenth Rule

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
Where Does Lisp Come From?

John McCarthy, 1958

You're Doing It Completely Wrong.
LISP is now the second oldest programming language in present widespread use (after FORTRAN and not counting APT, which isn’t used for programming per se).

WHERE DOES LISP COME FROM?

IBM 704, 1960/61
A Lisp Program From ca. 1961

```
DEFINE ((
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
           ((EQ A (CAR X)) T) (T (MEMBER A (CDR X))))))
(UNION (LAMBDA (X Y) (COND ((NULL X) Y) ((MEMBER
          (CAR X) Y) (UNION (CDR X) Y)) (T (CONS (CAR X)
          (UNION (CDR X) Y)))))
(INTERSECTION (LAMBDA (X Y) (COND ((NULL X) NIL)
          ((MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECTION
          (CDR X) Y))) (T (INTERSECTION (CDR X) Y)))))
))
INTERSECTION ((A1 A2 A3) (A1 A3 A5))
UNION ((X Y Z) (U V W X))
```
What Is Lisp?

Homoiconic
Code = Data
What Is Lisp?

Homoiconic
Code = Data
What Is Lisp?

Homoiconic
Code = Data
What Is Lisp?

Garbage Collection
WHAT IS LISP?

Programmable Programming Language
Multi-paradigm
What Is Lisp?

Programmable Programming Language
Multi-paradigm
What Is Lisp?

Programmable Programming Language
Multi-paradigm
What Is Lisp?
Dialects

Mother Tongues

Tracing the roots of computer languages through the ages

Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

Appeals to a wide audience  C (boiled down by the popularity of Unix)
Gets a job done  Cobol (designed for business-report writing)
Delivers near-functionality  Java (runs on any hardware platform)
Fills a niche  Mathematica (speeds up complex calculations)
Offers a medium of elegance  Icon (has friendly, line-oriented syntax)
Has a powerful user base or backer  C# (developed by Microsoft for .Net)
Has a charismatic leader  Perl (programmer-author Larry Wall)

Source: Paul Quinlan, Beryl Hatton, associates director of computer science at IBM Research; The NextComputing Museum, Todd Presnell, senior researcher at Microsoft; Gino Winderholt, computer scientist, Stanford University
Common Lisp

1984
these are 4 atoms

null? call/cc set!

"hello, world"

+ 3.5 − foo@bar
these are 4 atoms

null?  call/cc  set!

"hello, world"

+ 3.5 - foo@bar
these are 4 atoms

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
these are 4 atoms

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
these are 4 atoms

null? call/cc set!

"hello, world"

+ 3.5 − foo@bar
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)

(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

(((nested (list)) (with . pair)))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

(((nested (list)) (with . pair))
S-EXPRESSIONS: PAIRS, CONS CELLS, LISTS

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ())))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() \text{nil}

(1 . 5) (<car> . <cdr>)

(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ())))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

(((nested (list)) (with . pair)))
S-Expressions: Pairs, Cons Cells, Lists

() nil

(1 . 5) (<car> . <cdr>)
(<first> . <rest>)

(1 . ()) = (1)

(1 . (2 . (3 . ()))) = (1 2 3)

(a list of 5 elements)

((nested (list)) (with . pair))
Semantics

((operator operand operand ...))

(+ 12.5 (/ 93.8 20 (+ 10 (* 10 67)))))

(car (list 12.5 "hey"))
Semantics

(operator operand operand ...)

(+ 12.5 (/ 93.8 20 (+ 10 (* 10 67))))

(car (list 12.5 "hey"))
(operator operand operand operand ...)

(+ 12.5 (/ 93.8 20 (+ 10 (* 10 67)))))

(car (list 12.5 "hey"))
Binding: (define lotto-gewinn (* 1000 1000))

Functions: (lambda (x y) (/ (* x x) y))

Quoting: (quote (foo bar)) = '(foo bar)

Conditionals:

(cond ((< jackpot 100)
    (display "whatever")))
((> jackpot 100000)
    (display "not bad!"))
(else
    (display "could be worse")))
Binding: (define lotto-gewinn (* 1000 1000))

Functions: (lambda (x y) (/ (* x x) y))

Quoting: (quote (foo bar)) = '(foo bar)

Conditionals:

(cond ((< jackpot 100)
 (display "whatever"))
 ((> jackpot 100000)
 (display "not bad!"))
 (else
 (display "could be worse"))))
Binding: (define lotto-gewinn (* 1000 1000))

Functions: (lambda (x y) (/ (* x x) y))

Quoting: (quote (foo bar)) = '(foo bar)

Conditionals:

(cond ((< jackpot 100)
   (display "whatever"))
  ((> jackpot 100000)
   (display "not bad!")))
  (else
   (display "could be worse"))))
Binding: (define lotto-gewinn (* 1000 1000))

Functions: (lambda (x y) (/ (* x x) y))

Quoting: (quote (foo bar)) = '(foo bar)

Conditionals:

(cond ((< jackpot 100)
       (display "whatever"))
      ((> jackpot 100000)
       (display "not bad!"))
      (else
       (display "could be worse"))))
Some functions

(cons a b) => (a . b)

(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(eq? a b) => #t / #f
(cons a b) => (a . b)

(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(eq? a b) => #t / #f
Some functions

(cons a b) => (a . b)

(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(eq? a b) => #t / #f
Some functions

(cons a b) => (a . b)

(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(eq? a b) => #t / #f
Defining Functions

Functions are values!

(define square (lambda (x) (* x x)))

(square 11) => 121

(define (square x) (* x x))

((lambda (x) (* x x)) 11) => 121
Defining Functions

Functions are values!

```
(define square (lambda (x) (* x x)))

(square 11) => 121
```

```
(define (square x) (* x x))

((lambda (x) (* x x)) 11) => 121
```
Functions are values!

(\texttt{define square (lambda (x) (* x x)))}

(square 11) => 121

(\texttt{define (square x) (* x x))}

(((lambda (x) (* x x)) 11) => 121}
Functions are values!

(define square (lambda (x) (* x x)))

(square 11) => 121

(define (square x) (* x x))

(((lambda (x) (* x x)) 11) => 121
Defining Functions

Functions are values!

(define square (lambda (x) (* x x)))

(square 11) => 121

(define (square x) (* x x))

((lambda (x) (* x x)) 11) => 121
Functions are values!

(define square (lambda (x) (* x x)))

(square 11) => 121

(define (square x) (* x x))

((lambda (x) (* x x)) 11) => 121
Higher Order Functions

Functions which take functions as arguments and/or return functions.

(map square '(3 4 5)) => (9 16 25)

(for-each (lambda (x y)
            (display (format "~A ~A" x y))
            (newline))
          '(99 1001 42)
          '(red-balloons nights wtf/s))

99 red-balloons
1001 nights
42 wtf/s
Functions which take functions as arguments and/or return functions.

\[(\text{map square } '(3 4 5)) \Rightarrow (9 16 25)\]

\[(\text{for-each (lambda (x y)}
  (display (format "~A ~A" x y))
  (newline))
  '(99 1001 42)
  '(red-balloons nights wtf/s))\]

99 red-balloons
1001 nights
42 wtf/s
Higher Order Functions

Functions which take functions as arguments and/or return functions.

(map square '(3 4 5)) => (9 16 25)

(for-each (lambda (x y)
    (display (format "~A ~A" x y))
    (newline))
  '(99 1001 42)
  '(red-balloons nights wtf/s))

99 red-balloons
1001 nights
42 wtf/s
Higher Order Functions

Functions which take functions as arguments and/or return functions.

(map square '(3 4 5)) => (9 16 25)

(for-each (lambda (x y)
            (display (format "~A ~A" x y))
            (newline))
         '(99 1001 42)
         '(red-balloons nights wtf/s))

99 red-balloons
1001 nights
42 wtf/s
(unless (too-late?)
  (do-taxes)
  (go-shopping))

(cond ((not (too-late?))
  (do-taxes)
  (go-shopping)))

(define-syntax unless
  (syntax-rules ()
    ((unless condition exp ...)
      (cond ((not condition) exp ...)))))
(unless (too-late?)
  (do-taxes)
  (go-shopping))

(cond ((not (too-late?))
  (do-taxes)
  (go-shopping)))

(define-syntax unless
  (syntax-rules ()
    (((unless condition exp ...)
      (cond ((not condition) exp ...))))))
(unless (too-late?)
  (do-taxes)
  (go-shopping))

(cond ((not (too-late?))
  (do-taxes)
  (go-shopping)))

(define-syntax unless
  (syntax-rules ()
    ((unless condition exp ...)
      (cond ((not condition) exp ...))))))
Arithmetische Ausdrücke

39 - 210 / (3 + 10 * 67)

(− 39 (/ 210 (+ 3 (* 10 67))))
Arithmetische Ausdrücke

39 - 210 / (3 + 10 * 67)

(- 39 (/ 210 (+ 3 (* 10 67)))

)}
XML

<html>
<head>
  <title>Lisp &gt; all</title>
</head>
<body>
<h1>Welcome!</h1>
<p>Here you will find:</p>
<ul>
  <li><a href="/lispy-times">Lispy Times!</a></li>
  <li><a href="/other-stuff">Some other stuff</a></li>
  <li><a href="/more">and MORE</a></li>
</ul>
</body>
</html>

SXML

(HTML
  (HEAD
    (TITLE "Lisp > all"))
  (BODY (H1 "Welcome!")
    (P "Here you will find:")
    (UL
      (LI (A (@ (HREF "/lispy-times")) "Lispy Times!"))
      (LI (A (@ (HREF "/other-stuff")) "Some other stuff"))
      (LI (A (@ (HREF "/more")) "and MORE"))))
XML

```html
<html>
<head>
  <title>Lisp &gt; all</title>
</head>
<body>
<h1>Welcome!</h1>
<p>Here you will find:</p>
<ul>
  <li><a href="/lispy-times">Lispy Times!</a></li>
  <li><a href="/other-stuff">Some other stuff</a></li>
  <li><a href="/more">and MORE</a></li>
</ul>
</body>
</html>
```

SXML

```sml
(html
  (head
    (title "Lisp > all"))
  (body
    (h1 "Welcome!")
    (p "Here you will find:")
    (ul
      (li (a (@ (href "/lispy-times")) "Lispy Times!"))
      (li (a (@ (href "/other-stuff")) "Some other stuff"))
      (li (a (@ (href "/more")) "and MORE"))))
)```
Regular Expressions

`/^(foo|bar|baz)\s+\d+/`

SREs

(seq bol (submatch (or "foo" "bar" "baz"))
 (+ space) (+ number))
Regular Expressions

`/^\(foo|bar|baz\)\s+\d+/$`

SREs

(seq bol (submatch (or "foo" "bar" "baz"))
 (+ space) (+ number))
The World Seen Through
The Eyes Of A Lisper

SQL

SELECT firstname, lastname, company
FROM members AS m
LEFT JOIN interests AS i ON i.member_id = m.id
WHERE age > 18
ORDER BY lastname, firstname;

SSQL

(select (columns firstname lastname company)
 (from (join left (as members m) (as interests i)
 (on (= (col m id) (col i member_id))))))
(where (> age 19))
(order (desc lastname) firstname))
SQL

SELECT firstname, lastname, company
FROM members AS m
LEFT JOIN interests AS i ON i.member_id = m.id
WHERE age > 18
ORDER BY lastname, firstname;

SSQL

(select (columns firstname lastname company)
 (from (join left (as members m) (as interests i)
   (on (= (col m id) (col i member_id))))))
(where (> age 19))
(order (desc lastname) firstname)
So why would you use that?

Paul Graham: Beating The Averages
http://www.paulgraham.com/avg.html
So why would you use that?

Paul Graham: Beating The Averages
http://www.paulgraham.com/avg.html
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, "eggs"
- `chicken-install postgresql`

http://www.call-cc.org/
Chicken Scheme

http://www.call-cc.org/

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, "eggs"
- chicken-install postgresql
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
  - portable and embeddable
  - more than 400 extensions, “eggs”
  - chicken-install postgresql

http://www.call-cc.org/
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, "eggs"
- `chicken-install postgresql`

http://www.call-cc.org/
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, “eggs”
  - `chicken-install postgresql`

http://www.call-cc.org/
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, "eggs"
- chicken-install postgresql

http://www.call-cc.org/
Chicken Scheme

- Compiler translates Scheme to C
- Interpreter, mixing is possible
- Portable and embeddable
- More than 400 extensions, “eggs”
- chicken-install postgresql

http://www.call-cc.org/
Recommended Reading

Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Recommended Reading

Recommended Reading

http://www.cs.berkeley.edu/~bh/simply-toc.html
The Little Schemer

Fourth Edition

Daniel P. Friedman and Matthias Felleisen

Foreword by Gerald J. Sussman
REFERENCES

John McCarthy: History of Lisp, 1979
http://www-formal.stanford.edu/jmc/history/lisp/lisp.html

Association of Lisp Users http://lisp.org/

Scheme http://schemers.org/
http://schemewiki.org/

Common Lisp http://www.cliki.net/