# Lisp

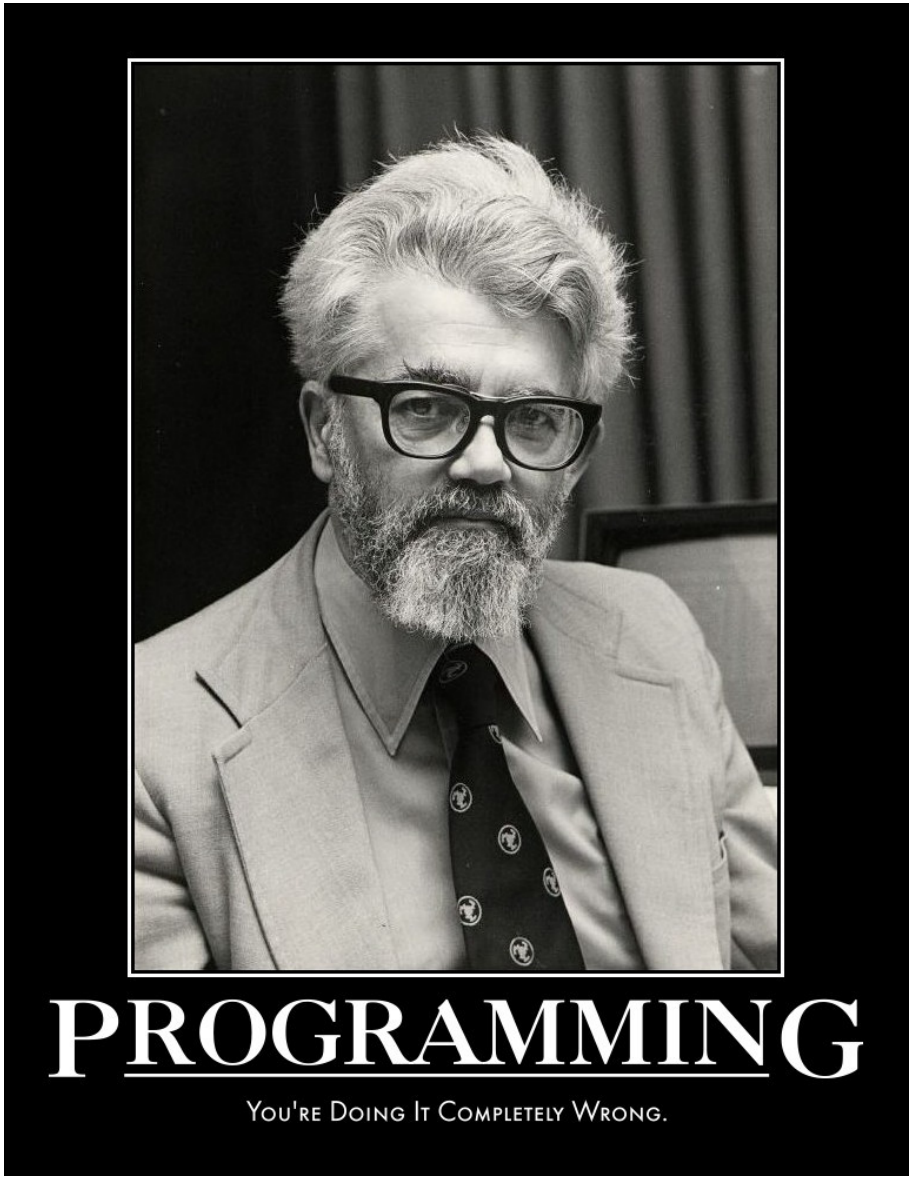
Moritz Heidkamp
moritz@twoticketsplease.de

25. April 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.*

# Woher kommt Lisp?




John McCarthy, 1958

# Woher kommt Lisp?

*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).*

John McCarthy: History of Lisp, 1979.

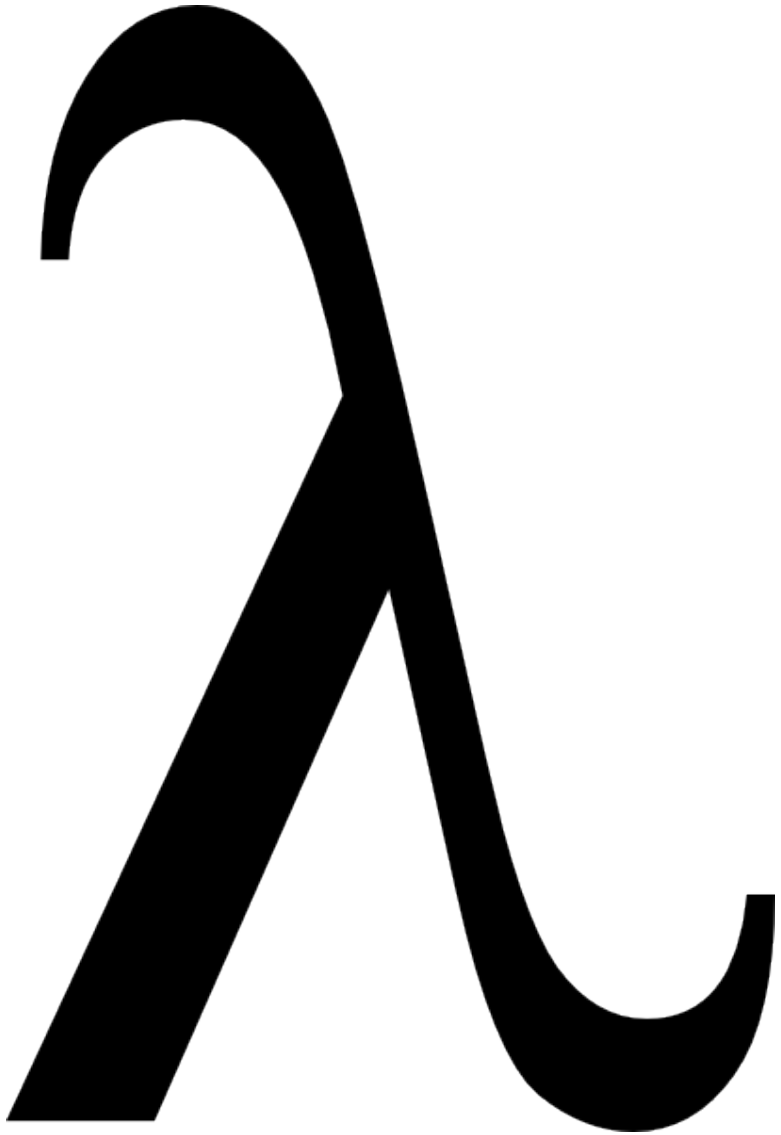
# Woher kommt Lisp?

IBM 704, 1960/61

# Ein Lisp-Programm von ca. 1961

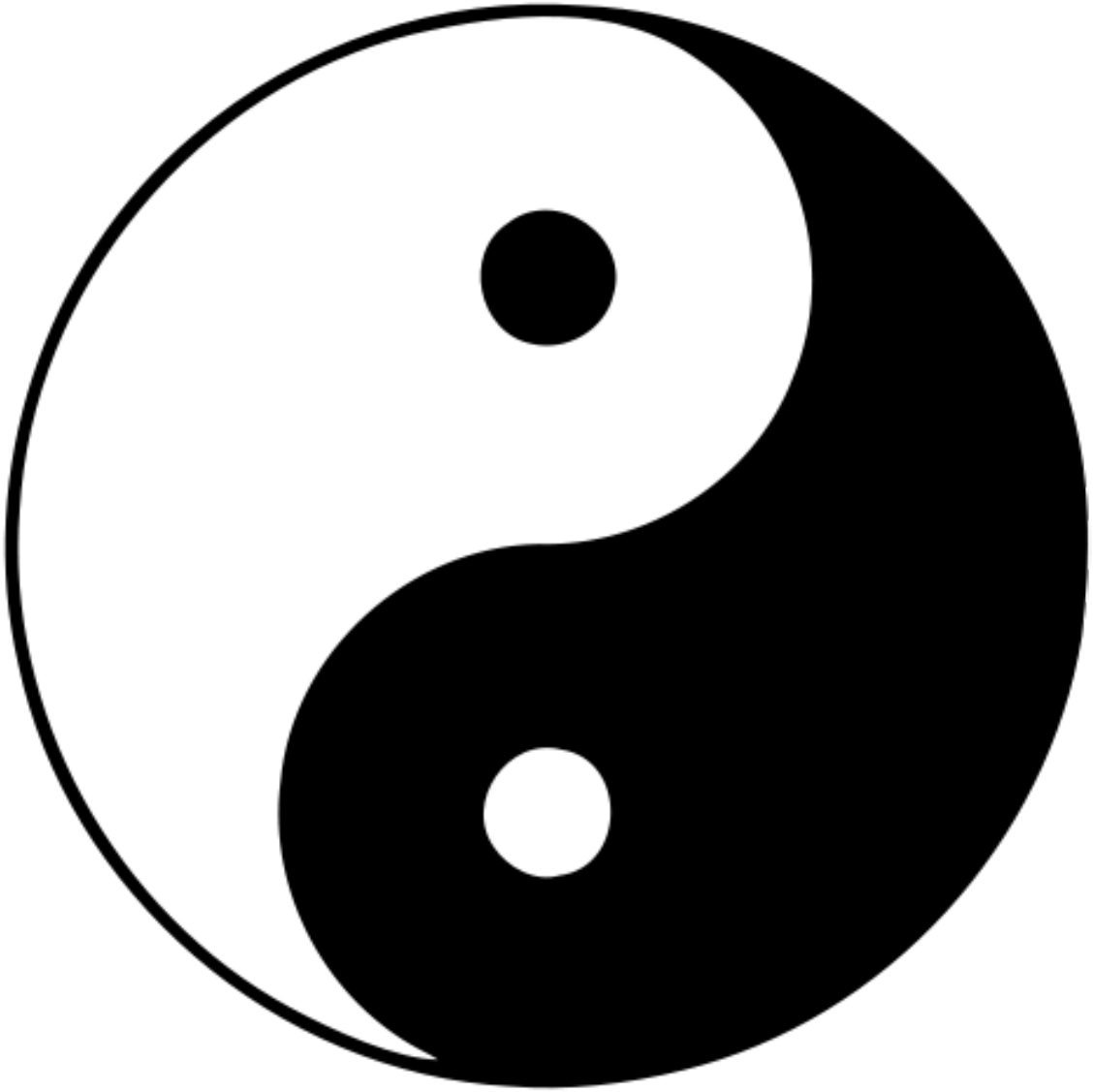
```
DEFINE ((
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
      ((EQ A (CAR X)) T) (T (MEMBER A (CDR X))) )))
(UNION (LAMBDA (X Y) (COND ((NULL X) Y) ((MEMBER
      (CAR X) Y) (UNION (CDR X) Y)) (T (CONS (CAR X)
      (UNION (CDR X) Y))) )))
(INTERSECTION (LAMBDA (X Y) (COND ((NULL X) NIL)
      ((MEMBER (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))
```

# Was ist Lisp?



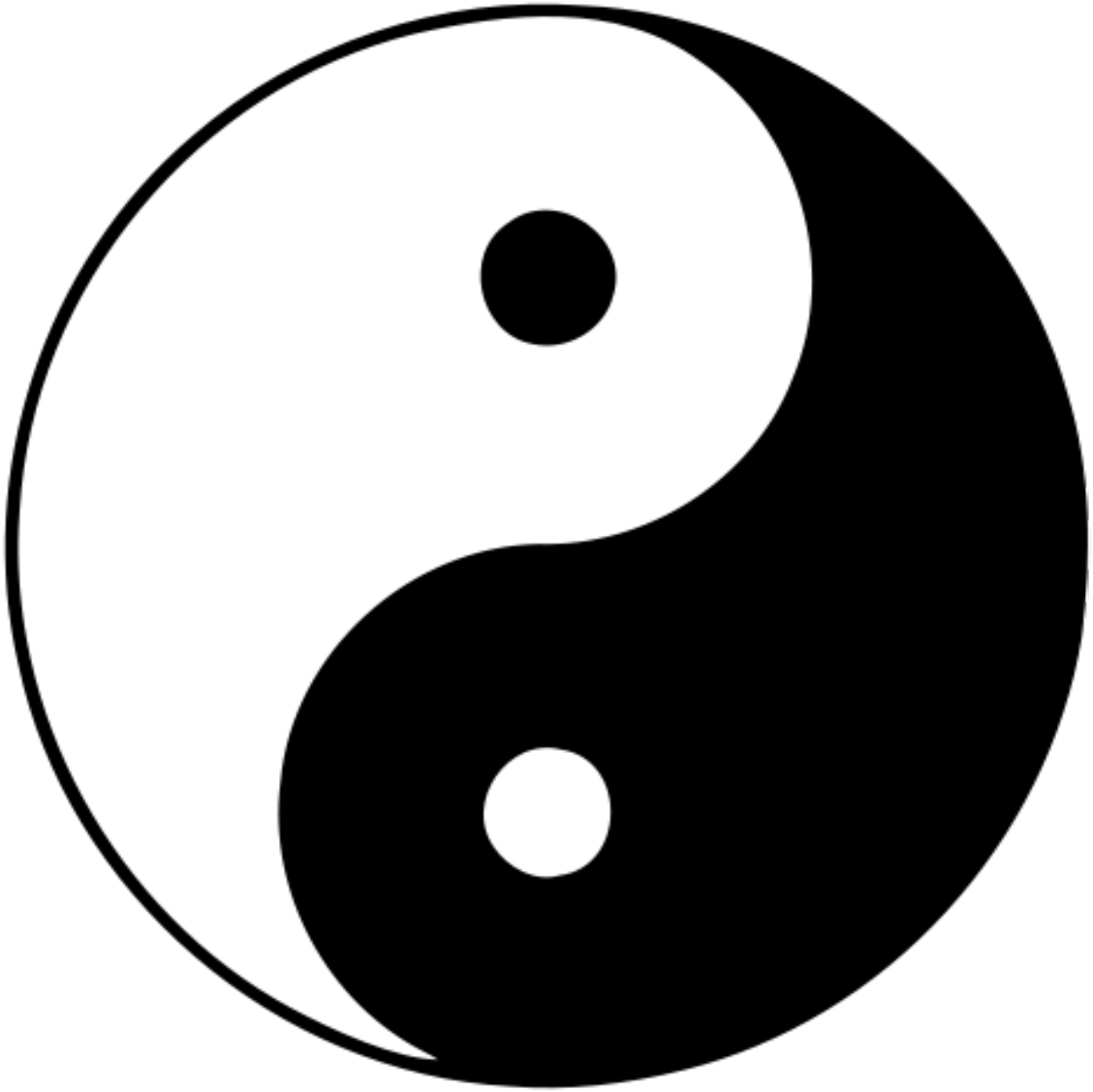
Funktional

# Was ist Lisp?



Homoikonizität
Code = Daten

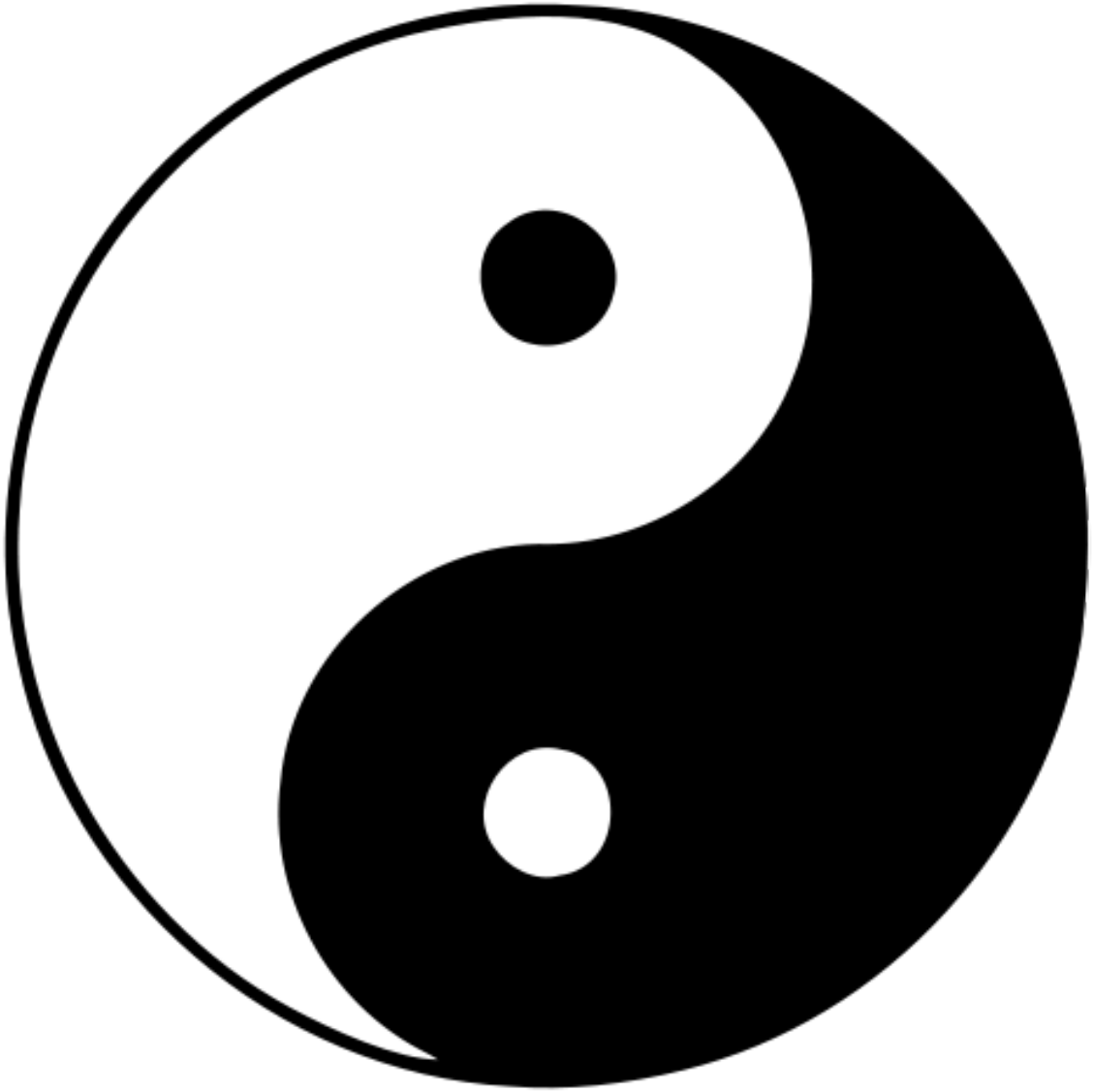
# Was ist Lisp?



Homoikonizität

Code = Daten

# Was ist Lisp?

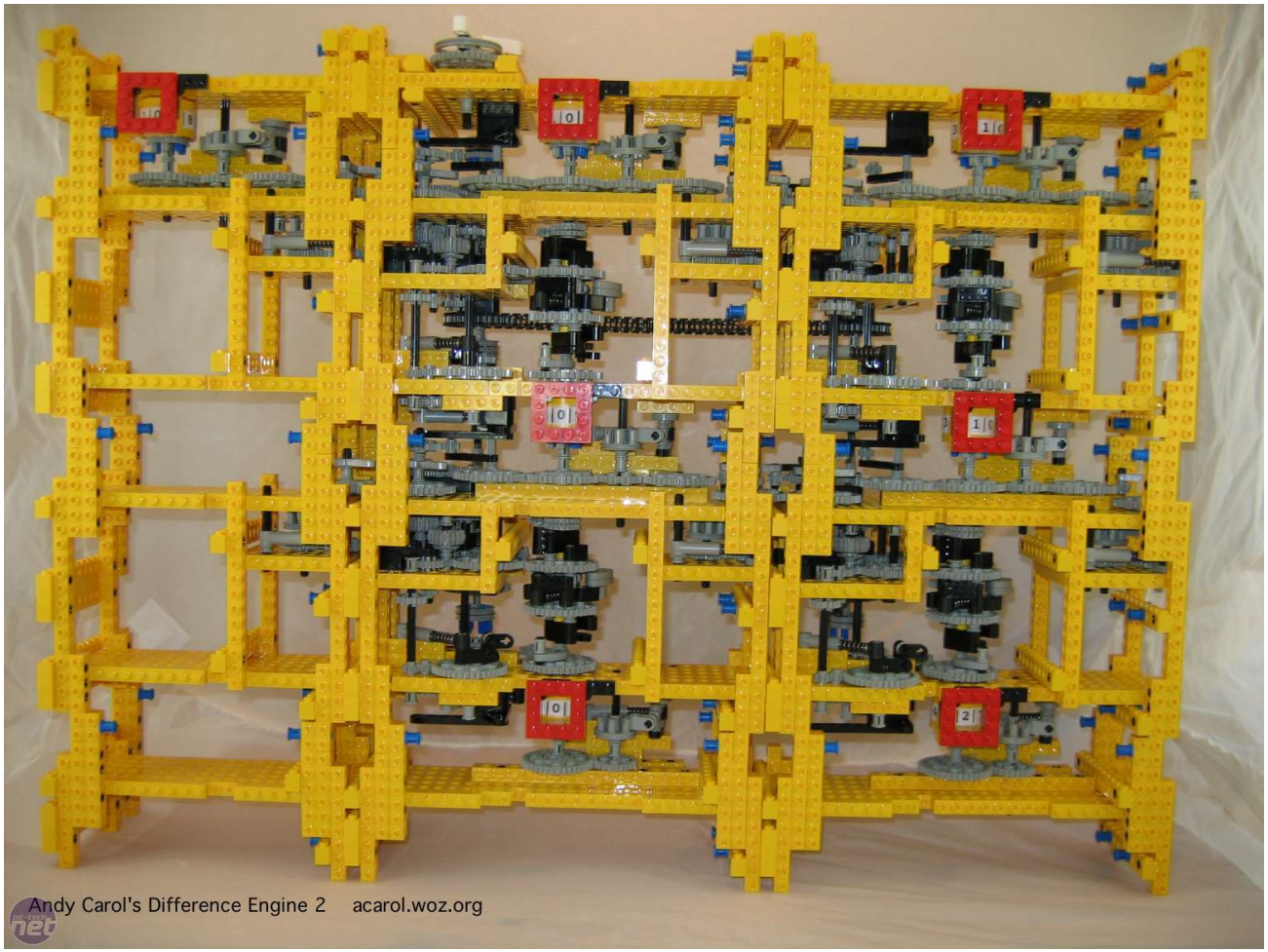


Homoikonizität
Code = Daten

# Was ist Lisp?

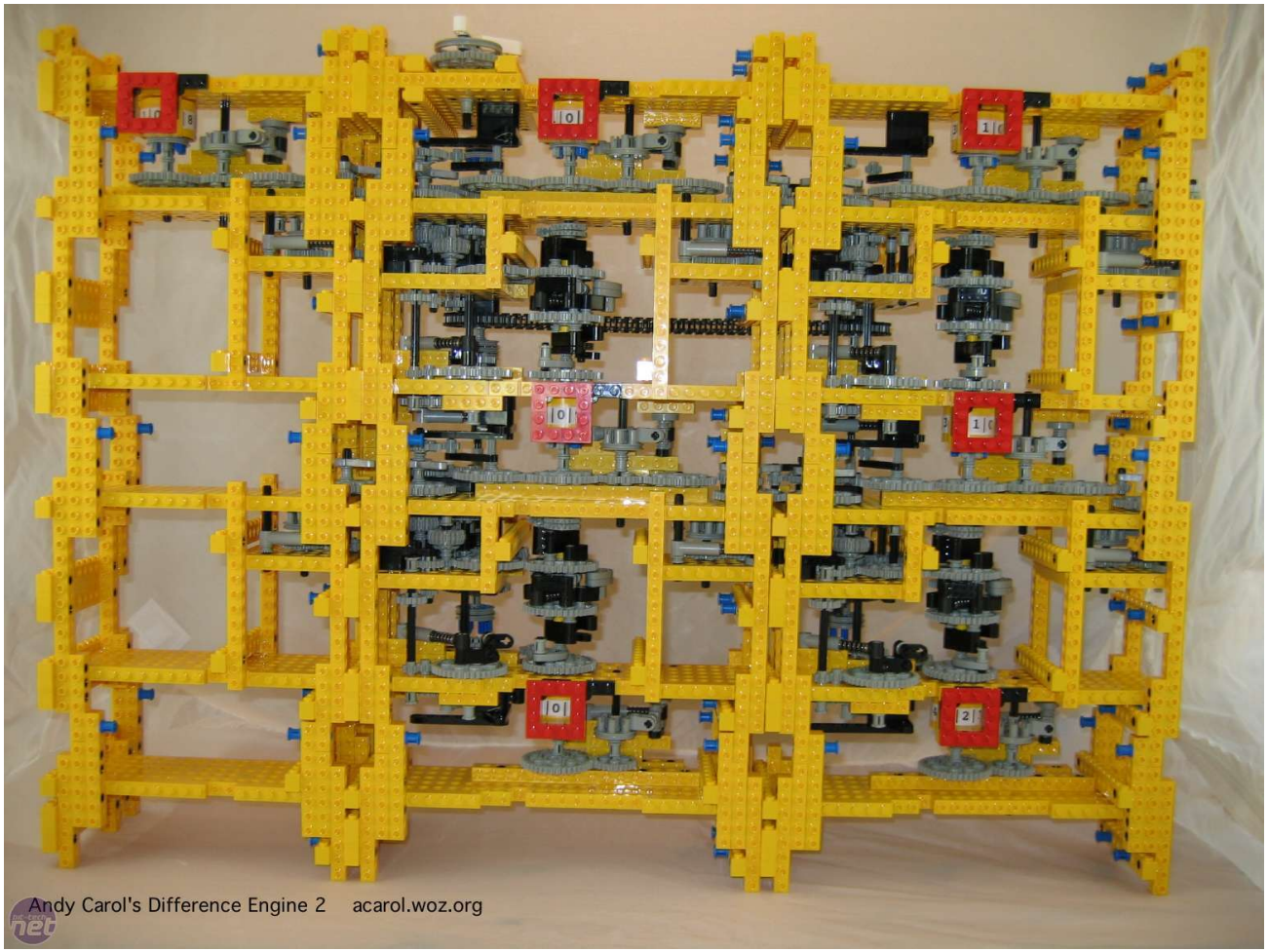
Garbage Collection

# Was ist Lisp?




Programmierbare Programmiersprache
multiparadigmatisch

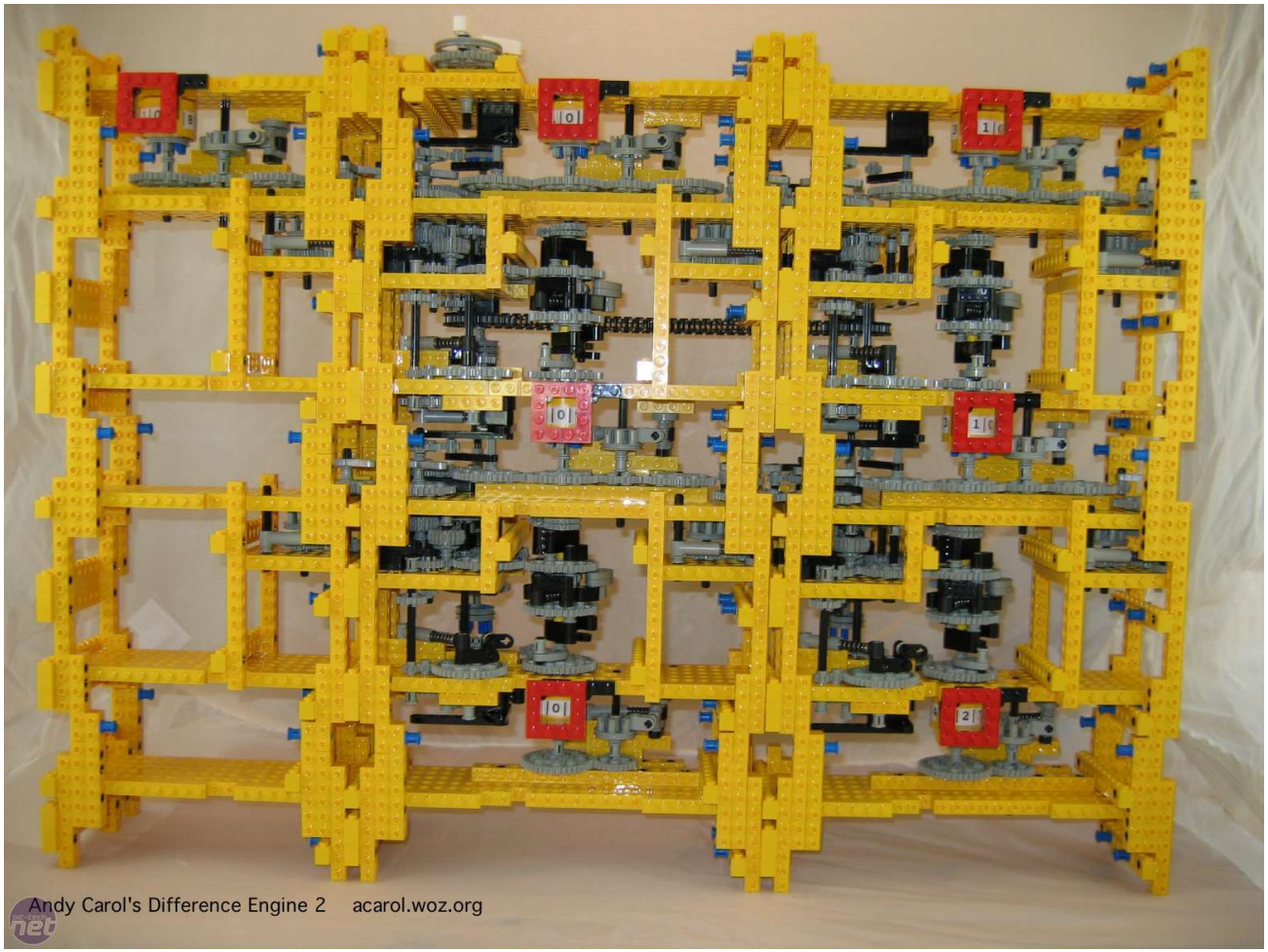
# Was ist Lisp?




Programmierbare Programmiersprache

multiparadigmatisch

# Was ist Lisp?




Programmierbare Programmiersprache
multiparadigmatisch

# Was ist Lisp?

# Dialekte

# Mother Tongues

**Tracing the roots of computer languages through the ages**

**Just like** half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers – electronic lexicographers, if you will – aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. **– Michael Menduno**

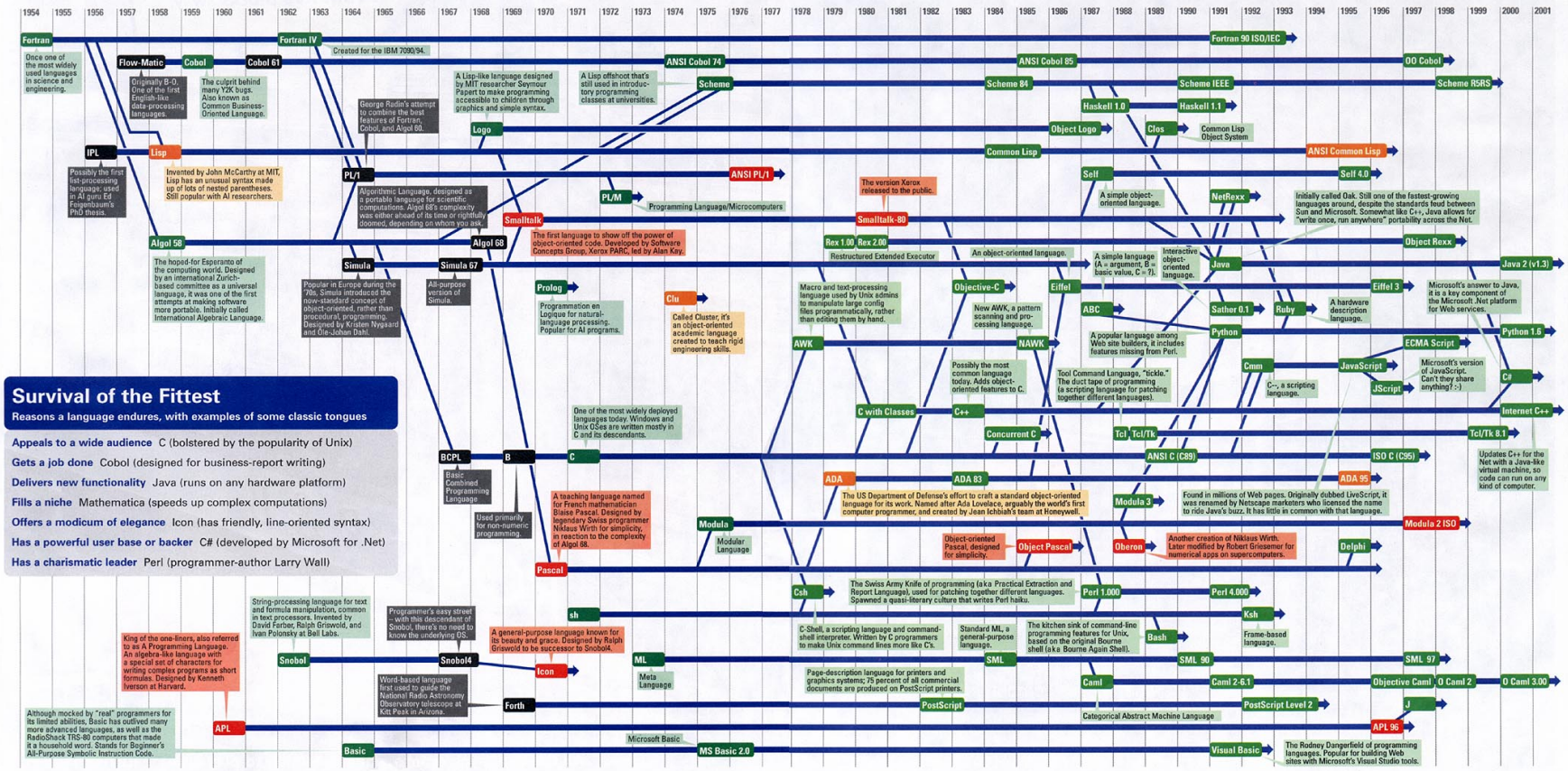
**Key**

- 1954 **Year Introduced**
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- **Lineage continues**

## Survival of the Fittest

**Reasons a language endures, with examples of some classic tongues**

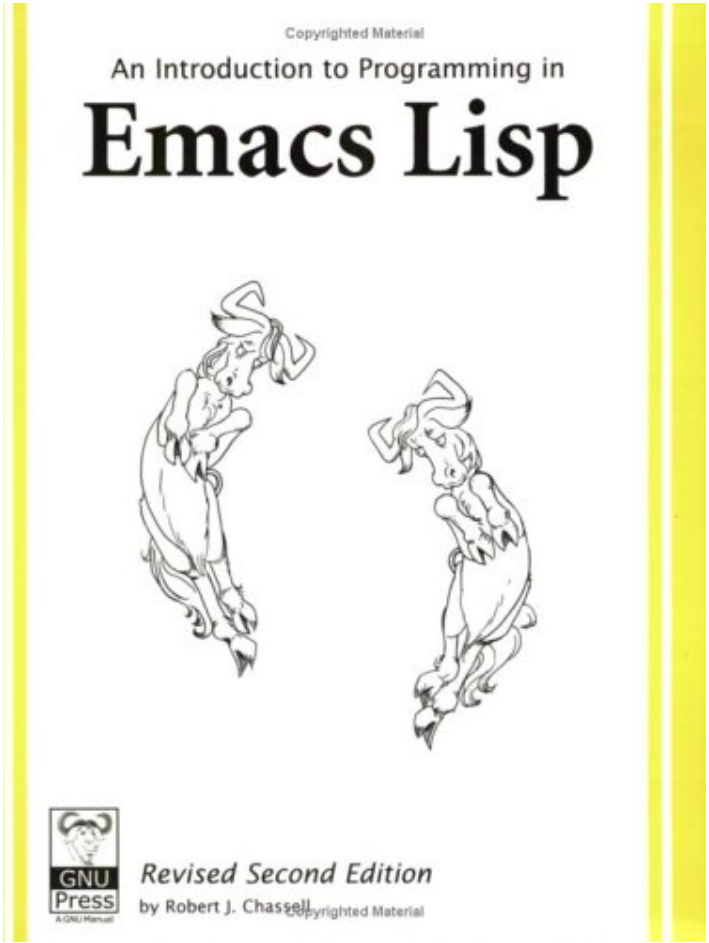
- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** 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)




Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University


1984

# Emacs Lisp




1984

# Scheme


1975

# Clojure

2007

# S-Expressions: Atome

```
dies sind 4 atome

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
```

# S-Expressions: Atome

```
dies sind 4 atome

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
```

# S-Expressions: Atome

```
dies sind 4 atome

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
```

# S-Expressions: Atome

```
dies sind 4 atome

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
```

# S-Expressions: Atome

```
dies sind 4 atome

null? call/cc set!

"hello, world"

+ 3.5 - foo@bar
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ()) = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ())  = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ())  = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

# S-Expressions: Paare, Cons-Zellen, Listen

```
() nil

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

(1 . ())  = (1)

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

(eine liste mit 5 elementen)

((geschachtelte (liste)) (mit . paar))
```

```
(operator operand operand ...)
```

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

```
(car (list 12.5 "hey"))
```

```
(operator operand operand ...)
```

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

```
(car (list 12.5 "hey"))
```

```
(operator operand operand ...)
```

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

```
(car (list 12.5 "hey"))
```

# Special Forms

Binding: `(define lotto-gewinn (* 1000 1000))`

Funktionen: `(lambda (x y) (/ (* x x) y))`

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

Bedingungen:

```
(cond ((< lotto-gewinn 100)
       (display "pfff"))
      ((> lotto-gewinn 100000)
       (display "nicht schlecht!"))
      (else
       (display "naja")))
```

# Special Forms

Binding: `(define lotto-gewinn (* 1000 1000))`

Funktionen: `(lambda (x y) (/ (* x x) y))`

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

Bedingungen:

```
(cond ((< lotto-gewinn 100)
       (display "pfff"))
      ((> lotto-gewinn 100000)
       (display "nicht schlecht!"))
      (else
       (display "naja")))
```

# Special Forms

Binding: `(define lotto-gewinn (* 1000 1000))`

Funktionen: `(lambda (x y) (/ (* x x) y))`

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

Bedingungen:

```
(cond ((< lotto-gewinn 100)
       (display "pfff"))
      ((> lotto-gewinn 100000)
       (display "nicht schlecht!"))
      (else
       (display "naja")))
```

# Special Forms

Binding: `(define lotto-gewinn (* 1000 1000))`

Funktionen: `(lambda (x y) (/ (* x x) y))`

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

Bedingungen:

```
(cond ((< lotto-gewinn 100)
       (display "pfff"))
      ((> lotto-gewinn 100000)
       (display "nicht schlecht!"))
      (else
       (display "naja")))
```

# Einige Funktionen

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

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

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

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

# Einige Funktionen

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

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

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

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

# Einige Funktionen

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

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

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

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

# Einige Funktionen

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

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

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

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

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen definieren

Funktionen sind Werte!

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

```
(square 11) => 121
```

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

```
((lambda (x) (* x x)) 11) => 121
```

# Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente entgegennehmen und/oder zurückgeben.

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


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



99 luftballons
1001 nacht
42 wtf/s
```

# Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente entgegennehmen und/oder zurückgeben.

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

```
(for-each (lambda (x y)
            (display (format "~A ~A" x y))
            (newline))
          '(99 1001 42)
          '(luftballons nacht wtf/s))
```

```
99 luftballons
1001 nacht
42 wtf/s
```

# Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente entgegennehmen und/oder zurückgeben.

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

```
(for-each (lambda (x y)
            (display (format "~A ~A" x y))
            (newline))
          '(99 1001 42)
          '(luftballons nacht wtf/s))
```

```
99 luftballons
1001 nacht
42 wtf/s
```

# Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente entgegennehmen und/oder zurückgeben.

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

```
(for-each (lambda (x y)
            (display (format "~A ~A" x y))
            (newline))
          '(99 1001 42)
          '(luftballons nacht wtf/s))
```

```
99 luftballons
1001 nacht
42 wtf/s
```

# Macros

```
(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 ...)))))
```

# Macros

```
(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 ...)))))
```

# Macros

```
(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 ...)))))
```

# Die Welt gesehen durch die Brille eines Lispers

Arithmetische Ausdrücke

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

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

# Die Welt gesehen durch die Brille eines Lispers

Arithmetische Ausdrücke

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

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

# Die Welt gesehen durch die Brille eines Lispers

## 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")))))
```

# Die Welt gesehen durch die Brille eines Lispers

## 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")))))
```

# Die Welt gesehen durch die Brille eines Lispers

Reguläre Ausdrücke

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

SREs

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

# Die Welt gesehen durch die Brille eines Lispers

Reguläre Ausdrücke

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

SREs

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

# Die Welt gesehen durch die Brille eines Lispers

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))
```

# Die Welt gesehen durch die Brille eines Lispers

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))
```

# Warum Lisp?

Und was macht man dann damit?

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

Und was macht man dann damit?

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

# Chicken Scheme

CHICKENscheme

A PRACTICAL AND PORTABLE SCHEME SYSTEM

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

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

# Chicken Scheme

CHICKENscheme

A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

# Chicken Scheme

CHICKENscheme
A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

# Chicken Scheme

CHICKENscheme
A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

# Chicken Scheme

CHICKENscheme

A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

# Chicken Scheme

CHICKENscheme
A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

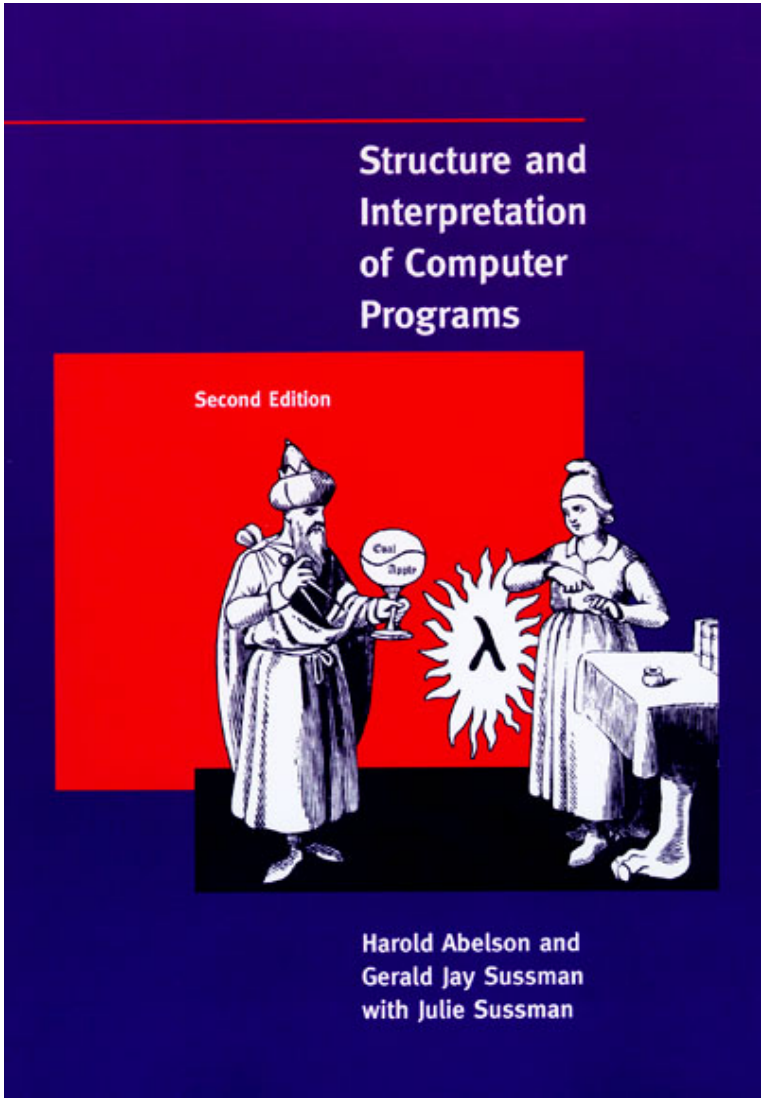
# Chicken Scheme

CHICKENscheme
A PRACTICAL AND PORTABLE SCHEME SYSTEM

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

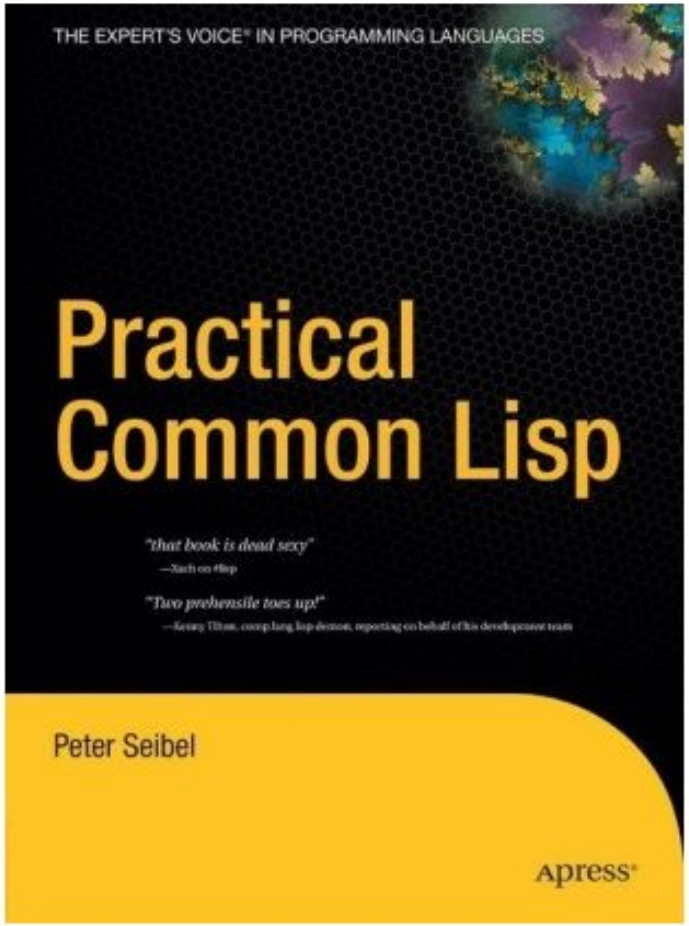
- Compiler übersetzt Scheme nach C
- Interpreter, Mischbetrieb möglich
- portabel und embeddable
- mehr als 400 Erweiterungen, “eggs”
- `chicken-install postgresql`

# Literaturempfehlungen



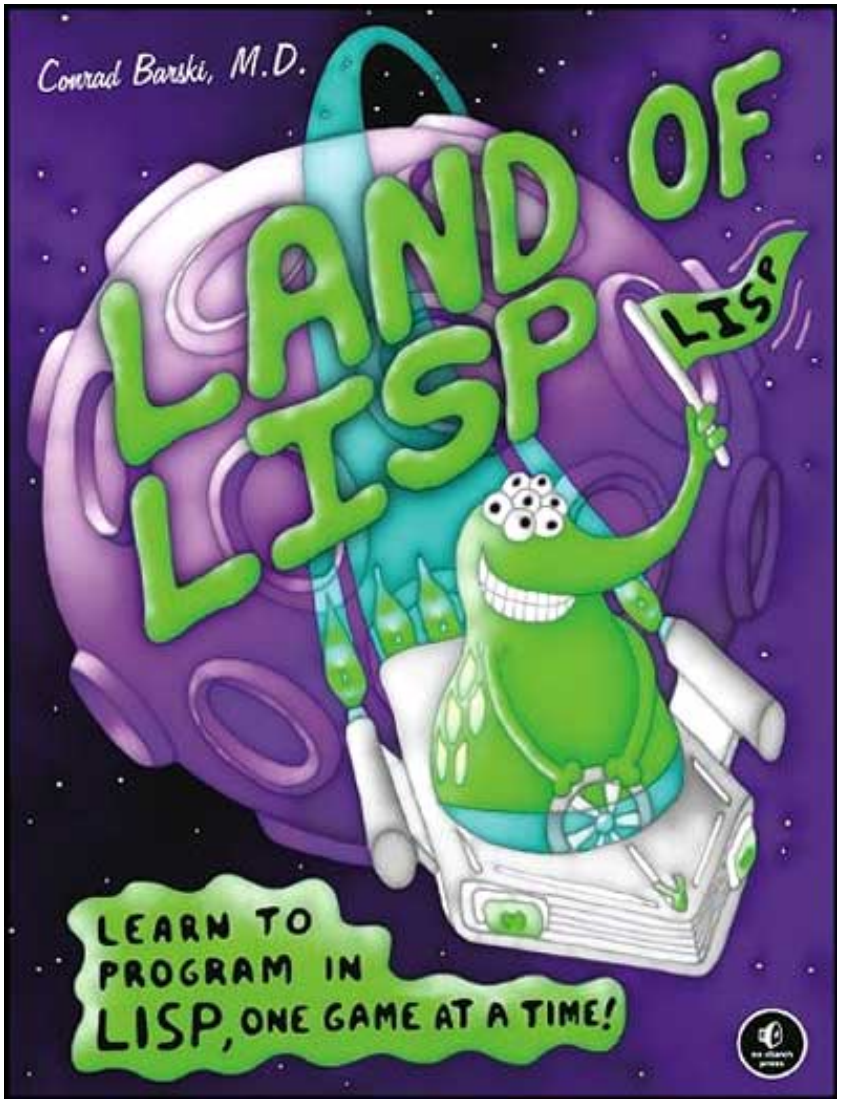

http://mitpress.mit.edu/sicp/full-text/book/book.html

# Literaturempfehlungen

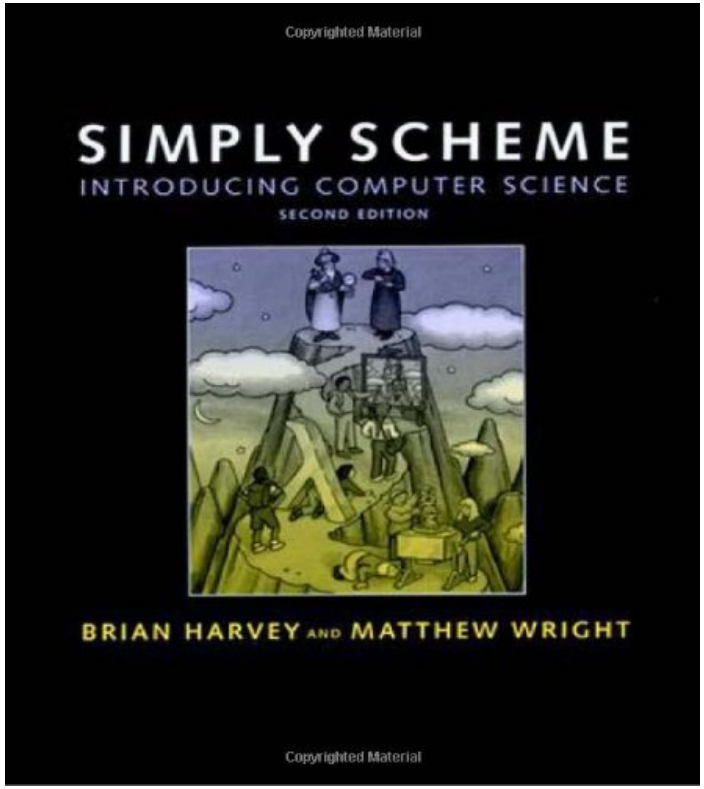



`http://www.gigamonkeys.com/book/`

# Literaturempfehlungen

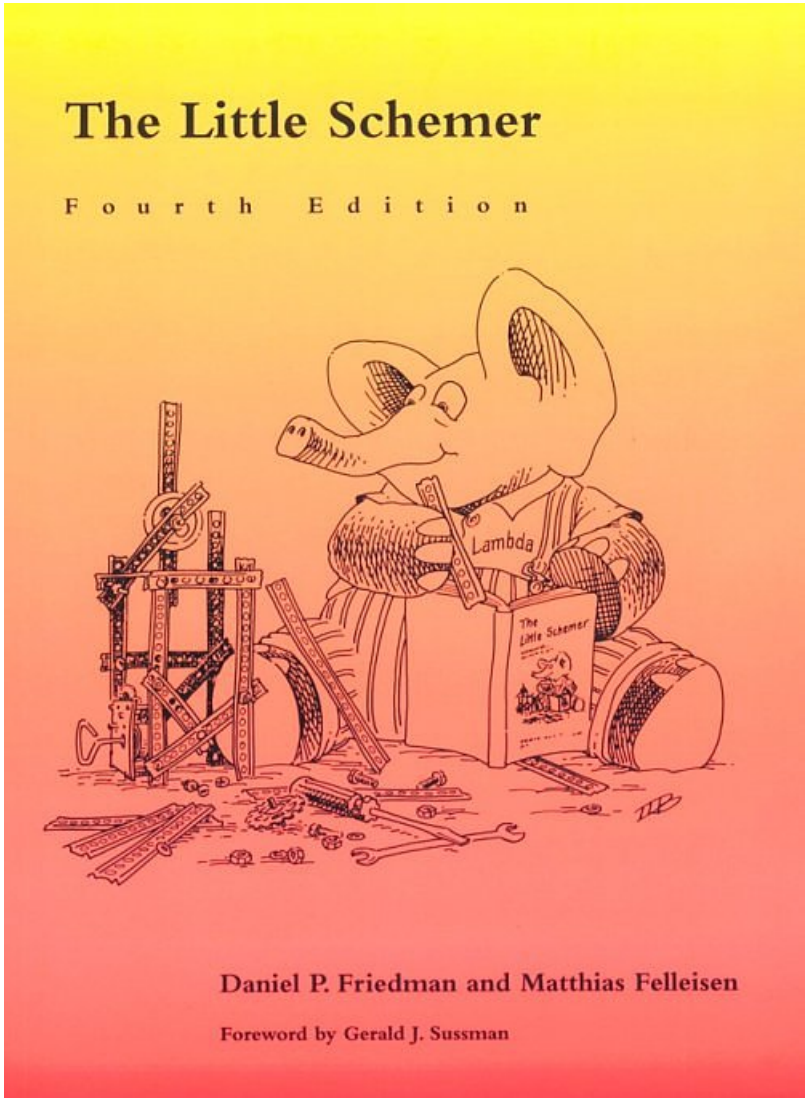


# Literaturempfehlungen




`http://www.cs.berkeley.edu/~bh/simply-toc.html`

# Literaturempfehlungen



# Referenzen

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/