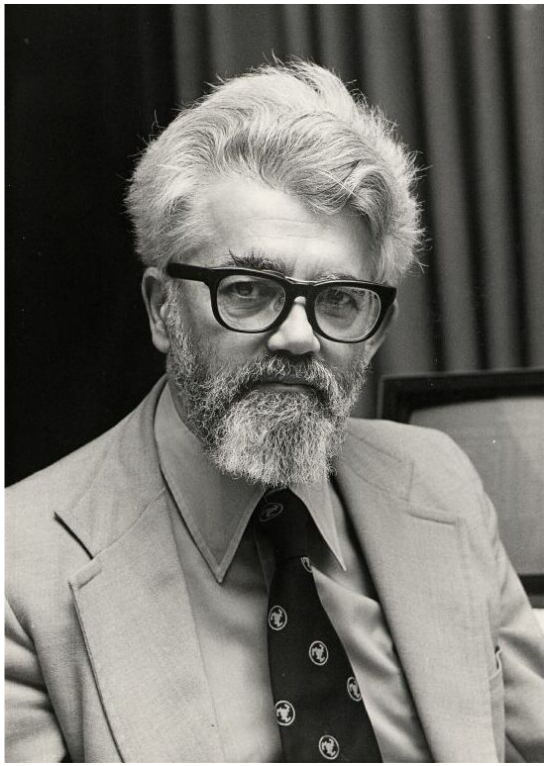
Racket

Lisp is λ-calculus done right.

Scheme is Lisp done right.

Racket is Scheme done right.

Origins of Scheme: Lisp



John McCarthy

- Lisp created in 1958
- Based on λ-calculus
- Made for list-processing
- S-Expression syntax
- Code acts as data
- Data acts as code

Core S-Expression syntax

```
<s-exp> ::= <symbol>
          |  (<s-exp> ...)
```

S-Expressions: Benefits

- Human-readable
- Human-writable
- Easy to parse

S-Expression example

```
(ships
 (ship (name Enterprise)
       (x    120)
       (y    150))
 (ship (name Galactica)
       (x    130)
       (y    42)))
```

XML: High-calorie S-Expressions

```
<ships>
 <ship>
  <name>Enterprise</name>
  <x>120</x>
  <y>150</y>
 </ship>
 <ship>
  <name>Galactica</name>
  <x>130</x>
  <y>42</y>
 </ship>
</ships>
```

S-Expressions: Universal syntax

```
class Dog extends Pet {
 void bark() {
  print("woof") ;
}}
```

S-Expressions: Universal syntax

```
class Dog extends Pet {
 void bark() {
  print("woof") ;
}}
```

```
(class Dog extends Pet
 (void (bark)
  (print "woof")))
```

S-Expressions: Universal syntax

```
  class Dog extends Pet {
   void bark() {
    print("woof") ;
  }}
```

```
(class
 (name    Dog)
 (extends Pet)
 (methods
  (method
   (name bark)
   (args ())
   (type (-> () void))
   (body (print "woof"))))))
```

Scheme

Guy Steele

- Lisp done right, 1978
- Fixed scoping oversight
- Purified formal semantics
- Hygienic macro system

Racket

Matthew Flatt

- Scheme done right, 2010
- Added huge library system
- Adds modules, structs, etc.
- Batteries included

Racket example: Factorial

```
(define (fact n)
  (cond
    ((<= n 0)  1)
    (else       (* n (fact (- n 1))))))
```

Racket programs

```
<body> ::= <def> ... <exp> ...
```

Definitions

```
<def> ::= (define (<id> <params>) <body>)
        |   (define <id> <exp>)
        |   (define-struct <struct-spec>)
        |   (define-syntax <id> <syntax-transformer>)
```

A tour of Racket

Numbers

- Integers: `1`, `-3`
- Rationals: `(/ 8 10)` => `4/5`
- "Reals": `(sqrt 2)` => `1.4142135623730951`
- Imaginary: `(sqrt -1)` => `0+1i`
- Complex: `(* 1+3i 1-22/7i)` => `73/7-1/7i`
- Precise: `(fact 18)` => `6402373705728000`

Booleans

- Boolean literals `#t`, `#f`
- Only `#f` is false
- Anything else is true

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '())` => `(1)`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`
- `'(3 1) => (3 1)`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`
- `'(3 1) => (3 1)`
- `(car (cons 3 (cons 1 '()))) => 3`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`
- `'(3 1) => (3 1)`
- `(car (cons 3 (cons 1 '()))) => 3`
- `(cdr (cons 3 (cons 1 '()))) => (1)`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`
- `'(3 1) => (3 1)`
- `(car (cons 3 (cons 1 '()))) => 3`
- `(cdr (cons 3 (cons 1 '()))) => (1)`
- `(pair? (cons 3 4)) => #t`

Lists

- `cons`, `car`, `cdr`, `'()`, `pair?`
- `(cons 1 '()) => (1)`
- `(cons 3 (cons 1 '())) => (3 1)`
- `'(3 1) => (3 1)`
- `(car (cons 3 (cons 1 '()))) => 3`
- `(cdr (cons 3 (cons 1 '()))) => (1)`
- `(pair? (cons 3 4)) => #t`
- `(null? '()) => #t`

Quoted S-Expressions

- `'<s-exp>` == `(quote <s-exp>)`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`
- `(cdr '(1 2 3)) => (2 3)`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`
- `(cdr '(1 2 3)) => (2 3)`
- `'(f (g x)) => (f (g x))`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`
- `(cdr '(1 2 3)) => (2 3)`
- `'(f (g x)) => (f (g x))`
- `(car '(f (g x))) => f`

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`
- `(cdr '(1 2 3)) => (2 3)`
- `'(f (g x)) => (f (g x))`
- `(car '(f (g x))) => f`
- `'() => ()`

Primitives

- Arithmetic: `+`, `modulo`, `quotient`
- Types: `null?`, `pair?`, `number?`, `integer?`
- Lists: `list`, `cons`, `car`, `cdr`
- Equality: `=`, `eq?`, `eqv?`, `equal?`
- Ex: `(gcd 248 184) => 8`
- Ex: `(cons 3 (cons 4 '())) => (3 4)`

`let`

- **`<let-exp> ::=`**
  `(let ((<var> <exp>) ...)`
  `<body>)`
- **`let`** binds values to variable names

let

- `<let-exp> ::=`
  `(let ((<var> <exp>) ...)`
  `<body>)`
- `let` binds values to variable names
-
```
(let ((π   3.14)
      (two 2))
  (* two π))      => 6.28
```

let

- **<let-exp> ::=**
  (let ((<var> <exp>) ...)
  <body>)
- **let** binds values to variable names
- **(let ((π 3.14)**
  (two 2))
  (* two π)) **=> 6.28**
- **(let ((π 3.14)**
  (2*π (* 2 π)))
  2*π) **=/=> 6.28**

let*

- **<let*-exp> ::=**
  (let* ((<var> <exp>) ...)
  <body>)
- **let*** binds values to variable names sequentially

let*

- **<let*-exp> ::=**

```
(let* ((<var> <exp>) ...)
 <body>)
```

- **let*** binds values to variable names sequentially

```
(let* ((π   3.14)
       (π   3.14159))
  (* 2 π))                    => 6.28318
```

let*

- **<let*-exp> ::=**
  (let* ((<var> <exp>) ...)
  <body>)
- **let*** binds values to variable names sequentially
- **(let* ((π 3.14)**
  (π 3.14159))
  (* 2 π)) **=> 6.28318**
- **(let* ((π 3.14)**
  (2*π (* 2 π)))
  2*π) **=> 6.28**

Variables

- Any symbol allowed
- Roughly: `[^#()'` `,]+`
- Ex: `foo`
- Ex: `foo-bar`
- Ex: `π`, `Δ`, `∞`, `Σ`, `⊑`, `∅`, `→`
- Ex: `pair?`
- Ex: `call-with-current-continuation`
- Ex: `(let ((let 3)) let) => 3`

λ

- λ produces anonymous functions
- **<lambda> ::= (λ (<var> ...) <body>)**
- **((λ (x) (* x x)) 3) => 9**
- **(let ((λ (λ (λ) (λ λ)))) (λ λ))**

Keyword, optional args

- λ produces anonymous functions
- **<lambda> ::= (λ (<formal> ...) <body>)**
- **<formal> ::= #:<keyword> <param>**
- **| <param>**
- **<param> ::= <var>**
- **| [<var> <default>]**
-
```
((λ (x #:debug [debug? #f])
    (when debug?
      (printf "x: ~a~n" x))
    (* x x))
  300)                              => 90000
```

Variable-arity args

<lambda> ::= (λ <var> <body>)

Variable-arity args

```
(let ((x 200))
  (define f (λ vars vars))
  (f 1 2 3 x))                    => (1 2 3 200)
```

Variable-arity args

```
(let ((x 200))
  (define f (λ vars vars))
  (apply f '(a b c d)))     => (a b c d)
```

Lexical scope

To what does the following evaluate?

```
(let ((x 20))
  (let* ((x 3)
         (f (λ () x))
         (x 10))
    (f)))
```

Lexical scope

- λ-terms capture bindings in scope
- WYSIWYG for anonymous functions

Obligatory derivative example

```
(define ε 0.001)
(define (D f) (λ (x) (/ (- (f (+ x ε)) (f x))
                        ε)))
(define (g x) (* x x))
((D g) 0) ≈ 0
((D g) 2) ≈ 4
```

Conditionals

- **<if-exp> ::= (if <exp> <exp> <exp>)**
- **<when-exp> ::= (when <exp> <body>)**
- **<cond-exp> ::=**
  (cond (<exp> <exp>) ...)
- **(if 'a 'b 'c) => b**
- **(if #f 'b 'c) => c**
- **(cond**
  ((< 3 0) 'negative)
  ((= 3 0) 'zero)
  ((> 3 0) 'positive)) => positive

Quasiquotes

- Special syntax for constructing literals
- `` `<qq-sexp> == (quasiquote <qq-sexp>) ``
- `,<exp> == (unquote <exp>)`
- `` `(3 + 4 is ,(+ 3 4)) => (3 + 4 is 7) ``

Quasiquotes

- Special syntax for constructing literals
- `` `<qq-sexp> == (quasiquote <qq-sexp>) ``
- `,<exp> == (unquote <exp>)`
- `` `(3 + 4 is ,(+ 3 4)) => (3 + 4 is 7) ``

```
(let ((new-ship (λ (name x y)
                  `(ship (name ,name)
                         (x    ,x)
                         (y    ,y)))))
  (new-ship 'Enterprise 42 1701))
```

Quasiquotes

- Special syntax for constructing literals
- `` `<qq-sexp> == (quasiquote <qq-sexp>) ``
- `,<exp> == (unquote <exp>)`
- `` `(3 + 4 is ,(+ 3 4)) => (3 + 4 is 7) ``

```
(let ((new-ship (λ (name x y)
                  `(ship (name ,name)
                         (x    ,x)
                         (y    ,y)))))
  (new-ship 'Enterprise 42 1701))

      => (ship (name Enteprise)
               (x 42)
               (y 1701))
```

Quasiquotes

```
(let ((x 10))
  `(The is ,x)) => (The is 10)
```

Quasiquotes – splicing

- `,@<exp> -- splices in list`

```
(let ([x '(1 2 3 4)])
  `(a ,@x b))          => (a 1 2 3 4 b)
```

Quasiquotes – splicing

- ,@<exp> -- splices in list

```
(let ([x '(1 2 3 4)])
  `(a ,@x b))          => (a 1 2 3 4 b)

(let ([x '(1 2 3 4)])
  `(a ,x b))           => (a (1 2 3 4) b)
```

Mutating variables

- **`begin`** sequences expressions, like **`{ }`** in C

```
(begin
  1
  2
  3)    => 3
```

- **`set!`** assigns to a variable, like **`=`** in C

```
(let ((π 3.14))
  (begin
    (set! π 3.14159)
    π))                    => 3.14159
```

Pattern-matching

- Powerful tool for dissecting data like trees

Pattern-matching

- Powerful tool for dissecting data like trees

```
(match '(1 2 3)
  [`(1 ,x 4)  x]
  [`(1 2 ,z)  z]) => 3
```

Pattern-matching

- Powerful tool for dissecting data like trees

```
(match '(1 2 3)
  [`(1 ,x 4)  x]
  [`(1 2 ,z)  z]) => 3

(match '(1 3 5 9)
  [(list 1 (? even?) ...)
   'all-even]
  [(list 1 (? odd?) ...)
   'all-odd])                    => all-odd
```

Macros

Code as data

```
(unless (> 3 x)
  (printf "foo"))
```

Code as data

```
'(unless (> 3 x)
   (printf "foo"))
```

Code as data

```
(cdr '(unless (> 3 x)
        (printf "foo"))) => ((> 3 x) (printf "foo"))
```

Once upon a time

define-macro

```
(define-macro (unless test body)
`(if ,test #f (begin ,body)))
```

define-macro

```
(define-macro (or exp1 exp2)
  `(if ,exp1 ,exp1 ,exp2))
```

Oops

```
(or (read) (read))
```

define-macro

```
(define-macro (or exp1 exp2)
  `(let ((tmp ,exp1))
     (if tmp tmp ,exp2)))
```

Oops

```
(or foo tmp)
```

define-macro

```
(define-macro (or exp1 exp2)
  (let (($tmp (gensym 'tmp)))
    `(let ((,$tmp ,exp1))
       (if ,$tmp ,$tmp ,exp2))))
```

Oops

```
(let ([unless fire-the-missiles])
  (unless world-is-ending?
    (lock-the-missiles)))
```

Fixing bad hygiene: syntax-rules

syntax-rules

```
(define-syntax <name>
  (syntax-rules (<keyword> ...)
    [<pattern>  <rewrite>]
    ...))
```

or

```
(define-syntax and
  (syntax-rules ()
    [(or exp1 exp2)
     (let ((tmp exp))
       (if tmp tmp exp2))]))
```

and

```
(define-syntax and
  (syntax-rules ()
    [(and exp1 exp2)    (if exp1 exp2 #f)]))
```

and

```
(define-syntax and
  (syntax-rules ()
    [(and)                          #t]
    [(and exp)                      exp]
    [(and exp1 exp2 ...)            (if exp1
                                        (and exp2 ...)
                                        #f)]))
```

Teaser

amb

```
(let ((x (amb 1 3 4))
      (y (amb 4 5 6))
      (z (amb 5 6 7)))
  (assert (= (+ (* x x) (* y y)) (* z z)))
  (printf "~a ~a ~a" x y z))
```