Scheme Lisp - Feel the Cool

Andy Balaam, OpenMarket
artificialworlds.net/blog

Intro

Scheme is

- simple,
- weird, and
- cool

Simple

Scheme is simple by design.

- Designed for teaching[1]
- Used as the basis of the Computer Science course at MIT
- Based on Lambda Calculus[2]

---

[1] by Sussman and Steele, 1970s

[2] Alonzo Church, 1930

Simple to try

```
$ sudo apt install racket
$ mzscheme
>
```

Simple to use

Scheme has:

- One thing you can do
- One data structure

Both are actually the same.

Simple syntax

```
(operator operand1 operand2 ...)
```

Simple expressions

```
> (+ 3 4)
```

Simple expressions

```
> (+ 3 4)
7
```

Simple expressions

```
> (+ 3 4)
7
> (* 3 4)
```

Simple expressions

```
> (+ 3 4)
7
> (* 3 4)
12
```

Simple expressions

```
> (+ 3 4)
7
> (* 3 4)
12
> (+ 5 (* 2 2))
```

Simple expressions

```
> (+ 3 4)
7
> (* 3 4)
12
> (+ 5 (* 2 2))
9
```

Simple definitions

```
> (define foo 3)
```

Simple definitions

```
> (define foo 3)

> foo
```

Simple definitions

```
> (define foo 3)

> foo
3
```

Simple definitions

```
> (define foo 3)

> foo
3
> (* foo 4)
```

Simple definitions

```
> (define foo 3)

> foo
3
> (* foo 4)
12
```

Simple functions

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

Simple functions

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

> (square 4)
```

Simple functions

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

> (square 4)
16
```

Simple functions

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

> (square 4)
16
> (+ (square 2) (square 3))
```

Simple functions

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

> (square 4)
16
> (+ (square 2) (square 3))
13
```

Simple flow control

```
(define (abs x)
        (if (< x 0)
            (- x)
            x))
```

Simple flow control

```
(define (abs x)
        (if (< x 0)
            (- x)
            x))
```

```
> (abs -3)
```

Simple flow control

```
(define (abs x)
        (if (< x 0)
            (- x)
            x))
```

```
> (abs -3)
3
```

Simple flow control

```
(define (abs x)
        (if (< x 0)
            (- x)
            x))
```

```
> (abs -3)
3
> (abs 3)
```

Simple flow control

```
(define (abs x)
        (if (< x 0)
            (- x)
            x))
```

```
> (abs -3)
3
> (abs 3)
3
```

Simple data structure

```
> (list 9 3 5)
```

Simple data structure

```
> (list 9 3 5)
(9 3 5)
```

Simple data structure

```
> (list 9 3 5)
(9 3 5)
> (sort (list 9 3 5) <)
```

Simple data structure

```
> (list 9 3 5)
(9 3 5)
> (sort (list 9 3 5) <)
(3 5 9)
```

Simple data structure

```
> (list 9 3 5)
(9 3 5)
> (sort (list 9 3 5) <)
(3 5 9)
> (length (list 3 2))
```

Simple data structure

```
> (list 9 3 5)
(9 3 5)
> (sort (list 9 3 5) <)
(3 5 9)
> (length (list 3 2))
2
```

Weird

Scheme is weird.

- Building lists from pairs
- Recursion for everything
- Passing functions into functions
- Data/code duality

Weird lists

```
> (define x (list 1 2 3))
```

Weird lists

```
> (define x (list 1 2 3))

> (car x)
```

Weird lists

```
> (define x (list 1 2 3))

> (car x)
1
```

Weird lists

```
> (define x (list 1 2 3))

> (car x)
1
> (cdr x)
```

Weird lists

```
> (define x (list 1 2 3))

> (car x)
1
> (cdr x)
(2 3)
```

Weird pairs

```
> (cons "a" "b")
```

Weird pairs

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

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
> (define p (cons 1 2))
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
> (define p (cons 1 2))

> (car p)
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
> (define p (cons 1 2))

> (car p)
1
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
> (define p (cons 1 2))

> (car p)
1
> (cdr p)
```

Weird pairs

```
> (cons "a" "b")
("a" . "b")
> (cons (cons 5 6) 7)
((5 . 6) . 7)
> (define p (cons 1 2))

> (car p)
1
> (cdr p)
2
```

Weird list-building

```
> null
```

Weird list-building

```
> null
()
```

Weird list-building

```
> null
()
> (cons 2 null)
```

Weird list-building

```
> null
()
> (cons 2 null)
(2)
```

Weird list-building

```
> null
()
> (cons 2 null)
(2)
> (cons 1 (cons 2 null))
```

Weird list-building

```
> null
()
> (cons 2 null)
(2)
> (cons 1 (cons 2 null))
(1 2)
```

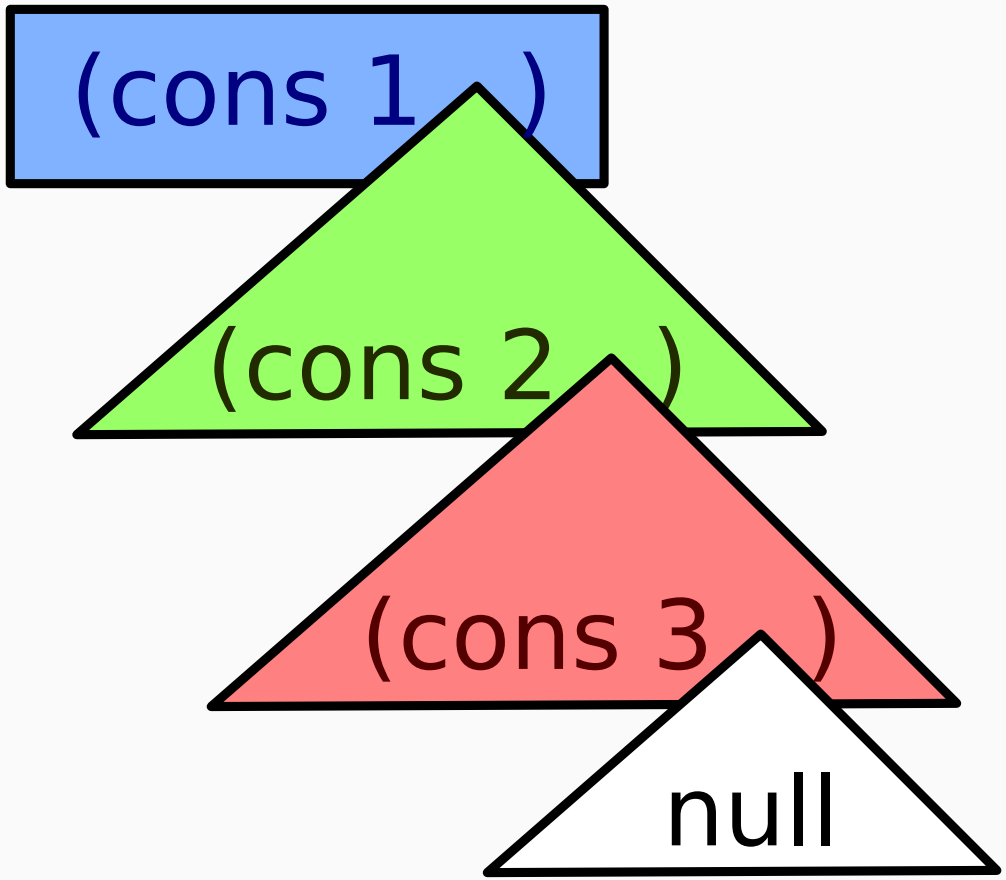
Weird list-building

```
> null
()
> (cons 2 null)
(2)
> (cons 1 (cons 2 null))
(1 2)
> (list 1 2)
```

Weird list-building

```
> null
()
> (cons 2 null)
(2)
> (cons 1 (cons 2 null))
(1 2)
> (list 1 2)
(1 2)
```

Weird lists made of pairs




```
(list 1 2 3)
```

Weirdly named functions

- `cons` sticks things together
- `car` gives you the first thing
- `cdr` gives you the second thing

Weirdly named functions

- `cons` sticks things together
- `car` gives you the “head”
- `cdr` gives you the “tail”

Weirdly named functions

- So `cadddr`'s meaning should be obvious

Weirdly named functions

- So `cadddr`'s meaning should be obvious
- Right?

Weirdly named functions

```
(caaddr v)
Returns (car (cdr (cdr (cdr v))))
```

Weird recursion

```
(define (sum vs)
        (if (= 1 (length vs))
            (car vs)
            (+ (car vs)
               (sum (cdr vs)))))
```

Weird recursion

```
(define (sum vs)
        (if (= 1 (length vs))
            (car vs)
            (+ (car vs)
               (sum (cdr vs)))))

> (sum (list 5 6 7))
```

Weird recursion

```
(define (sum vs)
        (if (= 1 (length vs))
            (car vs)
            (+ (car vs)
               (sum (cdr vs)))))
```

```
> (sum (list 5 6 7))
18
```

Weird recursion

```
(define (sum vs)
        (if (= 1 (length vs))
            (car vs)
            (+ (car vs)
               (sum (cdr vs)))))
```

Weird recursion

```
(define (sum vs)
        (if (= 1 (length vs))
            (car vs)
            (+ (car vs)
               (sum (cdr vs)))))
```

Weird meta-functions

```
(define (double value)
        (* 2 value))

(define (apply-twice fn value)
        (fn (fn value)))

> (apply-twice double 2)
```

Weird meta-functions

```
(define (double value)
        (* 2 value))

(define (apply-twice fn value)
        (fn (fn value)))

> (apply-twice double 2)
8
```

Weird meta-functions

```
(define (apply-twice fn value)
        (fn (fn value)))
```

Weird functions as values

```
> (map double (list 3 4 5))
```

Weird functions as values

```
> (map double (list 3 4 5))
(6 8 10)
```

Weird code as data

```
> (define s (list '+ 4 7))
```

Weird code as data

```
> (define s (list '+ 4 7))
> s
```

Weird code as data

```
> (define s (list '+ 4 7))
> s
(+ 4 7)
```

Weird code as data

```
> (define s (list '+ 4 7))
> s
(+ 4 7)
> (eval s)
```

Weird code as data

```
> (define s (list '+ 4 7))
> s
(+ 4 7)
> (eval s)
11
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
> (define s2 (switchop s))
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
> (define s2 (switchop s))
> s2
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
> (define s2 (switchop s))
> s2
(* 4 7)
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
> (define s2 (switchop s))
> s2
(* 4 7)
> (eval s2)
```

Weird code as data

```
> (define (switchop a) (cons '* (cdr a)))
> (define s2 (switchop s))
> s2
(* 4 7)
> (eval s2)
28
```

Cool

- Quoting
- Better names
- Duck typing (generics)
- Lambdas & Closures
- Metaprogramming

Cool quoting

```
> '(* 3 6)
```

Cool quoting

```
> '(* 3 6)
(* 3 6)
```

Cool quoting

```
> '(* 3 6)
(* 3 6)

> '(foo (bar "a" 3))
```

Cool quoting

```
> '(* 3 6)
(* 3 6)

> '(foo (bar "a" 3))
(foo (bar "a" 3))
```

Cool names

- These are all valid names in Scheme:

```
equal?
boom!
a*b
co-ordinates
<10
+
```

This is cool.

Cool replacement

- This works:

```
> (define (+ x y) 5)
> (+ 2 2)
5
```

This is cool.

Cool Duck Typing

```
> (sort (list 5 4 3 2 1) <)
(1 2 3 4 5)
> (sort (list "abc" "a" "ab") string<?)
("a" "ab" "abc")
```

This is somewhat uncool, but useful.

Cool lambdas

```
> (map
        (lambda (x) (+ x 1))
        (list 1 2 3))
(2 3 4)
```

Cool closures

```
(define (counter)
        (define c 0)
        (lambda ()
                (set! c (+ c 1))
                c))
```

Cool closures

```
(define (counter)
        (define c 0)
        (lambda ()
                (set! c (+ c 1))
                c))
```

Cool closures

```
> (define a (counter))
> (a)
```

Cool closures

```
> (define a (counter))
> (a)
1
```

Cool closures

```
> (define a (counter))
> (a)
1
> (a)
```

Cool closures

```
> (define a (counter))
> (a)
1
> (a)
2
```

Cool closures

```
> (define a (counter))
> (a)
1
> (a)
2
> (a)
```

Cool closures

```
> (define a (counter))
> (a)
1
> (a)
2
> (a)
3
```

Cool closures

```
> (define b (counter))
> (b)
```

Cool closures

```
> (define b (counter))
> (b)
1
```

Cool closures

```
> (define b (counter))
> (b)
1
> (a)
```

Cool closures

```
> (define b (counter))
> (b)
1
> (a)
4
```

Cool metaprogramming

Metaprogramming is just programming.

```
> (define (times-n n) (lambda (x) (* n x)))
```

Cool metaprogramming

Metaprogramming is just programming.

```
> (define (times-n n) (lambda (x) (* n x)))
> (define times3 (times-n 3))
```

Cool metaprogramming

Metaprogramming is just programming.

```
> (define (times-n n) (lambda (x) (* n x)))
> (define times3 (times-n 3))
> (define (trpl lst) (map times3 lst))
```

Cool metaprogramming

Metaprogramming is just programming.

```
> (define (times-n n) (lambda (x) (* n x)))
> (define times3 (times-n 3))
> (define (trpl lst) (map times3 lst))
> (trpl (list 1 2 3))
```

Cool metaprogramming

Metaprogramming is just programming.

```
> (define (times-n n) (lambda (x) (* n x)))
> (define times3 (times-n 3))
> (define (trpl lst) (map times3 lst))
> (trpl (list 1 2 3))
(3 6 9)
```

Cool things I haven't mentioned

- Macros
- Streams
- The Metacircular Evaluator

Cool reading

- *Structure and Interpretation of Computer Programs*[3] changed my life

[3] https://mitpress.mit.edu/sicp/full-text/book/book.html

Questions

Extra - data from functions (1)

```
(define (mcons a b)
    (lambda (cmd)
        (if (equal? cmd "car")
            a
            b)))

(define (mcar pair) (pair "car"))
(define (mcdr pair) (pair "cdr"))
```

Extra - data from functions (2)

```
> (define foo (mcons 1 2))
> (mcar foo)
1
> (mcdr foo)
2
```

Extra - numbers from functions (1)

```
(define n0 (lambda () null))

(define (minc x) (lambda () x))

(define (mdec x) (x))
```

Extra - numbers from functions (2)

```
(define n1 (minc n0))
(define n2 (minc n1))
(define n3 (minc n2))
(define n4 (minc n3))
(define n5 (minc n4))
```

Extra - numbers from functions (3)

```
(define (mzero? x) (null? (x)))

(define (mequal? x y)
    (cond
        ((mzero? x) (mzero? y))
        ((mzero? y) (mzero? x))
        (else (mequal? (mdec x) (mdec y)))))
```

Extra - numbers from functions (4)

```
> (mequal? n1 n0)
#f
> (mequal? n1 n1)
#t
```

Extra - numbers from functions (5)

```
(define (m+ x y)
    (if (mzero? y)
        x
        (m+ (minc x) (mdec y))))
```

Extra - numbers from functions (6)

```
> (mequal? (m+ n0 n2) n2)
#t
> (mequal? (m+ n0 n2) n3)
#f
> (mequal? (m+ n0 n2) (m+ n1 n2))
#f
> (mequal? (m+ n2 n3) n5)
#t
```