# Scheme

# Announcements

# Scheme

# Scheme is a Dialect of Lisp

What are people saying about Lisp?

- "If you don't know Lisp, you don't know what it means for a programming language to be powerful and elegant."

  - Richard Stallman, created Emacs & the first free variant of UNIX

- "The only computer language that is beautiful."

  –Neal Stephenson, DeNero's favorite sci-fi author

- "The greatest single programming language ever designed."

  –Alan Kay, co-inventor of Smalltalk and OOP (from the user interface video)

## Scheme Expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2   3.3   true   +   quotient
- Combinations: (quotient 10 2)   (not true)

Numbers are self-evaluating; symbols are bound to values

Call expressions include an operator and 0 or more operands in parentheses

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn't matter)

(Demo)

# Special Forms

# Special Forms

A combination that is not a call expression is a special form:

- **if** expression: (if <predicate> <consequent> <alternative>)
- **and** and **or**: (and <e1> **...** <en>), (or <e1> **...** <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

Evaluation:

(1) Evaluate the predicate expression

(2) Evaluate either the consequent or alternative

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The symbol “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

A procedure is created and bound to the symbol “abs”

(Demo)

# Scheme Interpreters

(Demo)

# Lambda Expressions

# Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

λ

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a call expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

12

Evaluates to the $x+y+z^2$ procedure

# Sierpinski's Triangle

(Demo)

# More Special Forms

## Cond & Begin

The cond special form that behaves like if-elif-else statements in Python

```
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```
(cond ((> x 10) (print 'big))
      ((> x 5)  (print 'medium))
      (else     (print 'small)))
```

```
(print
  (cond ((> x 10) 'big)
        ((> x 5)  'medium)
        (else     'small)))
```

The begin special form combines multiple expressions into one expression

```
if x > 10:
    print('big')
    print('guy')
else:
    print('small')
    print('fry')
```

```
(cond ((> x 10) (begin (print 'big)   (print 'guy)))
      (else     (begin (print 'small) (print 'fry))))
```

```
(if (> x 10) (begin
                (print 'big)
                (print 'guy))
             (begin
                (print 'small)
                (print 'fry)))
```

# Let Expressions

The let special form binds symbols to values temporarily; just for one expression

```python
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
```

*a and b are **still** bound down here*

```scheme
(define c (let ((a 3)
                (b (+ 2 2)))
               (sqrt (+ (* a a) (* b b)))))
```

*a and b are **not** bound down here*

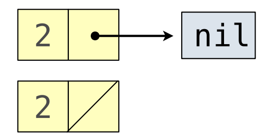
# Lists

# Scheme Lists

In the late 1950s, computer scientists used confusing names

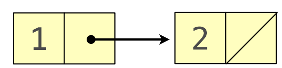
- **cons:** Two-argument procedure that creates a linked list
- **car:** Procedure that returns the first element of a list
- **cdr:** Procedure that returns the rest of a list
- **nil:** The empty list

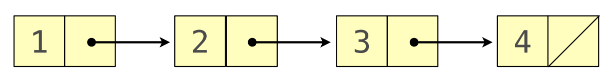
(cons 2 nil)




**Important! Scheme lists are written in parentheses with elements separated by spaces**

```
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```




(Demo)

# Symbolic Programming

# Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

(Demo)

# Programs as Data

## A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)

scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

(Demo)

# Generating Code

## Quasiquotation

There are two ways to quote an expression

```
  Quote:      '(a b)   =>   (a b)

  Quasiquote: `(a b)   =>   (a b)
```

They are different because parts of a quasiquoted expression can be unquoted with ,

```
              (define b 4)

  Quote:      '(a ,(+ b 1))  =>   (a (unquote (+ b 1))

  Quasiquote: `(a ,(+ b 1))  =>   (a 5)
```

Quasiquotation is particularly convenient for generating Scheme expressions:

```
              (define (make-add-procedure n) `(lambda (d) (+ d ,n)))

              (make-add-procedure 2)  => (lambda (d) (+ d 2))
```

## Example: While Statements

What's the sum of the squares of even numbers less than 10, starting with 2?

```
x = 2
total = 0
while x < 10:
    total = total + x * x
    x = x + 2
```

```
(begin
  (define (f x total)
    (if (< x 10)
      (f (+ x 2) (+ total (* x x)))
      total))
  (f 2 0)))
```

What's the sum of the numbers whose squares are less than 50, starting with 1?

```
x = 1
total = 0
while x * x < 50:
    total = total + x
    x = x + 1
```

```
(begin
  (define (f x total)
    (if (< (* x x) 50)
      (f (+ x 1) (+ total x))
      total))
  (f 1 0)))
```

(Demo)