Scheme is a Dialect of Lisp

What are people saying about Lisp?

• "The greatest single programming language ever designed."  
  – Alan Kay, co-inventor of Smalltalk and OOP

• "The only computer language that is beautiful."  
  – Neal Stephenson, John's favorite sci-fi author

• "God's programming language."  
  – Brian Harvey, Berkeley CS instructor extraordinaire

http://imgs.xkcd.com/comics/lisp_cycles.png
Scheme Fundamentals

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 have an operator and 0 or more operands.

```
> (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)
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 names**: `(define <name> <expression>)`
- **New procedures**: `(define (<name> <formal parameters>) <body>)`

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

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

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

A procedure is created and bound to the name “abs”
Lambda Expressions

Lambda expressions evaluate to anonymous functions.

$\lambda \langle \text{formal-parameters} \rangle \langle \text{body} \rangle$

Two equivalent expressions:

$\text{(define (plus4 x) (+ x 4))}$

$\text{(define plus4 (lambda (x) (+ x 4)))}$

An operator can be a call expression too:

$\text{((lambda (x y z) (+ x y (square z))) 1 2 3)}$

Evaluates to the $\text{add-x-} \& \text{-y-} \& \text{-z}^2$ procedure
Pairs and Lists

In the late 1950s, computer scientists used confusing names.

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

They also used a non-obvious notation for recursive lists.

• A (recursive) Scheme list is a pair in which the second element is nil or a Scheme list.
• Scheme lists are written as space-separated combinations.
• A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

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

Not a well-formed list!
Symbols normally refer to values; how do we refer to symbols?

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

Quotation is used to refer to symbols directly in Lisp.

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

Quotation can also be applied to combinations to form lists.

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```
Dots can be used in a quoted list to specify the second element of the final pair.

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

However, dots appear in the output only of ill-formed lists.

> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)

What is the printed result of evaluating this expression?

> (cdr '(((1 2) . (3 4 . (5))))
(3 4 5)
Coercing a Sorted List to a Binary Search Tree

Divide length \( n \) into 3 parts: \( \left[ \frac{n-1}{2}, 1, \frac{n-1}{2} \right] \)

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
The Let Special Form

(define (entry tree) ...)
(define (left-branch tree) ...)
(define (right-branch tree) ...)
(define (make-tree entry left right) ...)
(define (list->tree elements)
  (car (partial-tree elements (length elements))))

(define (partial-tree elts n)
  (if (= n 0)
      (cons nil elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
            (non-left-elts (cdr left-result))
            (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
              (right-result (partial-tree (cdr non-left-elts) right-size)))
              (let ((right-tree (car right-result))
                (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree)
                  remaining-elts)))))))
The Begin Special Form

\[(\text{begin} \ <\text{exp}_1> \ <\text{exp}_2> \ ... \ <\text{exp}_n>)\]