Scheme Is a Dialect of 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, sci-fi author

*The most powerful programming language is Lisp. If you don’t know Lisp or its variant, Scheme, you don’t appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.*
  - Richard Stallman, founder of the Free Software movement

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

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

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

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

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

Lambda Expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

An operator can be a combination too:

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

Evaluates to the add-x-y-z-\(x^2\) procedure

Lambda Expressions
Pairs

We can implement pairs functionally:

```scheme
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also has built-in pairs that use weird 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

A pair is represented by a dot between the elements, all in parens

```scheme
> (cons 1 2)
  (1 . 2)
> (car (cons 1 2))
  1
> (cdr (cons 1 2))
  2
```

Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list

Scheme lists are recursive lists:

• `nil` is the empty list
• A non-empty Scheme list is a pair in which the second element is `nil` or a Scheme list

Scheme lists are written as space-separated combinations

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

Not a well-formed list!

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

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

Quotation prevents something from being evaluated by Lisp

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

Quotation can also be applied to combinations to form lists

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

What is the printed result of evaluating this expression?

```scheme
> (cdr '(1 2 . (3 4 . (5)))))
  (3 4 5)
```

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

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

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

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

What is the printed result of evaluating this expression?

```scheme
> (cdr '((1 2 . (3 . 4 . (5))))
  (3 4 5)
```