Scheme
Announcements
Scheme
Scheme is a Dialect of Lisp
Scheme is a Dialect of Lisp

What are people saying about Lisp?
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
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
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 Expressions

Scheme programs consist of expressions, which can be:
Scheme Expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2  3.3  true  +  quotient
Scheme Expressions

Scheme programs consist of expressions, which can be:

- **Primitive expressions**: `2`  `3.3`  `true`  `+`  `quotient`
- **Combinations**: `(quotient 10 2)`  `(not true)`
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
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
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
```
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” names Scheme’s built-in integer division procedure (i.e., function)
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

```scheme
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
```

“quotient” names Scheme’s built-in integer division procedure (i.e., function)
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)
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

```scheme
> (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)
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

```scheme
> (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)
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).
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)
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)
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:
A combination that is not a call expression is a special form:

- **if** expression: \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>\)
A combination that is not a call expression is a special form:

- **if** expression:  
  \[(\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)\]

**Evaluation:**
1. Evaluate the predicate expression
2. Evaluate either the consequent or alternative
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>)

Evaluation:
1. Evaluate the predicate expression
2. Evaluate either the consequent or alternative
Special Forms

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

- **if** expression:  \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)\)
- **and** and **or**:  \((\text{and} \ <e1> \ldots \ <en>), \ (\text{or} \ <e1> \ldots \ <en>)\)
- **Binding symbols**:  \((\text{define} \ <\text{symbol}> \ <\text{expression}>)\)

Evaluation:
1. Evaluate the predicate expression
2. Evaluate either the consequent or alternative
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>)`

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

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

- **if** expression:  
  \[
  (\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)
  \]

- **and** and **or**:  
  \[
  (\text{and } <e1> ... <en>), (\text{or } <e1> ... <en>)
  \]

- Binding symbols:  
  \[
  (\text{define } <\text{symbol}> <\text{expression}>)
  \]

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
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>)

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

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative

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

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

- **if** expression:  \((\text{if } \text{<predicate>} \; \text{<consequent>} \; \text{<alternative>})\)
- **and** and **or**:  \((\text{and } \text{<e1>} \ldots \; \text{<en>}), \; (\text{or } \text{<e1>} \ldots \; \text{<en>})\)
- Binding symbols:  \((\text{define } \text{<symbol>} \; \text{<expression>})\)
- New procedures:  \((\text{define } (\text{<symbol>} \; \text{<formal parameters>}) \; \text{<body>})\)

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

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

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative
A combination that is not a call expression is a special form:

- **if** expression:  
  \[ (\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}> ) \]

- **and** and **or**:  
  \[ (\text{and} \ <e1> \ldots \ <en> ), \ (\text{or} \ <e1> \ldots \ <en> ) \]

- Binding symbols:  
  \[ (\text{define} \ <\text{symbol}> \ <\text{expression}> ) \]

- New procedures:  
  \[ (\text{define} \ (<\text{symbol}> \ <\text{formal parameters}> ) \ <\text{body}> ) \]

```
> (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:  
  \[(\text{if } \text{<predicate>} \text{ <consequent>} \text{ <alternative>})\]

- **and** and **or**:  
  \[(\text{and } \text{<e1>} \ldots \text{ <en>}), (\text{or } \text{<e1>} \ldots \text{ <en>})\]

- **Binding symbols**:  
  \[(\text{define } \text{<symbol>} \text{ <expression>})\]

- **New procedures**:  
  \[(\text{define } (\text{<symbol>} \text{ <formal parameters>}) \text{ <body>})\]

Evaluation:

1. Evaluate the predicate expression
2. Evaluate either the consequent or alternative

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

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

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

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

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

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

\[
\begin{align*}
> & \ (\text{define } \text{pi} \ 3.14) \\
> & \ (* \ \text{pi} \ 2) \\
& \ 6.28 \\
> & \ (\text{define } (\text{abs } x) \\
& \quad (\text{if } (< \ x \ 0) \\
& \quad \quad (- \ x) \\
& \quad \quad x)) \\
> & \ (\text{abs} \ -3) \\
& \ 3
\end{align*}
\]

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative
Scheme Interpreters (Demo)
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to anonymous procedures
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

(lambda (<formal-parameters>) <body>)
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[(\text{lambda} (<\text{formal-parameters}>)) <\text{body}>\]
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
\text{(lambda (<formal-parameters>) <body>)}
\]

Two equivalent expressions:

\[
\text{(define (plus4 x) (+ x 4))}
\]

\[
\text{(define plus4 (lambda (x) (+ x 4)))}
\]
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

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

An operator can be a call expression too:
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
\text{(lambda (<formal-parameters>) <body>)}
\]

Two equivalent expressions:

\[
\text{(define (plus4 x) (+ x 4))}
\]

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

An operator can be a call expression too:

\[
((\text{lambda (x y z) (+ x y (square z))) 1 2 3})
\]
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
\lambda (\text{<formal-parameters>}) \text{<body>}
\]

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 \(x+y+z^2\) procedure
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
\lambda (\text{formal-parameters}) \text{ body}
\]

Two equivalent expressions:

\[
\text{(define (plus4 x) (+ x 4))}
\]

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

An operator can be a call expression too:

\[
((\text{lambda (x y z) (+ x y (square z)))} 1 2 3) \rightarrow 12
\]

Evaluates to the \(x+y+z^2\) procedure
More Special Forms
Cond & Begin
Cond & Begin

The cond special form that behaves like if–elif–else statements in Python
Cond & Begin

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

```python
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```
The cond special form that behaves like if-elif-else statements in Python

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

```prolog
(cond (> x 10) (print 'big))
    (> x 5) (print 'medium))
    (else (print 'small)))
```
The cond special form that behaves like if–elif–else statements in Python

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

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

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

(\texttt{cond \((> x \ 10) \ (print \ 'big')\)} \texttt{(cond \((> x \ 5) \ (print \ 'medium')\)} \texttt{(\texttt{else} \ (print \ 'small')))}\texttt{(print \ 'small'))})
The cond special form that behaves like if–elif–else statements in Python

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

The begin special form combines multiple expressions into one expression
The cond special form that behaves like if–elif–else statements in Python

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

The begin special form combines multiple expressions into one expression

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

The begin special form combines multiple expressions into one expression
The cond special form that behaves like if–elif–else statements in Python

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

```
(cond (> x 10) 'big)
  ((> x 5) 'medium)
  (else 'small)))
```
**Cond & Begin**

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

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

The begin special form combines multiple expressions into one expression

```lisp
(if x > 10:
  print('big')
  print('guy')
  (cond ((> x 10) (begin (print 'big)   (print 'guy)))
         ((> x 5)  (print 'medium))
         (else   (print 'small)))  
  (begin
   (print 'small)
   (print 'fry))))
 Else:
  print('small')
  print('fry'))
```

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

```lisp
(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.
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)
```
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*
Let Expressions

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

\[
\begin{align*}
    a &= 3 \\
    b &= 2 + 2 \\
    c &= \text{math.sqrt}(a \times a + b \times b)
\end{align*}
\]

\textit{a and b are still bound down here}
Let Expressions

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

\[
\begin{align*}
a &= 3 \\
b &= 2 + 2 \\
c &= \text{math.sqrt}(a \times a + b \times b)
\end{align*}
\]

\[
\begin{align*}
\text{a and b are } \textbf{still} \text{ bound down here} \\
(\text{define} \ c \ (\text{let} \ ((a \ 3) \ (b \ (+ \ 2 \ 2))) \ (\text{sqrt} \ (+ \ (* \ a \ a) \ (* \ b \ b))))
\end{align*}
\]

\[
\begin{align*}
\text{a and b are } \textbf{not} \text{ bound down here}
\end{align*}
\]
Lists
Scheme Lists
In the late 1950s, computer scientists used confusing names.
Scheme Lists

In the late 1950s, computer scientists used confusing names
- **cons**: Two-argument procedure that creates a linked list
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
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
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
Schema 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) 2 nil
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
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

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

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

(cons 2 nil)
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

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

> (cons 1 (cons 2 nil))
Scheme Lists

In the late 1950s, computer scientists used confusing names

• **cons**: Two-argument procedure that creates a linked list
  (cons 2 nil)  
  ![Diagram](cons-diagram.png)

• **car**: Procedure that returns the first element of a list
  ![Diagram](car-diagram.png)

• **cdr**: Procedure that returns the rest of a list
  ![Diagram](cdr-diagram.png)

• **nil**: The empty list

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

```plaintext
> (cons 1 (cons 2 nil))
(1 2)
```
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

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))
```
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

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

```scheme
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil)))
> x
```
Scheme Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a linked list
  - `(cons 2 nil)`
- **car**: Procedure that returns the first element of a list
  - `(1 2)`
- **cdr**: Procedure that returns the rest of a list
  - `(define x (cons 1 (cons 2 nil)))`
- **nil**: The empty list

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

> `(cons 1 (cons 2 nil))`

> `(define x (cons 1 (cons 2 nil)))`

> `x`

> `(1 2)`
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

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

```scheme
> (cons 1 (cons 2 nil))  
(1 2)  
> (define x (cons 1 (cons 2 nil)))  
> x  
(1 2)  
> (car x)
```
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

**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`
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

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)
```
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

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

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

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))))
```
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

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

```scheme
> (cons 1 (cons 2 nil))  (cons 2 nil)  1  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
1  2  3  4
```
Scheme Lists

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

- **cons**: Two-argument procedure that creates a linked list.
  - (cons 2 nil)

- **car**: Procedure that returns the first element of a list.
  - > (define x (cons 1 (cons 2 nil)))
  - > x
  - > (car x)
  - 1

- **cdr**: Procedure that returns the rest of a list.
  - > (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
  - (1 2 3 4)

- **nil**: The empty list.

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)
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

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

```
> (cons 1 (cons 2 nil))          (cons 2 nil)       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
Symbolic Programming

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

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

> (define a 1)
Symbolic Programming

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

> (define a 1)
> (define b 2)
Symbolic Programming

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

> (define a 1)
> (define b 2)
> (list a b)
Symbolic Programming

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

> (define a 1)
> (define b 2)
> (list a b)
(1 2)
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
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.
Symbolic Programming

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)
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)
Symbolic Programming

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)
Symbolic Programming

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)
Symbolic Programming

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)
```

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

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

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.
Symbolic Programming

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.

> '(a b c)
Symbolic Programming

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.

> '(a b c)
(a b c)
Symbolic Programming

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

```lisp
> (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.

```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.

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

Symbolic Programming

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.

> '(a b c)
(a b c)
> (car '(a b c))
a
Symbolic Programming

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.

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

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.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```
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
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
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)
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
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)
```
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.

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

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

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

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

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

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

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

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

There are two ways to quote an expression
Quasiquotation

There are two ways to quote an expression

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

There are two ways to quote an expression

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

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

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:
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)))
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
Example: While Statements

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

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

```python
x = 2
total = 0
while x < 10:
    total = total + x * x
    x = x + 2
```
Example: While Statements

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

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

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

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

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

```scheme
(define (f x total)
    (if (< x 10)
        (f (+ x 2) (+ total (* x x)))
        total))
(f 2 0))
```
Example: While Statements

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

```plaintext
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)))
Example: While Statements

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

\[
x = 2 \\
total = 0 \\
while x < 10:
    total = total + x * x \\
x = x + 2
\]

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

\[
(begin \\
    (define (f x total)
        (if (< x 10)
            (f (+ x 2) (+ total (* x x)))
            total))
    (f 2 0)))
\]
Example: While Statements

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

\[
x = 2 \\
\text{total} = 0 \\
\text{while } x < 10: \\
\quad \text{total} = \text{total} + x \times x \\
\quad x = x + 2
\]

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

\[
x = 1 \\
\text{total} = 0 \\
\text{while } x \times x < 50: \\
\quad \text{total} = \text{total} + x \\
\quad x = x + 1
\]
Example: While Statements

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

\[
\begin{align*}
& x = 2 \\
& \text{total} = 0 \\
& \textbf{while} \ x < 10: \\
& \quad \text{total} = \text{total} + x \times x \\
& \quad x = x + 2 \\
& \end{align*}
\]

\[
\begin{align*}
&\text{(define} \ (f \ x \ \text{total}) \\
&\quad \text{(if} \ (< \ x \ 10) \\
&\quad \quad (f \ (+ \ x \ 2) \ (+ \ \text{total} \ (* \ x \ x))) \\
&\quad \quad \text{total})) \\
&\quad \text{(f} \ 2 \ 0))
\end{align*}
\]

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

\[
\begin{align*}
& x = 1 \\
& \text{total} = 0 \\
& \textbf{while} \ x \times x < 50: \\
& \quad \text{total} = \text{total} + x \\
& \quad x = x + 1 \\
& \end{align*}
\]

\[
\begin{align*}
&\text{(define} \ (f \ x \ \text{total}) \\
&\quad \text{(if} \ (< \ (* \ x \ x) \ 50) \\
&\quad \quad (f \ (+ \ x \ 1) \ (+ \ \text{total} \ x)) \\
&\quad \quad \text{total})) \\
&\quad \text{(f} \ 1 \ 0))
\end{align*}
\]
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
```

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
```

(Demo)