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 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:** $2 \ 3.3 \ true \ + \ \text{quotient}$
- **Combinations:** $(\text{quotient} \ 10 \ 2) \ (\text{not} \ true)$
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

```scheme
> (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

```
> (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

```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)

(Demo)
Special Forms
Special Forms
A combination that is not a call expression is a special form:
Special Forms

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

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

- **if** expression: `(if <predicate> <consequent> <alternative>)`

**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: (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**: \((\text{and} \ <e_1> \ldots \ <e_n>), \ (\text{or} \ <e_1> \ldots \ <e_n>)\)
- Binding symbols: \((\text{define} \ <\text{symbol}> \ <\text{expression}>)\)

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

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

```scheme
> (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**: \((\text{and} \ <\text{e}1> ... <\text{e}n>), (\text{or} <\text{e}1> ... <\text{e}n>)\)
- Binding symbols: \((\text{define} <\text{symbol}> <\text{expression}>)\)

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

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

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:  (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>)

 evaluating:
  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: \((\text{if} \ <\text{predicate}\> \ <\text{consequent}\> \ <\text{alternative}\>)\)
- **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: \((\text{define} \ (<\text{symbol}\> \ <\text{formal parameters}\>) \ <\text{body}\>)\)

```
> (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

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

\[
\begin{align*}
> (\text{define} \ \text{pi} \ 3.14) \\
> (* \ \text{pi} \ 2) \\
& 6.28
\end{align*}
\]

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

```lisp
> (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

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: `(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

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

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

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

Two equivalent expressions:

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

Lambda expressions evaluate to anonymous procedures

\(\lambda\) (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

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

Two equivalent expressions:

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

(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

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

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 \text{x+y+z^2} procedure
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 (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
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

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

```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

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

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

The begin special form combines multiple expressions into one expression

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

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

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

The begin special form combines multiple expressions into one expression

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

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))
   ((> x 5)  (print 'medium))
   (else    (print 'small) (print 'fry)))
```
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))))
```

The begin special form combines multiple expressions into one expression

```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

```
    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

```plaintext
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

\[
\begin{align*}
    a &= 3 \\
    b &= 2 + 2 \\
    c &= \text{math.sqrt}(a \times a + b \times b) \\
    \text{a and b are } \textit{still} \text{ bound down here}
\end{align*}
\]

\[
\begin{align*}
    \text{define } c \ (\text{let} \ ((a 3) \\
    \hspace{1cm} (b (+ 2 2)) \\
    \hspace{1cm} (\text{sqrt} (+ (* a a) (* b b)))))
\end{align*}
\]
Let Expressions

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

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

\(a\) and \(b\) are \textbf{still} bound down here

\[
\begin{align*}
\text{define }\ c \ &= \text{(let }\ ((a \ 3) \\
& \quad (b \ (+ \ 2 \ 2)) \\
& \quad (\text{sqrt} \ (+ \ (* \ a \ a) \ (* \ b \ b))))
\end{align*}
\]

\(a\) and \(b\) are \textbf{not} bound down here
Turtle Graphics
Drawing Stars

(forward 100) or (fd 100) draws a line

(right 90) or (rt 90) turns 90 degrees
Sierpinski's Triangle

(Demo)