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?

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

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

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2, 3.3, true, +, quotient, ...
Scheme Fundamentals

Scheme programs consist of expressions, which can be:

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

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

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

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

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

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

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e_1> ... <e_n>), (or <e_1> ... <e_n>)`
A combination that is not a call expression is a special form:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e_1> ... <e_n>), (or <e_1> ... <e_n>)`
- **Binding names**: `(define <name> <expression>)`
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>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```
A combination that is not a call expression is a special form:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e_1> ... <e_n>), (or <e_1> ... <e_n>)`
- **Binding names**: `(define <name> <expression>)`

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

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

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

The name “pi” is bound to 3.14 in the global frame
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 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 combination that is not a call expression is a *special form*:

- **If expression:** \((\text{if } \text{<predicate>} \text{<consequent>} \text{<alternative>})\)
- **And and or:** \((\text{and } \text{<e}_1\ldots \text{<e}_n\text{)}, \text{(or } \text{<e}_1\ldots \text{<e}_n\text{)}\)
- **Binding names:** \((\text{define } \text{<name> } \text{<expression>})\)
- **New procedures:** \((\text{define } (\text{<name> } \text{<formal parameters>}) \text{<body>})\)

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

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

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

A procedure is created and bound to the name “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:**  
  \[
  (\text{and} \ <\text{e}_1> \ ... \ <\text{e}_n>), \ (\text{or} \ <\text{e}_1> \ ... \ <\text{e}_n>)
  \]

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

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

> (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”

Demo
Lambda Expressions

Lambda expressions evaluate to anonymous functions.
Lambda Expressions

Lambda expressions evaluate to anonymous functions.

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

Lambda expressions evaluate to anonymous functions.

(\texttt{lambda (<formal-parameters>) <body>})
Lambda Expressions

Lambda expressions evaluate to anonymous functions.

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

Two equivalent expressions:

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

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

Lambda expressions evaluate to anonymous functions.

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

Two equivalent expressions:

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

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

An operator can be a call expression too:
Lambda Expressions

Lambda expressions evaluate to anonymous functions.

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

Two equivalent expressions:

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

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

An operator can be a call expression too:

\[
((\text{lambda} \ (x \ y \ z) \ (\text{+} \ x \ y \ (\text{square} \ z))) \ 1 \ 2 \ 3)
\]
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 \textit{add-x--y--z^2} procedure
Pairs and Lists
Pairs and Lists

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

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

• **cons**: Two-argument procedure that *creates a pair*
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
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
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
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.
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.
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.
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.
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)
  ```
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)
```

Not a well-formed list!
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
```

Not a well-formed list!
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
```

Not a well-formed list!
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!
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!
Symbolic Programming
Symbolic Programming

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

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)

Quotation is used to refer to symbols directly in Lisp.

> (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)

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)
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)

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

Symbols are now values
Symbolic Programming

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

\[
\begin{align*}
> & \text{(define a 1)} \\
> & \text{(define b 2)} \\
> & \text{(list a b)} \\
& (1 \ 2)
\end{align*}
\]

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

Quotation is used to refer to symbols directly in Lisp.

\[
\begin{align*}
> & \text{(list 'a 'b)} \\
& (a \ b) \\
> & \text{(list 'a b)} \\
& (a \ 2)
\end{align*}
\]

Symbols are now values

Quotation can also be applied to combinations to form lists.
Symbols normally refer to values; how do we refer to symbols?

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

Quotation is used to refer to symbols directly in Lisp.

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

Quotation can also be applied to combinations to form lists.

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

No sign of “a” and “b” in the resulting value
Symbols are now values
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)
```

Quotation is used to refer to symbols directly in Lisp.

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

Quotation can also be applied to combinations to form lists.

```lisp
> (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.

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

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

Symbols are now values
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)
Scheme Lists and Quotation
Dots can be used in a quoted list to specify the second element of the final pair.
Dots can be used in a quoted list to specify the second element of the final pair.

> (cdr (cdr '(1 2 . 3)))
Scheme Lists and Quotation

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

> (cdr (cdr '(1 2 . 3)))
3
Scheme Lists and Quotation

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.
Scheme Lists and Quotation

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

\[
> \text{(cdr (cdr '(1 2 . 3)))}
3
\]

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

\[
> '(1 2 . 3)
\]

1 2 3
Scheme Lists and Quotation

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)
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))
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))
```

![Diagram](image-url)
Scheme Lists and Quotation

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))
Scheme Lists and Quotation

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

\[
\texttt{> (cdr (cdr '(1 2 . 3)))} \\
\texttt{3}
\]

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

\[
\texttt{> '(1 2 . 3)} \\
\texttt{(1 2 . 3)}
\]

\[
\texttt{> '(1 2 . (3 4))} \\
\texttt{(1 2 3 4)}
\]
Scheme Lists and Quotation

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

\[
> (\text{cdr} \ (\text{cdr} \ '(1\ 2\ .\ 3)))
\]

3

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

\[
> '(1\ 2\ .\ 3)
(1\ 2\ .\ 3)
\]

\[
> '(1\ 2\ .\ (3\ 4))
(1\ 2\ 3\ 4)
\]

\[
> '(1\ 2\ 3\ .\ \text{nil})
\]
Scheme Lists and Quotation

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

\[ \texttt{> (cdr (cdr '(1 2 . 3)))} \]
\[ 3 \]

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

\[ \texttt{> '(1 2 . 3)} \]
\[ (1 2 . 3) \]

\[ \texttt{> '(1 2 . (3 4))} \]
\[ (1 2 3 4) \]

\[ \texttt{> '(1 2 3 . nil)} \]
\[ (1 2 3 nil) \]
Scheme Lists and Quotation

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

\[
> \text{(cdr (cdr '(1 2 . 3)))}
\]

3

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

\[
> '(1 2 . 3) \\
(1 2 . 3)
\]

\[
> '(1 2 . (3 4)) \\
(1 2 3 4)
\]

\[
> '(1 2 3 . nil) \\
(1 2 3)
\]
Scheme Lists and Quotation

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?
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))))
```

Scheme Lists and Quotation

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

\[
> \text{(cdr (cdr '}(1 2 . 3)\))}
3
\]

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

\[
> '(1 2 . 3) \quad (1 2 . 3)
> '(1 2 . (3 4)) \quad (1 2 3 4)
> '(1 2 3 . nil) \quad (1 2 3)
\]

What is the printed result of evaluating this expression?

\[
> \text{(cdr '}(1 2) . (3 4 . (5)))\)}
(3 4 5)
Coercing a Sorted List to a Binary Search Tree
Coercing a Sorted List to a Binary Search Tree

1 → 2 → 3 → 4 → 5 → 6 → 7 →
Coercing a Sorted List to a Binary Search Tree

Divide length n into 3 parts: \( \left( \frac{n-1}{2} , 1 , \frac{n-1}{2} \right) \)
Coercing a Sorted List to a Binary Search Tree

Divide length $n$ into 3 parts: $[(n-1)/2, 1, (n-1)/2]$

Recursively coerce the left part
Divide length $n$ into 3 parts: $\left[ \frac{n-1}{2}, 1, \frac{n-1}{2} \right]$

Recursively coerce the left part
Coercing a Sorted List to a Binary Search Tree

Divide length $n$ into 3 parts: $[(n-1)/2, 1, (n-1)/2]$.

Recursively coerce the left part.
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
Coercing a Sorted List to a Binary Search Tree

Divide length n into 3 parts: \([ (n-1)/2, 1, (n-1)/2 ]\)

Recursively coerce the left part

The next element is the entry
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
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
Coercing a Sorted List to a Binary Search Tree

Divide length n into 3 parts: [ (n-1)/2 , 1 , (n-1)/2 ]

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
Coercing a Sorted List to a Binary Search Tree

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

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
Coercing a Sorted List to a Binary Search Tree

Divide length n into 3 parts: \([ (n-1)/2, 1, (n-1)/2 \)\]

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
Coercing a Sorted List to a Binary Search Tree

Divide length n into 3 parts: \([ (n-1)/2 , 1 , (n-1)/2 ] \)

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
Coercing a Sorted List to a Binary Search Tree

Divide length \( n \) into 3 parts: \([ (n-1)/2 , 1 , (n-1)/2 ]\)

Recursively coerce the left part

The next element is the entry

Recursively coerce the right part
The Let Special Form
The Let Special Form

(define (entry tree) ...)
(define (left-branch tree) ...)
(define (right-branch tree) ...)
(define (make-tree entry left right) ...)
The Let Special Form

(define (entry tree) ...)
(define (left-branch tree) ...)
(define (right-branch tree) ...)
(define (make-tree entry left right) ...)

1 → 2 → 3 → 4 → 5 → 6 → 7 → ...
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)))))
```
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))
```
(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)
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)))
          ...)))
```
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))))
        ...))
...
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)))  
                  (let ((non-left-elts (cdr left-result)))  
                      (right-size (- n (+ left-size 1))))))))
```
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))))))
```
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)))
```
(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))
        (left-tree (car left-result))
        (non-left-elts (cdr left-result))
        (right-size (- n (+ left-size 1))))
        (let ((this-entry (car non-left-elts)))
          (...))))
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)))))
      ...))
```
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))))
            ...))
  ...))
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))))))
```
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))))))
      ...))
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

\[(\text{begin } \text{<exp}_1\text{> } \text{<exp}_2\text{> } \ldots \text{ <exp}_n\text{>})\]
The Begin Special Form

\((\text{begin} \ <\exp_1> \ <\exp_2> \ ... \ <\exp_n>)\)

Demo