Announcements
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
Announcements

- Homework 7 due Tuesday 11/5 @ 11:59pm.
- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Midterm 2 is graded.
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
• Midterm 2 is graded.
  • (And yes, it was very challenging.)
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
• Midterm 2 is graded.
  • (And yes, it was very challenging.)
  • Mean: 30
Announcements

• Homework 7 due Tuesday 11/5 @ 11:59pm.
• Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
• Midterm 2 is graded.
  • (And yes, it was very challenging.)
  • Mean: 30
  • Solutions will be posted and exams distributed soon.
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?

"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, DeNero'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, DeNero'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, DeNero's favorite sci-fi author

"God's programming language."
– Brian Harvey, Berkeley CS instructor extraordinaire
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 include an operator and 0 or more operands in parentheses.
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 include an operator and 0 or more operands in parentheses.

```
> (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 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 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 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 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 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 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 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 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 include an operator and 0 or more operands in parentheses.

> (quotient 10 2)
> (quotient (+ 8 7) 5)
> (+ (* 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 include an operator and 0 or more operands in parentheses.

> (quotient 10 2)       "quotient" names Scheme’s built-in integer division procedure (i.e., function)
  5
> (quotient (+ 8 7) 5)  Combinations can span multiple lines (spacing doesn’t matter)
  3
> (+ (* 3 (+ (* 2 4) (+ 3 5)))
  (+ (- 10 7) 6))
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 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 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 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 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 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)
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.
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**: \((\text{and} <e_1> ... <e_n>), (\text{or} <e_1> ... <e_n>)\)

**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**: \( (\text{and} \ <\text{e}_1> \ ... \ <\text{e}_n> ), \ (\text{or} \ <\text{e}_1> \ ... \ <\text{e}_n> ) \)
- **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>)`

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

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

> (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**:
  \((\text{and } \text{<e}_1 \text{ ... } \text{<e}_n \text{)}, (\text{or } \text{<e}_1 \text{ ... } \text{<e}_n \text{)})\)

• **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

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

- **If** expression:  `(if <predicate> <consequent> <alternative>)`
- **And and or**:
  - And: `(and <e₁> ... <eₙ>)`
  - Or: `(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.
Special Forms

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

• **If expression:**  
  \[(\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\]

• **And and or:**  
  \[(\text{and } \langle e_1 \rangle \ldots \langle e_n \rangle), \ (\text{or } \langle e_1 \rangle \ldots \langle e_n \rangle)\]

• **Binding symbols:**  
  \[(\text{define } \langle\text{symbol}\rangle \ \langle\text{expression}\rangle)\]

• **New procedures:**  
  \[(\text{define } (\langle\text{symbol}\rangle \ \langle\text{formal parameters}\rangle) \ \langle\text{body}\rangle)\]

```
> (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>)

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

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

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

(Demo)
Counting Trees
The structure of a sentence can be described by a tree. Each sub-tree is a *constituent*.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

a long noun phrase
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

a long noun phrase
The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- a long noun phrase
  - a two word modifier
The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

Example: Counting Binary Trees

a long noun phrase

a two word modifier
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

```
a long noun phrase
  a two word modifier

some trees are balanced
```
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- a long noun phrase
  - a two word modifier
- some trees are balanced
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

```
   +--- a long noun phrase (+--- a two word modifier)
       |                 +--- some trees are balanced
       |                   +--- the other trees lean
```
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- a long noun phrase
  - a two word modifier
- some trees are balanced
- the other trees lean
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- a long noun phrase
- a two word modifier
- some trees are balanced
- the other trees lean
- so many trees exist
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- a long noun phrase
  - a two word modifier
- some trees are balanced
- the other trees lean
- so many trees exist
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

```
W X Y Z
```

a long noun phrase

a two word modifier

some trees are balanced

the other trees lean

so many trees exist
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

```
    W     X     Y     Z
  /       /       /       /
 a long noun phrase some trees are balanced so many trees exist
  /       /       /       /
 a two word modifier the other trees lean
```
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a *constituent*.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a *constituent*.

The number of trees over $n$ leaves with $k$ leaves in the left and $n-k$ in the right is:
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a constituent.

- A long noun phrase
- A two word modifier
- Some trees are balanced
- The other trees lean
- So many trees exist
- The number of trees over \( n \) leaves with \( k \) leaves in the left and \( n-k \) in the right is:
  (The number of trees with \( k \) leaves) * (The number of trees with \( n-k \) leaves)
Example: Counting Binary Trees

The structure of a sentence can be described by a tree. Each sub-tree is a *constituent*.

The number of trees over \( n \) leaves with \( k \) leaves in the left and \( n-k \) in the right is:

\[
\text{(The number of trees with } k \text{ leaves) } \times \text{(The number of trees with } n-k \text{ leaves)}
\]

(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.

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

Lambda expressions evaluate to anonymous procedures.

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

Two equivalent expressions:

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

An operator can be a call expression too:
Lambda Expressions

Lambda expressions evaluate to anonymous procedures.

\[
\lambda \text{ (}\begin{array}{c}
\text{formal-parameters} \\
\end{array}\text{)} \begin{array}{c}
\text{body} \\
\end{array}\n\]

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.

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

Two equivalent expressions:

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

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) list in Scheme 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) list in Scheme 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) list in Scheme 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 any value for the second element of the last pair; maybe not a 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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```scheme
> (define x (cons 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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```scheme
> (define x (cons 1 2))
> x
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

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


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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```scheme
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
> (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```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)
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```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)
```
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) list in Scheme 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 any value for the second element of the last pair; maybe not a list!

```
> (define x (cons 1 2))
> x
(1 . 2)  # Not a well-formed list!
> (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)
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.
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)

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)
(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?

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

Quotation is used to refer to symbols directly in Lisp.

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

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?

> (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.

> (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)

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)

Symbols are now values

Quotation can also be applied to combinations to form lists.

> (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))
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
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.

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

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

\[
> '(1 2 . 3)
\]
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
```
Dots can be used in a quoted list to specify the second element of the final pair.

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

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

```lisp
> '(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.

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

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

```lisp
> '(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)
> ')(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))
```

16
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.

> (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)
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)
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)
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.

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

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

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

\[
(1 \ 2 \ 3 \ 4)
\]

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

\[
(1 \ 2 \ 3)
\]

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)
(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?
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?

> (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.

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

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

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

What is the printed result of evaluating this expression?

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

(Demo)