Announcements

- HW9 due Wednesday

- Ants extra credit due Wednesday
  - See Piazza for submission instructions

- Hog revisions out, due next Monday
Scheme Is a Dialect of Lisp
Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of Smalltalk and OOP
Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of Smalltalk and OOP

“The only computer language that is beautiful.”
-Neal Stephenson, sci-fi author
Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of Smalltalk and OOP

“The only computer language that is beautiful.”
-Neal Stephenson, sci-fi author

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

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of Smalltalk and OOP

“The only computer language that is beautiful.”
-Neal Stephenson, sci-fi author

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

http://imgs.xkcd.com/comics/lisp_cycles.png
Scheme programs consist of expressions, which can be:
Scheme programs consist of expressions, which can be:

- Primitive expressions: \(2, 3.3, \text{true}, +, \text{quotient}, \ldots\)
Scheme programs consist of expressions, which can be:

- Primitive expressions: \(2, 3.3, \text{true}, +, \text{quotient}\), ...
- Combinations: \((\text{quotient} \ 10 \ 2), (\text{not} \ \text{true})\), ...
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 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)
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
```
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)
```
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

\[
\begin{align*}
\text{> (quotient 10 2)} \quad & \quad 5 \\
\text{> (quotient (+ 8 7) 5)} \quad & \quad 3
\end{align*}
\]
Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: \( 2, 3.3, \text{true}, +, \text{quotient}, \ldots \)
- Combinations: \((\text{quotient} 10 2), (\text{not} \ \text{true}), \ldots\)

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

\[
\begin{align*}
\text{> (quotient 10 2)} & \quad 5 \\
\text{> (quotient (+ 8 7) 5)} & \quad 3 \\
\text{> (+ (* 3
\quad (+ (* 2 4)
\quad (+ 3 5)))
\quad (+ (- 10 7)
\quad 6))}
\end{align*}
\]
Scheme Fundamentals

Scheme programs consist of expressions, which can be:
• Primitive expressions: \(2, 3.3, \text{true}, +, \text{quotient}, \ldots\)
• Combinations: \((\text{quotient} \ 10 \ 2), (\text{not} \ \text{true}), \ldots\)

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

\[
\begin{align*}
> & (\text{quotient} \ 10 \ 2) \\
& 5 \\
> & (\text{quotient} \ (+ \ 8 \ 7) \ 5) \\
& 3 \\
> & (+ \ (* \ 3 \\
& \quad (+ \ (* \ 2 \ 4) \\
& \quad (+ \ 3 \ 5))) \\
& (+ \ (- \ 10 \ 7) \\
& \quad 6)) \\
& 57
\end{align*}
\]
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:

\[ \text{> (quotient 10 2)} \]
\[ 5 \]
\[ \text{> (quotient (+ 8 7) 5)} \]
\[ 3 \]
\[ \text{> (+ (* 3}
  \[ \text{\quad (+ (* 2 4)}
  \[ \text{\quad (+ 3 5))})
  \[ \text{(+ (- 10 7)}
  \[ \text{\quad 6))} \]
\]

57
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 programs consist of expressions, which can be:
• Primitive expressions: \(2, \ 3.3, \ true, +, \ quotient, \ldots\)
• Combinations: \((quotient \ 10 \ 2), \ (not \ true), \ldots\)

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

\[
> \ (quotient \ 10 \ 2) \\
5 \\
> \ (quotient \ (+ \ 8 \ 7) \ 5) \\
3 \\
> \ (+ \ (* \ 3 \ \\
\quad \ (+ \ (* \ 2 \ 4) \\
\quad \ (+ \ 3 \ 5))) \\
\quad \ (+ \ (- \ 10 \ 7) \ 6))
\]

“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)
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)
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:  
  
  \[
  (\text{if } <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>)
  \]
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:** $(\text{and } <e_1> ... <e_n>), (\text{or } <e_1> ... <e_n>)$
A combination that is not a call expression is a *special form*:

- **If expression:** \((if \ <\text{predicate}\> \ <\text{consequent}\> \ <\text{alternative}\>)\)
- **And and or:** \((\text{and} \ \ <\text{e}_1\> \ ... \ <\text{e}_n\>), (\text{or} \ \ <\text{e}_1\> \ ... \ <\text{e}_n\>)\)
- **Binding names:** \((\text{define} \ \ <\text{name}\> \ <\text{expression}\>)\)
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**: \((\text{define } (<\text{name}> <\text{formal parameters}>) <\text{body}>))\)
A combination that is not a call expression is a *special form*:

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

- **And and or:**  
  
  \[
  \text{(and <e_1> ... <e_n>), (or <e_1> ... <e_n>)}
  \]

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

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

> (define pi 3.14)
Special Forms

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

- **If expression:**  
  \[(if \ <predicate> \ <consequent> \ <alternative>)\]

- **And and or:**  
  \[(and \ <e_1> \ldots \ <e_n>), (or \ <e_1> \ldots \ <e_n>)\]

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

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

> (define pi 3.14)  

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>)`

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

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

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
```

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:** \((\text{if } \langle \text{predicate} \rangle \ \langle \text{consequent} \rangle \ \langle \text{alternative} \rangle)\)
- **And and or:** \((\text{and } \langle e_1 \rangle \ ... \ \langle e_n \rangle), (\text{or } \langle e_1 \rangle \ ... \ \langle e_n \rangle)\)
- **Binding names:** \((\text{define } \langle \text{name} \rangle \ \langle \text{expression} \rangle)\)
- **New procedures:** \((\text{define } (\langle \text{name} \rangle \ \langle \text{formal parameters} \rangle) \ \langle \text{body} \rangle)\)

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

> (define (abs x)
  (if (< x 0)
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:** \((\text{if } <\text{predicate}> <\text{consequent}> <\text{alternative}>)\)
- **And** and **or:** \((\text{and } <e_1> \ldots <e_n>), (\text{or } <e_1> \ldots <e_n>)\)
- **Binding 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)
\quad (if (< x 0)
\quad (\text{-} x)
\quad (- x))

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\ldots \text{<e}_n\ldots \text{<e}_n\ldots )\), (\text{or } \text{<e}_1\ldots \text{<e}_n\ldots \text{<e}_n\ldots \text{<e}_n\ldots )\]

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

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

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

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:**
  \[(and \ <e_1> \ ... \ <e_n>), \ (or \ <e_1> \ ... \ <e_n>)\]

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

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

```scheme
top-level
> (define pi 3.14)
> (* pi 2)
6.28

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

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

- **And** and **or**:
  \[(and \ <e_1> \ ... \ <e_n>), \ (or \ <e_1> \ ... \ <e_n>)\]

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

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

\[
> (\text{define} \ \text{pi} \ 3.14)\\
> (* \ \text{pi} \ 2)\\
6.28
\]

\[
> (\text{define} \ (\text{abs} \ x)\\
\hspace{1em} (if \ (<\ x \ 0)\\
\hspace{2em} (- \ x)\\
\hspace{3em} x))\\
> (\text{abs} \ -3)
\]

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

A procedure is created and bound to the name “abs”
A combination that is not a call expression is a *special form*:

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

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

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

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

```lisp
> (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.”
Lambda Expressions

Lambda expressions evaluate to anonymous procedures
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

(lambda (<formal-parameters>) <body>)
Lambda expressions evaluate to anonymous procedures

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

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

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

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

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

An operator can be a combination too:
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

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

An operator can be a combination too:

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

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

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

An operator can be a combination too:

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

Evaluates to the \textit{add-x-&-y-&-z^2} procedure
We can implement pairs functionally:
We can implement pairs functionally:

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

We can implement pairs functionally:

\[
\begin{align*}
&(\text{define} \ (\text{pair} \ x \ y) \ (\lambda m \ (\text{if} \ (= m 0) \ x \ y))) \\
&(\text{define} \ (\text{first} \ p) \ (p \ 0)) \\
&(\text{define} \ (\text{second} \ p) \ (p \ 1))
\end{align*}
\]

Scheme also has built-in pairs that use weird names:
We can implement pairs functionally:

\[
\begin{align*}
(\text{define} \ (\text{pair} \ x \ y) \ (\lambda (m) \ (\text{if} \ (= m \ 0) \ x \ y))) \\
(\text{define} \ (\text{first} \ p) \ (p \ 0)) \\
(\text{define} \ (\text{second} \ p) \ (p \ 1))
\end{align*}
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
Pairs

We can implement pairs functionally:

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

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
Pairs

We can implement pairs functionally:

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

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair
We can implement pairs functionally:

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

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

A pair is represented by a dot between the elements, all in parens
We can implement pairs functionally:

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

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

```
> (cons 1 2)
```
Pairs

We can implement pairs functionally:

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

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

```
> (cons 1 2)
(1 . 2)
```
Pairs

We can implement pairs functionally:

\[
\begin{align*}
& \text{(define (pair x y) \lambda(m) (if (= m 0) x y))} \\
& \text{(define (first p) (p 0))} \\
& \text{(define (second p) (p 1))}
\end{align*}
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

\[
\begin{align*}
& \text{> (cons 1 2)} \\
& (1 . 2) \\
& \text{> (car (cons 1 2))}
\end{align*}
\]
We can implement pairs functionally:

\[
\begin{align*}
\text{(define (pair x y) (lambda (m) (if (= m 0) x y))} \\
\text{(define (first p) (p 0))} \\
\text{(define (second p) (p 1))}
\end{align*}
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

\[
\begin{align*}
\text{> (cons 1 2)} \\
\text{(1 . 2)} \\
\text{> (car (cons 1 2))} \\
\text{1}
\end{align*}
\]
We can implement pairs functionally:

\[
\begin{align*}
\text{(define (pair x y) (lambda (m) (if (= m 0) x y)))} \\
\text{(define (first p) (p 0))} \\
\text{(define (second p) (p 1))}
\end{align*}
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

\[
\begin{align*}
\text{> (cons 1 2)} \\
\text{(1 . 2)} \\
\text{> (car (cons 1 2))} \\
\text{1} \\
\text{> (cdr (cons 1 2))}
\end{align*}
\]
Pairs

We can implement pairs functionally:

\[
(\text{define } (\text{pair } x \ y) (\lambda \ m \ (\text{if } (= \ m \ 0) \ x \ y)))
\]
\[
(\text{define } (\text{first } p) \ (p \ 0))
\]
\[
(\text{define } (\text{second } p) \ (p \ 1))
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

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

\[
> \ (\text{cons } 1 \ 2)
\]
\[
(1 . \ 2)
\]
\[
> \ (\text{car} \ (\text{cons } 1 \ 2))
\]
\[
1
\]
\[
> \ (\text{cdr} \ (\text{cons } 1 \ 2))
\]
\[
2
\]
Recursive Lists
A recursive list can be represented as a pair in which the second element is a recursive list or the empty list
Recursive Lists

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

Scheme lists are recursive lists:
Recursive Lists

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

Scheme lists are recursive lists:

- \texttt{nil} is the empty list
Recursive Lists

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

Scheme lists are recursive lists:

- \texttt{nil} is the empty list
- A non-empty Scheme list is a pair in which the second element is \texttt{nil} or a Scheme list.
Recursive Lists

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

Scheme lists are recursive lists:
• **nil** is the empty list
• A non-empty Scheme list is a pair in which the second element is **nil** or a Scheme list

Scheme lists are written as space-separated combinations
A recursive list can be represented as a pair in which the second element is a recursive list or the empty list.

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
```
Recursive Lists

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

Scheme lists are recursive lists:

- \texttt{nil} is the empty list
- A non-empty Scheme list is a pair in which the second element is \texttt{nil} or a Scheme list

Scheme lists are written as space-separated combinations

\begin{verbatim}
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
\end{verbatim}
Recursive Lists

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

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations

```scheme
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
(1 2 3 4)
```
Recursive Lists

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

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations

> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
> x
> (1 2 3 4)
> (cdr x)
Recursive Lists

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

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
```
Recursive Lists

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

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
```
Recursive Lists

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

Scheme lists are recursive lists:

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

Scheme lists are written as space-separated combinations:

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 4)
```
Recursive Lists

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

Scheme lists are recursive lists:

• **`nil`** is the empty list

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

Scheme lists are written as space-separated combinations

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

Not a well-formed list!
Symbolic Programming
Symbolic Programming

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

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

> (define a 1)
Symbolic Programming

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

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

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

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

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

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

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

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

Quotation prevents something from being evaluated by Lisp

No sign of “a” and “b” in the resulting value
Symbols are normally evaluated to produce values; how do we refer to symbols?

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

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
Symbolic Programming

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

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

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)

No sign of “a” and “b” in the resulting value
Symbols are normally evaluated to produce values; how do we refer to symbols?

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

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
```
Symbols are normally evaluated to produce values; how do we refer to symbols?

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

Quotation prevents something from being evaluated by Lisp

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

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

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

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

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
Symbols are normally evaluated to produce 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 prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)

Symbols are now values

> (list 'a b)
(a 2)

Quotation can also be applied to combinations to form lists
Symbolic Programming

Symbols are normally evaluated to produce 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 prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)

Symbols are now values

> (list 'a b)
(a 2)

Quotation can also be applied to combinations to form lists

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

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

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

Quotation prevents something from being evaluated by 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
```
Symbols are normally evaluated to produce 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 prevents something from being evaluated by 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
> (cdr '(a b c))
```
Symbols are normally evaluated to produce values; how do we refer to symbols?

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

Quotation prevents something from being evaluated by Lisp

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

Quotation can also be applied to combinations to form lists

\[
\begin{align*}
\texttt{> (car '(a b c))} \\
\texttt{a} \\
\texttt{> (cdr '(a b c))} \\
\texttt{(b c)}
\end{align*}
\]
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.
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)))
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
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 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

1 2 3
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
Scheme Lists and Quotation

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

\[
\begin{align*}
> (\text{cdr} (\text{cdr} '(1 2 . 3))) \\
3
\end{align*}
\]

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

\[
\begin{align*}
> '(1 2 . 3) \\
(1 2 . 3)
\end{align*}
\]
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)

\[ '(1 2 . (3 4)) \]
Dots can be used in a quoted list to specify the second element of the final pair

\[ \text{\textit{\textgreater{} (cd\textit{r} (cd\textit{r} ' (1 2 . 3)))}} \]

3

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

\[ \text{\textit{\textgreater{} ' (1 2 . 3) \quad 1 \quad \rightarrow \quad 2 \quad 3}} \]
\[ \text{(1 2 . 3)} \]
\[ \text{\textgreater{} ' (1 2 . (3 4)) \quad 1 \quad \rightarrow \quad 2} \]
Dots can be used in a quoted list to specify the second element of the final pair

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

\[
3
\]

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

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

\[
(1 2 . 3)
\]

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

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \textit{nil}
\]
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)
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

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

(1 2 . 3)

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

(1 2 3 4)

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

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

\[ 3 \]

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

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

\( (1\ 2\ .\ 3) \)

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

\( (1\ 2\ 3\ 4) \)

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

\( (1\ 2\ 3\ .\ \text{nil}) \)
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)
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)

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

(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 4))))
3

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

> '((1 2) . 3) (1 2 . 3)
1 2 3

> '((1 2) . (3 4)) (1 2 3 4)
1 2 3 4

> '((1 2) . nil) (1 2 3)
1 2 3 nil

What is the printed result of evaluating this expression?

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