Lists, higher order procedures, and symbols
6.037 - Structure and Interpretation of Computer Programs

Alex Vandiver (alexmv)

Massachusetts Institute of Technology

Lecture 2
Project 0 was due today

Reminder: Project 1 due at 7pm on Tuesday

Mail to 6.037-psets@mit.edu

If you didn’t sign up on Tuesday, let us know
Types

(+ 5 10) =>

Expects type <number> as 1st argument, given: "hi"; other arguments were: 15

Addition is not defined for strings

Only works for things of type number

Scheme checks types for simple built-in functions
(+ 5 10) => 15
(\(+\ 5\ 10\)) \Rightarrow 15

(+ "hi" 15) =>
Types

```
(+ 5 10) => 15

(+ "hi" 15) =>
   +: expects type <number> as 1st argument, given: "hi"; other arguments were: 15
```
Types

(+ 5 10) => 15

(+ "hi" 15) =>
  +: expects type <number> as 1st argument,
  given: "hi"; other arguments were: 15

- Addition is not defined for strings
Types

(+ 5 10) => 15

(+ "hi" 15) =>
  +: expects type <number> as 1st argument, given: "hi"; other arguments were: 15

- Addition is not defined for strings
- Only works for things of type number
- Scheme checks types for simple built-in functions
Simple data types

Everything has a type:
  - Number
Simple data types

Everything has a type:
- Number
- String
Simple data types

Everything has a type:
- Number
- String
- Boolean
Everything has a type:
- Number
- String
- Boolean
- Procedures?
Everything has a type:
- Number
- String
- Boolean
- Procedures?
  - Is the type of not the same type as + ?
What about procedures?

- Procedures have their own types, based on arguments and return value.
- \( \text{number} \mapsto \text{number} \) means “takes one number, returns a number.”
Type examples

(+ 5 10)     => 15

(+ "hi" 15)  =>

+: expects type <number> as 1st argument,
given: "hi"; other arguments were: 15

- What is the type of +?
Type examples

(+ 5 10) => 15

(+ "hi" 15) =>
  +: expects type <number> as 1st argument, given: "hi"; other arguments were: 15

- What is the type of +?
- number, number ↦ number
Type examples

(+ 5 10) => 15

(+ "hi" 15) =>
   +: expects type <number> as 1st argument,
   given: "hi"; other arguments were: 15

- What is the type of +?
- **number, number \(\mapsto\) number**
  
  (mostly)
Expression: ...is of type:
15
"hi"
square
>

Type of a procedure is a contract
If the operands have the specified types, the procedure will result
in a value of the specified type
Otherwise, its behavior is undefined
Expression: 15  "hi"  square >  
...is of type:  
number
Type examples

Expression:  
15
"hi"
square
>

...is of type: 
number
string
Expression: 15 "hi" square >

...is of type: number string number \rightarrow number
Type examples

Expression:  
15  
"hi"  
square  
>  

... is of type:  
number  
string  
number $\mapsto$ number  
number, number $\mapsto$ boolean
Type examples

Expression:  
15  ...is of type:  
number
"hi"  string
square  number ↦→ number
>  number, number ↦→ boolean

- Type of a procedure is a **contract**
- If the operands have the specified types, the procedure will result in a value of the specified type
- Otherwise, its behavior is undefined
More complicated examples

\[(\text{lambda } (a \ b \ c) \ (\text{if } (> a 0) \ (+ b c) \ (- b c)))\]

, ,    \rightarrow
More complicated examples

(l lambda (a b c)
  (if (> a 0) (+ b c) (- b c)))

, , →
More complicated examples

\[
\text{(lambda (a b c)}
\begin{array}{c}
\text{(if (> a 0) (+ b c) (- b c))}) \\
, , , \mapsto
\end{array}
\]
More complicated examples

\[(\lambda (a \ b \ c) \ (\text{if} \ (> \ a \ 0) \ (+ \ b \ c) \ (\ - \ b \ c)))\]

\[\text{number, } \to \]
More complicated examples

$(\lambda (a\ b\ c)\ (\text{if}\ (>\ a\ 0)\ (+\ b\ c)\ (\text{-}\ b\ c)))$

$\text{number, number, number} \mapsto$

$(\lambda (p)\ (\text{if}\ p\ \text{"hi"}\ \text{"bye"}))$

$\text{boolean, } \mapsto \text{string}$

$(\lambda (x)\ (*\ 3.14\ (*\ 2\ 5)))$

$\text{any, } \mapsto \text{number}$
More complicated examples

\[
\text{lambda (a b c)}
\quad \text{(if (> a 0) (+ b c) (- b c))}
\]

\[
\text{number, number, } \rightarrow
\]
More complicated examples

\[
\text{(lambda (a b c) } \\
\text{ (if (> a 0) (+ b c) (- b c)))}
\]

\text{number, number, number } \mapsto \text{number}
More complicated examples

\[
\text{(lambda (a b c)}
\text{(if (> a 0) (+ b c) (- b c)))}
\]

\text{number, number, number} \mapsto \text{number}
More complicated examples

\[(\lambda (a \ b \ c) \quad (\text{if} \ (> \ a \ 0) \ (+ \ b \ c) \ (- \ b \ c)))\]

\text{number, number, number} \mapsto
More complicated examples

\[
(lambda (a b c)
   (if (> a 0) (+ b c) (- b c)))
\]

number, number, number $\mapsto$ number
More complicated examples

\[\text{(lambda (a b c)} \]
\[\text{  (if (> a 0) (+ b c) (- b c)) \} \]

**number, number, number \(\mapsto\) number**

\[\text{(lambda (p)} \]
\[\text{  (if p "hi" "bye") \} \]

\(\mapsto\)
More complicated examples

\[
(lambda \ (a \ b \ c) \\
  \ (if \ (> \ a \ 0) \ (+ \ b \ c) \ (- \ b \ c)))
\]

\textbf{number, number, number} \mapsto \textbf{number}

\[
(lambda \ (p) \\
  \ (if \ p \ "\text{hi}" \ "\text{bye}"))
\]

\mapsto
More complicated examples

\[(\texttt{lambda } (a \ b \ c) \n\quad (\texttt{if } (> a 0) (+ b c) (- b c)))\]

\textbf{number, number, number} $\mapsto$ \textbf{number}

\[(\texttt{lambda } (p) \n\quad (\texttt{if } p \ "hi" \ "bye")\]

\textbf{boolean} $\mapsto$
More complicated examples

\[
\text{(lambda } (a\ b\ c) \\
  \text{(if}\ (>\ a\ 0)\ (+\ b\ c)\ (-\ b\ c)))
\]

\text{number, number, number} \rightarrow \text{number}

\[
\text{(lambda } (p) \\
  \text{(if}\ p\ \text{"hi"}\ \text{"bye"})
\]

\text{boolean} \rightarrow \text{string}
More complicated examples

\[
(lambda \ (a \ b \ c) \\
  (if (> a 0) (+ b c) (- b c)))
\]

**number, number, number \(\mapsto\) number**

\[
(lambda \ (p) \\
  (if p "hi" "bye"))
\]

**boolean \(\mapsto\) string**

\[
(lambda \ (x) \\
  (* 3.14 (* 2 5)))
\]

\(\mapsto\)
More complicated examples

\[(\text{lambda } (a \ b \ c) \ (\text{if } (\text{> } a \ 0) \ (+ \ b \ c) \ (- \ b \ c)))\]

**number, number, number \mapsto number**

\[(\text{lambda } (p) \ (\text{if } p \ "\text{hi}" \ "\text{bye}"))\]

**boolean \mapsto string**

\[(\text{lambda } (x) \ (* \ 3.14 \ (* \ 2 \ 5)))\]

**any \mapsto**
More complicated examples

\[
\text{(lambda (a b c)}
\quad \text{(if (> a 0) (+ b c) (- b c)))}
\]

\text{number, number, number} \mapsto \text{number}

\[
\text{(lambda (p)}
\quad \text{(if p "hi" "bye")})
\]

\text{boolean} \mapsto \text{string}

\[
\text{(lambda (x)}
\quad \text{(\ast 3.14 (\ast 2 5)\)\) }}
\]

\text{any} \mapsto \text{number}
Patterns across procedures

Procedural abstraction is finding patterns, and making procedures of them:

- \((\times \; 17 \; 17)\)
- \((\times \; 42 \; 42)\)
- \((\times \; x \; x)\)
- \(\ldots\)
Procedural abstraction is finding patterns, and making procedures of them:

- \((\ast 17 \ 17)\)
- \((\ast 42 \ 42)\)
- \((\ast x \ x)\)
- \(\ldots\)
- \((\text{lambda} \ (x) \ (\ast x \ x))\)
• $1 + 2 + \ldots + 100$
• $1 + 4 + 9 + \ldots + 100^2$
• $1 + \frac{1}{3^2} + \frac{1}{5^2} + \ldots + \frac{1}{99^2} \approx \frac{\pi^2}{8}$
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a) (sum-squares (+ 1 a) b))))
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a) (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b) 0
      (+ (/ 1 (square a))
         (pi-sum (+ 2 a) b))))
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a) (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b) 0
      (+ (/ 1 (square a))
          (pi-sum (+ 2 a) b))))
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a) (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b) 0
      (+ (/ 1 (square a))
          (pi-sum (+ 2 a) b))))
(define (sum-integers a b)
  (if (> a b) 0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a) (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b) 0
      (+ (/ 1 (square a))
          (pi-sum (+ 2 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

What is the type of this procedure?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

What is the type of this procedure?

- What type is the output?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

What is the type of this procedure?

→ number

What type is the output?

What type is the output?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

What is the type of this procedure?

What type is the output?

How many arguments does it have?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

What is the type of this procedure?

, , , , \(\mapsto\) \text{number}

- What type is the output?
- How many arguments does it have?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

What is the type of this procedure?

, , , , → number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
        (sum term (next a) next b))))

What is the type of this procedure?

( \rightarrow \text{number} ), \text{number}, \text{number}, \rightarrow \text{number}

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

What is the type of this procedure?

(number → number), , , → number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
Complex types

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

What is the type of this procedure?

(number \rightarrow number) , number , \rightarrow number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b)))
)

What is the type of this procedure?

(number → number) , number , (             ) ,                   → number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

What is the type of this procedure?

(number -> number) , number , (number -> number) ,           → number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

What is the type of this procedure?

(number ⇒ number), number, (number ⇒ number), number ⇒ number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?
(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
       (sum term (next a) next b))))

What is the type of this procedure?

(number → number), number, (number → number), number → number

- What type is the output?
- How many arguments does it have?
- What is the type of each argument?

Higher-order procedures take a **procedure** as an argument, or return one as a value
Higher-order procedures

\[ \sum_{k=a}^{b} k \]

(define (sum-integers a b)
  (if (> a b) 0
      (+ a
         (sum-integers (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

(define (new-sum-integers a b)
  (sum
   (lambda (x) x)
   a
   (lambda (x) (+ x 1))
   b))
Higher-order procedures

\[ \sum_{k=a}^{b} k \]

(define (sum-integers a b)
  (if (> a b) 0
      (+ a
          (sum-integers (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))
Higher-order procedures

\[ \sum_{k=a}^{b} k \]

(define (sum-integers a b)
  (if (> a b) 0
      (+ a
         (sum-integers (+ 1 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
         (sum term (next a) next b)))))

(define (new-sum-integers a b)
  (sum
   a
   b))
Higher-order procedures

$$\sum_{k=a}^{b} k$$

(define (sum-integers a b)
  (if (> a b) 0
    (+ a
      (sum-integers (+ 1 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b)))))

(define (new-sum-integers a b)
  (sum
    a

    b))
Higher-order procedures

\[ \sum_{k=a}^{b} k \]

(define (sum-integers a b)
  (if (> a b) 0
      (+ a
         (sum-integers (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

(define (new-sum-integers a b)
  (sum (lambda (x) x) a
       b))

Alex Vandiver (MIT)  Lists, higher order procedures, and symbols  Lecture 2  13 / 65
Higher-order procedures

\[ \sum_{k=a}^{b} k \]

(define (sum-integers a b)
  (if (> a b) 0
      (+ a
         (sum-integers (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
         (sum term (next a) next b))))

(define (new-sum-integers a b)
  (sum (lambda (x) x)
       a
       b))
Higher-order procedures

\[
\sum_{k=a}^{b} k
\]

(define (sum-integers a b)
  (if (> a b) 0
    (+ a
      (sum-integers (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b))))

(define (new-sum-integers a b)
  (sum (lambda (x) x)
    a
    (lambda (x) (+ x 1))
    b))

Alex Vandiver (MIT)  Lists, higher order procedures, and symbols  Lecture 2  13 / 65
Higher-order procedures

$$\sum_{k=a}^{b} k^2$$

(define (sum-squares a b)
  (if (> a b) 0
    (+ (square a)
      (sum-squares (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b))))

(define (new-sum-squares a b)
  (sum square a
    (lambda (x) (+ x 1)) b)))
\[ \sum_{k=a}^{b} k^2 \]

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a)
          (sum-squares (+ 1 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))
Higher-order procedures

\[ \sum_{k=a}^{b} k^2 \]

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a)
         (sum-squares (+ 1 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
         (sum term (next a) next b)))))

(define (new-sum-squares a b)
  (sum
   a
   b))
Higher-order procedures

\[ \sum_{k=a}^{b} k^2 \]

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a)
          (sum-squares (+ 1 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b)))))

(define (new-sum-squares a b)
  (sum
   a
   b))
Higher-order procedures

\[ \sum_{k=a}^{b} k^2 \]

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a)
          (sum-squares (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

(define (new-sum-squares a b)
  (sum square
    a
    b))
Higher-order procedures

$$\sum_{k=a}^{b} k^2$$

(define (sum-squares a b)
  (if (> a b) 0
      (+ (square a)
          (sum-squares (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
      (+ (term a)
          (sum term (next a) next b))))

(define (new-sum-squares a b)
  (sum square
    a

    b))
Higher-order procedures

\[ \sum_{k=a}^{b} k^2 \]

(define (sum-squares a b)
  (if (> a b) 0
    (+ (square a)
        (sum-squares (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
        (sum term (next a) next b))))

(define (new-sum-squares a b)
  (sum square
    a
    (lambda (x) (+ x 1))
    b))
Higher-order procedures

\[ \sum_{k=a}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8} \]

(define (pi-sum a b)
  (if (> a b) 0
      (+ (/ 1 (square a))
          (pi-sum (+ 2 a) b)))))
Higher-order procedures

\[ \sum_{k=a}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8} \]

(define (pi-sum a b)
  (if (> a b) 0
    (+ (/ 1 (square a))
      (pi-sum (+ 2 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b))))
Higher-order procedures

\[ \sum_{k=a}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8} \]

(define (pi-sum a b)
  (if (> a b) 0
       (+ (/ 1 (square a))
           (pi-sum (+ 2 a) b))))

(define (sum term a next b)
  (if (> a b) 0
       (+ (term a)
           (sum term (next a) next b))))

(define (new-pi-sum a b)
  (sum
   a
   b))

Alex Vandiver (MIT) Lists, higher order procedures, and symbols Lecture 2 15 / 65
Higher-order procedures

\[
\sum_{k=a \text{ odd}}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8}
\]

(define (pi-sum a b)
  (if (> a b) 0
    (+ (/ 1 (square a))
      (pi-sum (+ 2 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
      (sum term (next a) next b))")

(define (new-pi-sum a b)
  (sum
    a
    b))
Higher-order procedures

\[ \sum_{k=a, k \text{ odd}}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8} \]

(define (pi-sum a b)
  (if (> a b) 0
    (+ (/ 1 (square a))
        (pi-sum (+ 2 a) b))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
        (sum term (next a) next b))))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x)))
       a
       b))
Higher-order procedures

\[ \sum_{k=a}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8} \]

\[
\text{(define (pi-sum a b)}
  \text{ (if (> a b) 0)}
  \text{ (+ (/ 1 (square a))}
  \text{ (pi-sum (+ 2 a) b)))}
\]

\[
\text{(define (sum term a next b)}
  \text{ (if (> a b) 0)}
  \text{ (+ (term a)}
  \text{ (sum term (next a) next b)))}
\]

\[
\text{(define (new-pi-sum a b)}
  \text{ (sum (lambda (x) (/ 1 (square x)))}
  \text{ a}
  \text{ b))}
\]
\[
\sum_{k=a}^{b} \frac{1}{k^2} \approx \frac{\pi^2}{8}
\]

(define (pi-sum a b)
  (if (> a b) 0
    (+ (/ 1 (square a))
        (pi-sum (+ 2 a) b)))))

(define (sum term a next b)
  (if (> a b) 0
    (+ (term a)
        (sum term (next a) next b)))))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x)))
        a
        (lambda (x) (+ x 2))
        b)))
Returning procedures

... takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (new-sum-squares a b)
  (sum square a add1 b))

(define (add1 x) (+ x 1))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a add2 b))

(define (add2 x) (+ x 2))
Returning procedures

...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))

Returning procedures

...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))
(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))
(define (add2 x) (+ x 2))
(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a add2 b))
...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)  
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)  
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-pi-sum a b)  
  (sum (lambda (x) (/ 1 (square x))) a  
       (lambda (x) (+ x 2)) b))
(define (add2 x) (+ x 2))
(define (new-pi-sum a b)  
  (sum (lambda (x) (/ 1 (square x))) a add2 b))
Returning procedures

... takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b) 
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b) 
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))

(define (new-pi-sum a b) 
  (sum (lambda (x) (/ 1 (square x))) a 
       (lambda (x) (+ x 2)) b))
(define (add2 x) (+ x 2))
(define (new-pi-sum a b) (sum (lambda (x) (/ 1 (square x))) a add2 b))
Returning procedures

...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))
Returning procedures

...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))
...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))
(define (add2 x) (+ x 2))
...takes a procedure as an argument or returns one as a value

(define (new-sum-integers a b)
  (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (new-sum-squares a b)
  (sum square a (lambda (x) (+ x 1)) b))
(define (add1 x) (+ x 1))
(define (new-sum-squares a b) (sum square a add1 b))

(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a
       (lambda (x) (+ x 2)) b))
(define (add2 x) (+ x 2))
(define (new-pi-sum a b)
  (sum (lambda (x) (/ 1 (square x))) a add2 b))
(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))
Returning procedures

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) ... ))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
(define add3 (incrementby 3))
Returning procedures

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) ... ))

(define add1 (incrementby 1))
Returning procedures

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))
(define incrementby (lambda (n) ... ))
(define add1 (incrementby 1))
(define add2 (incrementby 2))

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) ... ))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
(define add37.5 (incrementby 37.5))
Returning procedures

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) ... ))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
(define add37.5 (incrementby 37.5))

**type of** incrementby:
Returning procedures

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) ... )))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
(define add37.5 (incrementby 37.5))

**type of incrementby:**
**number** ↦ (**number** ↦ **number**)
(define incrementby
  ; type: num -> (num->num)
  (lambda (n)
    (lambda (x) (+ x n))))

(incrementby 2)
((lambda (n) (lambda (x) (+ x n))) 2)
(lambda (x) (+ x 2))
((incrementby 2) 4)
((lambda (x) (+ x 2)) 4)
(+ 4 2)
6
(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n)))))
(define incrementby
   ;; type: num -> (num->num)
   (lambda (n) (lambda (x) (+ x n))))

( incrementby 2 )
Returning procedures

(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n))))

(incrementby 2)
((lambda (n) (lambda (x) (+ x n))) 2)
(+ 4 2)
6
(define incrementby
   ; type: num -> (num->num)
   (lambda (n) (lambda (x) (+ x n)))))

( incrementby 2 )
( (lambda (n) (lambda (x) (+ x n))) 2 )
( (lambda (x) (+ x 2)) )
(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n))))

( incrementby 2 )
( (lambda (n) (lambda (x) (+ x n))) 2 )
  (lambda (x) (+ x 2))

( (incrementby 2) 4)
Returning procedures

(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n)))))

( incrementby 2 )
( (lambda (n) (lambda (x) (+ x n))) 2 )
  (lambda (x) (+ x 2))

( (incrementby 2) 4)
((lambda (x) (+ x 2)) 4)
(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n)))))

(incrementby 2)
((lambda (n) (lambda (x) (+ x n))) 2)
  (lambda (x) (+ x 2))

((incrementby 2) 4)
((lambda (x) (+ x 2)) 4)
  (+ 4 2)
(define incrementby
  ; type: num -> (num->num)
  (lambda (n) (lambda (x) (+ x n)))))

( incrementby 2 )
( (lambda (n) (lambda (x) (+ x n))) 2 )
  (lambda (x) (+ x 2))

( (incrementby 2) 4)
((lambda (x) (+ x 2)) 4)
  (+ 4 2)
  6
Procedural abstraction

(define sqrt (lambda (x) (try 1 x)))

(define try (lambda (guess x)
  (if (good-enough? guess x)
    guess
    (try (improve guess x) x))))

(define good-enough? (lambda (guess x)
  (< (abs (- (square guess) x)) 0.001)))

(define improve (lambda (guess x)
  (average guess (/ x guess))))

(define average (lambda (a b)
  (/ (+ a b) 2)))
(define sqrt (lambda (x) (try 1 x))
(define try (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (try (improve guess x) x)))))

(define good-enough? (lambda (guess x)
                        (< (abs (- (square guess) x)) 0.001))
(define improve (lambda (guess x)
                   (average guess (/ x guess))))
(define average (lambda (a b) (/ (+ a b) 2)))
(define sqrt (lambda (x) (try 1 x)))
(define try (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (try (improve guess x) x))))
(define good-enough? (lambda (guess x)
    (< (abs (- (square guess) x))
        0.001)))

(define sqrt (lambda (x) (try 1 x)))
(define try (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (try (improve guess x) x))))
(define good-enough? (lambda (guess x)
    (< (abs (- (square guess) x))
        0.001)))
(define improve (lambda (guess x)
    (average guess (/ x guess))))
(define sqrt (lambda (x) (try 1 x)))
(define try (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (try (improve guess x) x)))
(define good-enough? (lambda (guess x)
    (< (abs (- (square guess) x)) 0.001))
(define improve (lambda (guess x)
    (average guess (/ x guess)))

(define average (lambda (a b)
    (/ (+ a b) 2)))
(define sqrt (lambda (x)
    (define try (lambda (guess x)
        (if (good-enough? guess x)
            guess
            (try (improve guess x) x))))
    (define good-enough? (lambda (guess x)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve (lambda (guess x)
        (average guess (/ x guess)))
    (try 1 x))

    (define average (lambda (a b)
        (/ (+ a b) 2))))
(define sqrt (lambda (x)
    (define try (lambda (guess)
        (if (good-enough? guess)
            guess
            (try (improve guess)))))
    (define good-enough? (lambda (guess)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve (lambda (guess)
        (average guess (/ x guess)))
    (try 1))

    (define average (lambda (a b)
        (/ (+ a b) 2)))

    (define sqrt 1)

    (define good-enough? lambda (guess)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve lambda (guess)
        (average guess (/ x guess)))
    (define try lambda (guess)
        (if (good-enough? guess)
            guess
            (try (improve guess)))
    (define sqrt lambda (x)
        (try 1))

    (define average lambda (a b)
        (/ (+ a b) 2))
A type is a set of values
Summary of types

- A type is a set of values
- Every value has a type
Summary of types

- A type is a set of values
- Every value has a type
- Procedure types (types which include $\rightarrow$) indicate:
  - Number of arguments required
  - Type of each argument
  - Type of the return value
Summary of types

- A type is a set of values
- Every value has a type
- Procedure types (types which include $\rightarrow$) indicate:
  - Number of arguments required
  - Type of each argument
  - Type of the return value
- They provide a mathematical theory for reasoning efficiently about programs
A type is a set of values

Every value has a type

Procedure types (types which include $\mapsto$) indicate:
- Number of arguments required
- Type of each argument
- Type of the return value

They provide a mathematical theory for reasoning efficiently about programs

Useful for preventing some common types of errors
Summary of types

- A type is a set of values
- Every value has a type
- Procedure types (types which include $\rightarrow$) indicate:
  - Number of arguments required
  - Type of each argument
  - Type of the return value
- They provide a mathematical theory for reasoning efficiently about programs
- Useful for preventing some common types of errors
- Basis for many analysis and optimization algorithms
Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element

Ideally want this "gluing" to have the property of closure: "The result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object."
Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) getting the pieces back out
Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) getting the pieces back out
- Need a contract between “glue” and “unglue”
Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) getting the pieces back out
- Need a contract between “glue” and “unglue”
- Ideally want this “gluing” to have the property of closure:
Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) getting the pieces back out
- Need a contract between “glue” and “unglue”
- Ideally want this “gluing” to have the property of closure: “The result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object.”
(cons <a> <b>) → <p>

Where <a> and <b> are expressions that map to <a-val> and <b-val>

Returns a pair whose car-part is <a-val> and whose cdr-part is <b-val>
(cons <a> <b>) → <p>

Where <a> and <b> are expressions that map to <a-val> and <b-val>
Pairs \((\mathbf{cons \ cells})\)

- \((\mathbf{cons \ \langle a \rangle \ \langle b \rangle}) \rightarrow \langle p \rangle\)
- Where \(\langle a \rangle\) and \(\langle b \rangle\) are expressions that map to \(\langle a-val \rangle\) and \(\langle b-val \rangle\)
- Returns a pair \(\langle p \rangle\) whose car-part is \(\langle a-val \rangle\) and whose cdr-part is \(\langle b-val \rangle\)
Pairs (cons cells)

- \((\text{cons } <a> <b>) \rightarrow <p>\)
- Where \(<a>\) and \(<b>\) are expressions that map to \(<a-val>\) and \(<b-val>\)
- Returns a pair \(<p>\) whose car-part is \(<a-val>\) and whose cdr-part is \(<b-val>\)
- \((\text{car } <p>) \rightarrow <a-val>\)
- \((\text{cdr } <p>) \rightarrow <b-val>\)
Pairs are tasty

(define p1 (cons 4 (+ 3 2)))

(car p1) ; -> 4

(cdr p1) ; -> 5
Pairs are tasty

\[
\text{(define p1 (cons 4 (+ 3 2)))}
\]

\[
\text{(car p1)} \quad ; \quad ->
\]
Pairs are tasty

(define p1 (cons 4 (+ 3 2)))
(car p1) ; -> 4
Pairs are tasty

(define p1 (cons 4 (+ 3 2)))
(car p1) ; -> 4
(cdr p1) ; ->
Pairs are tasty

(define p1 (cons 4 (+ 3 2)))
(car p1) ; -> 4
(cdr p1) ; -> 5
Pairs are a data abstraction

- **Constructor**
  
  \[(\text{cons } A \ B) \mapsto \text{Pair}<A,B>\]
Pairs are a data abstraction

- **Constructor**
  \[(\text{cons } A \ B) \mapsto \text{Pair} <A, B>\]

- **Accessors**
  \[(\text{car } \text{Pair} <A, B>) \mapsto A\]
  \[(\text{cdr } \text{Pair} <A, B>) \mapsto B\]
Pairs are a data abstraction

- **Constructor**
  \[(\text{cons } A \text{ } B) \mapsto \text{Pair}\langle A, B \rangle\]

- **Accessors**
  \[(\text{car } \text{Pair}\langle A, B \rangle) \mapsto A\]
  \[(\text{cdr } \text{Pair}\langle A, B \rangle) \mapsto B\]

- **Contract**
  \[(\text{car } (\text{cons } A \text{ } B)) \mapsto A\]
  \[(\text{cdr } (\text{cons } A \text{ } B)) \mapsto B\]

Alex Vandiver (MIT)

Lists, higher order procedures, and symbols
Lecture 2 25 / 65
Pairs are a data abstraction

- **Constructor**
  \[(\text{cons } A \; B) \rightarrow \text{Pair}<A,B>\]

- **Accessors**
  \[(\text{car } \text{Pair}<A,B>) \rightarrow A\]
  \[(\text{cdr } \text{Pair}<A,B>) \rightarrow B\]

- **Contract**
  \[(\text{car } (\text{cons } A \; B)) \rightarrow A\]
  \[(\text{cdr } (\text{cons } A \; B)) \rightarrow B\]

- **Operations**
  \[(\text{pair? } Q) \text{ returns } \#t \text{ if } Q \text{ evaluates to a pair, } \#f \text{ otherwise}\]
Once we build a pair, we can treat it as if it were a primitive.

Pairs have the property of closure — we can use a pair anywhere we would expect to use a primitive data element:

\[(\text{cons} (\text{cons} 1 2) 3)\]
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))

(define p1 (make-point 2 3))
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
Building data abstractions

(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))

What type is make-point?
Building data abstractions

(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))

What type is make-point?

number, number \rightarrow \text{Point}
(define make-point cons)
(define point-x car)
(define point-y cdr)

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
Building on earlier abstraction

;;; Point abstraction
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))
Building on earlier abstraction

;;; Point abstraction
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
Building on earlier abstraction

;;; Point abstraction
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))

;;; Segment abstraction
(define (make-seg pt1 pt2) (cons pt1 pt2))
(define (start-point seg) (car seg))
(define (end-point seg) (cdr seg))
Building on earlier abstraction

;;; Point abstraction
(define (make-point x y) (cons x y))
(define (point-x point) (car point))
(define (point-y point) (cdr point))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))

;;; Segment abstraction
(define (make-seg pt1 pt2) (cons pt1 pt2))
(define (start-point seg) (car seg))
(define (end-point seg) (cdr seg))
(define s1 (make-seg p1 p2))
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
    (* scale (point-y pt))))
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))

(stretch-point p1 2)  ->  

Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
    (* scale (point-y pt))))

(stretch-point p1 2)  ->  (4 . 6)
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))

(stretch-point p1 2)  ->  (4 . 6)
p1  ->
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))

(stretch-point p1 2)  ->  (4 . 6)
p1  ->  (2 . 3)
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))

What type is stretch-point?
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt))
               (* scale (point-y pt))))

What type is stretch-point?

Point, number \(\mapsto\) Point
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-seg seg scale)
  (make-seg (stretch-point (start-point seg) scale)
            (stretch-point (end-point seg) scale)))

(define (seg-length seg)
  (sqrt (+ (square
            (- (point-x (start-point seg))
              (point-x (end-point seg))))
       (square
        (- (point-y (start-point seg))
          (point-y (end-point seg)))))))
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-seg seg scale)
  (make-seg (stretch-point (start-point seg) scale)
    (stretch-point (end-point seg) scale)))

(define (seg-length seg)
  (sqrt (+ (square
    (- (point-x (start-point seg))
      (point-x (end-point seg))))
    (square
      (- (point-y (start-point seg))
        (point-y (end-point seg)))))))
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-seg seg scale)
  (make-seg (stretch-point (start-point seg) scale)
            (stretch-point (end-point seg) scale)))

(define (seg-length seg)
  (sqrt (+ (square
            (- (point-x (start-point seg))
               (point-x (end-point seg)))))
       (square
        (- (point-y (start-point seg))
           (point-y (end-point seg)))))))
(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (make-point (* scale (point-x pt)) (* scale (point-y pt))))

(stretch-point p1 2) -> (4 . 6)
p1 -> (2 . 3)
Using data abstractions

(define p1 (make-point 2 3))
(define p2 (make-point 4 1))
(define s1 (make-seg p1 p2))

(define (stretch-point pt scale)
  (cons (* scale (car pt))
        (* scale (cdr pt))))

(stretch-point p1 2) -> (4 . 6)
p1 -> (2 . 3)
Abstractions have two communities

- **Builders**
  
  ```scheme
  (define (make-point x y) (cons x y))
  (define (point-x point) (car point))
  ```

- **Users**
  
  ```scheme
  (* scale (point-x pt))
  ```
Abstractions have two communities

- **Builders**
  
  (define (make-point x y) (cons x y))
  (define (point-x point) (car point))

- **Users**
  
  (* scale (point-x pt))

- **Frequently the same person**
Pairs are a data abstraction

- **Constructor**
  
  \[(\text{cons} \ A \ B) \mapsto \text{Pair}\langle A, B \rangle\]

- **Accessors**
  
  \[(\text{car} \ \text{Pair}\langle A, B \rangle) \mapsto A\]
  
  \[(\text{cdr} \ \text{Pair}\langle A, B \rangle) \mapsto B\]

- **Contract**
  
  \[(\text{car} \ (\text{cons} \ A \ B)) \mapsto A\]
  
  \[(\text{cdr} \ (\text{cons} \ A \ B)) \mapsto B\]

- **Operations**
  
  \[(\text{pair?} \ Q) \ \text{returns} \ #t \ \text{if} \ Q \ \text{evaluates to a pair,} \ #f \ \text{otherwise}\]
Pairs are a data abstraction

- **Constructor**
  \[(\text{cons}\ A\ B) \mapsto \text{Pair}<A,B>\]

- **Accessors**
  \[(\text{car}\ \text{Pair}<A,B>) \mapsto A\]
  \[(\text{cdr}\ \text{Pair}<A,B>) \mapsto B\]

- **Contract**
  \[(\text{car}\ (\text{cons}\ A\ B)) \mapsto A\]
  \[(\text{cdr}\ (\text{cons}\ A\ B)) \mapsto B\]

- **Operations**
  \[(\text{pair?}\ Q)\ \text{returns}\ \#t\ \text{if}\ Q\ \text{evaluates to a pair},\ \#f\ \text{otherwise}\]

- **Abstraction barrier**
Rational number abstraction

- A rational number is a ratio $\frac{n}{d}$.
Rational number abstraction

- A rational number is a ratio \( \frac{n}{d} \)
- Addition:
  \[
  \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}
  \]
A rational number is a ratio $\frac{n}{d}$

Addition:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

$$\frac{2}{3} + \frac{1}{4} = \frac{2 \cdot 4 + 3 \cdot 1}{12} = \frac{11}{12}$$
A rational number is a ratio \( \frac{n}{d} \).

**Addition:**

\[
\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}
\]

\[
\frac{2}{3} + \frac{1}{4} = \frac{2 \cdot 4 + 3 \cdot 1}{12} = \frac{11}{12}
\]

**Multiplication:**

\[
\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}
\]
Rational number abstraction

- A rational number is a ratio \( \frac{n}{d} \)
- Addition:
  \[
  \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}
  \]
  \[
  \frac{2}{3} + \frac{1}{4} = \frac{2 \cdot 4 + 3 \cdot 1}{12} = \frac{11}{12}
  \]
- Multiplication:
  \[
  \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}
  \]
  \[
  \frac{2}{3} \cdot \frac{1}{3} = \frac{2}{9}
  \]
Rational number abstraction

- **Constructor**
  
  ; make-rat: integer, integer -> Rat
  
  (make-rat <n> <d>) -> <r>
Rational number abstraction

- **Constructor**
  
  ; make-rat: integer, integer -> Rat
  
  (make-rat <n> <d>) -> <r>

- **Accessors**
  
  ; numer, denom: Rat -> integer
  
  (numer <r>)
  
  (denom <r>)
Rational number abstraction

- **Constructor**
  
  \[
  \text{make-rat: integer, integer -> Rat} \\
  (\text{make-rat } \langle n \rangle \; \langle d \rangle) \rightarrow \langle r \rangle
  \]

- **Accessors**

  \[
  \text{numer, denom: Rat -> integer} \\
  (\text{numer } \langle r \rangle) \\
  (\text{denom } \langle r \rangle)
  \]

- **Contract**

  \[
  (\text{numer (make-rat } \langle n \rangle \; \langle d \rangle)) \implies \langle n \rangle \\
  (\text{denom (make-rat } \langle n \rangle \; \langle d \rangle)) \implies \langle d \rangle
  \]
Rational number abstraction

- **Constructor**
  
  \[
  \text{make-rat: integer, integer -> Rat} \\
  (\text{make-rat } <n> <d>) \rightarrow <r>
  \]

- **Accessors**
  
  \[
  \text{numer, denom: Rat -> integer} \\
  (\text{numer } <r>) \\
  (\text{denom } <r>)
  \]

- **Contract**
  
  \[
  (\text{numer (make-rat } <n> <d>)) \implies <n> \\
  (\text{denom (make-rat } <n> <d>)) \implies <d>
  \]

- **Operations**
  
  \[
  (+\text{rat } x \ y) \\
  (*\text{rat } x \ y)
  \]
Rational number abstraction

- **Constructor**
  
  ```scheme
  ; make-rat: integer, integer -> Rat
  (make-rat <n> <d>) -> <r>
  ```

- **Accessors**
  
  ```scheme
  ; numer, denom: Rat -> integer
  (numer <r>)
  (denom <r>)
  ```

- **Contract**
  
  ```scheme
  (numer (make-rat <n> <d>)) \implies <n>
  (denom (make-rat <n> <d>)) \implies <d>
  ```

- **Operations**
  
  ```scheme
  (+rat x y)
  (*rat x y)
  ```

- **Abstraction barrier**
Rational number abstraction

- Constructor
- Accessors
- Contract
- Operations
- Abstraction barrier

### Implementation

; Rat = Pair<integer, integer>
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
Rational number abstraction

- Constructor
- Accessors
- Contract
- Operations
- Abstraction barrier

Implementation

; Rat = Pair<integer, integer>
(define (make-rat n d) (cons d n))
(define (numer r) (cdr r))
(define (denom r) (car r))
; What is the type of +rat?
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
             (* (numer y) (denom x)))
           (* (denom x) (denom y))))
Additional operators

; What is the type of +rat? Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
              (* (numer y) (denom x)))
             (* (denom x) (denom y))))
Additional operators

; What is the type of +rat? Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
             (* (denom x) (denom y))))

; The type of *rat:
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
Additional operators

; What is the type of +rat? Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
              (* (numer y) (denom x)))
             (* (denom x) (denom y))))

; The type of *rat: Rat, Rat -> Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
             (* (denom x) (denom y))))
Using our system

(define one-half (make-rat 1 2))
(define three-fourths (make-rat 3 4))

(define new (+rat one-half three-fourths))

(numer new) ; ?
(denom new) ; ?
Using our system

```
(define one-half (make-rat 1 2))
(define three-fourths (make-rat 3 4))

(define new (+rat one-half three-fourths))

(numer new) ; 10
(denom new) ; 8
```

We get $\frac{10}{8}$, not the simplified $\frac{5}{4}$
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b)))))
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))

(define (make-rat n d)
  (cons n d))

(define (numer r)
  (/ (car r) (gcd (car r) (cdr r))))

(define (denom r)
  (/ (cdr r) (gcd (car r) (cdr r))))
Rationalizing implementation

(define (gcd a b)
    (if (= b 0)
        a
        (gcd b (remainder a b)) ))

(define (make-rat n d)
    (cons n d))

(define (numer r)
    (/ (car r) (gcd (car r) (cdr r))))

(define (denom r)
    (/ (cdr r) (gcd (car r) (cdr r))))

Remove common factors when accessed
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))

(define (make-rat n d)
  (cons (/ n (gcd n d))
        (/ d (gcd n d))))

(define (numer r)
  (car r))

(define (denom r)
  (cdr r))

Remove common factors when created
We want to group a set of rational numbers
Grouping together larger collections

We want to group a set of rational numbers

(cons r1 r2)
We want to group a set of rational numbers

\[(\text{cons} \ (\text{cons} \ r1 \ r2) \\
(\text{cons} \ r3 \ r4))\]
We want to group a set of rational numbers

\[(\text{cons} \ (\text{cons} \ (\text{cons} \ r1 \ r2) \ \\
\quad \ (\text{cons} \ r3 \ r4)) \ \\
\quad r5)\]
Grouping together larger collections

We want to group a set of rational numbers

\[(\text{cons} \ (\text{cons} \ (\text{cons} \ r_1 \ r_2) \ (\text{cons} \ r_3 \ r_4)) \ (\text{cons} \ r_5 \ r_6))\]
We want to group a set of rational numbers

(cons (cons (cons (cons r1 r2) (cons r3 r4)) (cons r5 r6)) (cons r7 r8))
Grouping together larger collections

We want to group a set of rational numbers

$$(\text{cons} \ (\text{cons} \ (\text{cons} \ r_1 \ r_2) \ (\text{cons} \ r_3 \ r_4)) \ (\text{cons} \ (\text{cons} \ r_5 \ r_6) \ (\text{cons} \ r_7 \ r_8)))$$
Grouping together larger collections

We want to group a set of rational numbers

\[(\text{cons} \ (\text{cons} \ (\text{cons} \ r1 \ r2) \ \\
\qquad \ (\text{cons} \ r3 \ r4)) \ \\
(\text{cons} \ (\text{cons} \ r5 \ r6) \ \\
\qquad \ (\text{cons} \ r7 \ r8))\)\]

\[
\ldots
\]
A list is a type that can hold an arbitrary number of ordered items.
Conventional interfaces — lists

- A list is a type that can hold an arbitrary number of ordered items.
- Formally, a list is a sequence of pairs with the following properties:
  - The \textit{car}-part of a pair holds an item
  - The \textit{cdr}-part of a pair holds the rest of the list
  - The list is terminated by the empty list: \texttt{’()}

Lists are closed under \texttt{cons} and \texttt{cdr}
A list is a type that can hold an arbitrary number of ordered items. Formally, a list is a sequence of pairs with the following properties:

- The car-part of a pair holds an item
- The cdr-part of a pair holds the rest of the list
- The list is terminated by the empty list: ’()’

Lists are closed under cons and cdr.
(cons <el1> <el2>)

(list <el1> <el2> ... <eln>)

(list 1 2 3 4) ; -> (1 2 3 4)

(null? <z>) ; -> #t if <z> evaluates to empty list
(cons <el1> <el2>)
(cons <el1> <el2>)

\[ \text{node} \rightarrow \text{el2} \]

\[ \downarrow \]

\[ \text{el1} \]

(list <el1> <el2> ... <eln>)
Lists and pairs as pictures

\[(\text{cons} \ <\text{el}1>\ <\text{el}2>)\]

\[
\begin{array}{c}
\text{el}2 \\
\downarrow \\
\text{el}1
\end{array}
\]

\[(\text{list} \ <\text{el}1>\ <\text{el}2> \ ... \ <\text{el}n>)\]

\[
\begin{array}{c}
\text{el}1 \\
\downarrow \\
\text{el}2 \\
\downarrow \\
\cdot \cdot \cdot \\
\downarrow \\
\text{el}n
\end{array}
\]
Lists and pairs as pictures

(\texttt{cons} \ <\texttt{el1}> \ <\texttt{el2}>)

(\texttt{list} \ <\texttt{el1}> \ <\texttt{el2}> \ ... \ <\texttt{eln}>)

(\texttt{list} \ 1 \ 2 \ 3 \ 4) \ ; \ \rightarrow
Lists and pairs as pictures

(cons <el1> <el2>)

\[\text{el1} \rightarrow \text{el2}\]

(list <el1> <el2> ... <eln>)

\[\text{el1} \rightarrow \text{el2} \rightarrow \cdots \rightarrow \text{eln}\]

(list 1 2 3 4) ; -> (1 2 3 4)
Lists and pairs as pictures

(cons <el1> <el2>)

\[ \begin{array}{c}
\text{el1} \\
\downarrow \\
\text{el2}
\end{array} \]

(list <el1> <el2> ... <eln>)

\[ \begin{array}{c}
\text{el1} \\
\downarrow \\
\text{el2} \\
\downarrow \\
\cdots \\
\downarrow \\
\text{eln}
\end{array} \]

(list 1 2 3 4) ; -> (1 2 3 4)
(null? <z>) ; -> #t if <z> evaluates to empty list
Sequences of cons cells

Better, and safer, to abstract:

(define first car)
(define rest cdr)
(define adjoin cons)
- Sequences of `cons` cells

- Better, and safer, to abstract:
  - `(define first car)`
  - `(define rest cdr)`
  - `(define adjoin cons)`

- ... but we don’t for lists and pairs
(define 1thru4 (list 1 2 3 4))
(define 1thru4 (list 1 2 3 4))
(define 2thru7 (list 2 3 4 5 6 7))
(define 1thru4 (list 1 2 3 4))
(define 2thru7 (list 2 3 4 5 6 7))

(define (enumerate from to)
  (if (> from to)
      '()
      (cons from (enumerate (+ 1 from) to)))))
cdr’ing down lists

(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))

(append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1) list2)))
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))

(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1)
                    list2))))
(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst)))))

(define (double-list lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst))
            (double-list (cdr lst)))))

(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

Alex Vandiver (MIT)
Lists, higher order procedures, and symbols
Lecture 2
Transforming lists

(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst))))

(define (double-list lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst))
            (double-list (cdr lst))))
(define (square-list lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-list (cdr lst))))

(define (double-list lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst))
            (double-list (cdr lst)))))

Transforming lists

Alex Vandiver (MIT)
Lists, higher order procedures, and symbols
Lecture 2 49 / 65
(define (square-list lst)
  (if (null? lst)
      ()
      (cons (square (car lst))
            (square-list (cdr lst))))))

(define (double-list lst)
  (if (null? lst)
      ()
      (cons (* 2 (car lst))
            (double-list (cdr lst))))))

(define (map proc lst)
  (if (null? lst)
      ()
      (cons (proc (car lst))
            (map proc (cdr lst))))))
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
           (map proc (cdr lst))))

What is the type of `map`?
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

What is the type of `map`?

, $\mapsto$
(define (map proc lst)
   (if (null? lst)
       ()
       (cons (proc (car lst))
             (map proc (cdr lst)))))

What is the type of map?

(  →  ),  →
(define (map proc lst)
  (if (null? lst)
      ()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

What is the type of map?
(↦→, List<→↦)
(define (map proc lst)
  (if (null? lst)
      ()
      (cons (proc (car lst))
        (map proc (cdr lst)))))

What is the type of map?
(↦→), List<A> ↦
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst))))

What is the type of \texttt{map}?
\((A \mapsto \_), \texttt{List}<A> \mapsto\)
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst))))))

What is the type of \texttt{map}?
\((A \rightarrow B), \text{List}<A> \rightarrow\)
(define (map proc lst)
  (if (null? lst)
      ()
      (cons (proc (car lst))
              (map proc (cdr lst))))

What is the type of \texttt{map}? 
\texttt{(A \mapsto B), List\langle A \rangle \mapsto List\langle \rangle)}
(define (map proc lst)
  (if (null? lst)
    ()
    (cons (proc (car lst))
      (map proc (cdr lst))))
)

What is the type of map?
(A → B), List<A> → List<B>
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))

(filter even? (list 1 2 3 4 5 6))
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
  ((pred (car lst))
   (cons (car lst)
     (filter pred (cdr lst))))
  (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) ’())
        ((pred (car lst))
         (cons (car lst)
              (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;;-> (2 4 6)

What is the type of filter?
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
                (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)

What is the type of filter?
(→, List< → →)
(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)

What is the type of filter?
(↦Boolean), List<A> ↦
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;;-> (2 4 6)

What is the type of filter?
(A ↦ ), List<A> ↦
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
                (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)

What is the type of filter?
(A ↦ Boolean), List<A> ↦
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
    ((pred (car lst))
    (cons (car lst)
        (filter pred (cdr lst)))))
  (else (filter pred (cdr lst)))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)

What is the type of filter?
(A \rightarrow \text{Boolean}), \text{List}<A> \rightarrow \text{List}< >
Choosing just part of a list

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst))))))

(filter even? (list 1 2 3 4 5 6))
;-> (2 4 6)

What is the type of filter?
(A ↦ Boolean), List<A> ↦ List<A>
Data Types in Scheme

- Conventional
  - Numbers: 29, −35, 1.34, 1.2e5
Data Types in Scheme

- Conventional
  - Numbers: 29, −35, 1.34, 1.2e5
  - Characters and Strings: \texttt{#\\textbackslash{}a "this is a string"}
Data Types in Scheme

- **Conventional**
  - Numbers: 29, −35, 1.34, 1.2e5
  - Characters and Strings: \texttt{\textbackslash a "this is a string"}
  - Booleans: \texttt{#t}, \texttt{#f}
Data Types in Scheme

Conventional
- Numbers: 29, −35, 1.34, 1.2e5
- Characters and Strings: \a "this is a string"
- Booleans: #t, #f
- Vectors: #(1 2 3 "hi" 3.7)
Data Types in Scheme

- **Conventional**
  - Numbers: 29, −35, 1.34, 1.2e5
  - Characters and Strings: \( \text{"this is a string"} \)
  - Booleans: \#t, \#f
  - Vectors: \#\( (1, 2, 3, \text{"hi"}, 3.7) \)

- **Scheme-specific**
  - Procedures: value of +, result of evaluating \( \text{\lambda} (x) \ x \)
Data Types in Scheme

- Conventional
  - Numbers: 29, −35, 1.34, 1.2e5
  - Characters and Strings: \\a "this is a string"
  - Booleans: #t, #f
  - Vectors: #(1 2 3 "hi" 3.7)

- Scheme-specific
  - Procedures: value of +, result of evaluating (lambda (x) x)
  - Pairs and lists: (42 . 8), (1 1 2 3 5 8 13)
Data Types in Scheme

- Conventional
  - Numbers: 29, −35, 1.34, 1.2e5
  - Characters and Strings: \\a "this is a string"
  - Booleans: #t, #f
  - Vectors: #(1 2 3 "hi" 3.7)

- Scheme-specific
  - Procedures: value of +, result of evaluating (lambda (x) x)
  - Pairs and lists: (42 . 8), (1 1 2 3 5 8 13)
  - Symbols: pi, +, x, foo, hello-world
Symbols

- So far, we’ve seen them as the names of variables
  - `(define foo (+ bar 2))"
So far, we’ve seen them as the names of variables

(\texttt{define \texttt{foo} (+ \texttt{bar} 2)})

But, in Scheme, all data types are \underline{first class}, so we should be able to:
Symbols

So far, we’ve seen them as the names of variables

(define foo (+ bar 2))

But, in Scheme, all data types are first class, so we should be able to:

Pass symbols as arguments to procedures
Symbols

- So far, we’ve seen them as the names of variables
  - `(define foo (+ bar 2))`
- But, in Scheme, all data types are **first class**, so we should be able to:
  - Pass symbols as arguments to procedures
  - Return them as values of procedures
Symbols

- So far, we’ve seen them as the names of variables
  - `(define foo (+ bar 2))`
- But, in Scheme, all data types are first class, so we should be able to:
  - Pass symbols as arguments to procedures
  - Return them as values of procedures
  - Associate them as values of variables
Symbols

- So far, we’ve seen them as the names of variables
  
  `(define foo (+ bar 2))`

- But, in Scheme, all data types are first class, so we should be able to:
  
  - Pass symbols as arguments to procedures
  - Return them as values of procedures
  - Associate them as values of variables
  - Store them in data structures

  - For example: `(chocolate caffeine sugar)`

![Diagram showing data structures for chocolate, caffeine, and sugar]
How do we refer to Symbols?

- Evaluation rule for symbols
  - Value of a symbol is the value it is associated with in the environment.
How do we refer to Symbols?

- Evaluation rule for symbols
  - Value of a symbol is the value it is associated with in the environment.
  - We associate symbols with values using the special form `define`.
How do we refer to Symbols?

- **Evaluation rule for symbols**
  - Value of a symbol is the value it is associated with in the environment.
  - **We associate symbols with values using the special form** `define`
  - `(define pi 3.1415926535)`
How do we refer to Symbols?

- Evaluation rule for symbols
  - Value of a symbol is the value it is associated with in the environment.
  - We associate symbols with values using the special form `define`.
  - `(define pi 3.1415926535)`
  - `(* pi 2 r)`
How do we refer to Symbols?

- **Evaluation rule for symbols**
  - Value of a symbol is the value it is associated with in the environment.
  - We associate symbols with values using the **special form** `define`.
    - `(define pi 3.1415926535)
    - `(* pi 2 r)`

- **But how do we get to the symbol itself?**
  - `(define baz pi)` ??
  - `baz →`
How do we refer to Symbols?

- **Evaluation rule for symbols**
  - Value of a symbol is the value it is associated with in the environment.
  - We associate symbols with values using the special form `define`.
  
  ```scheme
  (define pi 3.1415926535)
  (* pi 2 r)
  ```

- **But how do we get to the symbol itself?**
  
  ```scheme
  (define baz pi) ??
  baz → 3.1415926535
  ```
Referring to Symbols

Say your favorite color
Referring to Symbols

- Say your favorite color
- Say “your favorite color”
Say your favorite color
Say “your favorite color”
In the first case, we want the meaning associated with the expression
In the second, we want the expression itself
Referring to Symbols

- Say your favorite color
- Say “your favorite color”
- In the first case, we want the meaning associated with the expression
- In the second, we want the expression itself
- We use the concept of quotation in Scheme to distinguish between these two cases
New special form: quote

We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

(quote foo)
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

\[(\text{quote} \ \text{foo}) \rightarrow \text{foo}\]
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

  (quote foo) → foo
  (define baz (quote pi)) → undefined
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”
  
  `(quote foo) → foo
  (define baz (quote pi)) → undefined

  baz
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

  (quote foo) → foo
  (define baz (quote pi)) → undefined
  baz → pi
New special form: quote

We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

(quote foo) → foo
(define baz (quote pi)) → undefined
baz → pi
(+ pi baz)
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

(quote foo) → foo
(define baz (quote pi)) → undefined
baz → pi
(+ pi baz) → ERROR
We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”

(quote foo) \rightarrow foo
(define baz (quote pi)) \rightarrow undefined
baz \rightarrow pi
(+ pi baz) \rightarrow ERROR

+: expects type <number> as 2nd argument, given: pi; other arguments were: 3.1415926535
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”
  
  \[(quote \text{foo}) \rightarrow \text{foo}\]
  
  \[(\text{define baz (quote pi)}) \rightarrow \text{undefined}\]
  
  \[
  \text{baz} \rightarrow \text{pi}
  \]
  
  \[(+ \text{pi baz}) \rightarrow \text{ERROR}\]

  - \(+:: \text{expects type <number> as 2nd argument, given: pi; other arguments were: } 3.1415926535\]

  \[(\text{list (quote foo) (quote bar) (quote baz)})\]
New special form: quote

- We want a way to tell the evaluator: “I want the following object as whatever it is, not as an expression to be evaluated”
  
  (quote foo) → foo
  (define baz (quote pi)) → undefined
  baz → pi
  (+ pi baz) → ERROR

- +: expects type <number> as 2nd argument, given: pi; other arguments were: 3.1415926535

  (list (quote foo) (quote bar) (quote baz))
  → (foo bar baz)
Syntactic sugar

- The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
Syntactic sugar

- The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
- When it sees ‘pi it acts just like it had read (quote pi)
- The latter is what is actually evaluated

Examples:

- ‘pi → pi
- ’17 → 17
- ’"Hello world" → "Hello world"
- ’(1 2 3) → (1 2 3)
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut

When it sees ‘pi it acts just like it had read (quote pi)

The latter is what is actually evaluated

Examples:

’pi
Syntactic sugar

- The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
- When it sees ‘pi it acts just like it had read (quote pi)
- The latter is what is actually evaluated
- Examples:
  ‘pi → pi
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
When it sees ’pi it acts just like it had read (quote pi)
The latter is what is actually evaluated
Examples:
’pi → pi
’17
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut

When it sees ‘pi it acts just like it had read (quote pi)

The latter is what is actually evaluated

Examples:

‘pi → pi
‘17 → 17
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
When it sees ‘pi it acts just like it had read (quote pi)
The latter is what is actually evaluated
Example:
’pi → pi
’17 → 17
’"Hello world"
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut

When it sees 'pi it acts just like it had read (quote pi)

The latter is what is actually evaluated

Examples:

'pi → pi
'17 → 17
'"Hello world" → "Hello world"
The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
When it sees ‘pi it acts just like it had read (quote pi)
The latter is what is actually evaluated
Examples:
‘pi → pi
‘17 → 17
‘"Hello world" → "Hello world"
’(1 2 3)
Syntactic sugar

- The Reader (part of the Read-Eval-Print Loop, REPL) knows a short-cut
- When it sees \texttt{’pi} it acts just like it had read \texttt{(quote pi)}
- The latter is what is actually evaluated
- Examples:
  \texttt{’pi} \rightarrow \texttt{pi}
  \texttt{’17} \rightarrow \texttt{17}
  \texttt{’"Hello world"} \rightarrow \texttt{"Hello world"}
  \texttt{’(1 2 3)} \rightarrow \texttt{(1 2 3)}
Making list structures with symbols

(list (quote brains) (quote caffeine) (quote sugar))
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)

(define x 42) (define y '(x y z))
(list (list 'foo 'bar) (list x y) (list 'baz 'quux 'squee))
; -> ((foo bar) (42 (x y z)) (baz quux squee))
'(list 'foo 'bar) (x y) (baz quux squee)
Making list structures with symbols

(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
; -> (brains caffeine sugar)
Making list structures with symbols

(list (quote brains) (quote caffeine) (quote sugar))
  ; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
  ; -> (brains caffeine sugar)
'(brains caffeine sugar)
Making list structures with symbols

(list (quote brains) (quote caffeine) (quote sugar))
  ; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
  ; -> (brains caffeine sugar)
'(brains caffeine sugar)
  ; -> (brains caffeine sugar)
(list (quote brains) (quote caffeine) (quote sugar))  
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)  
; -> (brains caffeine sugar)
'(brains caffeine sugar)  
; -> (brains caffeine sugar)
(define x 42) (define y '(x y z))
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
; -> (brains caffeine sugar)
'(brains caffeine sugar)
; -> (brains caffeine sugar)
(define x 42) (define y '(x y z))
(list (list 'foo 'bar) (list x y)
     (list 'baz 'quux 'squee))
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
; -> (brains caffeine sugar)
'(brains caffeine sugar)
; -> (brains caffeine sugar)
(define x 42) (define y '(x y z))
(list (list 'foo 'bar) (list x y)
 (list 'baz 'quux 'squee))
; -> ((foo bar) (42 (x y z))
 (baz quux squee))
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
; -> (brains caffeine sugar)
'(brains caffeine sugar)
; -> (brains caffeine sugar)
(define x 42) (define y '(x y z))
(list (list 'foo 'bar) (list x y)
 (list 'baz 'quux 'squee))
; -> ((foo bar) (42 (x y z))
 (baz quux squee))
'((foo bar) (x y) (bar quux squee))
(list (quote brains) (quote caffeine) (quote sugar))
; -> (brains caffeine sugar)
(list 'brains 'caffeine 'sugar)
; -> (brains caffeine sugar)
'(brains caffeine sugar)
; -> (brains caffeine sugar)
(define x 42) (define y '(x y z))
(list (list 'foo 'bar) (list x y)
(list 'baz 'quux 'squee))
; -> (((foo bar) (42 (x y z))
    (baz quux squee))
'((foo bar) (x y) (bar quux squee))
; -> (((foo bar) (x y) (bar quux squee))
Confusing examples

(define x 20)

(+ x 3) -> 23
(+ x 3) -> (+ x 3)
(list (quote +) x '3) -> (+ 20 3)
(list '+ x 3) -> (+ 20 3)
(list + x 3) -> (#<procedure:+> 20 3)
Confusing examples

(define x 20)
(+ x 3) ; ->

(list (quote +) x '3) ; ->
(list '+ x 3) ; ->
(list + x 3) ; ->

Alex Vandiver (MIT)
Lists, higher order procedures, and symbols
Lecture 2 59 / 65
Confusing examples

\( \text{(define } x \ 20) \) \\
\( (+ \ x \ 3) \quad ; \quad \rightarrow \quad 23 \)
(define x 20)
(+ x 3) ; -> 23
'(+ x 3) ; ->
Confusing examples

```
(define x 20)
(+ x 3) ; -> 23
'(+ x 3) ; -> (+ x 3)
```
Confusing examples

(define x 20)
(+ x 3) ; → 23
'(+ x 3) ; → (+ x 3)
(list (quote +) x '3) ; →
Confusing examples

(define x 20)
(+ x 3) ; -> 23
'(+ x 3) ; -> (+ x 3)
(list (quote +) x '3) ; -> (+ 20 3)
Confusing examples

(define x 20)
(+ x 3) ;  ->  23
'(+ x 3) ;  ->  (+ x 3)
(list (quote +) x '3) ;  ->  (+ 20 3)
(list '+ x 3) ;  ->
(define x 20)
(+ x 3) ; → 23
'(+ x 3) ; → (+ x 3)
(list (quote +) x '3) ; → (+ 20 3)
(list '+ x 3) ; → (+ 20 3)
Confusing examples

(define x 20)
(+ x 3) ; -> 23
'(+ x 3) ; -> (+ x 3)
(list (quote +) x '3) ; -> (+ 20 3)
(list '+ x 3) ; -> (+ 20 3)
(list + x 3) ; ->
Confusing examples

\[(\text{define } x 20)\]
\[(+ x 3) \quad ; \quad \rightarrow \quad 23\]
\[\text{'}(+ x 3)\quad ; \quad \rightarrow \quad (+ x 3)\]
\[(\text{list } (\text{quote } +) x \text{'}3)\quad ; \quad \rightarrow \quad (+ 20 3)\]
\[(\text{list } '+ x 3)\quad ; \quad \rightarrow \quad (+ 20 3)\]
\[(\text{list } + x 3)\quad ; \quad \rightarrow \quad (#<\text{procedure:+}> 20 3)\]
Operations on symbols

- **symbol?** has type `anytype` → `boolean`, returns `#t` for symbols

  ```scheme
  (symbol? (quote foo)) → #t
  (symbol? 'foo) → #t
  (symbol? 4) → #f
  (symbol? '(1 2 3)) → #f
  (symbol? foo) → It depends on what value foo is bound to
  ```
**Operations on symbols**

- **symbol?** has **type anytype** → **boolean**, **returns #t for symbols**
  - (symbol? (quote foo)) → #t
  - (symbol? (quote 'foo)) → #t
  - (symbol? 4) → #f
  - (symbol? '(1 2 3)) → #f
  - (symbol? foo) → It depends on what value foo is bound to

Alex Vandiver (MIT)
Operations on symbols

- symbol? has type anytype → boolean, returns #t for symbols
  - (symbol? (quote foo)) → #t
  - (symbol? 'foo) → #t
Operations on symbols

- `symbol?` has type `anytype -> boolean`, returns `#t` for symbols
  - `(symbol? (quote foo)) -> #t`
  - `(symbol? 'foo) -> #t`
  - `(symbol? 4) -> #f`

It depends on what value `foo` is bound to.

`eq?` tests the equality of symbols.
Operations on symbols

- `symbol?` has type `anytype → boolean`, returns `#t` for symbols
  
  `(symbol? (quote foo)) → #t
  (symbol? ’foo) → #t
  (symbol? 4) → #f
  (symbol? ’(1 2 3)) → #f

It depends on what value `foo` is bound to.
symbol? has type anytype → boolean, returns #t for symbols

(symbol? (quote foo)) → #t
(sимвol? 'foo) → #t
(symbol? 4) → #f
(symbol? '(1 2 3)) → #f
(symbol? foo) → It depends on what value foo is bound to
Operations on symbols

- `symbol?` has type `anytype` → boolean, returns `#t` for symbols
  - `(symbol? (quote foo)) → #t`
  - `(symbol? 'foo) → #t`
  - `(symbol? 4) → #f`
  - `(symbol? '(1 2 3)) → #f`
  - `(symbol? foo) → It depends on what value foo is bound to`

- `eq?` tests the equality of symbols
An aside: Testing for equality

- `eq?` tests if two things are exactly the same object in memory. Not for strings or numbers.
An aside: Testing for equality

- `eq?` tests if two things are exactly the same object in memory. Not for strings or numbers.
- `=` tests the equality of numbers
An aside: Testing for equality

- `eq?` tests if two things are exactly the same object in memory. Not for strings or numbers.
- `=` tests the equality of numbers
- `equal?` tests if two things print the same—symbols, numbers, strings, lists of those, lists of lists
(= 4 10) ; ->
(= 4 10) ; -> #f
(= 4 10) ; -> #f
(= 4 4) ; ->
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; ->
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; ->
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> 
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t

define a '(1 2)
define b '(1 2)
eq? a b ; → #f
define a b
eq? a b ; → #t
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t
(eq? (expt 2 70) (expt 2 70)) ; →
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t
(eq? (expt 2 70) (expt 2 70)) ; → #f

(= "foo" "foo") ; →
(= 4 10) ; -> #f
(= 4 4)  ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4)  ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
\[
\begin{align*}
\text{(= 4 10)} & \quad ; \quad \rightarrow \quad \#f \\
\text{(= 4 4)} & \quad ; \quad \rightarrow \quad \#t \\
\text{(equal? 4 4)} & \quad ; \quad \rightarrow \quad \#t \\
\text{(equal? (/ 1 2) 0.5)} & \quad ; \quad \rightarrow \quad \#f \\
\text{(eq? 4 4)} & \quad ; \quad \rightarrow \quad \#t \\
\text{(eq? (expt 2 70) (expt 2 70))} & \quad ; \quad \rightarrow \quad \#f \\
\text{(= "foo" "foo")} & \quad ; \quad \rightarrow \quad \text{Error!} \\
\text{(eq? "foo" "foo")} & \quad ; \quad \rightarrow\ 
\end{align*}
\]
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; ->
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t
(eq? (expt 2 70) (expt 2 70)) ; → #f

(= "foo" "foo") ; → Error!
(eq? "foo" "foo") ; → #f
(equal? "foo" "foo") ; → #t

(eq? '(1 2) '(1 2)) ; →
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t
(eq? (expt 2 70) (expt 2 70)) ; → #f

(= "foo" "foo") ; → Error!
(eq? "foo" "foo") ; → #f
(equal? "foo" "foo") ; → #t

(eq? '(1 2) '(1 2)) ; → #f
(equal? '(1 2) '(1 2)) ; →
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(define b '(1 2))
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(define b '(1 2))
(eq? a b) ; ->
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(define b '(1 2))
(eq? a b) ; -> #f
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(define b '(1 2))
(eq? a b) ; -> #f
(define a b)
(= 4 10) ; → #f
(= 4 4) ; → #t
(equal? 4 4) ; → #t
(equal? (/ 1 2) 0.5) ; → #f
(eq? 4 4) ; → #t
(eq? (expt 2 70) (expt 2 70)) ; → #f

(= "foo" "foo") ; → Error!
(eq? "foo" "foo") ; → #f
(equal? "foo" "foo") ; → #t

(eq? '(1 2) '(1 2)) ; → #f
(equal? '(1 2) '(1 2)) ; → #t
(define a '(1 2))
(define b '(1 2))
(eq? a b) ; → #f
(define a b)
(eq? a b) ; →
(= 4 10) ; -> #f
(= 4 4) ; -> #t
(equal? 4 4) ; -> #t
(equal? (/ 1 2) 0.5) ; -> #f
(eq? 4 4) ; -> #t
(eq? (expt 2 70) (expt 2 70)) ; -> #f

(= "foo" "foo") ; -> Error!
(eq? "foo" "foo") ; -> #f
(equal? "foo" "foo") ; -> #t

(eq? '(1 2) '(1 2)) ; -> #f
(equal? '(1 2) '(1 2)) ; -> #t
(define a '(1 2))
(define b '(1 2))
(eq? a b) ; -> #f
(define a b)
(eq? a b) ; -> #t
Tagged data

- Attaching a symbol to all data values that indicates the type
- Can now determine if something is the type you expect

(define (make-point x y)  
  (list x y))

(define (make-rat n d)  
  (list x y))
Tagged data

- Attaching a symbol to all data values that indicates the type
- Can now determine if something is the type you expect

```
(define (make-point x y)
  (list 'point x y))

(define (make-rat n d)
  (list 'rat x y))
```
Tagged data

- Attaching a symbol to all data values that indicates the type
- Can now determine if something is the type you expect

(define (make-point x y)
  (list 'point x y))

(define (make-rat n d)
  (list 'rat x y))

(define (point? thing)
  (and (pair? thing)
       (eq? (car thing) 'point)))

(define (rat? thing)
  (and (pair? thing)
       (eq? (car thing) 'rat)))
Benefits of tagged data

- **Data-directed programming** - decide what to do based on type

```scheme
(define (stretch thing scale)
  (if (point? thing)
      (stretch-point thing scale)
      (stretch-seg thing scale)))
```

Defensive programming - Determine if something is the type you expect, give a better error

```scheme
(define (stretch-point pt)
  (if (not (point? pt))
      (error "stretch-point passed a non-point:" pt)
      ;; ... carry on
)
```

Alex Vandiver (MIT)
Benefits of tagged data

- **Data-directed programming** - decide what to do based on type

```scheme
(define (stretch thing scale)
  (if (point? thing)
      (stretch-point thing scale)
      (stretch-seg thing scale))
)
```

- **Defensive programming** - Determine if something is the type you expect, give a better error

```scheme
(define (stretch-point pt)
  (if (not (point? pt))
      (error "stretch-point passed a non-point:" pt)
      ;; ...carry on
  ))
```
Recitation time!