Lists, higher order procedures, and symbols

6.037 - Structure and Interpretation of Computer Programs

Alex Vandiver (alexmv)

Massachusetts Institute of Technology

Lecture 2

Administrivia

- 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)      => 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
- 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
- **number** $\mapsto$ **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 +?
- **number, number** $\mapsto$ **number**
  (mostly)

Type examples

Expression:	...is of type:
`15`	**number**
`"hi"`	**string**
`square`	**number** $\mapsto$ **number**
`>`	**number, number** $\mapsto$ **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

```
(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)))
```

any $\mapsto$ **number**

Patterns across procedures

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

- `(* 17 17)`
- `(* 42 42)`
- `(* x x)`
- ...
- `(lambda (x) (* x x))`

Summation

- $1 + 2 + \ldots + 100$
- $1 + 4 + 9 + \ldots + 100^2$
- $1 + \frac{1}{3^2} + \frac{1}{5^2} + \ldots + \frac{1}{99^2} \approx \frac{\pi^2}{8}$

Summation

```
(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))))
```


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↦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

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

```
(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

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

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

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)

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)
             (+ 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)))
```


Procedural abstraction

```
(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)))
```

Summary of types

- 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
- Basis for many analysis and optimization algorithms

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."

Pairs (`cons` cells)

- `(cons <a> <b>)` → `<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>`
- `(car <p>)` → `<a-val>`
- `(cdr <p>)` → `<b-val>`

Pairs are tasty

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

(car p1)   ; ->  4

(cdr p1)   ; ->  5
```

Pairs are a data abstraction

- Constructor
  (cons A B) $\mapsto$ Pair<A,B>
- Accessors
  (car Pair<A,B>) $\mapsto$ A
  (cdr Pair<A,B>) $\mapsto$ B
- Contract
  (car (cons A B)) $\mapsto$ A
  (cdr (cons A B)) $\mapsto$ B
- Operations
  (pair? Q) returns #t if Q evaluates to a pair, #f otherwise
- Abstraction barrier

Pair abstraction

- 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:
  (cons (cons 1 2) 3)

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 $\mapsto$ Point

Building data abstractions

```
(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))
(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 (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`?

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

(stretch-point p1 2)  -> (4 . 6)
p1 -> (2 . 3)
```


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
  `(cons A B)` ↦ `Pair<A,B>`
- Accessors
  `(car Pair<A,B>)` ↦ `A`
  `(cdr Pair<A,B>)` ↦ `B`
- Contract
  `(car (cons A B))` ↦ `A`
  `(cdr (cons A B))` ↦ `B`
- Operations
  `(pair? Q)` returns `#t` if `Q` evaluates to a pair, `#f` otherwise
- Abstraction barrier

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>
```

- Accessors

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

- Contract

```
(numer (make-rat <n> <d>)) ⟹ <n>
(denom (make-rat <n> <d>)) ⟹ <d>
```

- Operations

```
(+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))
```


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)    ; ?
```

We get $\frac{10}{8}$, not the simplified $\frac{5}{4}$

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

Rationalizing implementation

```
(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

Grouping together larger collections

We want to group a set of rational numbers

```
(cons r1 r2)
```

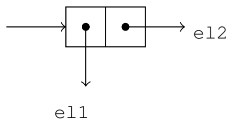
...

Conventional interfaces — lists

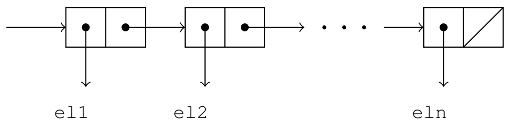
- A list is a type that can hold an arbitrary number of ordered items.
- Formally, a list is a sequence of pairs with the following properties:
  - The `car-part` of a pair holds an item
  - The `cdr-part` of a pair holds the rest of the list
  - The list is terminated by the empty list: `'()`
- Lists are closed under `cons` and `cdr`

Lists and pairs as pictures

```
(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
```


Lists

- 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

`cons`'ing up lists

```
(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)))))

(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
              (append (cdr list1)
                      list2))))
```

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


Map

```
(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))
;-> (2 4 6)
```

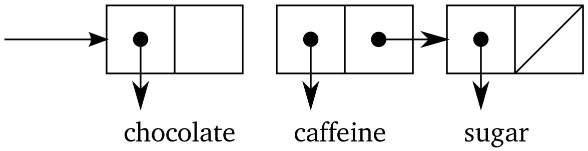
What is the type of `filter`?
(A ↦ Boolean), List<A> ↦ List<A>

Data Types in Scheme

- Conventional
  - Numbers: 29, −35, 1.34, 1.2*e*5
  - 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))`
- 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)`



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 3.1451926535)`
  - `(* pi 2 r)`
- But how do we get to the symbol itself?
  - `(define baz pi)` ??
  - `baz` → `3.1451926535`

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"

  `(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
- 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)`

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)
(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)
```


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.
- `=` tests the equality of numbers
- `equal?` tests if two things print the same– symbols, numbers, strings, lists of those, lists of lists


```
(= 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 '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

```
(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

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

Recitation time!