6.001 SICP

- Today's topics
  - Types of objects and procedures
  - Procedural abstractions
  - Capturing patterns across procedures – Higher Order Procedures

Types

```
(+ 5 10) ==> 15

(+ "hi" 5)
 ;The object "hi", passed as the first argument to
 integer-add, is not the correct type
```

- Addition is not defined for strings

Types – simple data

- We want to collect a taxonomy of expression types:
  - Simple Data
    - Number
      - Integer
      - Real
      - Rational
    - String
    - Boolean
    - Names (symbols)
- We will use this for notational purposes, to reason about our code. Scheme checks types of arguments for built-in procedures, but *not for user-defined ones.*

Types – compound data

- Pair<A,B>
  - A compound data structure formed by a cons pair, in which the first element is of type A, and the second of type B: e.g. (cons 1 2) has type Pair<number, number>
- List<A>=Pair<A, List<A> or nil>
  - A compound data structure that is recursively defined as a pair, whose first element is of type A, and whose second element is either a list of type A or the empty list.
    - E.g. (list 1 2 3) has type List<number>; while (list 1 "string" 3) has type List<number or string>

Examples

25 ; Number
3.45 ; Number
"this is a string" ; String
(> a b) ; Boolean
(cons 1 3) ; Pair<Number, Number>
(list 1 2 3) ; List<Number>
(cons "foo" (cons "bar" nil)) ; List<String>

Types – procedures

- Because procedures operate on objects and return values, we can define their types as well.
- We will denote a procedures type by indicating the types of each of its arguments, and the type of the returned value, plus the symbol → to indicate that the arguments are mapped to the return value
- E.g. **number → number** specifies a procedure that takes a number as input, and returns a number as value

Types

- (+ 5 10) ==> 15
  (+ "hi" 5)
  ;The object "hi", passed as the first argument to integer-add, is not the correct type
- Addition is not defined for strings
- The type of the integer-add procedure is

  number, number → number

  two arguments, both numbers — result value of integer-add is a number

Why "integer-add"?

Type examples

- expression: evaluates to a value of type:

expression	evaluates to a value of type:
15	number
"hi"	string
square	number → number
>	number,number → boolean

(> 5 4) ==> #t

- The 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
    - maybe an error, maybe random behavior

Types, precisely

- A type describes a set of scheme values
  - number → number describes the set:

    all procedures, whose result is a number, which require one argument that must be a number

- Every scheme value has a type
  - Some values can be described by multiple types
  - If so, choose the type which describes the largest set
- Special form keywords like **define** do not name values
  - therefore special form keywords have no type

Your turn

- The following expressions evaluate to values of what type?

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

```
(lambda (p) (if p "hi" "bye"))
```

```
(* 3.14 (* 2 5))
```


Your turn

- The following expressions evaluate to values of what type?

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

number, number, number → number

```
(lambda (p) (if p "hi" "bye"))
```

Boolean → string

```
(* 3.14 (* 2 5))
```

number

End of part 1

- type: a set of values
- every value has a type
- procedure types (types which include →) indicate
  - number of arguments required
  - type of each argument
  - type of result of the procedure
- Types: a mathematical theory for reasoning efficiently about programs
  - useful for preventing certain common types of errors
  - basis for many analysis and optimization algorithms

What is procedure abstraction?

Capture a common pattern

(* 2 2)

(* 57 57)

(* k k)

(lambda (x) (* x x))

Actual pattern

Formal parameter for pattern

Give it a name (define square (lambda (x) (* x x)))

Note the type: number → number

Other common patterns

- 1 + 2 + … + 100 = (100 * 101)/2
- 1 + 4 + 9 + … + 100² = (100 * 101 * 201)/6
- 1 + 1/3² + 1/5² + … + 1/101² = π²/8

```
(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 (+ a 2) b))))
```


A quick sidebar

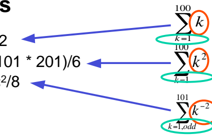
```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))
```

This is the same as:

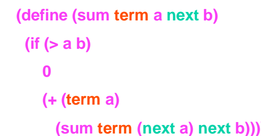
```
(define sum-integers
  (lambda (a b)
    (if (> a b)
        0
        (+ a (sum-integers (+ 1 a) b)))))
```


Other common patterns

- 1 + 2 + … + 100 = (100 * 101)/2
- 1 + 4 + 9 + … + 100² = (100 * 101 * 201)/6
- 1 + 1/3² + 1/5² + … + 1/101² = π²/8



```
(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 (+ a 2) b))))
```



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


Let's check this new procedure out!

```
(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

procedure

procedure

procedure

Higher order procedures

- A higher order procedure: takes a procedure as an argument or returns one as a value

```
(define (sum-integers1 a b)
      (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (sum-squares1 a b)
      (sum square a (lambda (x) (+ x 1)) b))

(define (pi-sum1 a b)
    (sum (lambda (x) (/ 1 (square x))) a
         (lambda (x) (+ x 2)) b))
```

Be careful

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

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

(define (sum-integers1 a b)
     (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
```

6.001 SICP 19/41

Let's check to be sure

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

```
(sum-integers 1 4)

(sum (lambda (x) x) 1 (lambda (x) (+ x 1)) 4)

(if (> 1 4)
  0
  (+ ((lambda (x) x) 1)
     (sum (lambda (x) x)
          ((lambda (x) (+ x 1)) 1)
          (lambda (x) (+ x 1))
          4)))

(+ 1 (sum (lambda (x) x) 2 (lambda (x) (+ x 1)) 4))

(+ 1 (+ 2 (sum (lambda (x) x) 3 (lambda (x) (+ x 1)) 4)))
```

6.001 SICP 20/41

Computing derivatives

$$f : x \rightarrow x^2 \qquad f : x \rightarrow x^3$$
$$Df : x \rightarrow 2x \qquad Df : x \rightarrow 3x^2$$

We can easily write f in either case:

(define f (lambda (x) (* x x x)))

But what is D??

6.001 SICP 21/41

Computing derivatives

$$f : x \rightarrow x^2 \qquad f : x \rightarrow x^3$$
$$Df : x \rightarrow 2x \qquad Df : x \rightarrow 3x^2$$

But what is D??

- maps a function (or procedure) to a different function
- here is a good approximation:

$$Df(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

6.001 SICP 22/41

Computing derivatives

$$f : x \rightarrow x^2 \qquad Df : x \rightarrow 2x$$

$$Df(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                   epsilon)) ))
```

(number → number) → (number→ number)

6.001 SICP 23/41

Using "deriv"

```
(define square (lambda (y) (* y y)) )
(define epsilon 0.001)
```

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                   epsilon)) ))
```

```
((deriv square) 5)

( (lambda (x) (/ (- ((lambda (y) (* y y)) (+ x epsilon) )
                    ((lambda (y) (* y y)) x) ) )
                 epsilon))
  5 )

(/ (- ((lambda (y) (* y y)) (+ 5 epsilon) )
      ((lambda (y) (* y y)) 5) ) )
   epsilon))

10.001
```

6.001 SICP 24/41

`deriv` composes

```
(define (cube x) (* x x x))

(define epsilon 0.001)

(define (deriv f)
  (lambda (x)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))

((deriv cube) 3)
>> 27.009000999996147
```

$Dx^3 = 3x^2$

```
((deriv (deriv cube)) 3)
>> 18.006000004788802
```

$D^2x^3 = D3x^2 = 6x$

Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (adjoin (proc (first lst))
              (map proc (rest lst)))))

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

Transforms a list to a list, replacing each value by the procedure applied to that value

Common Pattern #2: Filtering a List

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (first lst))
         (adjoin (first lst)
                 (filter pred (rest lst))))
        (else (filter pred (rest lst)))))

(filter even? (list 1 2 3 4 5 6))
;Value: (2 4 6)
```


Common Pattern #3: Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
         (add-up (rest lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
         (mult-all (rest lst)))))

(define (FOLD-RIGHT op init lst)
  (if (null? lst)
      init
      (op (first lst)
          (fold-right op init (rest lst)))))

(define (add-up lst)
  (fold-right + 0 lst))
```


Using common patterns over data structures

- We can more compactly capture our earlier ideas about common patterns using these general procedures.
- Suppose we want to compute a particular kind of summation:

$$\sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta)$$

Using common patterns over data structures

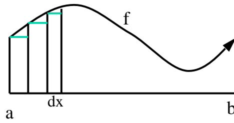
```
(define (generate-interval a b)
  (if (> a b)
      nil
      (cons a (generate-interval (+ 1 a) b))))

(define (sum f start inc terms)
  (fold-right + 0
    (map (lambda (x) (f (+ start (* x inc))))
         (generate-interval 0 terms))))
```

$$\sum_{i=0}^{n} f(a+i\delta)$$

Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))




```
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* (sum f a delta n) delta)))
(define atan (lambda (a)
              (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```


Finding fixed points of functions

Square root of x is defined by $\sqrt{x} = x / \sqrt{x}$

Think of as a transformation $f : y \rightarrow \frac{x}{y}$ then if we can find a $y = \sqrt{x}$, then $f(y) = y$, and such a y is called a fixed point of f.

- Here's a common way of finding fixed points
  - Given a guess x_1, let new guess by $f(x_1)$
  - Keep computing f of last guess, till close enough

```
(define (close? u v)   (< (abs (- u v)) 0.0001))
(define (fixed-point f i-guess)
   ;; (number->number, number) -> number
   (define (try g)
     ;; number -> number
     (if (close? (f g) g)
          (f g)
          (try (f g))))
   (try i-guess))
```


Using fixed points

(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1) → 1.6180

or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

Unfortunately if we try (sqrt 2), this oscillates between 1, 2, 1, 2,

So damp out the oscillation

```
(define average-damp
  (lambda (f)
    (lambda (x)
      (average x (f x)))))
```

Check out the type:

(number → number) → (number → number)

that is, this takes a procedure as input, and returns a **NEW** procedure as output!!! (… just like `deriv`)

- ((average-damp square) 10)
- ((lambda (x) (average x (square x))) 10)
- (average 10 (square 10))
- 55

… which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
    (average-damp
      (lambda (y) (/ x y)))
    1))
```

Compare this to Heron's algorithm in textbook – same process, but ideas intertwined with code

```
(define (cbrt x)
  (fixed-point
    (average-damp
      (lambda (y) (/ x (square y))))
    1))
```


Higher order procedures

- A higher order procedure:
  takes a procedure as an argument or returns one as a value

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
    (hop1 square 3)
    (+ 2 (square (+ 3 1))
    (+ 2 (square 4)
    (+ 2 (* 4 4))
    (+ 2 16)
    18

    (hop1 (lambda (x) (* x x)) 3)
    ...
    18
```

Type of hop1

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
```

- (number → number), number → number

1st arg must be a procedure

2nd arg must be a number

result is a number

A more interesting higher-order procedure

```
(define (compose f g) (lambda (x) (f (g x))))
((compose square double) 3)
(square (double 3))
(square (* 3 2))
(square 6)
(* 6 6)
36
```

What is the type of compose? Is it:

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

No! Nothing in compose requires a number

Compose works on other types too

```
(define (compose f g) (lambda (x) (f (g x))))
((compose
 (lambda (p) (if p "hi" "bye"))   boolean → string
 (lambda (x) (> x 0)))            number → boolean
 -5                               number
 ) ==> "bye"                      result: a string
```

Will any call to compose work? No!

```
((compose < square) 5)    wrong number of args to <
    <: number, number → boolean
((compose square double) "hi")
                          wrong type of arg to double
    double: number → number
```


Type of compose

```
(define compose (lambda (f g)
                 (lambda (x) (f (g x)))))
```

- Use type variables.

compose: (A → B), (C → A) → (C → B)

- Meaning of type variables:
  All places where a given type variable appears must match when you fill in the actual operand types

- The constraints are:
  - F and G must be functions of one argument
  - the argument type of G matches the type of X, C
  - the argument type of F matches the result type of G, A
  - the result type of compose is the function that maps the type of X, C, to the result type of F, B: C → B

Higher order procedures

- Procedures may be passed in as arguments
- Procedures may be returned as values
- Procedures may be used as parts of data structures
  - E.g., `(cons (lambda (x) x) +)` is perfectly legit

- Procedures are first class objects in Scheme!!