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 does not directly check types of arguments as part of its processing.

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

- Since 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
```

- The type of the integer-add procedure is

```
number, number → number
```

two arguments, both numbers

result value of integer-add is a number

- Addition is not defined for strings

7

Type examples

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

8

Types, precisely

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

    all procedures, whose result is a number,
    that also 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

9

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))
```

10

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`

11

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

12

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

13

Other common patterns

- 1 + 2 + … + 100 $\sum_{k=1}^{100} k$
- 1 + 4 + 9 + … + 100^2 $\sum_{k=1}^{100} k^2$
- 1 + $1/3^2$ + $1/5^2$ + … + $1/101^2$ (= $\pi^2/8$) $\sum_{k=1,odd}^{101} k^{-2}$

```
(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))))
```

14

Let's examine this new 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?

(num → num, num, num → num, num) → num

1. What type is the output?
2. How many arguments?
3. What type is each argument?

Is deducing types mindless, or what?

15

Let's examine this new 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?

Is deducing types mindless, or what?

16

Higher order procedures

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

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

17

Higher order procedures

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

18

Higher order procedures

```
(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))))

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

19

Higher order procedures

- 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 (add1 x) (+ x 1))

(define (sum-squares1 a b)(sum square a add1 b))

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

(define (add2 x) (+ x 2))

(define (pi-sum1 a b)
    (sum (lambda (x) (/ 1 (square x))) a add2 b))
```

20

Returning A Procedure As A Value

```
(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))

 incrementby: # → (# → #)

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

21

Returning A Procedure As A Value

```
(define incrementby
     (lambda(n)                    ))

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

      (lambda (x) (+ x 2))

(incrementby 2) → a procedure of one var (x) that
                  increments x by 2

((incrementby 3) 4) → ?

( (lambda(x)(+ x 3))  4) →
```

22

Returning A Procedure As A Value

```
(define incrementby
     (lambda(n)(lambda (x) (+ x n))))

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

      (lambda (x) (+ x 2))

(incrementby 2) →

((incrementby 3) 4) → ?

( (lambda(x)(+ x 3))  4) →
```

23

Nano-Quiz/Lecture Problem

```
(define incrementby
      (lambda(n)(lambda (x) (+ x n))))

(define f1 (incrementby 6)) → ?

(define f1 (lambda (x)(incrementby 6))) → ?
```

24

Nano-Quiz/Lecture Problem

```
(define incrementby
     (lambda(n)(lambda (x) (+ x n))))

(define f1 (incrementby 6)) → ?

(f1 4) →


(define f2 (lambda (x)(incrementby 6))) → ?

(f2 4)      → ?
((f2 4) 6) → ?
```

25

Procedures as values: Derivatives

$$f : x \to x^2 \qquad f : x \to x^3$$

$$f' : x \to 2x \qquad f' : x \to 3x^2$$

- Taking the derivative is a function: $D(f) = f'$
- What is its *type?*

D: (# → #) → (# → #)

26

Computing derivatives

- A good approximation:

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

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

(number → number) → (number→ number)

27

Using "deriv"

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

((deriv square) 5)
```

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

28

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)))))
```

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

```
(define (square-list lst)
  (map square lst))
(square-list (list 1 2 3 4)) → ?

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

29

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)
```

30/7

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)))))
```

31

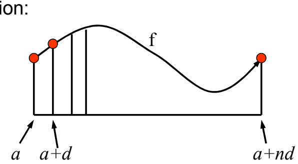
Common Pattern #3: Accumulating Results

```
(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))
```

32

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:



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

33

Using common patterns over data structures

```
(define (generate-interval a b)
  (if (> a b)
      nil
      (cons a (generate-interval (+ 1 a) b))))
(generate-interval 0 6) → ?

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

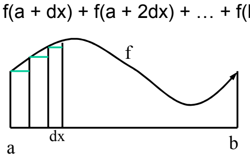
$$\sum_{i=0}^{n} f(a+i\delta)$$

34

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)))
    (* delta (sum f a delta n))))
```

35

Computing Integrals

```
(define (integral f a b n)
   (let ((delta (/ (- b a) n)))
     (* (sum f a delta n) delta)))
```

$$\int_0^a \frac{1}{1+x^2}dx = ?$$

```
(define atan (lambda (a)
  (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```

36

Procedures as arguments: a more complex example

- (define compose (lambda (f g 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

Procedures as arguments: a more complex example

- (define compose (lambda (f g 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

Compose works on other types too

(define compose (lambda (f g 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?

(compose < square 5)

wrong *number of args* to <

<: number, number → boolean

(compose square double "hi")

wrong *type of arg* to double

double: number → number

Compose works on other types too

(define compose (lambda (f g 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?

(compose < square 5)

(compose square double "hi")

Type of compose

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

- Use type variables.

  compose: (B → C), (A → B), A → C

- 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
  - the argument type of F matches the result type of G
  - the result type of compose is the result type of F

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

- Procedures are first class objects in Scheme!!