6.001 SICP

• Today’s topics
  • Procedural abstractions
  • Capturing patterns across procedures – Higher Order Procedures

What is procedure abstraction?

Capture a common pattern

\( (* 2 2) \)
\( (* 57 57) \)
\( (* k k) \)

\( \text{(lambda} (x) (* x x)) \)

Actual pattern

Formal parameter for pattern

Give it a name \( \text{(define square} (\text{lambda} (x) (* x x))) \)

Note the type: number → number

Other common patterns

• 1 + 2 + … + 100
• 1 + 4 + 9 + … + 100²
• 1 + 1/3² + 1/5² + … + 1/10!² (≈ ¼/8)

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

\( \sum = \frac{1}{1^2} + \frac{1}{3^2} + \frac{1}{5^2} + \ldots + \frac{1}{101^2} \approx \frac{\pi^2}{8} \)

Let’s examine this new procedure

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

What is the type of this procedure?

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

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

Is deducing types mindless, or what?

Higher order procedures

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

\( \text{(define sum-integers a b)} \)
\( \text{(if} (> a b) \)
\( \text{0} \)
\( \text{(sum-integers (} [+ 1 a] b))) \)

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

\( \text{(define sum-integers1 a b)} \)
\( \text{(sum (lambda} (x) x) a (lambda} (x) (x + 1)) b))) \)

\( \text{(define sum-squares1 a b)} \)
\( \text{(sum square a (lambda} (x) (x + 1)) b)) \)
Higher order procedures

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

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

\[
\text{(define (pi-sum1 a b)} \newline\text{ (sum (lambda (x) (/ 1 (square x))) a \newline\text{ (lambda (x) (+ x 2)) b))}
\]

\[
\text{(define (sum-integers1 a b)} \newline\text{ (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))}
\]

\[
\text{(define (sum-squares1 a b)} \newline\text{ (sum square a (lambda (x) (+ x 1)) b))}
\]

\[
\text{(define (pi-sum1 a b)} \newline\text{ (sum (lambda (x) (/ 1 (square x))) a \newline\text{ (add2 x) b))}
\]

\[
\text{Returning A Procedure As A Value}
\]

\[
\text{(define (add1 x) (+ x 1))}
\]

\[
\text{(define (add2 x) (+ x 2))}
\]

\[
\text{(define incrementby (lambda (n) \ldots \ldots ))}
\]

\[
\text{(define add1 (incrementby 1))}
\]

\[
\text{(define add2 (incrementby 2))}
\]

\[
\text{(define add37.5 (incrementby 37.5))}
\]

\[
\text{incrementby: \# \rightarrow (# \rightarrow \#)}
\]

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

\[
\text{Nano-Quiz/Lecture Problem}
\]

\[
\text{(define incrementby (lambda (n) (lambda (x) (+ x n)))))}
\]

\[
\text{(define f1 (incrementby 6)) \rightarrow ?}
\]

\[
\text{(f1 4) \rightarrow}
\]

\[
\text{(define f2 (lambda (x) (incrementby 6))) \rightarrow ?}
\]

\[
\text{(f2 4) \rightarrow ?}
\]

\[
\text{((f2 4) 6) \rightarrow ?}
\]

\[
\text{Procedures as values: Derivatives}
\]

\[
f : x \rightarrow x^2 \quad f' : x \rightarrow 2x
\]

\[
f : x \rightarrow x^3 \quad f'' : x \rightarrow 3x^2
\]

\[
\text{• Taking the derivative is a function: } D(f) = f' \quad \text{• What is its type?}
\]

\[
D: (\# \rightarrow \#) \rightarrow (\# \rightarrow \#)
\]
Computing derivatives

- A good approximation:

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

\[
\text{(define deriv}
  \begin{array}{l}
\lambda (f) \\
  (\lambda (x) (/ (- (f (+ x epsilon)) (f x)) epsilon))
\end{array} )
\]

Using “deriv”

\[
\text{(define square } (\lambda (y) (* y y)) )
\]

\[
\text{(define epsilon 0.001)}
\]

\[
((\text{deriv square}) 5)
\]

Common Pattern #1: Transforming a List

\[
\text{(define (square-list lst)}
  \begin{array}{l}
\text{(if (null? lst) '())}
\end{array}
\]

\[
\text{transform each value by the procedure applied to that value}
\]

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

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

(generate-interval 0 6)
```

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

Integration as a procedure

Integration under a curve $f$ is given roughly by

\[
\int_a^b f(x) \, dx \approx \sum_{i=0}^{n} f(a + i \cdot dx) \cdot dx
\]

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

Computing Integrals

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

Finding fixed points of functions

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

Think of as a transformation $y \rightarrow \frac{y}{x}$ 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 is a common way of finding fixed points
  • Given a guess $x_1$, use new guess by $f(x_1)$
  • Keep computing $f$ of last guess, till close enough

```scheme
(define (close? u v)   (< (abs (- u v)) 0.0001))
```

```scheme
(define (fixed-point f i-guess)
  (define (try g)
    (if (close? (f g) g)
      (f g)
      (try (f g))))

  (try i-guess))
```

```scheme
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1)
```

```
1.6180
```

So damp out the oscillation

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

Check out the type:

```scheme
(number ➔ number) ➔ (number ➔ number)
```

that is, this takes a procedure as input, and returns a NEW procedure as output!!!

```scheme
((average-damp 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 (the one we saw earlier) – same process, but ideas intertwined with code

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

Procedures as arguments: a more complex example

```
(define compose (lambda (f g x) (f (g x))))
(compose 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 (lambda (f g x) (f (g x))))
(compose
  (lambda (p) (if p "hi" "bye"))
  (lambda (x) (> x 0))
  -5 ) ==> "bye"
```

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
```

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!!