CS339: Abstractions and Paradigms for Programming

The Procedural Paradigm

Manas Thakur
CSE, IIT Bombay

Autumn 2024

Recall Substitution Model of Procedure Application

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
> (define (foo x y)
    (* x (+ (* x 2) y)))

> (foo 4 5)
```

```
(* 4 (+ (* 4 2) 5))

(* 4 (+ 8 5))

(* 4 13)

52
```

A “Procedural” Example

```
> (define (square x) (* x x))

> (define (sum-of-squares x y)
    (+ (square x) (square y)))

> (define (f a)
    (sum-of-squares (+ a 1) (* a 2)))
```

Solve using substitution

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (square x) (* x x))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

```
(f 5)

(sum-of-squares (+ 5 1) (* 5 2))

(+ (square (+ 5 1)) (square (* 5 2)))

(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

Could we have substituted this way?

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (square x) (* x x))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Applicative Order of Evaluation

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Evaluate arguments then apply

Solve using substitution

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (square x) (* x x))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Normal Order of Evaluation

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Fully substitute then reduce

Applicative vs Normal Orders of Evaluation

- Applicative order *Call by value* *Call by name (need)*
  - avoids redundant computation
  - takes lesser memory

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

- However, there are advantages of normal order too, and there are ways to make it more efficient (post mid-sem).
- Which one does Scheme use?
- How can you find out for any language?

Conditionals

- Absolute value of x:

$$|x| = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -x, & \text{if } x < 0 \end{cases}$$

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Clause

Predicate

Action

$$|x| = \begin{cases} -x, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases}$$

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

OR

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Predicate

Consequent

Alternate

Example: Newton's method for computing square roots

- Start with a guess and "improve" the guess until it is "good enough"
- Square root of 2:

Guess (y)	Quotient (x/y)	Average ((y+x/y)/2)
1	2/1 = 2	(1+2)/2 = 1.5
1.5	2/1.5 = 1.3333	1.4167
1.4167	1.4118	1.4142
1.4142	…	…

- Say we stop when the square of the guess is equal to the number up to three decimal places.

Example: Newton's square root [Cont.]

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

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

(define (average x y)
  (/ (+ x y) 2))

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

(define (sqrt x)
  (sqrt-iter 1.0 x))
```

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))

(define (square x)
  (* x x))
```

- Can you identify two "bad" things in this code?

Example: Newton's square root [Cont.]

```
(define (average x y)
  (/ (+ x y) 2))

(define (square x)
  (* x x))

(define (sqrt x)
  (define (improve guess)
    (average guess (/ x guess)))
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Packaged together

Namespace Abstraction

Next class:

Why did we have `sqrt-`"**`iter`**"?