Good programming practices

- Code design
- Documentation
- Debugging
- Evaluation and verification

Code layout and design

- Design of
  - Data structures
    - Natural collections of information
    - Suppression of detail from use of data
  - Procedural modules
  - Interfaces

Code layout and design

- Design of
  - Data structures
    - Natural collections of information
      - What are appropriate selectors and constructors?
        - » E.g. points in the plane (x, y), line segments as pairs of points
      - Can one naturally think about operations on constructs as a unit?
        - » E.g. translation, rotation, scaling of points and segments
    - Suppression of detail from use of data
      - Do the operations on data constructs actually depend on individual pieces?
    - In attacking a problem, try to lay out the collections of objects you will need, the relationships between them and the operations on them

Code layout and design

- Design of
  - Data structures
  - Procedural modules
    - Computation to be reused
    - Suppression of detail from use of procedure
  - Interfaces

Code layout and design

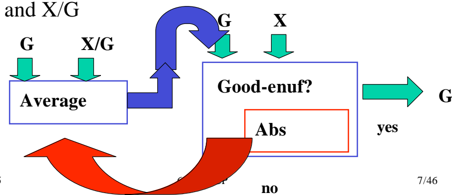
- Design of
  - Procedural modules
    - Computation to be reused
      - What computations appear to be specific to this problem?
      - What computations are likely to be used elsewhere?
    - Suppression of detail from use of procedure
      - Can you specify a contract between the input and output of a computation? If so, does this form a natural breakpoint – i.e. does anything depend on how this is done, or only on the contract?

Code layout and design

- Design of
  - Data structures
  - Procedural modules
  - Interfaces
    - “types” of inputs and outputs

An example of code modules

- Finding the sqrt of X
  - Make a guess, G
  - If it is good enough (i.e. G^2 close to X), stop
  - Otherwise, get a new guess by averaging G and X/G



Documenting code

- Supporting code maintenance
  - Can you read your code a year after writing it and still understand why you made particular design decisions?
  - Can you read your code a year after writing it and even understand what it is supposed to do?
- Identifying input/output behaviors
  - Specify expectations on input and the associated contract on output of a procedure

Documenting code

- Description of input/output behavior
- Expected or required types of arguments
- Type of returned value
- List of constraints that must be satisfied by arguments or stages of computation
- Expected state of computation at key points in code

An example of code documentation

```
(define sqrt-helper
  (lambda (X guess)
    ;;; compute approximate square root by
    ;;; successive refinement, guess is current
    ;;; approximation, X is number whose square
    ;;; root we are seeking.
    ;;; Type: (number, number) → number
   ;;; constraint: guess^2 == X
     (if (good-enuf? X guess) ; can we stop?
         guess                ; if yes, then return
         (sqrt-helper X
                      (improve X guess)
                 ; if not, then get better guess
                 ; and repeat process
           ))))
```


Debugging errors

- Common sources of errors
- Common tools to debug

Common errors

- Unbound variable
  - Cause: typo
  - Solution: search for instance
- Unbound variable
  - Cause: reference outside scope of binding
  - Solution:
    - Search for instance
    - Use debugging tools to isolate instance

The Debugger

- Places user inside state of computation at time of error
- Can step through
  - Reductions (computation reduced to a simpler expression)
  - Substitutions (computation converted to a simpler version of itself)
- Can examine bindings of variables and parameters

Debugger example

```
Lines identify stack frames, most recent first.
   Sx means frame is in subproblem number x
   Ry means frame is reduction number y
The buffer below describes the current subproblem or reduction.
-----------
The *ERROR* that started the debugger is:
  Unbound variable: bar
>S0  bar
    R0  bar
    R1  (if (= n 0) bar (+ n (foo (- n 1))))
    R2  (foo (- n 1))
 S1  (foo (- n 1))
    R0  (+ n (foo (- n 1)))
    R1  (if (= n 0) bar (+ n (foo (- n 1))))
    R2  (foo (- n 1))
 S2  (foo (- n 1))
    R0  (+ n (foo (- n 1)))
    R1  (if (= n 0) bar (+ n (foo (- n 1))))
    R2  (foo 2)
--more--
```

```
(define foo
  (lambda (n)
    (if (= n 0)
        bar
        (+ n (foo (- n 1))))))
```


Syntax errors

- Wrong number of arguments
  - Source: programming error
  - Solution: use debugger to isolate instance
- Type errors
  - As procedure
  - As arguments
    - Source: calling error
    - Solution: trace back through chain of calls

Structure errors

- Wrong initialization of parameters
- Wrong base case
- Wrong end test
- … and so on

Evaluation and verification

- Choosing good test cases
  - Pick values for input parameters at limits of legal range
    - Base case of recursive procedure
  - Pick values that span legal range of parameters
  - Pick values that reflect different kinds of input
    - Odd versus even integers
    - Empty list, versus single element list, versus many element list
- Retest prior cases after making code changes

Debugging tools

- The **ubiquitous** print/display expression
- Tracing
  - Print out values of parameters on input to a procedure(s)
  - Print out value return on exit of procedure(s)
- Stepping
  - Show the state of computation at each stage of substitution model

A debugging example

- We want to compute sines, using the mathematical approximation

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Initial code example

```
(define (sine x)
  (define (aux x n current)
    (let ((next (/ (expt x n) (fact n))))
         ;; compute next term
      (if (small-enuf? next)  ;; if small
          current ;; just return current guess
         (aux x (+ n 1) (+ current next))
           ;; otherwise, create new guess
              )))
  (aux x 1 0))
```


Test cases

```
(sine 0)   ; should be 0
;Value: 0

(sine 3.1415927)  ; should be 0
;Value: 22.140666527138016

(sine (/ 3.1415927 2.0))   ; should be 1
;Value: 3.8104481565660486
```


Chasing down the error

```
(define (sine x)
  (define (aux x n current)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 1) (+ current next)))))
  (aux x 1 0))
```


Test cases

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

```
(sine 3.1415927)
n is 1 current is 0
n is 2 current is 3.1415927
n is 3 current is 8.076395046346645
n is 4 current is 13.244108055421808
n is 5 current is 17.3028204216732
n is 6 current is 19.85298464991622
n is 7 current is 21.188247537124454
n is 8 current is 21.78751212841507
n is 9 current is 22.022842786585954
n is 10 current is 22.104988684118826
n is 11 current is 22.130795579321248
n is 12 current is 22.138166011464666
n is 13 current is 22.140095586116132
n is 14 current is 22.14056188901145
n is 15 current is 22.140666527138016
;Value: 22.140666527138016
```


Fixing the increments

```
(define (sine x)
  (define (aux x n current)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 2) (+ current next)))))
  (aux x 1 0))
```

Test cases

$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$

```
(sine 3.1415927)
n is 1 current is 0
n is 3 current is 3.1415927
n is 5 current is 8.309305709075163
n is 7 current is 10.859469937318183
n is 9 current is 11.4587345286088
n is 11 current is 11.54088042614167
n is 13 current is 11.548250858285089
n is 15 current is 11.548717161180408
;Value: 11.548717161180408
```

2/23/05 6.001 SICP 25/46

We need to alternate terms

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
               (+ n 2)
               (+ current (* addit next))
               (* addit -1)))))
  (aux x 1 0))
```

2/23/05 6.001 SICP 26/46

Test cases

```
(sine 3.1415927)
;The procedure #[compound-procedure 12 aux] has
  been called with 3 arguments; it requires
  exactly 4 arguments.
;Type D to debug error, Q to quit back to REP
  loop: q
```

2/23/05 6.001 SICP 27/46

Make sure procedure calls changed

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
               (+ n 2)
               (+ current (* addit next))
               (* addit -1)))))
  (aux x 1 0 -1))
```

2/23/05 6.001 SICP 28/46

```
(sine 3.1415927)    ; should be 0
n is 1 current is 0
n is 3 current is -3.1415927
n is 5 current is 2.026120309075164
n is 7 current is -.5240439191678563
n is 9 current is .07522067212275974
n is 11 current is -6.925225410112354e-3
n is 13 current is 4.452067333052508e-4
;Value: 4.452067333052508e-4

(sine (/ 3.1415927 2.0))   ; should be 1
n is 1 current is 0
n is 3 current is -1.57079635
n is 5 current is -.9248322238656045
n is 7 current is -1.004524855998199
n is 9 current is -.999843101378741
;Value: -.999843101378741
```

2/23/05 6.001 SICP 29/46

Make sure start off right

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 2)
               (+ current (* addit next)) (* addit -1)))))
  (aux x 1 0 1))
```

2/23/05 6.001 SICP 30/46

Test cases

```
(sine (/ 3.1415927 2.0))    ; should be 1
n is 1 current is 0
n is 3 current is 1.57079635
n is 5 current is .9248322238656045
n is 7 current is 1.004524855998199
n is 9 current is .999843101378741
;Value: .999843101378741
(sine 3.1415927)   ;; go  back and check test cases – should be 0
n is 1 current is 0
n is 3 current is 3.1415927
n is 5 current is -2.026120309075164
n is 7 current is .5240439191678563
n is 9 current is -.07522067212275974
n is 11 current is 6.925225410112354e-3
n is 13 current is -4.452067333052508e-4
;Value: -4.452067333052508e-4
(sine 0)   ;; go back and check test cases – should be 0
n is 1 current is 0
;Value: 0
```


Examine for Elegance(!)

```
(define (sine x)
  (define (aux x n current addit)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current  →  (+ current (* addit next))
          (aux x (+ n 2)
               (+ current (* addit next)) (* addit -1)))))
  (aux x 1 0 1))

(define (sine x)
  (define (aux n current addit)
    (let* ((next (/ (expt x n) (fact n)))
           (new (+ current (* addit next))))
      (if (small-enuf? next)
          new
          (aux (+ n 2) new (* addit -1)))))
  (aux 1 0 1))
```


Summary

- Display parameters to isolate errors
- Test cases to highlight errors
- Check range of test cases
- Be sure to retry test cases after corrections to ensure still are correct
- **Use these tricks and tools!**

Using types as a reasoning tool

- Types can help:
  – Planning code
  – As entry checks for debugging

Types as a planning tool

- Example: we want a procedure that repeatedly applies any procedure some number of times.

```
(define (mul a b)
   (if (= b 0)
       0
                 ; base case
       (+ a (mul a (- b 1)))))
 ; apply operation to input, and simpler version
(define (exp a b)
   (if (= b 0)
       1
                 ; base case
       (mul a (exp a (- b 1)))))
 ; apply operation to input, and simpler version
```


The use of repeated

```
(define mul
     (lambda (a b)
        ((repeated (lambda (x) (+ x a)) b) 0)))

(define exp
     (lambda (a b)
        ((repeated (lambda (x) (mul x a)) b) 1)))
```

Base case

Operation to repeat

Number of times to repeat

Types help us design repeated

- What is the type of `repeated`?

```
(define mul
    (lambda (a b)
       ((repeated (lambda (x) (+ x a)) b ) 0)))
```

[(A→A), Integer] → (A → A)

Designing repeated

```
(define (repeated proc n)
   (if (= n 0)
       [ ??? ]
       [?? repeated ??(- n 1) ??]))
```


Designing repeated

[(A→A), Integer] → (A → A)

```
(define (repeated proc n)
   (if (= n 0)
       (lambda (x) x)
       [?? repeated ??(- n 1) ??]))
```

```
(define mul
    (lambda (a b)
       ((repeated (lambda (x) (+ x a)) b ) 0)))
```


Designing repeated

[(A→A), Integer] → (A → A)

```
(define (repeated proc n)
   (if (= n 0)
       (lambda (x) x)
       (lambda (x)
         ?? repeated ?? (- n 1) ??)))
```


Designing repeated

[(A→A), Integer] → (A → A)

```
(define (repeated proc n)
   (if (= n 0)
       (lambda (x) x)
       (lambda (x)
         (proc
           ?? (repeated proc (- n 1)))) ))
```


Designing repeated

[(A→A), Integer] → (A → A)

```
(define (repeated proc n)
   (if (= n 0)
       (lambda (x) x)
       (lambda (x)
         (proc
           ((repeated proc (- n 1)) x)))))
```

Types as a debugging tool

- Check types of arguments on entry to ensure they meet specifications
- Check types of values returned to ensure they meet specifications
- (possibly) check constraints on values

An example of type checking

```
(define sqrt-helper
  (lambda (X guess)
    ;;; compute approximate square root by
    ;;; successive refinement, guess is current
    ;;; approximation, X is number whose square
    ;;; root we are seeking.
    ;;; Type: (number, number) → number
    ;;; constraint: guess^2 == X
    (if (or (not (number? X))
            (not (number? Guess)))
        (error "report this somehow")
        (if (good-enuf? X guess)
            guess
           (sqrt-helper X (improve X guess))))))
```


An example of type checking

```
(define sqrt-helper
  (lambda (X guess)
    ;;; compute approximate square root by
    ;;; successive refinement, guess is current
    ;;; approximation, X is number whose square
    ;;; root we are seeking.
    ;;; Type: (number, number) → number
    (if (not (>= x 0))
        (error "Not a positive number")
        (if (or (not (number? X))
                (not (number? Guess)))
            (error "report this somehow")
            (if (good-enuf? X guess)
                guess
               (sqrt-helper X
                       (improve X guess)))))))
```


Good programming practices

- Code design
- Documentation
- Debugging
- Evaluation and verification