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

Code layout and design

- Design of
  - Data structures
  - Procedural modules
    - "types" of inputs and outputs

In attacking a problem, try to lay out the collections of objects you will need, the relationships between them and the operations on them.
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

```
(define sqrt-helper
  (lambda (X guess)
    (if (good-enuf? X guess)
        guess
        (sqrt-helper X (improve X guess))))
```

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

```
;;; 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

```
(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

```scheme
(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
        current ;; just return current guess
        (aux x (+ n 1) (+ current next)))
    ;; otherwise, create new guess))
  (aux x 1 0))
```

Test cases

- \((\text{sine 0})\) ; should be 0
  ;Value: 0
- \((\text{sine 3.1415927})\) ; should be 0
  ;Value: 22.140666527138016
- \((\text{sine} (/ 3.1415927 2.0))\) ; should be 1
  ;Value: 3.8104481565660486

Chasing down the error

```scheme
(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) ;; if small current
        current ;; just return current guess
        (aux x (+ n 1) (+ current next)))
    ;; otherwise, create new guess)))
  (aux x 1 0))
```

Test cases

- \((\text{sine 3.1415927})\)
  n is 1 current is 0.1415927
  n is 2 current is 0.076395046436845
  n is 3 current is 0.032682042167352
  n is 4 current is 0.01207562128641507
  n is 5 current is 0.0046231105512596
  n is 6 current is 0.00182924645951622
  n is 7 current is 0.0007333437003057
  n is 8 current is 0.0002982285007953
  n is 9 current is 0.0001197243099117
  n is 10 current is 0.0000489108813805
  n is 11 current is 0.00001958168950145
  n is 12 current is 0.0000077564733642
  n is 13 current is 0.0000031026973451
  n is 14 current is 0.0000012401668900
  n is 15 current is 0.0000005000665271
  ;Value: 22.140666527138016

Fixing the increments

```scheme
(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) ;; if small current
        current ;; just return current guess
        (aux x (+ n 2) (+ current next)))
    ;; otherwise, create new guess)))
  (aux x 1 0))
```
Test cases

(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))

(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 .07522067212356525e-4
n is 11 current is -6.92522541012354e-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 -.445206733052508e-4
n is 9 current is -.99984101378741
;Value: -.99984101378741

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))

Test cases

(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))

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))

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))
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 3.1415927
n is 7 current is 4.7123889
n is 9 current is 6.28318531
;Value: 6.28318531

(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 -4.452067333052508e-4
n is 9 current is 6.925225410112354e-3
n is 11 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
          (aux x (+ n 2) (+ current (* addit next)) (* addit -1))))
  (aux x 1 0 1))

(define (sine x)
  (define (aux x 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 x 0 0))

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)))
Types help us design repeated

• What is the type of \texttt{repeated}?

\begin{verbatim}
(define mul
  (lambda (a b)
    ((repeated (lambda (x) (+ x a)) b ) 0)))
\end{verbatim}

Designing repeated

\begin{verbatim}
[(A\rightarrow A), Integer] \rightarrow (A \rightarrow A)
\end{verbatim}

\begin{verbatim}
(define (repeated proc n)
  (if (= n 0)
      lambda (x) x
    (proc
      ?? (repeated proc (- n 1)))))
\end{verbatim}

Designing repeated

\begin{verbatim}
[(A\rightarrow A), Integer] \rightarrow (A \rightarrow A)
\end{verbatim}

\begin{verbatim}
(define (repeated proc n)
  (if (= n 0)
      lambda (x) x
    (lambda (x)
      (proc
        ((repeated proc (- n 1)) x)))))
\end{verbatim}
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
    (if (not (number? X))
      (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