Good programming practices

- Why good design matters
- Code design
  - Behavioral spec
  - Data structures
  - Procedures
- Documentation
- Debugging
- Using types to isolate parameters
- Evaluation and verification

Old-school computer science called for methodical coding practices to ensure that the large computers used by banks, governments and scientists wouldn't break.

...Mr. Allechin craved discipline in code writing.

Good Design Matters

- Because you’ll never get big projects to work.
- Because they’ll become moribund: unfixable, unmodifiable.

Code Design: Behavior

- Figure out what it’s supposed to do.
- Example: Basic personal calendar manager
  - Capabilities?

Code Design: Data

- What basic objects are in this world?
- What are the relations among them?
Code Design: Data

- What basic objects are in this world?
  - dates, times, appointments
- What are the relations among them?
  - date with associated appointments
  - appointments with indication of date, time
  - Both?
- What operations on those objects?

Code Design: Data

- Data structure design
  - appointment?
    - Constructor
      - make-aptmt(date, time, place...)
    - Accessors
      - apptmt-date, apptmt-time...
    - Contract
      - (apptmt-date(make apptmt ...)) =
    - Operations
      - print-aptmt, move-aptmt, cancel-aptmt...

Code Design: Procedures

- Computation to be reused
  - What computations appear to be specific to this problem?
  - What computations are likely to be used elsewhere?

Code Design: Test Cases

- Write the test cases first
  - Helps you anticipate the tricky parts
  - Encourages you to write a general solution
  - As code is developed, keep retesting on test suite to ensure no accidental introduction of bugs

Code Design: Test Cases

- Choosing good test cases
  - Pick a few obvious values
  - Pick values at limits of legal range
    - Base case of recursive procedure
    - Pick values that span legal range
    - Pick values that reflect different kinds of input
      - Odd versus even integers
      - Empty list, versus single element list, versus many element list

Coding Style

- Write so it’s clear first, fast second
- Write so it’s clear first and second, fast third…
- Why
  - Code reading vs. code running
  - Moore’s Law
    - The computer, the desk lamp, and the speed of light

Moore’s Law is the empirical observation that the transistor density of integrated circuits, with respect to minimum component cost, doubles every 24 months[1]. It is attributed to Gordon E. Moore[2], a co-founder of Intel.
Coding Style

• Write so it’s clear first, fast second
• Write so it’s clear first and second, fast third...
• Why
  – Code reading vs. code running
  – Moore’s Law
  – The computer, the desk lamp, and the speed of light
    (11,871,781,320.96 inches/sec)

Documenting code

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

An example of code documentation
(define sqrt-helper
 (lambda (X guess)
   (if (good-enuf? X guess) ; can we stop?
     guess
     (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

Syntax errors

• Wrong number of arguments
  – Source: programming error
  – Solution: use debugger to isolate instance
• Type errors
  – As procedure \((5 \times 3)\)
  – As arguments
    • Source: calling error
    • Solution: trace back through chain of calls

Conceptual errors

• Wrong initialization of parameters
• Wrong base case
• Wrong end test
  • \ldots and so on

Evaluation and verification

• Test individual modules
• Retest prior cases after making code changes (regression testing)

Debugging tools

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

- Change language level to “Intermediate Student with Lambda”, press RUN.
- Input expression, press Stepper button.

Tracing in Dr. Scheme

- Change language level to “Essentials of Programming Languages”, press RUN.
- Add to top of definition window:
  `(require (lib "trace.ss"))`
- Indicate which function(s) to trace:
  `(trace fact)`

Debugging

- 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))

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 (* addit next))))
 (aux x 1 0))

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))
Make sure procedure calls change

```scheme
(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 ))
```

```scheme
(sine 3.1415927)   ; should be 0
n is 1 current is 0
n is 3 current is 2.2415927
n is 5 current is 2.026120309075164
n is 7 current is -.5240439191678563
n is 9 current is .07522067212273874
n is 11 current is -6.92523941011359e-3
n is 13 current is 4.452067333525058e-4
;Value: 4.452067333525058e-4

(sine (/ 3.1415927 2.0))  ; should be 1
n is 1 current is 0
n is 3 current is .57079635
n is 5 current is -.924322236658045
n is 7 current is -1.0443249395999199
n is 9 current is -.9984310137841
;Value: -.9984310137841
```

Make sure to start off right

```scheme
(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 ))
```

```scheme
(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 .924832236658045
n is 7 current is 1.0045248539998199
n is 9 current is .999843101378741
;Value: .999843101378741

(sine 3.1415927 2.0)  ; should be 0
n is 1 current is 0
n is 3 current is 2.2415927
n is 5 current is 2.026120309075164
n is 7 current is -.5240439191678563
n is 9 current is .07522067212273874
n is 11 current is -6.92523941011359e-3
n is 13 current is 4.452067333525058e-4
;Value: 4.452067333525058e-4

(sine 0)   ;; go back and check test cases – should be 0
n is 1 current is 0
;Value: 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!

Types

```scheme
(+ 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\langle A,B \rangle
  - 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\langle number, number \rangle
- List\langle A \rangle=Pair\langle A, List\langle A \rangle \rangle 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\langle number \rangle; while (list 1 "string" 3) has type List\langle number or string \rangle

Examples

25 ; Number
3.45 ; Number
“this is a string”; String
(> a b) ; Boolean
(cons 1 3) ; Pair\langle Number, Number \rangle
(list 1 2 3) ; List\langle Number \rangle
(cons “foo” (cons “bar” ‘())) ; List\langle String \rangle

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 $\rightarrow$ to indicate that the arguments are mapped to the return value
  - E.g. number $\rightarrow$ number specifies a procedure that takes a number as input, and returns a number as value

Types

- (+ 5 10) $\Rightarrow$ 15
  - 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 $\rightarrow$ number
  - two arguments, both numbers
  - result value of integer-add is a number
- Addition is not defined for strings

Type examples

- expression: evaluates to a value of type:
  15 number
  "hi" string
  square number $\rightarrow$ number
  (> 5 4) $\Rightarrow$ #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
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

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))

Summary of types

- 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

Using types as a reasoning tool

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

smallexpt(n): for n between 1 and 4, return a function that raises its argument to that power

(define (smallexpt n)
  (cond ((= n 1) x)
        ((= n 2) (* x x))
        ((= n 3) (* x x x))
        ((= n 4) (* x x x x))
        (else (error "invalid input"))))

Intended type of smallexpt?
  number → (number → number)

New kind of “beast” – a procedure that returns a procedure as output!!

(define (smallexpt n)
  (cond ((= n 1) (lambda(x) x))
        ((= n 2) (lambda(x) (* x x)))
        ((= n 3) (lambda(x) (* x x x)))
        ((= n 4) (lambda(x) (* x x x x)))
        (else (error "invalid input"))))

number → (number → number)
Types as a debugging tool

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

An example of type checking

```scheme
(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 (not (>= x 0))
          (error "Not a positive number")
        (if (good-enuf? X guess)
            guess
          (sqrt-helper X (improve X guess))))))
```

Good programming practices

- Code design
- Documentation
- Debugging
- Evaluation and verification