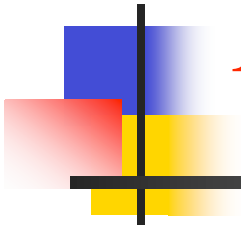


Simple Programs, Reduction, the Recipe, and Errors





Course Overview

- **Data-driven program design** 2-17
- Abstraction and good design 19-23
- Algorithms 25-28
- Accumulators 30-32
- Side effects 34-37
- Mutable structures, and objects 39-43

- Design Recipes
 - Introduced and used throughout the course



Today's Goals

- Numbers as examples of basic types
- Primitive operations on numbers
- Rules for reducing programs
- Simple programs
 - = Variables
 - + Function definitions
 - + Function application
- The design recipe
- Errors



Numbers as basic types

Used in math and in computing:

- Naturals: $0, 1, 2, \dots$ // “Number theory”
- Integers: $-1, -12, \dots$ // include negatives
- Rational numbers: $(3/4), (-5/6), \dots$
- Inexact numbers: `#i0.123`

For all but the last kind, Scheme computes exact results

COMP 490 project: What about the real numbers?



Primitive Computation

- Most languages provide basic operations on values of basic types:
 - `+`, `-`, `*`, `sqrt`, `expt`, `remainder`, `log`, `sin`
- Primitive computation = application of a basic operation
 - Basic operation = Basic function
 - Soon, we will see how to define our own (non-primitive) functions
- Function application
 - Syntax varies from language to language. In most languages, it is mixed.
 - Most languages use infix notation for math, and prefix notation in general.
- Scheme uses prefix notation uniformly for **everything**
 - `(+ A B)`, `(sqrt A)`, `(remainder A B)`
 - Bigger example: `(* (+ 1 2) (+ 3 4))`
 - How does this compare to writing `1+2*3+4` ?
- Scheme syntax keeps things simple and avoids possible ambiguity



Reduction for primitive functions

- A reduction = a computational “atom”
 - The smallest step of computation
- Example

$$(* (+ 1 2) (+ 3 4))$$

$$= (* 3 (+ 3 4))$$

$$= (* 3 7)$$

$$= 21 \quad \text{Goes left to right (why do we care?)}$$

- The following is **not** an atomic step, and so **not** a reduction

$$(- (+ 1 3) (+ 1 3))$$

$$= 0$$



Simple Programs

- Variables are simply names for values
 - `pi`, `my-SSN`, `album-name`, `tax-rate`, `x`
- Function definitions
 - `(define (area-of-box x) (* x x))`
 - `(define (half x) (/ x 2))`
- Function applications (just as we saw before)
 - `(area-of-box 2)`
 - `(half (area-of-box 3))`

Almost **all** functions can be written this way



Reductions for defined functions

- Assume we declared the two functions
 - `(define (area-of-box x) (* x x))`
 - `(define (half x) (/ x 2))`
- Then Scheme can perform these reductions

```
(half (area-of-box 3)) ←
= (half (* 3 3))
= (half 9) ←
= (/ 9 2)
= 4.5
```
- Reduction stops when we get to a value or an error



The Design Recipe

How should I go about writing programs?

1. Analyze problem and define data types
2. State contract and purpose for function
3. Give examples of function use and result
4. Write the function itself
5. Test it, and record actual results of tests

The order of the steps of the recipe is important



Example: Area of disk

```
;; Contract: area-of-ring : number number -> number Step 2
;; Purpose: To compute the area of a ring whose radius is
;;           outer and whose hole has a radius of inner
;; Examples: (area-of-ring 5 3) should produce 50.24 Step 3
;;              (area-of-ring 5 0) should produce 78.5
;; Definition: [refines steps 1-4] Step 4
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
;; Tests: Step 5
"Testing area-of-ring:" ;; Help your grader :)
(equal? (area-of-ring 5 3) 50.24)
(equal? (area-of-ring 5 0) 78.5)
;; ... and other examples
```

Note: Don't use `equal?` or strings in **Definition** yet! Use it only in **Tests** .



The Design Recipe (Big Picture)

- Encourages systematic problem solving
- Works best if keep our functions small
- We will learn how to decompose problems
 - “to decompose a problem”
= “to structure our understanding of a problem”
- Solution structure follows that of problem
 - This is part of datatype-driven program design



Syntax Errors

- A syntactically correct **program** can be
 - An atom, like
 - a number 17 , 4.5 , $\#i0.34$,
 - a variable *radius*, or
 - A compound **program**,
 - starting with (
 - followed by operator
 - then one or more **programs**
 - and, finally, ending with)
- Syntax errors:
 - $3)$, $(3 + 4)$, $(+ 3$, $)+($, ...



Runtime Errors

- Happen when operators are “surprised”
- Consider the following examples:
 - `(sqrt 1 2 3 4)` ;; syntax error
 - `(18 17)` ;; syntax error
 - `(/ 1 0)` ;; runtime error
 - `(+ 1 "a")` ;; runtime error
- Try things like that in DrScheme, and make a mental note of the error messages you get back.



Logistics

- New homework is posted online
 - Sign up for mailing list to get any updates, discussions
 - Make absolutely sure you follow the **recipe**
 - Partners: Talk to people after class, at lab, etc.
- Go to lab at any time this week (sign up there)
- Next online quiz:
 - Will cover Chapters 1, 2, 3
 - We'll let you know when the new system is up