Primitives; function and data definitions

Prof. Robert "Corky" Cartwright
Department of Computer Science
Rice University

Course Overview

- Functional program design in Scheme
  - Data-directed (functional) program design 2-12
  - Algorithm design 13-15
  - Applied functional programming 16-18
- Object-oriented (OO) program design in Java 19-45
  - …

Today’s Goals

- Common basic types
- Common primitive operations
- Rules for reducing programs
- Simple programs

  = Variable definitions

  + Function definitions
- The design recipe
- Errors
- Data definitions

Basic (primitive) types of data

numbers:

- naturals: `0, 1, 2, …` // number theory in mathematics
- integers: `…, -1, 0, 1, …` // include negatives
- rational numbers: `3/4, 0, -1/3, …` // include fractions
- inexact numbers: `#i0.123, #i0, …` // floating point numbers

Operations: +, -, *, /, expt, remainder

Scheme computes exact answers on exact inputs when possible

booleans: false, true

Operations: not, and, or, …

Symbols: `'A, 'a, 'Aa, 'Corky, …`

Operations: … // none important for now

Other basic types: strings, vectors , … // none important for now

Mixed-type Operations and Primitive Computation

- Basic relational operators
  - equal?   // all data values
  - =, <, >, <=, >= // only on numbers
- Primitive computation = application of a basic operation to constants
  - Basic operation = basic function
  - Soon, we will see how to define our own (non-primitive) functions
- Function application in Scheme: parenthesized prefix notation
  - Scheme uses parenthesized prefix notation uniformly for **everything**
  - `(+ 2 2), (sqrt 25), (remainder 7 3)`
  - Bigger example: `(* (+ 1 2) (+ 3 4))`
  - How does this compare to writing `1+2*3+4` ?
- Scheme syntax is simple, uniform, and avoids possible ambiguity

Computation is repeated reduction

- *Every Scheme program execution is the evaluation of a given expression constructed from primitive or defined functions and variables (constants).*
- *Evaluation proceeds by repeatedly performing the leftmost possible reduction (simplification) until the resulting expression is a **value.***
- *A **value** is any constant*. We will identify all of the expressions that are values as we explicate the language. Numbers, booleans, symbols are all values.

Reduction for primitive functions

- A *reduction* is an atomic computational step that replaces some expression by a simpler expression as specified by a Scheme evaluation rule (law). Every application of a basic operation to values yields a value (where run-time error is a special kind of value).
- Example

```
    (* (+ 1 2) (+ 3 4))
=> (reduces to) (* 3 (+ 3 4))
=>  (* 3 7) =>  21
```

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

```
    (- (+ 1 3) (+ 1 3))  =  0
```

Programs = Variable Definitions + Function Definitions

- Variables are simply names for values
  - `pi, my-SSN, album-name, tax-rate, x`
- Variable definitions
  - `(define freezing 32)`
  - `(define boiling 212)`
- 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 **any** function f used in a program can be written in the form
  - `(define (f v1 … vn)` *`<expression>`*`)`

where `<`*`expression`*`>` is constructed from constants, variables, function applications, and a few other constructs TBN.

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 any requisite data types
2. State contract (type) and purpose for *function* that solves the problem
3. Give examples of function use and result
4. Select a template for the function body
5. Write the function itself
6. Test it, and confirm that tests succeeded

The order of the steps of the recipe is important

Example: Area of ring

```
;; 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 :)
(check-expect (area-of-ring 5 3) 50.24)  ; reports error if not equal
(check-expect (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 repeatedly decompose problems into simpler problems until we reach problems that can be solved by simple expressions like we for `area-of-ring`
- Decomposition driven by structure of data being processed: *data-directed* design

Syntax Errors

- A syntactically correct **expression** can be
  - An *atomic* expression, like
    - a number `17, 4.5, #i0.34`
    - a variable `radius`
  - A *compound* **expression**,
    - starting with `(`
    - followed by basic or program-defined operation such as `+` or `f`
    - one or more **expression**s separated by spaces
    - ending with)
- Syntax errors:
  - `3)` , `(3 + 4)` , `(+ 3` , `)+(` , …

Runtime Errors

- Happen when basic operations are applied with manifestly illegal arguments
- 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.

Simple Data Definitions

- How do we define new forms of data in Scheme? For example, say we want to write a program for the registrar that maintains a directory of courses that can be searched …
- Problem description
  - "… Each university course will have an associated department and course numbers, as well as a class size. ...
- Data definition

```
;; A course is a structure (make-course dept num size)
;; where dept is a symbol, and num and size are numbers
(define-struct course (dept num size))
```

- Scheme processes this definition by creating the following operations:
  - *constructor:* `make-course`,
  - *accessors:* `course-dept`, `course-num`, `course-size`
  - *recognizer:* `course?`

Creating and Using Structures

- Syntax for creating a structure:

```
(define this-class (make-course 'COMP 211 41))
```

- A structure (a constructor applied to values) is a value (and hence is *not* reducible)
  - It's big. But it's just like `1, true, or 'Rabbit`
  - It's big. But it is NOT a reducible expression, like `(+ 1 2)`
- Syntax for extracting fields
  - `(course-dept this-class)`
    `(course-num this-class)`
- Reduction for field access

```
  (course-dept (make-course 'COMP 210 50))
⇒ 'COMP
```

- Notes:
  - `(make-course 'COMP 210 50)` is a value
  - `(make-course 'COMP 210 size)` is *not* a value (why not?)
  - `(make-course 'COMP 210 (+ 25 25))` is *not* a value (why not?)

Reminders

- New homework (HW1) is posted online
  - Due next Wednesday, so you will get to check it over in lab; don't wait until your lab to get started.
  - Sign up for mailing list to get any updates, discussions
  - Make absolutely sure you follow the **recipe** in writing Scheme programs.
  - Partners: Talk to people after class, at lab, etc.
  - Follow format of examples posted on the wiki in writing hand evaluations.
  - Submit your assignment using `svn` (the command line name for *subversion)*

Next Lecture

- Continue digesting chs. 1-10 in HTDP

- Next class
  - *Inductive* data definitions
  - Conditionals
  - Amplified design recipe