Racket Primitives and Function Definitions

Robert “Corky” Cartwright
Department of Computer Science
Rice University

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, sqrt

Racket computes exact answers on exact inputs when possible

Booleans: #false, #true // true => #true false => #false

Operations: not, and, or, …

Symbols: ‘A, ‘a, ‘Aa, ‘Corky, … // prefix quote marks: Racket!

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 Racket: 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` ?
- Racket syntax is simple, uniform, and avoids possible ambiguity

Computation is repeated reduction

- *Every Racket 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 the textual representation of 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 Racket 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 reduction of expression built from primitive functions

```
      (* (+ 1 2) (+ 3 4))
=>   (* 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
```

It is an equivalence in the transitive closure of reduction. (Every value reduced to itself.)

Programs = Variable Definitions + Function Definitions

- Variables are simply names for values; a few are predefined
  - **`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 to be covered in next lecture.

Reductions for defined functions

- Assume we defined the two functions

```
(define (area-of-box x) (* x x))
(define (half x) (/ x 2))
```

- Then Racket 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 including examples
2. State type contract and purpose for *function(s)* that solve the problem
3. Give examples of function use and result
4. Select and instantiate a template for the function body; many are *degenerate*
5. Write the function itself
6. Test it, and confirm that tests succeeded

The ordering of the steps of the recipe is important

Example: Solve quadratic equation

```
;; Type Contract: solve-quadratic: number number number -> number      Step 2
;; Purpose: (solve-quadratic a b c) finds the larger root of
   a*x*x + b*x + c = 0 given it has real roots and a != 0

;; Examples: (solve-quadratic 1 0 -25) = 5                        Step 3
;;           (solve-quadratic 5 0 -20) = 2
;;           (solve-quadratic 1 -10 25) = -4
;;           . . . and other examples

;; Template instantiation:  (degenerate)                          Step 4
;; (define (solve-quadratic a b c) ... )

;; Code                                                           Step 5
(define (solve-quadratic a b c)
   (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)))

;; Tests for solve-quadratic                                      Step 6
   (check-expect (solve-quadratic 1 0 -25) 5)
   (check-expect (solve-quadratic 5 0 -20) 2)
   (check-expect (solve-quadratic 1 -10 25) 5)
```

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 as in **`solve-quadratic`**
- 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 `b`
    - one or more **expression**s separated by spaces
    - ending with `)`
- Syntax errors:
  - `3) , (3 + 4) , (+ 3 , )+( ,` …
- Compound expressions:
  - `(+ 3 4) , (first (list 1 2 3))`

Runtime Errors

- Happen when basic operations are applied with manifestly illegal arguments
- Consider the following examples in Racket:
  - `(sqrt 1 2 3 4)` => `sqrt: expects only 1 argument, but found 4`
  - `(/ 1 0)` => `/: division by zero`
  - `(+ 1 ‘a)` => `+: expects a number as 2nd argument, given 'a`

  Racket prints error results in red. In hand evaluations (perhaps created using an editor) you can write use the prefix **ERROR** instead, e.g.,
  - `(/ 1 0)` => **ERROR** `/: division by zero`

  Your manually generated description of the error does not have to match Racket exactly: a paraphrase such as the following is fine:

  `(sqrt 1 2 3 4)` => **ERROR**`: wrong number of arguments to sqrt`

- Try examples in DrRacket

Reminders

- New homework (HW1) is posted online
  - Due next Tuesday, so you will get to check it over in lab; don't wait until your lab to get started.
  - Make absolutely sure you follow the **recipe** in writing Racket programs.
  - Follow format of examples posted on the wiki in writing hand evaluations.
  - Submit your assignment using **`svn`** (the command line name for *subversion)*

Epilog

- Reminder: continue digesting chs. 1-10 in HTDP
  Sections 8.3 and 9.4 are particularly important and they are not wordy.
- Next class (but read about them first and use them in working on HW1)
  - Most important primitive form of data: *lists*
  - *Data* definitions including self-reference (*recursive* data definitions)
  - Conditionals
  - Amplified design recipe supporting function definitions that use *recursion*
- Optional Challenge problem: What happens if we use rightmost reduction instead of leftmost? Can you devise an expression composed from primitive operations covered in this lecture such that standard Racket leftmost evaluation produces a different visible result than rightmost reduction produces (in terms the result generated by the computation). Hint: focus on pathological behavior. Not all pathological behavior is the same in terms of what Racket reports. Your solution does not have to an expression that a competent programmer would actually write. (Competent programmers do not write programs that contain pathological expressions.) Incentive: 20 extra credit points for hand-written example including step-by-step evaluations. Send your solution by email to comp311@rice.edu by 11:59 pm Friday.