Racket Primitives and Function Definitions

Robert “Corky” Cartwright
Department of Computer Science
Rice University

Today's Goals

- Common basic data values and types
- Common primitive operations
- Rules for reducing programs
- Simple program =

  Variable definitions + Function definitions + Expression
- 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
- primitive 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: // none important for now

Mixed-type Operations and Basic Computation

- Basic relational operators
  - `equal?` // well-defined for all data **values**
  - `=, <, >, <=, >=` // well-defined only on numbers (other **values** generate // runtime errors
- Primitive computation = application of a basic operation to constants
  - Basic operation = basic function (**if** not classified as basic function)
  - Soon, we will see how to define our own 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`** (common math notation)?
- 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 canonical textual representation of a constant (ignoring functions and lazy lists). We will identify all of the expressions that are values as we explicate the language. All basic Racket constants including numbers, booleans, symbols are **value**s.

Reduction for basic 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 a run-time error is a special kind of value that aborts a computation).
- Example reduction of expression built from basic functions

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

- Racket computation always performs **leftmost** reduction. If the leftmost pending reduction is ill-formed, Racket aborts reporting an error.
- The following is **not** an atomic step, and so **not** a reduction

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

  It is a pair in the transitive closure of the reduction relation (Note: every value reduces to itself!) It is **not** a valid reduction step because the operand expressions are **not** **value**s.
- All the rules for reducing the applications of primitive functions (excluding `if`) require **value**s as operands.

Program = Variable and Function Definitions + Subject Expression

- Variables are simply names for values; a few like **`true`** 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, basic function applications, and the applications of a few other non-basic but primitive functions that will 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 we go about writing programs?

1. Analyze problem and define any requisite data types; show examples for each type.
2. State type signatures and purpose [behavioral contract] [for all *function(s)* in the problem solution.
3. Give examples of function use and result.
4. Select and instantiate a template for the function body; many are *degenerate.* The data type of primary argument determines the form of the template instantiation.
5. Write the function itself by filling in the template instantiation.
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
;; Behavioral Contract: (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) = 5
;;           . . . 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 a primitive (such as **`+`** or **`-`**) or program defined function
    - one or more **expression**s separated by spaces (the operands)
    - ending with **`)`**
- Syntax errors:
  - **`3) , (3 + 4) , (+ 3 , )+( , …`**
- Compound expressions:
  - **`(+ 3 4) , (not x)`**

Runtime Errors

- Errors occur when basic operations are applied to manifestly illegal arguments (or wrong number of 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 (on GitHub Classroom) is posted online and distributed via Canvas
  - Due next W Sept 9 at 11:59 pm, easy for students who have previously used GitHub Classroom.
- Play with DrRacket
  - Extra credit exercise (+ 10pts):
    The **`expt`** (exponentiation) operation is basic but try defining your own implementation called **`exp`** that works only on natural numbers. Don't worry about templates or template instantiation.
    - Write a simple brute force definition for **`exp`** that is a good mathematical definition but slow when executed.
    - Write an efficient definition named **`fast-exp`** based on the binary representation of the exponent.
    - You obviously need to use recursion for both parts. Follow the design recipe but you should omit an inductive definition of the natural numbers and the template instantiation for **`fast-exp`**, which does not use simple structural recursion.
    - Send your Racket file that is a solution by email to cork@rice.edu by 11:59pm on Fri Sept 9.

Epilog

- Reference: chs. 1-10 in HTDP
  Sections 8.3 and 9.4 are particularly important and they are not wordy.
- Next class (read about them first; we will use them in 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*