3. Introduction to Racket

Robert Snapp
snapp@cs.uvm.edu

Department of Computer Science
University of Vermont

July 13, 2014

Introduction to *DrRacket*

DrRacket is a self-contained programming environment based on the racket programming language, which is an extension of scheme. Racket and scheme are programming languages that are well suited for manipulating symbols. Hence, the can readily simulate puzzles and games.

- *DrRacket* is free! (`www.racket-lang.org`).
- *DrRacket* is fun, (but not always easy).
- *DrRacket* is powerful.

Using DrRacket as a super calculator

We are accustomed to adding numbers using *infix* notation, e.g.,

$$1 + 2 + 3 + 4 + 5 = 15.$$

Racket uses parentheses and *prefix* notation, This means the operator comes before every one of its arguments:

`(+ 1 2 3 4 5)` $\Rightarrow$ `15`

Prefix notation is generally more efficient: one $+$ instead of four. Operations can be nested, e.g.,

`(* (+ 3 4) (- 9 6))` $\Rightarrow$ `21`

(The above is equivalent to $(3 + 4) \cdot (9 - 6) = 21.$)

Simple data types

DrRacket uses several different types of objects:

- Integers: e.g., `3`, `0`, `-12`. Integer arithmetic uses *arbitrary precision*.
- Fractions: e.g., `2/3`, `-5/6`.
- Floating point numbers: e.g., `2.5`, `-0.00303`
- Complex numbers: e.g., `0.0+1.0i` represents $i = \sqrt{-1}$, and x`+`y`i` represents $x + i\,y$.

Other *data types* include

- Strings: `"abc"`, `"Abc"`
- Characters: `#\a`, `#\b`, `#\c`
- Functions: `+`, `-`, `*`, `/`, `car`, `cdr`
- Boolean (true or false values): `#t`, `#f`.

Built-in numericalfunctions

DrRacket has as many built-in functions as do pocket calculators:

- Square root: `(sqrt` n`)` $\Rightarrow \sqrt{x}$, e.g., `(sqrt 9)` $\Rightarrow$ `3`
- Exponents: `(expt` x y`)` $\Rightarrow x^y$, e.g., `(expt 2 3)` $\Rightarrow$ `8`
- Exponentials: `(exp` x`)` $\Rightarrow e^x$, e.g., `(exp 2)` $\Rightarrow$ `7.38905609893065`
- Natural Logarithm: `(log` x`)` $\Rightarrow \log_e x$, e.g., `(log 2)` $\Rightarrow$ `0.6931471805599453`
- Trig functions: (argument must be in radians)
  - `(sin` x`)` $\Rightarrow \sin x$
  - `(cos` x`)` $\Rightarrow \cos x$
  - `(tan` x`)` $\Rightarrow \tan x$
  - `(asin` x`)` $\Rightarrow \arcsin x$
  - `(acos` x`)` $\Rightarrow \arccos x$
  - `(atan` x`)` $\Rightarrow \arctan x$
  - `(atan` x y`)` $\Rightarrow \arctan(x/y)$

Defining New Functions

It is easy to define new functions in DrRacket.

```
(define (add1 x)
     (+ x 1))
```

`define` is a special keyword that indicates that we are defining a new function. `x` is a dummy variable that represents a single argument.
Once `add1` has been evaluated (or executed), it can be used:

- `(add1 7)` ⇒ 8
- `(add1 -0.5)` ⇒ `0.5`
- `(add1 (add1 7))` ⇒ 9

`define` can also be used to define new symbols or variables:

```
(define my-name "Robert")
(define golden-mean (/ (+ (sqrt 5) 1) 2))
```

Alternate function definition, using `lambda`

Recall our `add1` function:

```
(define (add1 x)
      (+ x 1))
```

The following is equivalent

```
(define add1
      (lambda (x)
          (+ x 1)))
```

`lambda` is a special keyword that is used to define an *anonymous* function.

Defining a factorial function

```
(define 1! 1)
(define 2! (* 2 1!))
(define 3! (* 3 2!))
(define 4! (* 4 3!))
(define 5! (* 5 4!))
      ⋮
(define 100! (* 100 99!))
      ⋮
```

This seems tedious. Is there an easier way?

Yes there is. But we will need to learn two new concepts:

1. Conditional evaluation
2. Recursion

Conditions

Conditions are expressions that evaluate to either *true* (#t) or *false* (#f). A condition is thus like the answer to a yes-no question.

Racket has a number of built in conditions that return either true or false.

```
(null? '()) ⇒ #t
(null? '(1)) ⇒ #f
(string? "abc") ⇒ #t
(number? 3.145) ⇒ #t
(even? 4) ⇒ #t
(even? 5) ⇒ #f
(odd? 3) ⇒ #t
(odd? 4) ⇒ #f
(= 0 (- 3 3)) ⇒ #t
(> 5 3) ⇒ #t
(>= 5 5) ⇒ #t
(< -2 3) ⇒ #t
```

Conditional evaluation

Suppose we want to define a new function that adds 1 to all numbers except for zero. The special form `if` can be used:

```
(if condition
    true-expr
    false-expr)
```

Here, *condition* is an expression that evaluates to a true or false value. *true-expr* represents any scheme expression that should be evaluated only if the *condition* returns `#t`. *false-expr* is a scheme expression that is evaluated if *condition* returns `#f`.

```
(define (newadd1 x)
     (if (= x 0)
         x
         (+ x 1)))
```

Conditional evaluation using `cond`

The `cond` function allows one create branches with multiple options:

```
(cond (condition_1 expr_1)
      (condition_2 expr_2)
             ⋮
      (condition_n expr_n)
      (else        else-expr))
```

The `else` line is optional.

Recursive evaluation

Like a snake that swallows its own tail, a function in scheme can call itself:

```
(define (factorial n)
     (if (= n 0)
          1
          (* n (factorial (- n 1)))))
```

- (factorial 0) $\Rightarrow$ 1
- (factorial 1) $\Rightarrow$ 1
- (factorial 2) $\Rightarrow$ 2
- (factorial 3) $\Rightarrow$ 6
- (factorial 4) $\Rightarrow$ 24