3. Introduction to Racket

Robert Snapp
snapp@cs.uvm.edu

Department of Computer Science
University of Vermont

July 13, 2014
1 DrRacket

2 Simple data types

3 Functions
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](http://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 numerical functions**

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

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

Recall our \texttt{add1} function:

\begin{verbatim}
(define (add1 x)
  (+ x 1))
\end{verbatim}

The following is equivalent

\begin{verbatim}
(define add1
  (\lambda (x)
    (+ x 1)))
\end{verbatim}

\texttt{lambda} is a special keyword that is used to define an \textit{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.

\[
\begin{align*}
(null? ~(\)) & \Rightarrow #t \\
(null? ~(1)) & \Rightarrow #f \\
(string? "abc") & \Rightarrow #t \\
(number? 3.145) & \Rightarrow #t \\
(even? 4) & \Rightarrow #t \\
(even? 5) & \Rightarrow #f \\
(odd? 3) & \Rightarrow #t \\
(odd? 4) & \Rightarrow #f \\
(= 0 (- 3 3)) & \Rightarrow #t \\
(> 5 3) & \Rightarrow #t \\
(>= 5 5) & \Rightarrow #t \\
(< -2 3) & \Rightarrow #t
\end{align*}
\]
Conditional evaluation

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

\begin{verbatim}
(if condition
  true-expr
  false-expr)
\end{verbatim}

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

\begin{verbatim}
(define (newadd1 x)
  (if (= x 0)
      x
      (+ x 1)))
\end{verbatim}
Conditional evaluation using cond

The cond function allows one create branches with multiple options:

\[
\text{(cond (condition\_1 expr\_1)} \\
\quad \text{(condition\_2 expr\_2)} \\
\ldots \\
\quad \text{(condition\_n expr\_n)} \\
\text{(else else-expr))}
\]

The else line is optional.
Recursive evaluation

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

\[
\text{(define (factorial n)}
\begin{align*}
& (\text{if} \ (= \ n \ 0) \\
& \quad 1 \\
& \quad (* \ n \ (\text{factorial} \ (- \ n \ 1)))
\end{align*}
\)
\]

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