Racket Language Tutorial – The Basics

DOWNLOAD DRRACKET: Download from this link -- https://download.racket-lang.org/releases/8.2/

Make sure DrRacket launches. You'll see a screen similar to that shown in Figure 1. The top half of the screen is for writing programs, which are then executed using the "Run" button. The lower half is where you will work in this tutorial. The lower half executes an interactive read-eval-print loop. Whatever is typed there is executed as soon as the user hits the "Enter" key.

Under the "Language" tab, select "Choose Language" and select "The Racket Language."

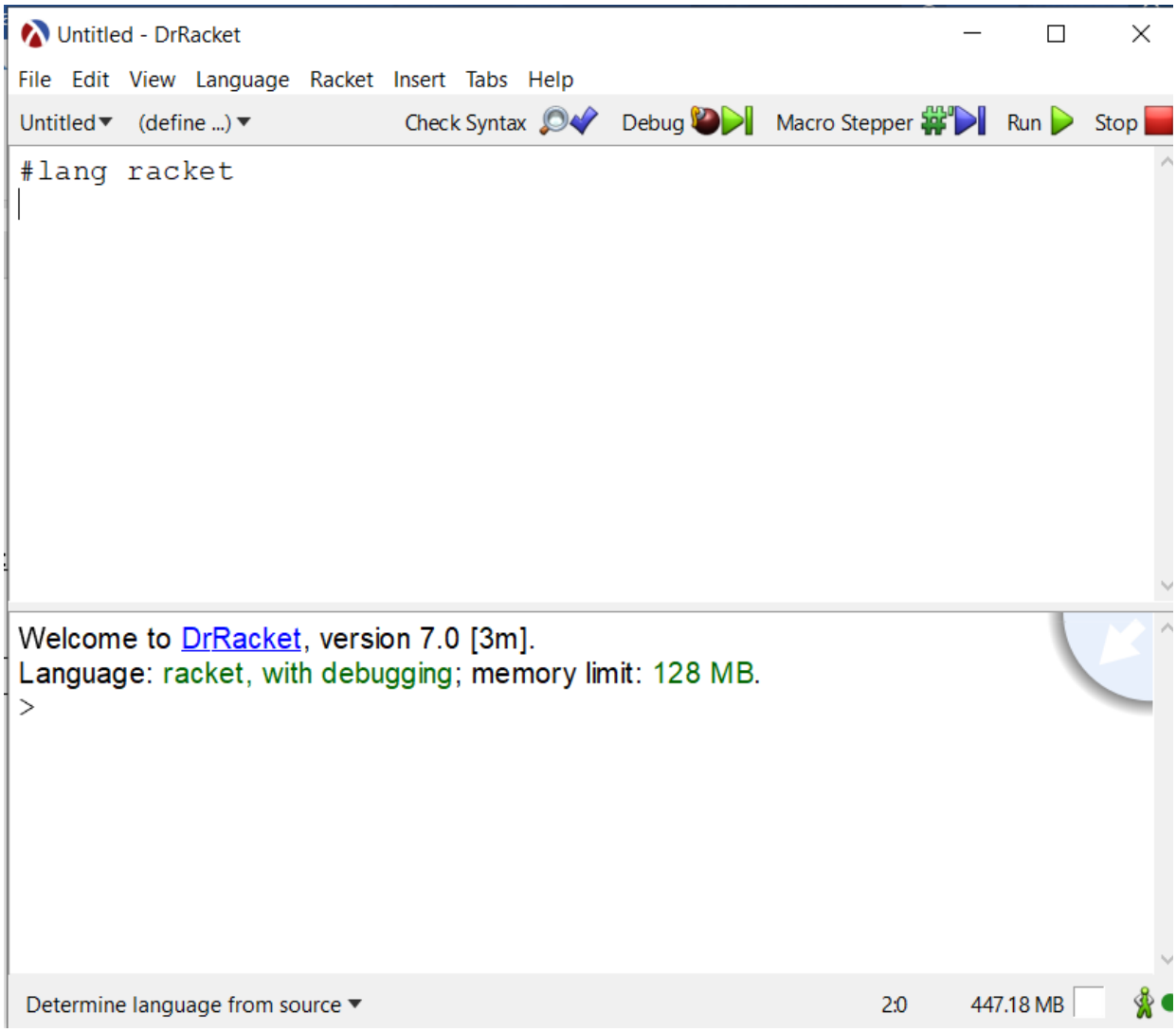

Figure 1: Dr Racket Screenshot, Version 7.0 Shown

CONSTANTS AND VARIABLES: In the bottom half of the screen type the following:

- An integer value + Enter (Racket echoes back the integer value)
- A letter of the alphabet + Enter (Racket says variable d is not defined)

- ‘a + Enter (Racket echoes back the literal character value of ‘a)
- A word, like dog, + Enter (Racket says variable not defined)
- A word in quotes, like “Dog” + Enter (Racket echoes back the string)

Summary: Racket distinguishes between primitive values (integers, numeric values, characters preceded by a carat, and strings in quotes) and variables (characters and strings without carats or quotes).

OPERATORS AND OPERANDS: Remember that the Lambda Calculus uses *application* to apply operators (functions) to operands. In Racket, this leads to *prefix* notation, as opposed to the infix notation used in other programming languages. For example “2 + 5” in infix notation, becomes “+ 2 5” in prefix notation. The prefix notation uses application to apply the plus operator (function), +, to the inputs 2 and 5. Type the following in Racket:

- (+ 2 5) + Enter (Racket prints the result of the operation, 7)
- + 2 5 + Enter (Racket echoes back that you’ve entered a procedure, the value 2 and the value 5) Note: The parentheses tells Racket to *apply* the plus operator to the operands 2 and 5. If you want anything to happen, use parentheses.
- (+ 2 2 2 2) + Enter (Racket prints the result of the operation, 8) Note the convenience of the prefix notation to apply the operator to numerous operators.
- (* 2 2 2 2) + Enter (Racket prints 16)
- (/ 2 2 2 2) + Enter (Racket prints ¼)
- (+ (* 2 4) (+ 6 -17)) + Enter (Racket prints -3)
- Play around with this!

DEFINITIONS: Racket would be of little use if everything needed to be typed out every time. The ability to *bind* values to variables solves that problem. Type the following into Racket:

- (Define x 10) + Enter (Racket provides no response)
- x + Enter (Racket echoes back the value bound to x → 10
- (define y (+ x 15)) + Enter (Racket provides no response)
- y + Enter (Racket echoes back the value bound to y → 25
- (define x 50) + Enter (Racket provides no response)
- x + Enter (Racket echoes back the value bound to x → 50)
- y + Enter (Racket echoes back the value bound to y → 25) (Why not 65?)
- Play with this!

FUNCTION DEFINITION: Functions are defined in the typical lambda calculus form. Type the following into Racket:

- (lambda (n) (+ n 1)) + Enter (Racket echoes back that this is a procedure)
- ((lambda (n) (+ n 1)) 5) + Enter (Racket applies the function to 5 and returns 6)
- (define add1 (lambda (n) (+ n 1))) + Enter (Racket provides no response)

- (add1 5) + Enter (Racket applies the add1 function to the operand 5 and returns 6)
- (add1 (add1 5)) + Enter (Racket applies the add1 function to the operand 5 twice, returning 7) Note: This is function composition.
- Play with this!

LISTS: Racket is derived from Scheme, which is derived from LISP. LISP stand for LISt Processing, and lists are the basic data structure used in all three of these languages. The two basic list operations are *car* and *cdr.* The *car* operator returns the first element of a list, and the *cdr* operator returns the list with the first element removed. Type the following in Racket:

- (list (1 2 3 4)) + Enter (Racket provides an error message)
- (list '(1 2 3 4)) + Enter (Racket echoes back the list (1 2 3 4))
- (define l1 '(1 2 3 4)) + Enter (Racket provides no response)
- l1 + Enter (Racket echoes back the list l1 = (1 2 3 4)
- (car l1) + Enter (Racket applies the car operator to l1, and returns the first element, 1)
- (cdr l1 + Enter (Racket applies the cdr operator to the list l1, and returns (2 3 4))
- (append l1 l1) + Enter (Racket appends l1 to itself and returns (1 2 3 4 1 2 3 4))
- (reverse l1) + Enter (Racket echoes back the reverse of l1 = (4 3 2 1)
- Play with it!

This tutorial has provided the basics of the Racket language. We'll go over more in class.

Discussion Questions:

1. Is definition assignment?
2. Does (reverse l1) change l1? What is it doing?
3. Lists are similar to 1-dimensional arrays (more properly called vectors, by the way) in an imperative language. How might you make the equivalent 2-dimensional arrays?
4. It is possible to create a function like *(lambda foo (op x))* that takes in a function (op) and an operand (x). Is this different from other languages that you've worked with? Are there advantages to this?