

Module 9

Functional Programming

Paradigm

CS 332 Organization of Programming Languages
Embry-Riddle Aeronautical University
Daytona Beach, FL

Mog Outcomes

- Download, install and run the DrRacket software
- Perform basic math operations in Racket using prefix notation
- Bind value to variables using the define function
- Create and manipulate lists
- Create user defined functions and compose functions
- Perform If-Then-Else operations using Racket syntax
- Perform repetition via recursion using Racket Syntax
- Demonstrate correspondence between λ calculus and Racket

Introduction / Background

- All Turing complete programming languages must ...
 - Define variables
 - Bind values to those variables *
 - Choose which statements execute through If-Then-Else features
 - Repeat statements (through iteration or recursion or both)
 - Produce output
- This tutorial addresses each of the above, in roughly that order.

* Evaluating expressions to produce a value is included in this.

Topic List

- Racket: background, downloading the software, GUI layout
- Basic math operations, prefix notation, defining variables, binding
- Lists: Creating and manipulating
- Functions: Creating and applying to input (relation to λ calculus)
- Selection: If-Then-Else syntax in Racket
- Repetition: Recursion in Racket, local definitions using Let and LetRec
- Sample programs, thinking functionally

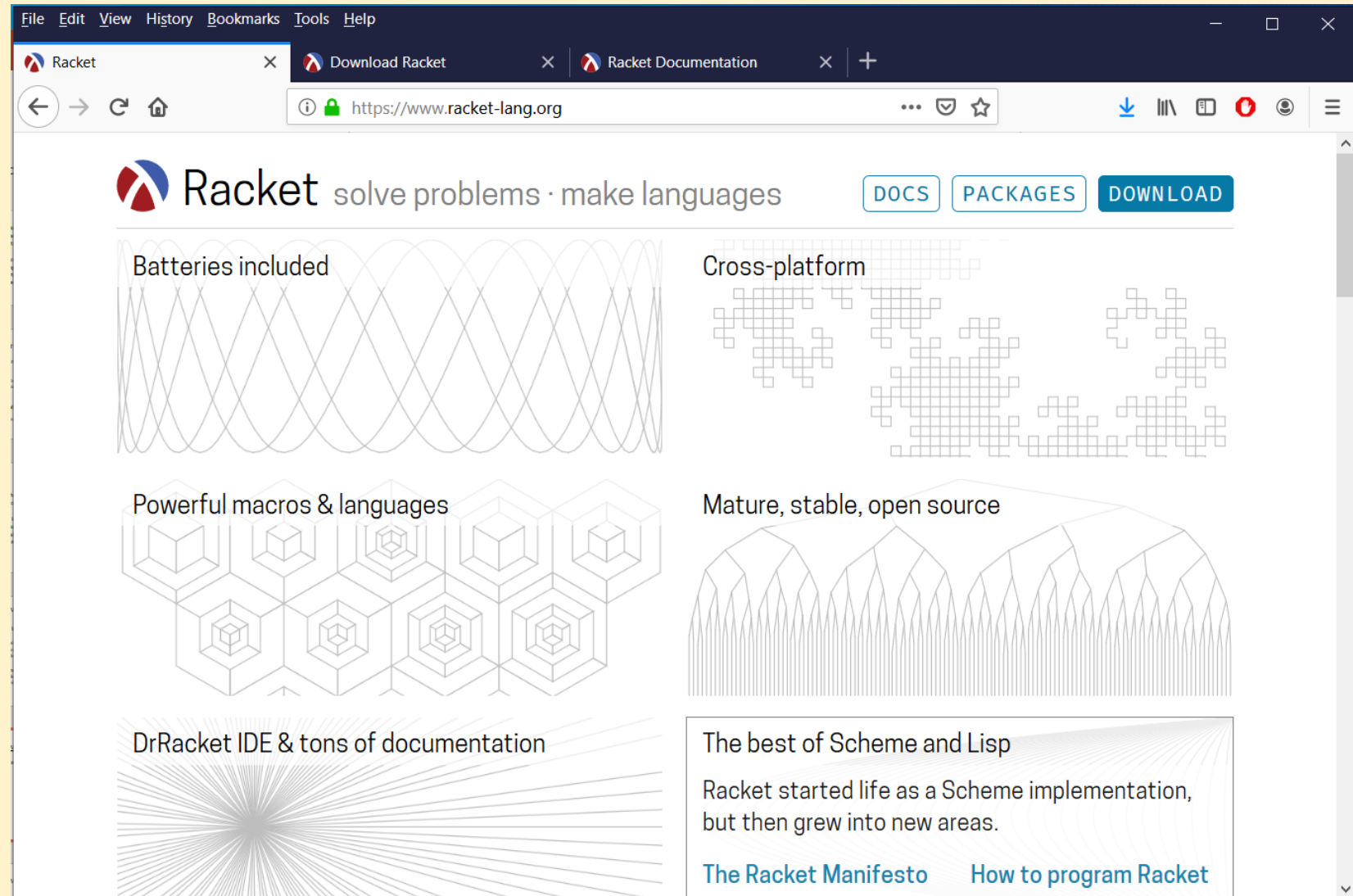
Racket: Background, Downloading, GUI layout

- Racket is a functional programming language.
- It is a descendant of LISP
 - LISP was developed in 1958 – functional programming is old!
 - LISP \rightarrow LISt Processing: data is represented as lists, not arrays.
 - LISP begat Scheme, Scheme begat Racket
- Functional programming syntax corresponds to the λ calculus
 - Everything is an expression, including lists (sequences of expressions)
 - $(\text{exp}_1 \text{ exp}_2) \rightarrow \text{exp}_1$ is a function, applied to its input, exp_2

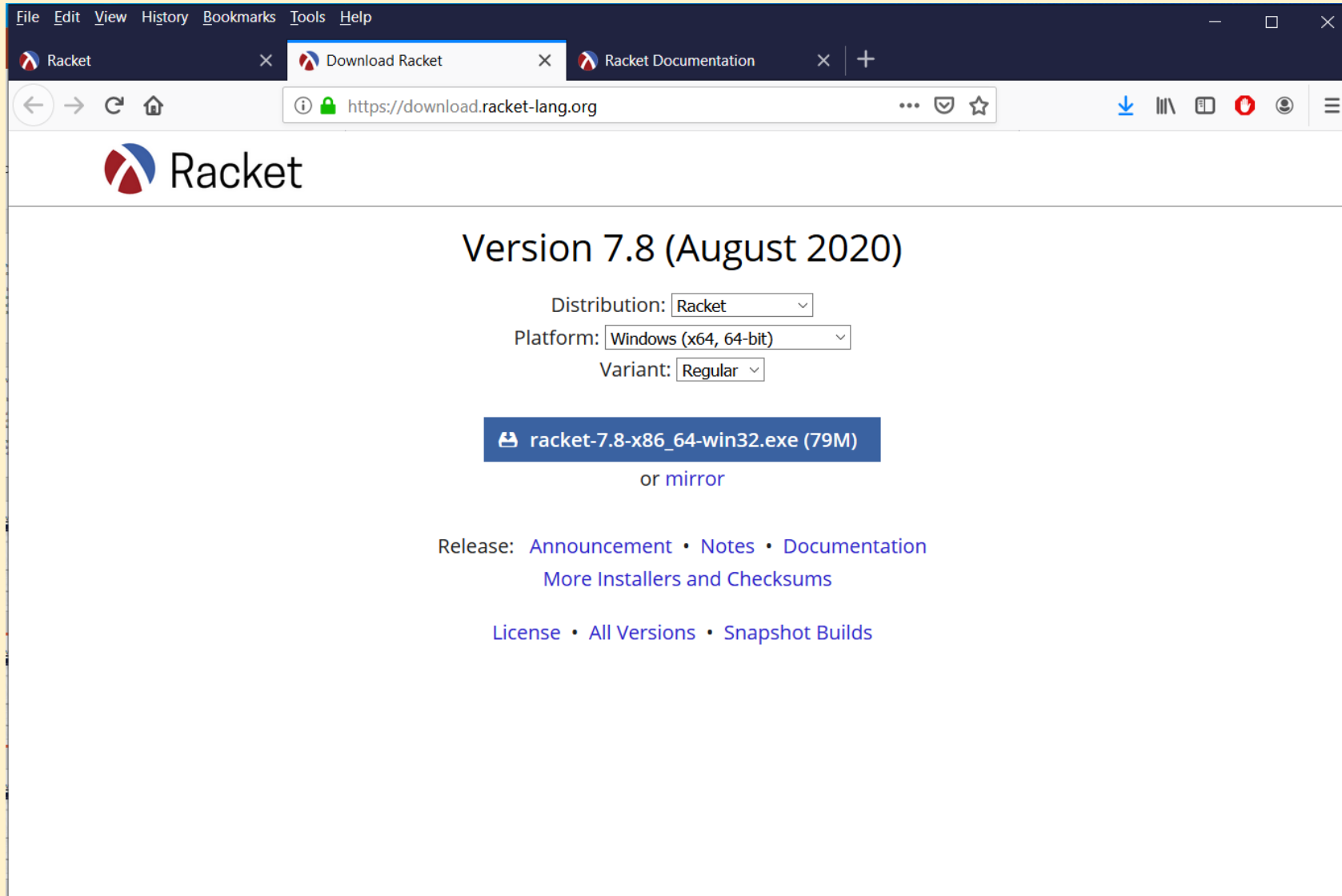
Racket: Background, Downloading, GUI layout

DrRacket is the software used to create Racket programs

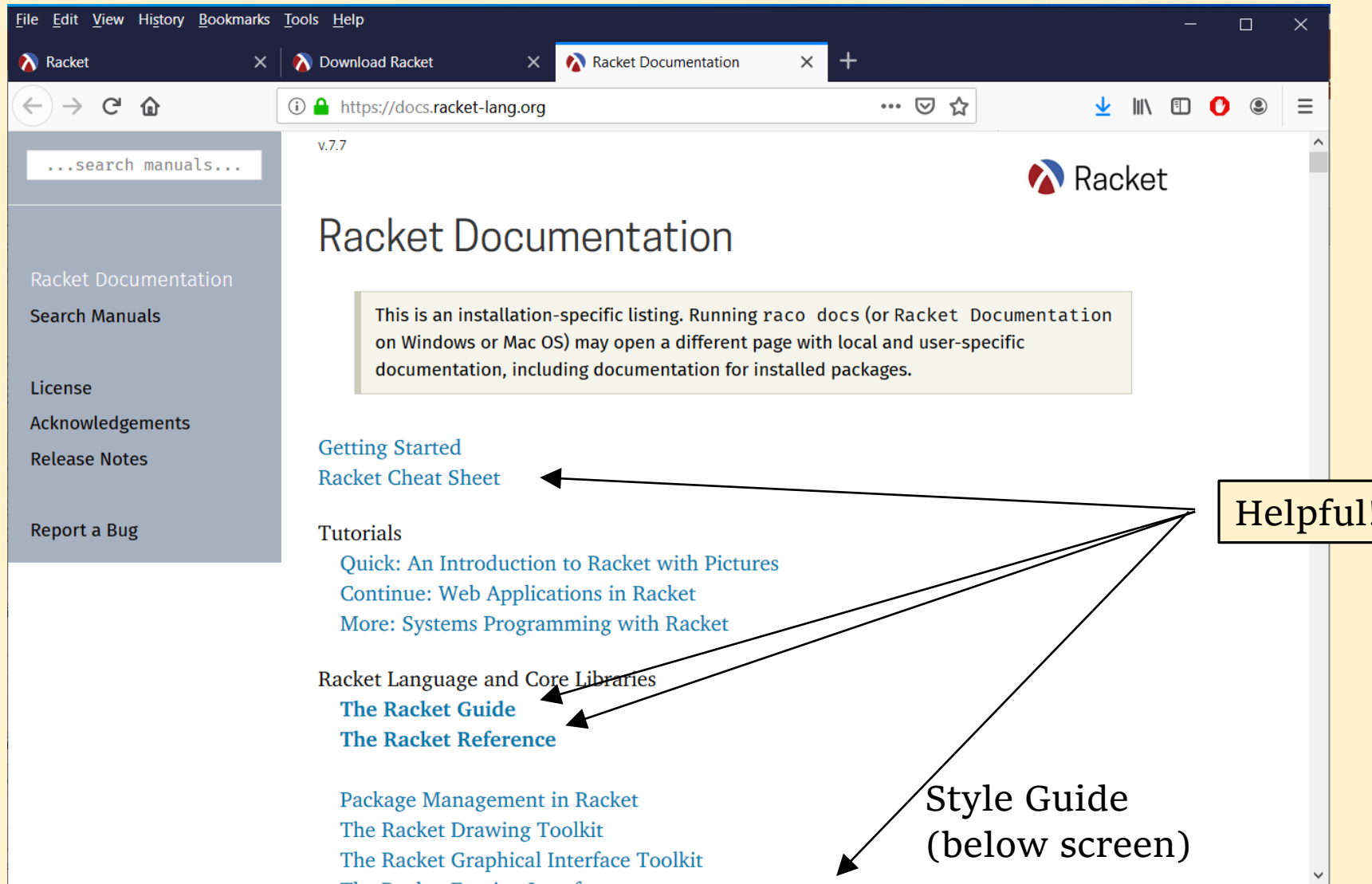
URL: <http://www.racket-lang.org>



Racket: Background, Downloading, GUI layout



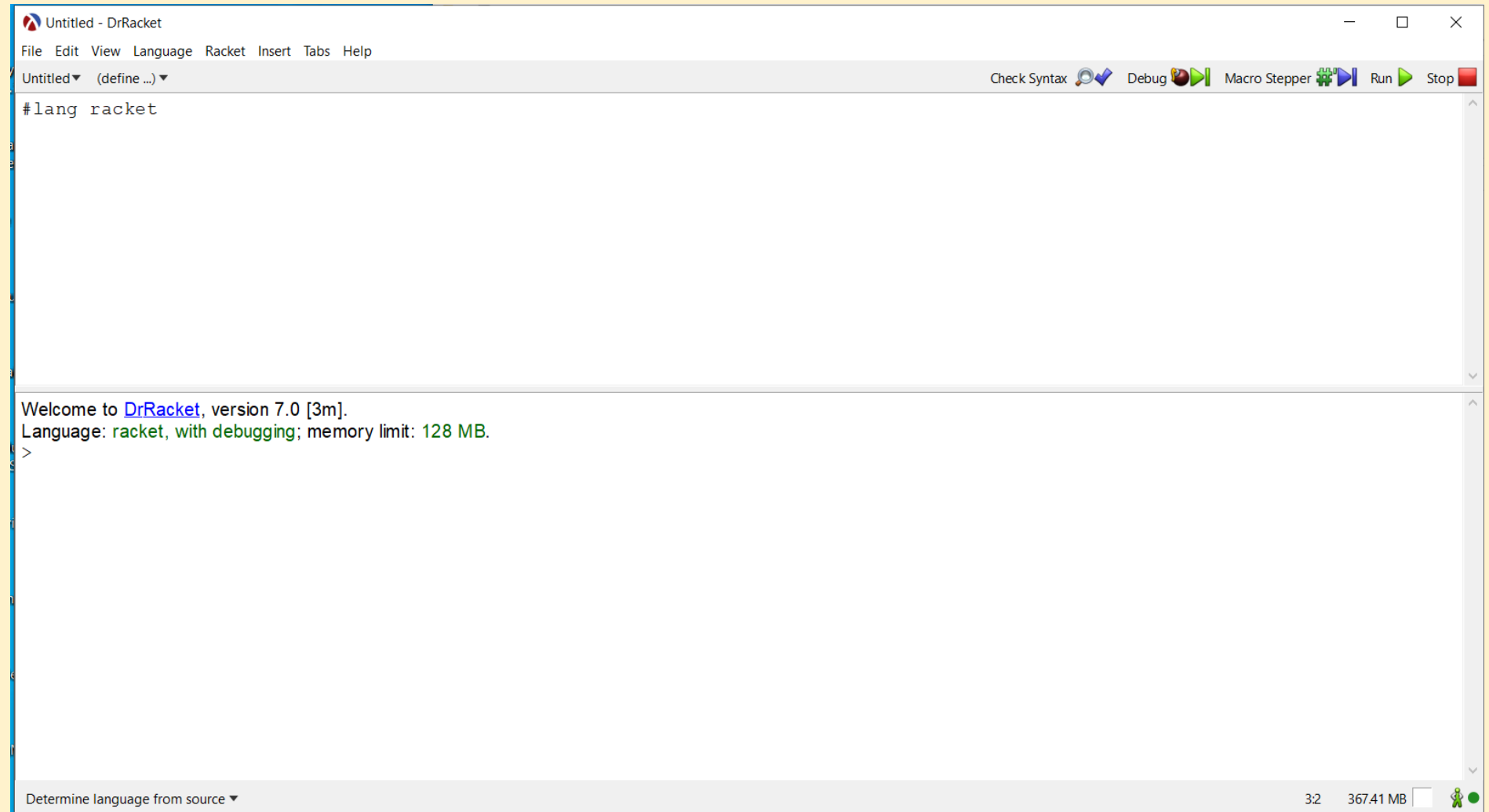
Racket: Background, Downloading, GUI layout



Racket: Background, Downloading, GUI layout

Write programs
here →

Interactive window,
Run programs
here →



Basic Math Operations – Prefix Notation

- Math you learned in math class (infix notation)
 - $7 + 5 + 3 + 2$
 - $7 + (5 - 3) + 2$
 - $7 * 5 * 3 * 2$
 - $7 * (5 - 3) * 2$
- Prefix notation
 - $7 + 5 + 3 + 2 \rightarrow (+\ 7\ 5\ 3\ 2)$
 - $7 + (5 - 3) + 2 \rightarrow (+\ 7\ (-\ 5\ 3)\ 2)$
 - $7 * 5 * 3 * 2 \rightarrow (*\ 7\ 5\ 3\ 2)$
 - $7 * (5 - 3) * 2 \rightarrow (*\ 7\ (-\ 5\ 3)\ 2)$
- Note correspondence with λ calculus: Applying operator to operands

Binding Values to Variables

- Imperative and Object-Oriented programming styles use assignment.
 - `x = 42; →` stores 42 in memory location associated with x
 - It also binds the value of 42 to x
- Functional programming uses binding.
 - `(define x 42) →` binds 42 to x
 - No need to think of memory storage, just the binding
 - Note correspondence with λ calculus:
 - “define” is an operator (function) applied to two operands (the variable name and value)
- Bindings can be global or local to functions, same as declarations in imperative programming

Lists: Definitions

- A single value is an atom.
- A list is a sequence of atoms enclosed in parentheses:
 - (2 4 6 8 10 12)
 - (“dog” “cat” “fish” “dogfish” “catfish”)
- Lists are always composed of two parts: (first element . rest of list)
- Lists can be empty: () is the empty list
- Lists can be made up of anything – no data type restrictions
 - A list of numbers: (34 -56 795 12 0 34 -2)
 - A list of words: (“cactus” “air” “force”)
 - A list of numbers and words: (34 “cactus” 42 “dog” “cat”)
 - A list of operators: (+ - + * * -)
 - A list of lists: ((1 3 5) (2 4 6) (7 8 9))

Lists: Creating and Combining

- Create lists using the “list” operator: `(list 1 3 9 17 3 6)`
 - The `'` tells the interpreter to not try to evaluate the list
 - `(+ 4 6 8 9)` means “add these numbers together”
 - `(list + 4 6 8 9)` means “make a list with the `+` operator as the first element
- Reuse created lists by binding them to a name:
 - `(define L1 '(56 42 93 12))`
 - Typing `L1` into the interactive screen returns `(56 42 93 12)` as a list.
- Add elements to the front of a list using the cons operator
 - `(cons 0 '(2 4 6 8)) = (0 2 4 6 8)`
 - `(cons '(1 2) '(3 4 5)) = ((1 2) 3 4 5)` (because `(1 2)` is one ‘thing’)
- Combine lists using the append operator
 - `(append '(1 2) '(3 4 5)) = (1 2 3 4 5)`

Lists: Two Basic Operations, car and cdr

- car returns the first element in a list. Often an atom, may be a list.
 - (car '(2 4 6 8 10)) – returns 2 (an atom)
 - (car '('(2 4) '(6 8) 10)) – returns '(2 4) (a list)
 - (car '()) – returns an error (empty lists has no first element)
 - (car 5) – returns an error. 5 is not a list.
- cdr returns the rest of the list, and is always a list.
 - (cdr '(2 4 6 8 10)) – returns '(4 6 8 10)
 - (cdr '('(2 4) '(6 8) 10)) – returns '('(6 8) 10)
 - (cdr '()) – returns an error (empty lists has no first element)
 - (cdr 5) – returns an error. 5 is an atom, not a list.
- These can be combined
 - (car (cdr '(2 4 6 8 10))) – returns 4

Lists: Additional Operators (Many not shown!)

- `(null? list)` – returns `#t` if the list is of length zero, `#f` otherwise
- `(length list)` - returns the length of a list
 - `(length '()) = 0`
 - `(length '(2 4 6 8)) = 4`
 - `(length '('(2 4 6) ('4 5 6))) = 2` (because the list is made up of two lists)
- `(reverse list)` returns a list with the order of the elements reversed
 - `(reverse '(2 4 6)) = (6 4 2)`
 - `(reverse '(“dog” “cat” “fish” “bullfrog”)) = (“bullfrog” “fish” “cat” “dog”)`

Creating and Composing Functions

- Per the λ calculus, a function (operator) is defined as $\lambda x.f(x)$.
 - $\lambda x.(+ x 1)$ – adds 1 to the input variable x (using prefix notation)
- Per the λ calculus, functions (operators) are applied to operands.
 - $\lambda x.(+ x 1) \ 7$ – apply the function to the input 7, using β reduction.
 - We could give the function a name: $\text{add1} \equiv \lambda x.(+ x 1)$
 - $\text{add1} \ 7$ – still application, apply the add1 function to 7
- Functional program languages use λ calculus application directly.

Creating and Composing Functions

- The form of λ calculus functions once more:
 - $\lambda x. (+\ 1\ x)$ – lambda, input variable x , $f(x)$
- The form of functions in Racket:
 - `(lambda (n) (+ 1 n))` – lambda, input variable n , $f(n)$
- Multivariable functions:
 - λ calculus: $\lambda xyz. (+\ x\ y\ z)$
 - Racket: `(lambda (x y z) (+ x y z))`
- To reuse functions, give them a name
 - `(define add1 (lambda (n) (+ 1 n)))`
 - `(add1 6) = 7`
- The λ calculus form is not needed in Racket (change from LISP, Scheme)
 - `(define (add1 n) (+ 1 n))`
 - `(define (average n m) (/ (+ n m) 2))`

Creating and Composing Functions

- Functions can serve as input to other functions
 - This aligns with the λ calculus (review if needed).
 - Languages that allow this are called higher order languages.
 - This is very powerful – see examples in later video
- Example functions that take functions (operators) as input
 - Admittedly silly examples
 - (define (foo operator input) (operator input 2))
 - (foo + 5) = (+ 5 2) = 7
 - (foo * 5) = (* 5 2) = 10
 - (define (foo operator input) (operator (operator input)))
 - (foo add1 7) = (add1 (add1 7)) = 9
 - (foo subtract1 7) = (subtract1 (subtract1 7)) = 5

Map Function

- The map operator “maps” the application of function to all elements in a list.
- The map operator is standard in Racket (and LISP and Scheme)
- Examples*:
 - `(map add1 '(2 3 4 5 6)) = (3 4 5 6 7)` (all elements have 1 added to them)
 - `(map reverse ((3 4 5) (6 7 8) (1 5 7))) = ((5 4 3) (8 7 6) (7 5 1))` (all lists reversed)
- If a function has two (or more) variables, you need two (or more) lists for input*:
 - `(map multiply (3 5 8) (2 8 6)) = (6 40 48)` (elements of each list multiplied)

* This discussion assumes the user has created the `add1` and `multiply` functions previously.

Selection: If-Then-Else in Racket

- All languages require three things for an if-then-else structure
 - A Boolean test
 - Something to do if the Boolean test evaluates to “true.”
 - Something to do if the Boolean test evaluates to “false.”
- Most imperative or object-oriented languages:
 - `if (Boolean test) { True branch} else {false branch}`
- Racket:
 - In terms of λ calculus: `(if expB expT expF)`
 - `(if (Boolean test) (true branch) (false branch))`
 - Example: `(if (equal? 5 3) “Yes!” “No!”)`

Selection: If-Then-Else in Racket

- Boolean constants in Racket:
 - `#t` = true, `#f` = false
- Boolean operators in Racket:
 - Testing for equality: `equal?` (see note at bottom of page)
 - Testing for empty lists: `null?`
 - Testing for inequality: `>`, `<`, `>=`, `<=` (much like other languages)
 - Operating on Boolean values: `and`, `or`, `xor`, `nand` (all are available)
- Examples:
 - `(equal? 5 3) = #f ; (> 5 3) = #t ; (< 5 3) = #f`
 - `(equal? (+ 2 3) (+ 4 1)) = #t`
- You can also check to see if an expression is an integer, list, string, etc.

Note: There are other forms (`eq?` `eqv?` and `=`), but they behave slightly differently. usually things work out fine, but sometimes not.
A good explanation at: <https://stackoverflow.com/questions/16299246/what-is-the-difference-between-eq-eqv-equal-and-in-scheme>

Repetition: Recursion in Racket

- For loops and while loops – not in Racket! It's all recursion
 - Technically, they have been added to Racket, but
 - It's not functional and not allowed in the course.
- Repetition always requires three things (always!)
 - A starting condition
 - A stopping condition
 - Progress from the start to the stop
- Recursion requires that a function call itself.

Repetition: Recursion in Racket

- Example: factorial function
- Factorial function, standard math definition:
 - $\text{fact}(0) = 1$
 - $\text{fact}(1) = 1 = 1 * \text{fact}(0)$
 - $\text{fact}(2) = 2 * 1 = 2 * \text{fact}(1)$
 - $\text{fact}(3) = 3 * 2 * 1 = 3 * \text{fact}(2)$
 - $\text{fact}(n) = \dots = n * \text{fact}(n - 1)$
- Factorial function in Racket:

```
(define (fact n) (if (> n 0)
                    (* n (fact (- n 1)))
                    1))
```

Stopping condition (return 1)

Recursive call with progress towards
stopping condition.

Example: sumall – Sums numbers in a list

- Thinking functionally:
 - What is the state at the end of the computation? Length of list is zero.
 - What should we return when the length of list is zero? Zero.
 - So, check length of list.
 - If zero, return zero.
 - Otherwise return first element added to sum of the rest of the list.
 - What is the sum of the rest of the list? (sumall (cdr list))

Function
name

Input: a list of numbers

Check if length of list is zero

```
(define (sumall l) (if (null? l)
                        0
                        (+ (car l) (sumall (cdr l)))))
)
```

If length is zero, return zero

Otherwise, add first value in list to
sum of remaining numbers in list.

Example: findMax – finds max value in a list

- Imperative thinking:
 - Must keep track of the “current max number” and the list: two things.
 - This is “memory based” thinking.
 - You don’t have to do that in a functional environment.
- Thinking functionally:
 - The max element in the list is either that first element, or the max of the rest of the list.
 - Return the larger of (car list) and (findmax (cdr list))
 - What if the list has only one element? It must be the max.
 - What is the state at the end of the computation? List has one element, return that as the max.
 - Code on next slide:

Example: findMax – finds max value in a list

```
(define (findmax l) (if (equal? (length l) 1)
  (car l)
  (if (> (car l) (findmax (cdr l)))
    (car l)
    (findmax (cdr l)))
  )
)
```

If only one element in list, it must be the max

If more than one element, the max is either first element, or the max of the rest of the list

Wasteful to do the calculation twice? No
Racket is LAZY – will only compute once, and use the result twice.

Example: Filter1 – return values $>$ threshold

- This function returns a list of values greater than some threshold.
- Inputs: a list of numbers and a threshold value.
- Compare first value in list (car list) to threshold
 - If greater, append to list returned by applying Filter1 to rest of list
 - If less than, return Filter1 applied to rest of list.
- You may be noticing a pattern here. If not, don't worry.
- Code on next slide

Example: Filter1 – return values > threshold

```
(define (filter1 l t) (if (equal? (length l) 0)
                          '()
                          (if (> (car l) t)
                              (append (list (car l)) (filter1 (cdr l) t))
                              (filter1 (cdr l) t)))
)
```

Explanation:

If the list is empty, return an empty list.

If the list is not empty, see if the first element is > the threshold

If true, append the first element with the list obtained from filtering the rest of the list

If false, simply return the list obtained from filtering the rest of the list.

Example: Filter1, filter 2, filter3

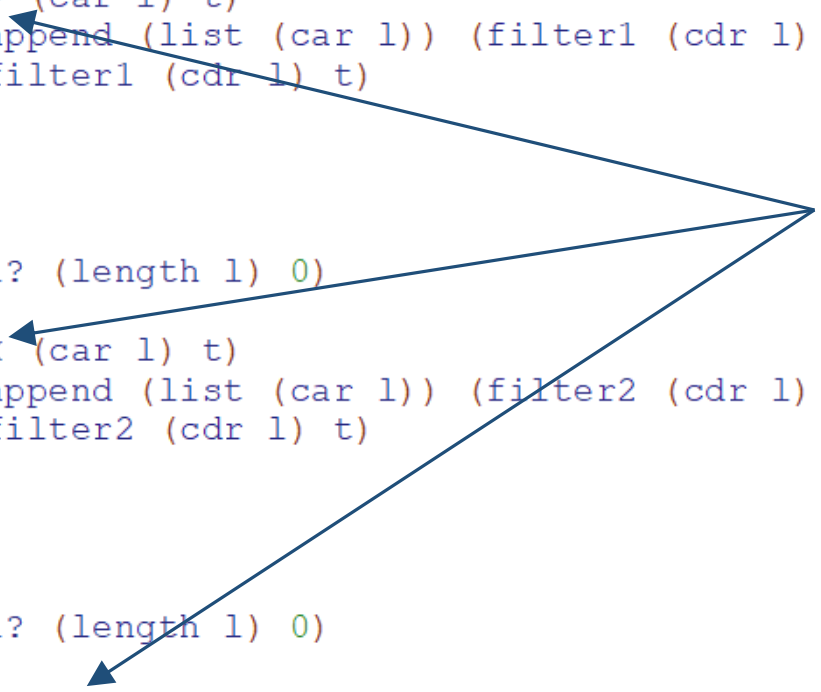
Note the similarities for filtering greater than, less than, equal to some value.

```
(define (filter1 l t) (if (equal? (length l) 0)
                          '()
                          (if (> (car l) t)
                              (append (list (car l)) (filter1 (cdr l) t))
                              (filter1 (cdr l) t))
                          )
)

(define (filter2 l t) (if (equal? (length l) 0)
                          '()
                          (if (< (car l) t)
                              (append (list (car l)) (filter2 (cdr l) t))
                              (filter2 (cdr l) t))
                          )
)

(define (filter3 l t) (if (equal? (length l) 0)
                          '()
                          (if (equal? (car l) t)
                              (append (list (car l)) (filter3 (cdr l) t))
                              (filter3 (cdr l) t))
                          )
)
```

Only real difference - -
One symbol in one line.



Example: Higher Order Thinking

- Rather than make three separate functions for filters, make one
 - Inputs are the list, the threshold, and the Boolean comparator (equal?, <, >)
 - One function can now do all three filters.
 - This technique can make for extremely efficient programming.

```
(define (genericFilter l t boolOp) (if (equal? (length l) 0)
    '()
    (if (boolOp (car l) t)
        (append (list (car l)) (genericFilter (cdr l) t boolOp))
        (genericFilter (cdr l) t boolOp)
    )
    )
)
```

Local Variables: let, let*, and letrec

- **let**: Binds variables and values, all at the same time.
 - `(let ([x 5] [y 2]) (* x y))` returns 10
 - `(let ([x 5] [y x]) (* x y))` returns an error – x undefined in [y x]
- **let***: Binds variables and values in sequential fashion.
 - `(let* ([x 5] [y 2]) (* x y))` returns 10
 - `(let* ([x 5] [y x]) (* x y))` returns 10
 - `(let* ([y x] [x 5]) (* x y))` returns an error – x undefined in [y x]
- **letrec**: Binds variables and values recursively
 - `(letrec ([x 5] [y 2]) (* x y))` returns 10
 - `(letrec ([x 5] [y x]) (* x y))` returns 10
 - `(letrec ([y x] [x 5]) (* x y))` returns an error – x undefined in [y x]