Racket Cribsheet – Version 10/4/21

Racket IDEA	EXPLANATION	Racket EXAMPLES

evaluation	Scheme will try to **interpret** the first element of a list inside () as a function to perform, and the rest of the elements as arguments to that function.	`(+ 1 2)`
		`(cdr L)`
		`(if (= n 2) 12 18)`

defining variables	**`define`** will create a memory location with the name of the first argument, and store the value of the second argument in that memory location	`(define a 12) -> a=12`
		`(define Harry (+ 10 2)) -> Harry=12`
		`(define L '(+ 10 2)) -> L=(+ 10 2)`

arithmetic operations	The ordinary 4 arithmetic functions will work with both whole numbers and floating points and fractions, and will usually take more that 2 arguments
	
	`abs` is the absolute value function.
		`(+ 1 2) -> 3  (+ 1 2 3) -> 6`
		`(- 5 3) -> 2  (- 5 3 1) -> 1`
		`(* 5 2) -> 10  (* 5 2 3) -> 30`
		`(/ 8 2) -> 4  (/ 7 2) -> 3 ½`
		`(abs -3) -> 3  (abs -2.3) -> 2.3`

integer arithmetic operations	**`remainder`** returns the remainder of its first argument divided by its second argument. **`quotient`** divides its first argument by its second, and ignores the remainder For positive values, **`modulo`** works the same as **`remainder`**.	`(remainder 13 5) -> 3` `(quotient 13 5) -> 2`

compound expressions	Lists within lists are evaluated from inside to outside	`(+ (* 3 4) 2) -> 14`
		`(remainder (quotient 543 10) 10) -> 4`
		`(quotient (remainder 543 10) 10) -> 0`

functions (without lambda)	Functions can be defined with one or more arguments. The argument names can be arbitrarily chosen, as long as they match the use of that argument inside the function body.	`(define (square n) (* n n))` `(define (square bleep) (* bleep bleep))`
functions (with lambda)	Another form of function definitions uses the Greek letter lambda (as **`λ`** or **`lambda`**).	`(define square (λ (n) (* n n)))` `(define square (lambda (n) (* n n)))`

relational operators	These operators compare numerical values and return **`#t`** or **`#f`**	`(= 12 13) -> #f`
		`(> 4 2) -> #t`
		`(>= 4 (+ 1 3)) -> #t`
		`(< 1 2 3) -> #t`
		`(<= 1 2 0) -> #f`

conjunctions	**`and`**, **`or`** and **`not`** combine several logical expressions into larger logical (boolean) expressions	`(and (= 4 4) (> 4 5)) -> #f`
		`(or (= 4 4) (> 4 5)) -> #t`
		`(not (= 4 4)) -> #f`

if	Make decisions with **`if`**. An if-list always has 4 elements: `(if test-part true-part false-part)` The "if-list" then returns the value of its true-part or its false-part.	`(if (= 12 13) 14 (+ 1 2)) -> 3` `(if (< 12 14) 14 (+ 1 2)) -> 14`

cond	Multi-decision lists can be composed with **`cond`**, which has any number of clauses, each with a test and a result. The clauses' tests are evaluated one after the other until one of the tests is true, then that clause's result is immediately returned as the **`cond`** list's value. An **`else`** clause may be included as the last clause and will be activated if all of the other clauses's tests are false.	`(cond`
		`  ( (> grade 100) "Extra credit")`
		`  ( (and (>= grade 65) (<= grade 100)) "Pass")`
		`  ( else "Fail")`
		`)`

functions with decisions		`(define (PrintGrade num)`
		`  (if (> num 100)`
		`  "Extra credit"`
		`  (if (>= num 65) "Pass" "Fail" )))`
		
		`(define (PrintGrade num)`
		`  (cond`
		`  ((> num 100) "Extra credit")`
		`  ((and (>= num 65) (<= num 100)) "Pass")`
		`  (else "Fail")`
		`  )`
		`)`

recursive functions	These are functions with call themselves.	`(define (factorial n)`
		`  (if (<= n 1) 1`
		`  (* n (factorial (- n 1)))))`