Scheme IDEA	EXPLANATION	Scheme EXAMPLES
lists	lists are formed by enclosing a sequence of items (atoms or lists) inside opening and closing parentheses	`(12 Harry (another list))`

evaluation	Scheme will try to interpret the first element of a list 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)`

preventing evaluation	Quoting a list using the apostrophe (') will prevent Scheme from evaluating a list	`(cdr '(do not evaluate)) -> (not evaluate)`

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 decimals 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	modulo returns the remainder of its first argument divided by its second argument. quotient divides its first argument by its second, and ignores the remainder	`(modulo 13 5) -> 3` `(quotient 13 5) -> 2`

compound expressions	Lists within lists are evaluated from inside to outside	`(+ (* 3 4) 2) -> 14`
		`(modulo (quotient 543 10) 10) -> 4`
		`(quotient (modulo 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. Note: in the current Windows version of DrScheme (299.x), that actual Greek letter "λ" can be entered but causes errors, and so the word "lambda" must be used instead.	`(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)`
		`  (cond`
		`  ((> num 100) "Extra credit")`
		`  ((and (>= num 65) (<= num 100)) "Pass")`
		`  (else "Fail")`
		`  ))`

list processing functions	- "car" returns the first element of a list
	- "cdr" returns the rest of a list once the first element has been removed
	- "cons" creates a larger list whose first element was cons's first argument, and the rest was cons's second argument
	- "list" creates a list from its arguments
	- "append" combines all of its arguments (which must be lists) into one larger list
		`(define L '(Harry (Tom Riddle) Ron))`
		
		`(car L) -> Harry`
		`(cdr L) -> ((Tom Riddle) Ron)`
		
		`(car (cdr L)) -> (Tom Riddle)`
		`(car (car (cdr L))) -> Tom`
		`(cdr (cdr L)) -> (Ron)`
		
		`(cons 'Fred L) -> (Fred Harry (Tom Riddle) Ron)`
		`(cons (car L) (cdr L)) -> L`
		
		`(list 1 2 3) -> (1 2 3)`
		
		`(append '(Fred) L) -> (Fred Harry (Tom Riddle) Ron)`

abbreviations of sequences of car and cdr	One can abbreviate up to 4 sets of these functions with a single function whose letters are formed from the middle letters of the sequence of car and cdr. For instance:	`(car (cdr (cdr (cdr L))))`
		`can be abbreviated:`
		`(cadddr L)`

recursive functions	A recursive function may make a call to itself during evaluation. When designing a recursive function, make sure that you include a **base case** which does not call the function itself. The typical recursive function tries to reduce a large problem to a smaller problem, and then call itself with the smaller problem, until the base case is reached.	`(define (mylength L)`
		`  (if (null? L)`
		`  0`
		`  (+ 1 (mylength (cdr L))))`
		`(define (factorial n)`
		`  (if (< n 2)`
		`  1`
		`  (* n (factorial (- n 1))))))`

miscellaneous functions	- (null? L) returns #t is the list is empty, else #f
	- (list? P) returns #t if P is a list, else #f
	- (sqrt n) returns the square-root of the number
	- (min a1 a2 a3 …) returns the minimum of arguments
	- (max a1 a2 a3 …) return the maximum of arguments
	- (even? n) return #t if the N is even, else #f
		`(null? '()) -> #t`
		`(list? '(a b)) -> #t  (list? 'a) -> #f`
		`(sqrt 2) -> 1.4142135623730951`
		`(min 5 3 9) -> 3`
		`(max 5 3 9) -> 9`
		`(even? 3) -> #f`