```
    ; equal => complete equality check for structure (value) for lists, strings, bit-vectors
    ; eql => stricter than equal, complete equality check for object identity (whether two arguments refer to the same object in memory)

(= 3 3.0) ; evaluates to t
(= 2 1) ; evaluates to nil
(/= 2 1) ; evaluates to t
(< 1 2) ; evalutes to t
(> 3 1) ; evalutes to t
(<= 1 1) ; evaluates to t
(>= 2 1) ; evaluates to t

(equal (list 3) (list 3)) ; this evaluates to t since equal checks for structural equality and compares the value of the contents of the lists instead of their place in memory
(equal (list 'a 'b) (list 'b 'a)) ; evaluates to nil

(eql 3 3) ; evaluates to t
(eql 3 3.0) ; evaluates to nil
(eql (list 3) (list 3)) ; this evaluates to nil since not same object in memory despite having structural equality

;; LOGICAL OPERATORS
    ; and
    ; or
    ; not

(and t t) ; evaluates to t
(and t nil) ; evaluates to nil
(or t nil) ; evaluates to t
(or nil nil) ; evaluates to nil
(not t) ; evaluates to nil
(not nil) ; evaluates to t
```

Control structures

```
;;; ---------- CONTROL STRUCTURE ----------

;; CONDITIONALS
    ; as established previously, only nil is false (and () empty list which evaluates to nil), everything else is true (t)
    ; conditional syntax => (if {TEST EXPRESSION} {IF TEST EXPRESSION TRUE} {ELSE EXPRESSION})
    ; cond => chains a series of conditional checks to arrive at a final result
    ; typecase => switch case statement but for type of value

(if t
    "this is true"
    "this is false") ; evaluates to "this is true"

(member 'Groucho '(Harpo Groucho Zeppo)) ; evaluates to '(GROUCHO ZEPPO)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope) ; evaluates to 'YEP since all non-nil values including '(GROUCHO ZEPPO) are t

(cond ((> 2 2) (error "wrong!"))
      ((< 2 2) (error "wrong again!"))
      (t 'ok)) ; evaluates to 'OK symbol since the first 2 checks were incorrect

(typecase 1
    (string :string)
    (integer :int)) ; evaluates to :int since 1 is of type integer

;; LOOPS
    ; loop => creates a loop iteratively that can be augmented with different keywords (:for :from :to :then :finally :across :collect)
    ; there is no while loop implementation by default
```