Lisp

- Lisp is a functional programming language based on **lists**, specifically linked lists
- The most basic data structure is Lisps is the `cons`, which is just a pair
  - i.e. `(cons "A" "B")` -> ("A" . "B")
    - The '.' is used to specify that a data structure is a pair rather than a list
- A list is formed by chaining `cons` structures so that the second value in the pair is another `cons`, though you can also just use the `(list)` function
  - i.e. `(list "A" "B" "C" "D" "E"`) -> ("A" "B" "C" "D" "E")
  - i.e. `(list "A" "B" (cons "C" "D"))` -> ("A" "B" . ("C" "D"))
- To access the first element of a cons cell, use `car`. Since a list is just a chaining of cons cells, `car` on a list will return the first element of the list
  - i.e. `(car (list "A" "B" "C"))` -> "A"
- To access the rest of a cons cell, use `cdr`. In the context of a list, this will return all elements after the first element.
  - You can chain `car` and `cdr`
  - i.e. `(cdr (list "A" "B" "C"))` -> ("B" "C")
- Parentheses expressions in Lisp will typically be evaluated when passed in as arguments - to not evaluate an expression (and to use its contents literally), include a single quote ' before the list
  - i.e. `(print '(+ (1 2 3)))`
- Common Functions:
  - `(+ x y)`, `(- x y)`, `(* x y)`, `(/ x y)`, `(floor x y)`, `(expt base power)`, ...
  - `(< x y)`, `(> x y)`, `(= x y)`, `(/= x y)`, ...
    - These are numerical comparisons
  - `(equal x y)`
    - General purpose equality
  - `(and ...)`, `(or ...)`, `(xor ...)`, `(not ...)`
  - `(setq VARIABLE VALUE)` sets the VARIABLE with VALUE - remember that VALUE doesn't necessarily need to be a single object, as it can also be a list
    - `setq` should be used to assign values to existing variables, though it can be used to create a global scope
  - `(let VARLIST BODY)` creates variables with values according to VARLIST and also creates a local scope for those variables to be used with other expressions in BODY

    -
      ```
      (let ((x 1) (y 2))
        (print (+ x y))
      )
      ```
    - If an item in VARLIST contains only a single value, then that variable is assigned `nil`
    - Use `let` to create local variables and use `setq` to actually change those variables
  - `(push ITEM LIST)` pushes ITEM to the beginning of LIST
  - `(defun function-name (ARGUMENTS) "OPTIONAL DOCUMENTATION" (interactive info) (body))` is used to create a function

  - `(progn body)` allows for multiple expressions to be performed in succession
  - `(if COND THEN ELSE)`

    -
      ```
      (if nil
            (print 'true)
            'very-false)
      ```
  - `(cond CLAUSES)` is like a switch statement

    -
      ```
      (cond ((eq a 'hack) 'foo)
            (t "default"))
      ```
  - `(while CONDITION BODY)`
- Programs in Lisp use data notation - everything is data (lists). However, since linked lists are fundamentally slow, repeatedly evaluating certain functions can be costly.
  - This where higher-level languages such as Lisp (and Javascript, Python, etc.) benefit from compiling expressions into bytecode, performing possible optimizations
    - Bytecode is typically portable - meaning it can be used with any architecture, though, compared to machine code, it may not be as fast

Emacs Lisp

- When creating an ELisp function, you can load it into your Emacs session by performing `M-x load-file` and then entering the file containing that function definition
  - Whenever you do `M-x`, you can then enter the function name and call it from your Emacs session
- the Emacs `(global-set-key "KEY" FUNCTION)` function binds "KEY" to FUNCTION
  - i.e. `(global-set-key "@" 'what-cursor-position)` will set the @ key to the 'what-cursor-position function (note the ' before the function, as we don't want to actually evaluate it)
    - If you wanted to undo this, you would copy the "@" key and use the command `(global-set-key "@" 'self-insert-command)`
- The command `byte-compile-file` compiles an Elisp file into bytecode
- `point-min` is a variable that points to the beginning of the buffer region
- `point-max` is a variable that points to the end of the buffer region