CS331
PLL

LISP Programming

Session by:

Prof. Pradip K.Das

Lisp: An Introduction

- Lisp – **List Processing language**
  - Developed by John McCarthy, MIT, in the late 1950s
    - The original idea was to implement a language based on mathematical functions that McCarthy was using to model problem solving.
    - A grad student implemented the first interpreter, and it was quickly adopted for AI programming:
      - Symbolic computing
      - List processing
      - Recursion
    - Early Lisp was based almost entirely on recursive and so
      - Lisp had no local variables nor iterative control structures (loops)
      - and since Lisp was interpreted, there was no compiler for early Lisp

More on LISP

In part because this was early in the history of high level languages, and in part because of the desire to support AI

- LISP used dynamic scoping
- LISP had no compiler
- LISP had no local variables
- LISP was slow

But, because LISP was interpreted

- It was easy to prototype systems and build larger scale systems than a compiled language
- **Lisp had features not available in other early languages**
  - **Recursion**
  - **Linked lists**
  - **Dynamic memory allocation**
  - **Typeless variables**

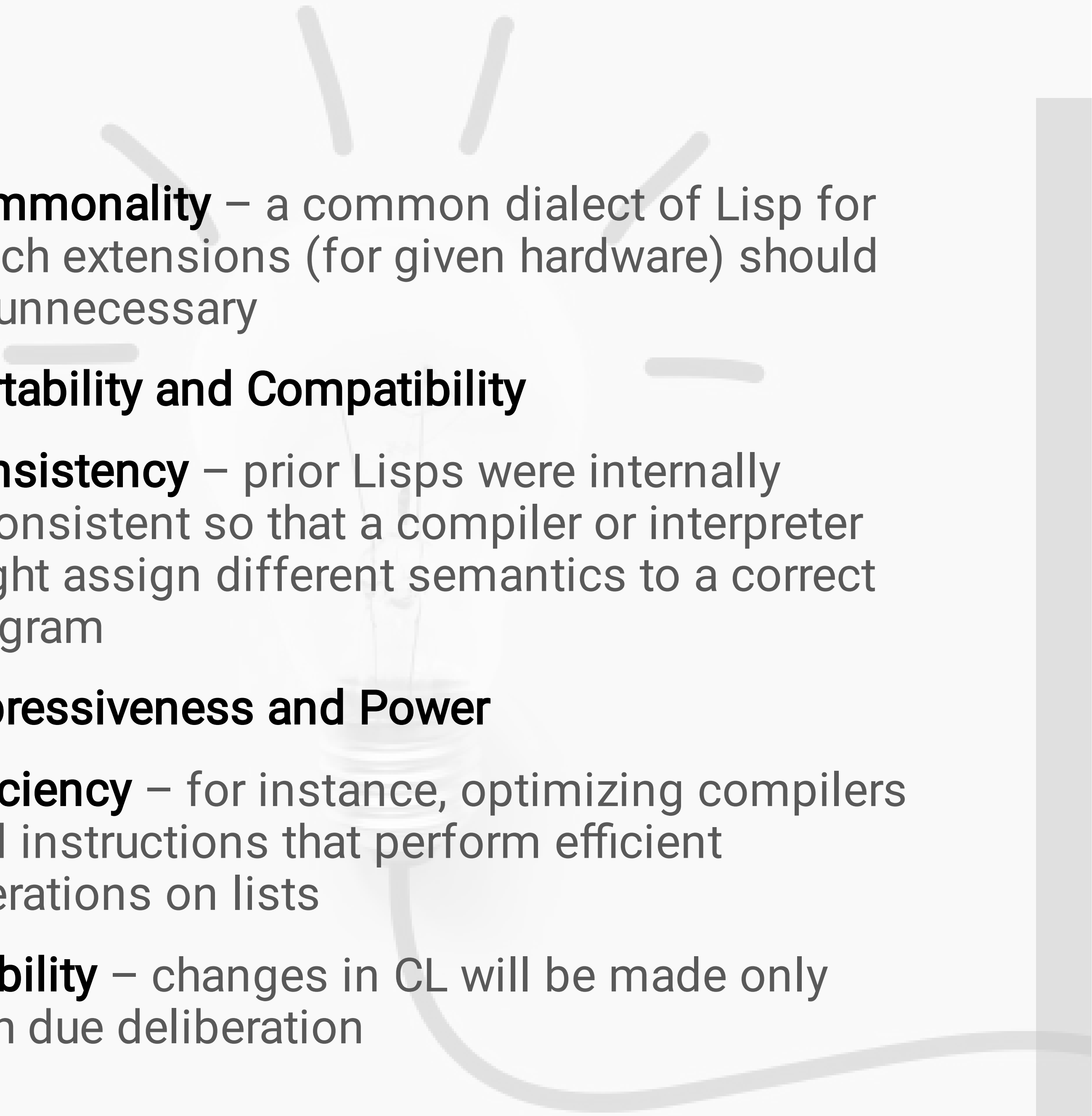
Lisps, Lisps and More Lisps

- The **original Lisp** was both **inefficient and difficult to use**
  - ✓ **Other Lisp dialects** were released by people trying to further the language
    - They added a compiler, local variables, static scoping, etc.
    - While Scheme is the most recognized Lisp to come out of these efforts, others include Qlisp, Elisp, Interlisp, etc.
  - ✓ Specialized hardware was manufactured to support Lisp:-
    - Hardware that had large heap memory spaces and optimized routines for garbage collection.
    - Interestingly, the first GUI was implemented for Lisp machines.

Toward a Standard: Common Lisp

- **DARPA sponsored an effort in the early 1980s to develop a standard for lisp, and so Common Lisp (CL) was born**
  - ✓ However, among the Lisp community, there were hard-fought battles
    - West coast users often used Interlisp.
    - East coast users often used MACLisp.
  - ✓ **CL would take the best attributes found in MacLisp with some influence from Zetalisp, Scheme and Interlisp.**
    - It would also provide a number of imperative features (e.g., loops)
    - ANSI created a standard for CL around 1990
    - The Common Lisp Object System (CLOS) was added to CL making the language roughly equivalent in size and capability to C++
    - While not as recognized as C++, Common Lisp is a very useful and flexible language.

Goals of CL



- **Commonality** – a common dialect of Lisp for which extensions (for given hardware) should be unnecessary
- **Portability and Compatibility**
- **Consistency** – prior Lisps were internally inconsistent so that a compiler or interpreter might assign different semantics to a correct program
- **Expressiveness and Power**
- **Efficiency** – for instance, optimizing compilers and instructions that perform efficient operations on lists
- **Stability** – changes in CL will be made only with due deliberation

Some CL Uses

CL is a very successful language in that

- It is extremely powerful as an AI tool

And yet CL is a very uncommon language because

- It is very complicated
- most software development is in C++ (with some in Java, C#, C, Python, Ada, etc.) such that few programmers would choose to use CL for development

The most important software developed from CL includes:

- Emacs
- G2 (a real-time expert system)
- AutoCAD
- Igor Engraver (musical notation editor and publisher)
- Yahoo Store

Up until the early 1990s, most AI research was still done in Lisp, most using Common Lisp

- However, once C++ added objects, many researchers switched to C++ because
  - Their graduate students had more familiarity with C++
  - Obtaining C++ compilers is often cheaper and easier than CL compilers and environments
  - C++ programs will run faster than CL programs (not always true, but generally true)

Why Study CL?

Learn about	Learn about functional programming
Learn	Learn how to utilize recursion better
Learn about	Learn about symbolic computing
Gain	Gain experience in other languages and syntax
Get	Get a better idea of such concepts as scope, binding, memory allocation, etc.
Learn	Learn a little about AI problem solving
Learn	But primarily: learn a new language • One that is **very very** different from what you have already experienced • But also a very rich and complex language

Lisp Basics

- Everything in a Lisp program is either
  - An atom
    - A single literal value which could be:
      1. a number
      2. a symbol like ’hello’
         - not to be confused with a string, a symbol cannot be subdivided in any way
      3. a character
      4. T or nil
  - A list
    - Lists are denoted by ( )
      - Lists can contain nothing (empty list), one or more atoms, sublists, or some combination of these
  - In Lisp, all variables are pointers that point to an atom or a list, but like Java, these pointers do not need dereferencing (unlike C or C++)
    - There are also sequences of different types (vectors, arrays, strings, structures, objects), these are neither lists nor atoms

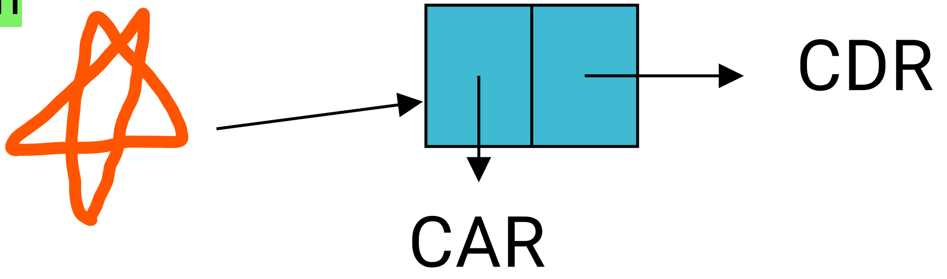
All variables are in fact pointers

A variable’s pointer points to an item in heap memory

- like Java, there is no dereferencing (unlike C or C++)
- unlike Java, *all* items are pointed to whether they are objects, strings, arrays, numbers, atoms, whatever

Lists are not only pointed to by the variable that defines the list, but each list element contains a pointer to the next item

- The structure of a list item is actually two pointers
  - the CAR is the pointer to the item
  - the CDR is the pointer to the next list item
  - this structure looks like this
- This structure is known as a *cons* cell



Lisp Code vs. Data

Lisp code is placed inside of lists

- All* Lisp instructions are function calls
- All Lisp function calls are written in prefix notation (name param1 param2 …)
- All Lisp function calls return a value, which can be used in another call
  - example: (square (+ 3 5)) (+ 3 5) returns 8, (square 8) returns 64

Lisp data are most commonl
y placed in lists

- So in Lisp, data and code have the same look to them, this makes it easy to write code that manipulates code rather than data (code that can generate code for instance)
  - * - there are exceptions to this statement that we will explore throughout this course

Lisp Interpreter

- One of the more unique aspects of Lisp (as opposed to say Java or C) is that **it is an interpreted language**
  1. You are given a command line where you can type in commands where a command can be
     - ✓a single executable statement (e.g., (+ x y))
     - ✓a definition (defining a new function, defining a variable, defining a class, instantiating an object, etc)
       - you see the response immediately and you don’t have to have already compiled anything, this lets you test out code while you write it
  2. The Interpreter works on a **REPL cycle**:
     - **Read –** read the latest item typed in at the command line (ended by <enter>)
     - **Eval –** evaluate what was typed in (execute it)
     - **Print –** print the result to the screen for immediate feedback
     - **Loop –** do it all over again

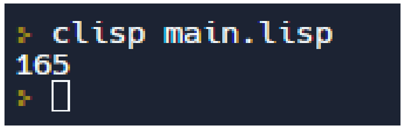
  Note: you can type definitions and instructions in an editor and load it all at once or compile it and load the compiled file

Examples

1. Let us write an **s-expression to find the sum of four numbers 17, 91 ,12 and 45.** To do this, we can type at the interpreter prompt.

✓ (write (+ 17 91 12 45))

Output:



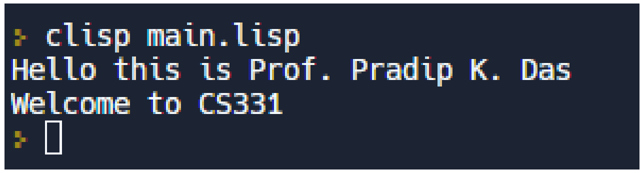
```
› clisp main.lisp
165
› 
```

2. Printing on console:

✓ (write-line "Hello this is Prof. Pradip K. Das")

✓ (write-line "Welcome to CS331")

Output:



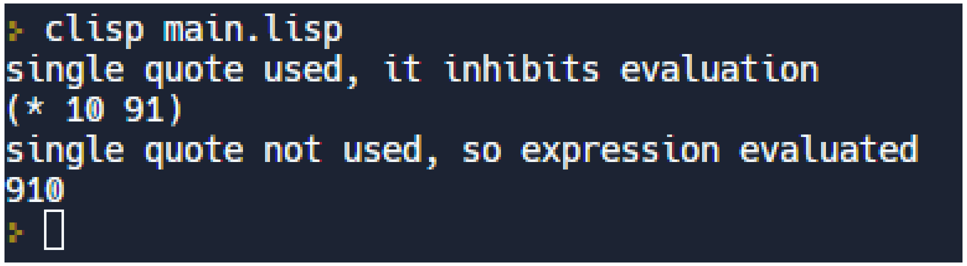
```
› clisp main.lisp
Hello this is Prof. Pradip K. Das
Welcome to CS331
› 
```

Examples

Use of Single Quotation Mark

- ✓ (write-line "single quote used, it inhibits evaluation")
- ✓ (write '(* 10 91))
- ✓ (write-line " ")
- ✓ (write-line "single quote not used, so expression evaluated")
- ✓ (write (* 10 91))

Output:



```
› clisp main.lisp
single quote used, it inhibits evaluation
(* 10 91)
single quote not used, so expression evaluated
910
›
```

LISP- Data types

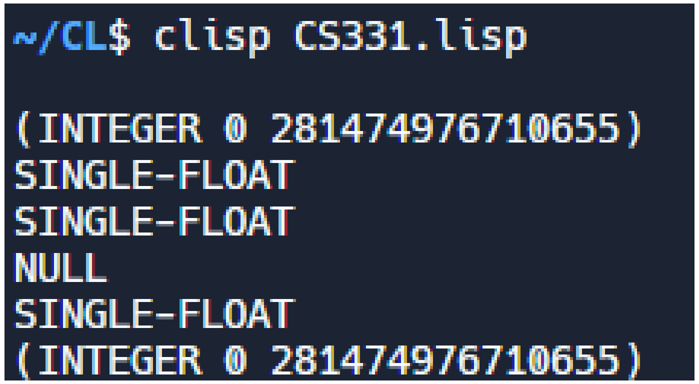
In LISP, variables are not typed, but data objects are.

LISP data types can be categorized as:-

- **Scalar types** – for example, number types, characters, symbols etc.
- **Data structures** – for example, lists, vectors, bit-vectors, and strings.

```
✓ (defvar x 10)
✓ (defvar y 34.567)
✓ (defvar ch nil)
✓ (defvar n 123.78)
✓ (defvar bg 11.0e+4)
✓ (defvar r 124/2)
✓ (print (type-of x))
✓ (print (type-of y))
✓ (print (type-of n))
✓ (print (type-of ch))
✓ (print (type-of bg))
✓ (print (type-of r))
```

Output:



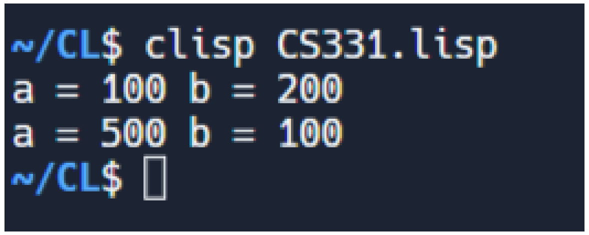
```
~/CL$ clisp CS331.lisp

(INTEGER 0 281474976710655)
SINGLE-FLOAT
SINGLE-FLOAT
NULL
SINGLE-FLOAT
(INTEGER 0 281474976710655)
```

Variables

- ❑ Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the **setq** construct.
- ❑ The **symbol-value function** allows you to extract the value stored at the symbol storage place.

✓ (setq x 100)
✓ (setq y 200)
✓ (format t "a = ~2d b = ~2d ~%" x y)
✓ (setq x 500)
✓ (setq y 100)
✓ (format t "a = ~2d b = ~2d" x y)

Output:



```
~/CL$ clisp CS331.lisp
a = 100 b = 200
a = 500 b = 100
~/CL$ 
```

```
CL-USER 1 > 'hello
HELLO

CL-USER 2 > (print 'hello)

HELLO
HELLO

CL-USER 3 > (setf a 'hello)
HELLO

CL-USER 4 > a
HELLO

CL-USER 5 > 'a
A
```

Type in a statement, it is evaluated
’hello evaluates the symbol hello

print is a function, it prints the item and also returns the item

setf is a function that has a side effect of adjusting a pointer and returns the value that the pointer is pointing to

evaluate a returns what a points to

notice here the quote mark says “do not evaluate, just return”

Example Session

```
CL-USER 41 > (defun sortlist (lis)
        (let (temp (size (length lis)) (sortedlist nil)
                (currentmin 1000))
        (dotimes (i size)
            (setf temp (findmin lis))
            (if (< temp currentmin) (setf currentmin temp))
            (setf lis (remove temp lis))
            (setf sortedlist (append sortedlist (list temp))))
        sortedlist))
SORTLIST

CL-USER 42 > (sortlist '(5 3 4 7 2 1 6))
(1 2 3 4 5 6 7)
```

The Debugger

- Unfortunately, the run-time environment is where most of your errors will be caught
  - Every time the interpreter does not understand something, you are thrown into the debugger
  - For now though, we will simply abort out of the debugger any time an error arises
    - This differs from implementation to implementation, but in LispWorks, look at the options and then type c: # where # is the number of the abort option (for instance, c: 1)

Example With Debugger

```
CL-USER 43 > (sortlist b)

Error: The variable B is unbound.
  1 (continue) Try evaluating B again.
  2 Return the value of :B instead.
  3 Specify a value to use this time instead of evaluating B.
  4 Specify a value to set B to.
  5 (abort) Return to level 0.
  6 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,  or :? for other options

CL-USER 44 : 1 > :c 3

Enter a form to be evaluated: '(5 3 2 4 1)
(1 2 3 4 5)
```

Example Continued

- CL-USER 62 : 1 > (sortlist b)

- Error: The variable B is unbound.
-   1 (continue) Try evaluating B again.
-   2 Return the value of :B instead.
-   3 Specify a value to use this time instead of evaluating B.
-   4 Specify a value to set B to.
-   5 (abort) Return to level 1.
-   6 Return to debug level 1.
-   7 Return a value to use.
-   8 Supply a new first argument.
-   9 Return to level 0.
- 10 Return to top loop level 0.
- Type :b for backtrace, :c <option number> to proceed, or :? for other options
- CL-USER 63 : 2 > :c 4
- Enter a form to be evaluated: '(5 3 4 6 2 1)
- (1 2 3 4 5 6)

Example Continued

```
CL-USER 66 > (sortlist)
Error: Call ((LAMBDA (#:LIS) (DECLARE (SPECIAL:SOURCE #)
        (LAMBDA-NAME SORTLIST))
        (BLOCK #:SORTLIST (LET # # #:SORTEDLIST)))) has the
        wrong number of arguments.
  1 (abort) Return to level 0.
  2 Return to top loop level 0.
Type :b for backtrace, :c <option number> to proceed,  or :? for other options
CL-USER 67 : 1 > :b
Interpreted call to SORTLIST
Call to SPECIAL::%EVAL-NOHOOK
Call to IV:PROCESS-TOP-LEVEL
Call to CAPI::CAPI-TOP-LEVEL-FUNCTION
Call to CAPI::INTERACTIVE-PANE-TOP-LOOP
Call to (SUBFUNCTION MP::PROCESS-SG-FUNCTION
        MP::INITIALIZE-PROCESS-STACK)
```

Defining Functions

- There are a number of built-in functions in CL
  - The basic form is:
    - (defun name (params) body)
      - name is the name of your function to have, it can be any legal CL name (and as long as the name is not a pre-defined CL name but it can be a name that you had previously used yourself)
      - params are the names of parameters being passed into the function, they can be any legal name, and there are no types associated with them in the definition, the list can be empty as in ( )
      - body is one or more CL function calls, after the last one is called, whatever value that last function call returns is returned by this function

Lisp and Variables

- There are generally three kinds of variables
  - Global variables – defined in the run-time environment (at the command line) or outside of a function in a file
    - these can be accessed within a function, but only having been declared as special
  - Local variables – defined in a function using the let statement, or within the scope of a specific instruction, such as the loop counter(s) in a do statement
    - these can only be accessed inside their scope, as soon as their scope ends, you can no longer access them
  - Parameters – much like local variables, but they are available throughout the entire function

Variable Types

- As described earlier, variables are actually pointers, but unlike C, the pointers are not typed
  - This means that the value being pointed to by a pointer can be of one type and later the pointer can point to a value of another type
  - With a few exceptions, variables in CL are not typed
    - notable exceptions to not typing variables are objects and structures
- However, all variables are type checked at run-time to make sure that a value is being treated correctly
  - So (* 5 'a) will yield a run-time error rather than an incorrect value

Operations on Numbers

- Arithmetic functions:
  - +, -, *, /
    - Note that / will always return quotient and remainder, there is nothing equivalent to "integer division" as we see in Java
    - But there are functions for truncate, round, ceiling, floor and mod
      - (truncate 5 3) → 1 (equivalent to 5 / 3)
      - (mod 5 3) → 2 (equivalent to 5 % 3)
    - Common Lisp will always reduce fractions as much as possible, so (/ 6 8) → ¾ and (/ 6 2) → 3
  - Now consider: (+ #c(3 5) #c(2 1)) → #c(5 6)
  - But (* #c(3 2) #c(3 -2)) → 13, why?

- Comparison functions:
  - <, >, =, /=, <=, >=
  - EQL is similar to = but more restrictive, it says "numbers must be equal and be the same type"
    - (= 5 5.0) is T, (EQL 5 5.0) is nil

Preventing Evaluation

- One problem in Lisp is that everything entered at the command line is evaluated
  - (+ 3 5) → evaluate □, apply it to the evaluation of 3 and 5
  - (□ a b) → evaluate □, apply it to the evaluation of a and b
    - a and b are variables, so evaluating a returns the value it points to (if a = 5 and b = 4, then (+ a b) returns 9)
  - (length lis) – returns the number of elements in lis
  - What if lis is not a variable, but the list (a b c)?
    - (length (a b c)) → this does not return 3, instead it will probably cause an error because (a b c) is interpreted as “call function a passing parameters b and c”

Basic I/O

- CL has very powerful I/O facilities
  - However, to get started, we will look at two simple I/O operations:
    - (read) – get the next input from the keyboard and return it
      - (setf x (read))   ;; accepts one input and stores it in x
    - (print x) – sends x to the run-time window, but only works on a single item
    - if you want to print multiple items, put them in a list as in
      - (print (list x y z))
      - Note: this output will look like this: (5 3 1) rather than 5 3 1
  - Later on, we will examine formatted input and output, inputting from other sources, outputting to different windows, and file I/O

Putting It All Together

- We will rely heavily on the REPL interpreter
  - Start up Common Lisp
  - Type in a function, if it is interpreted correctly, there will be no error
  - Now you can call your function to test it out
  - If it isn't right, redefine it with another defun statement
  - Evolve your code over time
- Alternatively, type your code into an editor
  - This allows you to write your code as a series of functions so that you can get a more global idea of what you have to do, and save your code in a text file
  - Copy and paste your code from the editor into the interpreter
  - Or, save the file and load the entire file at one time (usually you will only do this after you know all of your individual functions work!)

- Environment

That’s all for now!