Welcome to the MaL language!

The Interpreter was made using the Make a Lisp tutorial.

HOW TO USE MAL:

- Nothing is evaluated unless it is in a list.
- A list is signified by () parenthesis.
- All functions are prepend functions, unless otherwise specified
- (Meaning you write* (+ 1 2) *instead of* 1 + 2 *)*

QUIRKS:

- Functions are (mostly) pure, they cannot change the outside enviroment
- Functions that have ***** (asterick) after them create temporary enviroments
- Functions that have **!** (exclamation) after them mutate data (more on this and above in functions section)
- Since the tutorial was for multiple languages, it had you bind the name of the language you implemented it in to the token '*host-language*'

Types:

() signifies a list, unless quoted, the first element is assumed a function, the rest are arguments *e.g: (f, v1, v2) returns the result of passing v1, v2 into f, while `(f, v1, v2) returns the list (f, v1, v2), for use for something else*

[] signifies a vector, like a list, but elements aren't executed *e.g: [v1, v2, v3, ...], inputting this won't have the interpreter try to apply v2 and v3 as arguments to v1*

{} signifies a map, every odd element is taken as a key, every even, as the keys value *e.g: {k1 v1 k2 v2 k3 v3 ...}, must have even number of elements*

: signifies that the following characters are a key, a special type of string *e.g: :lk returns lk as a key symbol. This is mainly used in maps*

atoms are pointers to values.

"" signifies a string, like in python.

numbers are integers, there isn't support for floats

Boolean values are **true** or **false**, and are represented as written.

nil represents a None value.

ARGV represents the list of arguments passed to your MaL program, assuming it was run as a script.

Functions:

They are initialised by using **(fn*)** in the language A bit more important than in other languages, as the only native way to loop in MaL is recursion.

There are 2 types:

Normal functions	Macros

Acts on tokens after they are evaluated (turned into data)	Acts on tokens before they are evaluated (turned into data)

In other words, macros are just functions that receive tokens instead of the data those tokens represent.

They are treated like all the other types, can be passed into functions, and are immutable like everything else. Macros are not considered the same as functions.

Core Functions and Modifiers:

These definitions will take the form

[name]: [paramtype, paramtype, ..., *paramstype] -> [ret]

[definitions]

```
e: This is an example
```

The parameters will be referred to as p1, p2, etc or as *p if it takes in a variable amount of parameters. Optional parameters will be denoted by [, paramtype]. To represent multiple valid choices for name, paramtype, etc, the choices will be seperated by | and surrouneded with [].

What is the difference between Core Functions and Modifiers?

Modifiers modify the evaluation of the rest of the ast, or the working enviroment. Core functions are simple in-out functions. All of the core functions are resolved directly to functions. The modifiers aren't resolved until they are evaluated. This is because Modifiers either use tail call recursion, or act on the local enviroment, and thus their evaluation changes depending on the outside enviroment. In the case of quasiquote and quote, they need to act on the ast, and thus need to be evaluated outside of evalast, or else the ast is evaluated before it gets to them. What this means is that they aren't bound to functions, cannot be redefined, and aren't considered symbols. If core functions are evaluated using evalast, they return a function. Modifiers throw errors when evaluated using evalast, and must be evaluated using EVAL.

TL;DR: don't pass modifiers as parameters of a function.

CONVENTION:

! denotes functions that modify outside objects e.g def! modifies the enviroment you're running in, * denotes functions that create temporary enviroments for other things to run in e.g fn* creates an enviroment where parameter names are bound to parameters

Modifiers:

def!: [Symbol, unknown] -> unknown

Takes in a symbol, and binds it to the unknown value. The unknown value is returned. This is basically how you declare variables. These variables simple mapping of symbols to values, and cannot be changed. The symbol must be redefined.

```
e: (def! a 1) binds the symbol [a] to the value 1
e: (def! a 1) (def! a 2) binds the symbol [a] to 1, and then redefines [a] to be 2
```

defmacro!: [Symbol, Function] -> Function

Same as above, but specifically for macro functions. returns the macro function.

```
e: (defmacro! a (fn* (l) + l 1)) binds the function to [a] as a macro
```

fn*: [list, unknown] -> function

This returns a function that takes in parameters, maps them to the symbols, and then evaluates p2. p2 is usually a list, and p1 is usually not empty.

```
e: (fn* (a b) (+ a b)) returns a function that takes in 2 arguments, and adds them together.
```

let*: [seq, unknown] -> unknown

p1 is a list in the pattern (key, val, key val, ...). The keys are put into a temporary enviroment where they are mapped to the corresponding value in the list. p2 is evaluated in the new enviroment and returned.

```
e: (let* (a 1) a) returns 1, because a is defined as 1 in the enviroment.
```

do: [*unknown] -> unknown

takes in a variable amount of arguments, and evaluates them all. The last element in *p is evaluated and returned.

if: [unknown, unknown[, unknown]] -> unknown

evaluates p1. If it isn't nil, false, or 0, it returns the evaluation of p2. if provided, p3 will be evaluated in the event that p1 is nil, false, or 0.

```
e: (if true (+ 1 2) (- 1 2)) returns 3, as the result is not false, nil, or 0.
```

quote: [unknown] -> unknown

Marks its parameter to not be evaluated until later.

```
e: (quote (+ 1 2)) returns (+ 1 2), delays execution until later
```

quasiquote: [unknown] -> unknown

Marks its parameter to not be evaluated until later, Unless:

- unquote: [unknown] -> unknown marks its parameter for evaluation
- splice-update: [seq] -> *unknown takes everything out of the list

```
e: (quasiquote  ((1) (unquote (+ 1 2)) (splice-unquote (1 2 (+ 1 2))))) returns ((1) 3 1 2 3), as it evaluated (+ 1 2), which is 3, and unquotes and frees the elements of (1 2 (+ 1 2))
```

try*/catch*: [unknown[, list]] -> unknown

p1 is evaluated. If there exists p2, where the first token is catch*, exceptions are caught and passed into catch*.

- catch*: [Symbol, unknown] -> unknown: binds the exception that was caught to p1, and executes p2, with that new enviroment

If p2 doesn't exist, it just executes p1, without catching exceptions

```
e: (try* 123 (catch* e 456)) returns 123. If an exception occured, it would return 456
```

Core functions:

[+|-|*|/|%] as c: [number, number] -> number

returns p1 [c] p2

=: [unknown, unknown] -> bool

returns whether p1 and p2 hold the same value

```
e: (= 1 2) returns false
e: (= 2 "r") returns false
```

[<|>|<=|>=] as c: [number, number] -> bool

returns p1 [c] p2

[list|vector|map|sequential|atom|nil|keyword|symbol|string|number|macro|fn]? as c: [unknown] -> bool

returns whether p1 is the specified structure sequential is defined as either a list or a vector fn is for functions

```
e: (list? (1 2)) returns true
```

[true|false]? as c: [unknown] -> bool

returns whether the value is same as c

```
e: (false? true) returns false, and (false? 0) returns false
```

contains?: [map, key] -> bool

returns whether the specified key exists in the map, whether p2 is a key of p1

empty?: [unknown] -> bool

returns whether the given data structure is empty

not: [bool] -> bool

returns the inverse of a bool value

```
e: (not true) returns false
```

cond: [*unknown] -> unknown

the 1st, 3rd, 5th... element in *p1 is mapped to the element after it, henceforth referred to as key-value pairs the key is evaluated. If it is true, the value is evaluated and returned. preference determined by order in the list. basically just a series of if-else statements, or a switch if you think of it that way.

```
e: (cond false 7 true 8 true 9) returns 8
```

[vec|seq|hash-map|symbol|keyword|atom] as c: [unknown] -> c

seq represents list returns the result of trying to convert p1 into c

[vector|list] as c: [*unknown] -> c

returns the compilation of multiple elements into a [c]

```
e: (vector 1 2 3 4 5) returns vector [1 2 3 4 5]
```

deref: [atom] -> unknown

returns the value held in an atom

reset: [atom, unknown] -> unknwon

p1 is modified to point to p2, and p2 is returned

swap: [atom, function, *unknown] -> unknown

evalutes the result of passing *p3 into p2, and puts the return value into p1. returns the new value held by p1

cons: [seq, unknown] -> list

returns the result of adding p2 to the end of p1

concat: [*seq] -> list

returns the result of compiling the elements of all the lists and vectors in *p1 into 1 list

nth: [seq, number] -> unknown

returns element at the p2-th index in p1 in otherwords, given p1, return the nth element.

first: [seq] -> unknown

returns the first element of the sequential if it exists, otherwise nil

rest: [seq] -> unknown

returns everything in p1 except the first, if it exists, otherwise nil

apply: [function, *unknown, seq] -> unknown

p2 can have 0 or more elements similar to a function call, but p3 has its elements freed and passed into p1 as parameters

```
e: (apply (fn* (a b) (+ a b)) 4 (5)), 5 is in a list, but is freed from the list and pass into the function
```

map: [function, seq] -> list

applies the function to all the elements in p2. The return values are compiled into a list and returned

conj: [seq, unknown] -> seq

if p1 is a list, prepends p2 to p1 and returns it (assumes p2 is a function) if p1 is a vector, appends p2 to p1 and returns it

assoc: [map, *unknown] -> map

takes in a map, and then uses every other value as a key for the one after it. appends these pairs to the map and returns it.

```
e: (assoc {} :k 1 :l 2) returns {:k 1 :l 2}
with :k being associated with 1, and :l associated with 2
```

dissoc: [map, *unknown] -> map

removes all pairs in p1 whose key is in p2.

get: [map, unknown] -> unknown

gets the value from p1 whose key is p2

keys: [map] -> list

returns the list of keys in p1

vals: [map] -> list

returns the list of values in p1

pr-str: [*unknown] -> string

converts all elements in *p1 into a string, seperated with spaces. Shows the escape sequences

```
e: (pr-str "hi\nj" 2) returns:
"hi\nj 2"
```

str: [*unknown] -> string

converts all elements in *p1 into a strong, and concatonates them. Obeys escape sequences

```
e: (str "hi\nj" 2) returns:
"hi
j2"
```

prn: [*unknown] -> nil

applies pr-str on *p1, and prints it

println: [*unknown] -> nil

applies str on *p1, and prints it

read-string: [string] -> ast

converts the string into the tokenised representation mainly used with eval function

```
e: (read-string "( + 1 2)") returns (+ 1 2)
```

eval: [ast] -> unknown

evaluates a token in MaL when combined with above, can evaluate a string in MaL

load-file: [string] -> nil

reads the file with the name in p1, and trys to run it as a MaL program.

slurp: [string] -> string

reads string, tries to open the file with the corresponding name, and returns its contents as a string

spit!: [string, string] -> nil

reads the file with the name p1, and writes p2 to it. DOES NOT APPEND!

with-meta: [[list|vector|map|function], unknown] -> [list|vector|map|function]

p1 gets a new attribute - meta - which contains that value in p2 the new p1 gets returned

meta: [unknown] -> unknown

returns the meta value of p1 if exists. otherwise nil

throw: [string] -> Exception

throws an exception with the msg given in p1

readline: [string] -> string

reads a line of input. If the line is End of File, return nil. Otherwise, return the input

time-ms: [] -> number

returns the unix timestamp of the current time

python-eval: [string] -> unknown

evaluates a string in python

exit: [] -> N/A

exits the program. Is mainly for use in the interactive case.

Special Characters

Symbol	Definition
semicolon (;)	makes comments, everything on the line after this is not evaluated
colon (:)	makes a key for dicts, a special type of string.
at-sign (@)	derefernce the value in an atom
single quote (')	quotes the next token
backtick (`)	quasiquotes the next token
tilde (~)	unquotes the element in a quasiquote (only for use in a quasiquote)
tilde + atsign (~@)	splice-unquotes the element in a quasiquote (onlt for use in a quasiquote)
caret (^)	makes the next token the meta value for the token after it (in other words, ^ meta_data meta_holder)