I'm going to lay down everything i know and think about this interpreter.

I will focus on explaining everything as much as possible from all different aspects of the language.

What i think, what i know, why exists, how it works, etc.

But there will be no particular order and all thoughts will be scattered, so it is a raw brain dump.

The name of the language is "oh"

"oh" and "meh" are my "foo" and "bar"

the same way "oh my cat is very nice" is my lorem ipsum

So instead of "foo.js"

The file is "oh.js"

Instead of a command named "foo", the command that loads "oh.js" is named "oh"

Actually the command is an alias in my .profile

```
alias oh='rlwrap -a -n node /home/vms/oh/oh.js'
```

What this language is?

It is a rpn stack based language written in javascript.

It tries to rely as much as possible on javascript, to provide as much as interoperation with it as possible.

The main goal is to inherit access to all the browser APIs and do everything js can do but in a weirder way.

It is also my playground to experiment with programming.

In a way it is a reflection of my current understanding about programming. Everything i know about programming is being reflected either as part of the core interpreter or as a word definition.

It grows when my understanding grows.

When my understanding grows and the current implementation does not match my understanding anymore, it gets rewritten from the scratch.

But it also amplifies my understanding.

It is a playground for my mind and a training process to teach me about almost any concept about programming.

I can test any idea without friction, integrate it to the language, combine it, drop it or evolve it.

For me this interpreter is just a playground for ideas.

An idea can be tested immediately by just creating a word definition, either in the language itself or in javascript.

New syntax can be added by just creating a function that reads the source code and returns another function that will perform the logic, or performs the logic right away.

The language does not care about almost anything.

It only cares about reading a token and turning it into a function.

Some tokens provoke arbitrary execution of code while the rest are delayed and stored as functions.

The interpreter just reads tokens and collects functions, which will be iterated and called in sequence later.

If for example it reads a number like "1", it will create a closure in javascript that stores that number and when executed will push it on the stack.

1 becomes in a way:

```
let value = 1; return () => stack.push(value)
```

So the interpreter just reads a number and captures it, returning a function, that will get executed later.

I call the process of generating functions the compile procedure or compile time, and I see the execution of those functions as the runtime of the interpreter.

It is not compiling anything, but delaying executions and precomputing decisions.

The goal of the compiling procedure is to avoid as much as possible runtime overhead, mainly runtime lookups.

It is not the same to lookup what a word means once, than 200 times in a loop.

Although performance is not the real reason the compiling procedure actually exists.

The main reason is to freeze definitions so when a definition relies on another definition and gets "compiled", if the other definition gets redefined, this defintion that used the other one will not be affected by the other definition being updated.

In other words, it avoids dynamic binding.

All the words are found in an environment in the form of a function. An environment is just:

```
{ parent: pointer_to_another_environment_or_undef, word: {} }
```

The compiling procedure just searches for the chain of environments and looks whether the token is found in the word: {} object that serves as a lookup table for words. All words are stored in that word: {} object, the interpreter just finds and returns them.

When we are creating a definition, we are just reading tokens, turning them into functions, then storing those functions in an array.

The colon word is the main interface for the language to create a new definition.

It reads the name of the word that is going to be generated in the current environment and starts gathering functions and storing them into an array.

For example the definition of a word that pushes three numbers on the stack when executed:

```
: push-three-numbers 1 2 3 ;
```

The colon word ":" is an immediate word.

An immediate word executes at compile time.

The main procedure to turn a token into a function is a javascript function named compile element

compile element takes one argument and tries to return a function. It tries to unify everything into a function.

A token in the language will always represent an action.

This action gets represented internally by a function.

A number becomes a function that will push the number on the stack.

A word defined in the environment is just a javascript function.

A primitive word is a word defined in javascript directly like:

```
env.word.name = () => console.log("hi i am the word named 'name'")
```

env is the current environment pointer. It is where compile element starts searching for words.

When compile element is given the argument: "name" like in: compile_element("name") in js

It will start looking from the environment pointer:

```
if (env.word.name)
{
  return env.word.name
}
```

In this case if we defined that env.word.name, it will be a javascript function, compile element will just return that function.

compile element is the whole interpreter.

It only cares turning everything into a function or excuting immediate words.

There are two types of immediate words.

What i call immediate 0 and immediate 1

The 0 or 1 is just a flag compile element uses to distinguish between those two types.

immediate 0 means "execute this right now"

immediate 1 means "execute this right now, it will return a function on the stack which you will pop from the stack and return to the caller"

If a word is immediate it will execute it, if it's immediate 0, compile element will return undefined, if it's immediate 1, that word when executed will push a function on the stack and compile element will return it

If the token does not map to a word, compile element will fallback to compile atom, compile atom tries to provide syntax sugar for literals and fancy stuff.

A token is what read word returns, read word just reads the source code until a space, newline or tab, and returns the token it has read.

Some characters are special for read word, they do not require spaces between them and constitute an entire token by themselves.

The special characters are ( ) [ ] { } ` and "

This is because those characters are used for special syntax provided in the form of immediate words.

The starting delimiters like "(" "[" and "{", are immediate words that execute at compile time and start reading from the source code, to implement a different syntax. The closing delimiters ")" "]" and "}" are just a sentinel character for those reader words. Those immediate words will read until they find in the source code the closing delimiter.

"(" "[" and "{" just call the read word function to retrieve a word and do something with it. The read word function knows both the starting and closing delimiters are special characters, so when it finds one of them it will return it without expecting spaces betwen them.

That means that for the interpreter, reading

```
"(1 2 3)"
```

is equivalent to reading

```
 ( 1 2 3 )
```

The simplest example is "(" which is an immediate 1 word. Being an immediate word means compile element will execute it when it receives the "(" token as its argument, and being immediate 1 means this word has to push a function on the stack, which compile element will take and return to the caller.

The read eval print loop in node will just read a line of source code and make this to become the source of the interpreter,

Then it will call read word and give whatever read word returns as the argument to compile element.

Compile element will either return undef or a function. The repl will store that function into an array.

After all the source code has been exhausted, the repl will iterate through this array and call all the functions compile element returned.

If we were to type something in the repl like:

```
1 2 3
```

The repl will set the source to be the string "1 2 3", then create an empty array, then start calling read word until read word returns undefined, meaning the source is exhausted, giving whatever read word returns to compile element and if compile element returns a function, it will get appended to that array, After read word signals end of source code by returning undefined, the repl will take this array and iterate it, calling all the functions.

In this case what it would do is to call compile element three times

```
compile_element("1")
compile_element("2")
compile_element("3")
```

Compile element will search in the environments starting from the current environment, fail because there is no word defined 1, 2 or 3 in any environment, then fallback to compile atom.

Compile atom only applies several regular expression checks on that atom to determine what it is.

In the case of 1 2 and 3, they match the number_regex regular expression which is:

```
const number_regex = /^-?\d+(\.?\d*)$/
```

And it will return a function that will push that number on the stack:

```
if (atom.match(number_regex))
{
  const number = parseFloat(atom)
  return () => put(number)
}
```

So compile element will return those closures for those numbers and the repl will store them into an array

Similar in a way to this, but the numbers are lexical js variables instead of the literal numbers:

```
[() => put(1), () => put(2), () => put(3)]
```

That's what the repl will generate from reading the line of input "1 2 3"

Then it will just iterate this array and call all those functions in order.

That moment is what i call runtime, the execution of the collected functions.

Compile time is just when we are generating them, when we call compile_element(read_word()) and gather the results into an array.

If we are gathering functions into an array, and we give compile_element the name of an immediate 0 word, compile_element will return nothing, it will just evaluate that word right away, and we won't get anything to push on our array.

If we instead give it the name of an immediate 1 word, compile element will execute that word, the word will generate a function and return it on the stack, compile element will return us that function and we will append it to our array

This is all the interpreter does, the rest is just to create words, either normal words, immediate 0 or immediate 1 words and maybe extend compile atom with more regular expressions.

The interpreter does not know what syntax is, syntax is just tokens separated by spaces, newlines or tabs, and some characters that are a token by themselves.

It does not know what an if statement is, what a loop is, etc. It just knows there are environments which store words, those words are either returned or executed and some words might return a compilation unit in form of a function when executed. Plus the compile atom function which is just a bunch of regular expressions.

If compile atom runs all the chain of checks on this atom and does not recognize it, it will trigger an error.

New syntax not integrated in compile atom as a regular expression is implemented by immediate words that execute at compile time, read the source code either by characters or by calling read_word and perform logic at compile time, add entries into the compilation time environments, and/or return a function on the stack that will get inserted to the compilation array.

The "(" is the simplest example of an immediate 1 word that implements new syntax and returns a function on the stack that will get appended to our compilation array.

The "(" internally calls a js function named build_list, that creates an array and calls read word until read word returns the ")" token.

```
(1 2 3)
```

If we were to type this in the node repl, the repl will call read word and read word will return "("

Note that "(" does not require spaces and read_word returns "(" alone instead of "(1" because "( ) [ ] { }" are special characters that read word returns when it finds them without expecting any space.

So for the interpreter this will always be seen as if we type: " (1 2 3) "

Then the repl calls compile_element("(") and compile element finds this word in the root environment and sees that is an immediate 1 word, so it executes it and expects it to return a function on the stack, which will return to the repl and the repl will store in the array of functions is generating.

the "(" word will execute and will start calling read word until read word returns ")", it will just store all those results into an array.

This is an aproximation of what it does when compile element calls it:

```
const list = []
let word
```

```
while ((word = read_word()) !== ")")
{
  list.push(word)
}
stack.push(() => put(list.slice())
// list.slice() is a js idiom to copy a list, similar to the spread opera
// but the spread operator chokes with really long lists in v8 at least b
```

- The closure returns a new copy of the list
  instead of the original one to avoid mutation of that list, so every time you execute the closure that "(" generated, it will return a fresh copy of the list that was been built at compile time.

When "(" receives the ")" token from read word, it stops reading words and generates a javascript closure that when executed will push that array on the stack, then it just pushes this closure on the stack, compile element takes this closure and returns it to the repl and the repl stores it into the array of functions it is gathering. Once the repl knows there is no more input to compile, it will run the functions in this array. In this case the array will contain:

```
[() => put(list.slice())]
```

And it will just iterate it and call that function.

That function is the closure the "(" word generated from reading words and pushed on the stack at compile time.

That's how syntax, control operators, loops, etc is added into the language.

The language does not know what "(" does more than that "it executes and returns something at compile time"

The "(" extends compile element in a way, since compile element trusts this word to generate a compilation unit.

I like how the language does not care about anything, yet it can be extended almost infinitely by immediate words.

For example the interpreter does not know what "if" means, except that "if" is an immediate 1 word.

the "if" word will read the source code and generate a closure that when executed, will perform the logic of the "if" statement.

Everything in this language is mainly delayed execution and partial evaluation.

I tried my best into unifying everything into a function. Including variables.

Variables do not exist in this language, the word bind creates something that acts as a dynamic variable, but it's just a closure that pushes a value on the stack and this value can be mutated.

The reason i say i tried to unify even variables into a function is because bind generates a function that behaves as a variable.

The function is just this:

```
const binding = () => put(binding.value)
```

It relies on the fact that javascript functions are objects to just store the value property on the function object itself, making that closure become a cell or storage unit that when executed will push that value on the stack.

In order to mutate a value we just need a reference to that function and do something like:

```
binding.value = 3
```

And now when executed this function will push the number 3 on the stack.

That means that everything that stores a reference of that function is also storing a way to access and mutate this value.

It also implies that this binding is dynamic, if you mutate the value, everything that has a reference to that function will see the effects, since the function just pushes that value on the stack when executed, the value will be whatever value it has the moment this function gets executed, so the value will always be the last value being set.

The environments are cool.

Any definition that starts reading words and compiling them by calling compile atom and read word, will likely create a new environment before doing anything, and set this new environment as the current environment by overriding the env pointer, which is a global pointer used by the interpreter to point to whatever the current environment is and gets overwritten several times during compilation to always point to the current compilation environment. After a definition has been compiled, the current environment pointer will point back to the previous environment. Every definition that messes with the current environment pointer will store a reference to what it was pointing to and later restore it to point back to that reference.

The main example of a definition is the colon word, which is an immediate 0 word:

```
env.word[":"] = () =>
{
  const name = read_word()
  const code = []
  const old_environment_pointer = env
  env = make_env(env)
  let word
  while ((word = read_word()) !== ';')
  {
    const value = compile_element(word)
    if (value)
    {
      code.push(value)
    }
  }
  env = old_environment_pointer
  env.word[name] = () => { for (let fun of code) { fun() }}
}
env.word[":"].immediate = 0
```

That's mainly what the ":" word does, although it internally uses a function that performs this behavior, since it's a common pattern for any word that creates a definition.

The colon word executes at compile time, it's an immediate 0 word, so compile element executes it and returns undefined, because immediate 0 words just execute at compile time, but do not need to return any function on the stack, since they are mainly executed to perform some sort of side effect.

In the case of the colon word, the side effect is to generate a new word definition in the current environment.

The current environment when the interpreter starts, points to the root environment, which is an environment that has no parent:

```
  let env = { parent: undefined, word: {} }
  const root = env
```

root is just a pointer to the first environment ever created, which is the root environment.

When we type in the repl:

```
  : some-word-name 1 2 3 ;
```

The repl calls compile_element(":") and compile element finds ":" which is defined in the root environment and is an immediate 0 word.

Then the function stored in root.word[":"] is executed

This function stores a reference of whatever "env" was pointing to, then creates a new environment and overrides "env" to point to it, so now compile element will start searching words there, this environment inherits from whatever was the current environment when ":" gets executed at compile time.

Since it's a top level definition, the environment created will be a child of the root environment, and now the current environment will be this one.

Then it will start compiling 1, 2 and 3, until it reads the semicolon that signals the end of this definition.

Then it restores the current environment pointer to point back to the root environment and registers this definition into the root environment since it was the current environment when ":" was executed.

Now the root environment is:

```
root = { parent: undefined, word { 'some-word-name': () => { for (let fur
```

The environment that was generated by the colon word has been lost and garbage collected by javascript, since that environment was only temporarily referenced by the current environment pointer, but once the colon word ends compiling, the current environment pointer now points to the root environment again and no one has stored a reference to that new environment, so js schedules that environment to get collected for the gc.

That means that the environments that are created in definitions exist only during the compile time of that definition, they inherit from whatever environment was the current environment at the time of that definition and they get destroyed after the definition ends.

It is lexical scope at compile time.

A definition can be created inside another definition. It will generate a new environment that inherits from the parent, then compile stuff, then restore the environment pointer to point to the previous one. If that definition gets inserted into the previous environment, the outer definition will find it because it will call compile element and compile element will see that word registered in the environment.

The colon word creates definitions at compile time and injects them into whatever was the current environment at that point.

Note that in our some-word-name example, we were typing this into the repl and the repl called compile element to evaluate ":" at compile time, since ":" is an immediate 0 word, which means that just executes and returns nothing, compile element returned undefined and the repl did not append any function on its compilation array, since the repl just calls compile element until the source code is exhausted, then stores those functions in an array, and after there is no more input it iterates this array, calling each function in sequence.

The repl and a definition of a word are similar in the way that they collect functions into an array and their execution is just to run those functions in sequence later.

In the case of the line that only contains a definition like ": some-word-name 1 2 3 ;" the repl calls read word, read word returns ":" and the repl gives that to compile element, compile element executes ":" and returns undefined, so the compilation array of the repl is empty.

If the line were instead

```
: some-word name 1 2 3 ; some-word-name
```

The repl will call compile element with ":" as before, ":" will read words until it finds the ";" semicolon, then compile element will return nothing, the repl will call read word again and this time read word returns the string "some-word-name" which the repl gives to compile element and since ":" has registered this word in the root environment, compile element finds it and returns it to the repl, so the compile array of the repl contains the javascript function that ":" created:

```
const the_repl_execution_array = [() => { for (let fun of code) { fun() }
```

and the repl will keep gathering functions until read word signals eof by returning undefined.

Then the repl will iterate that array and call all the functions.

```
for (let fun of the_repl_execution_array)
{
  fun()
}
```

Which is exactly the same kind of function the colon word and any other definition generates.

Just note that when compile element returns undefined, which is when an immediate 0 gets executed, nothing is pushed in those arrays.

And when an immediate 1 gets executed, the function that immediate word returns on the stack gets appended to those arrays.

If the repl had the line ": some-word-name 1 2 3 ;"

The word definition will be like:

```
env.word['some-word-name'] = () => { for (let fun of [() => put(1), () =>
```

And the repl execution array will be empty, since it only has read ":", compile element returned nothing because ":" is immediate 0, so the repl did not append anything, then the repl calls read word again, but read word returns undefined too, because the line had no more input and the ":" word has exhausted the input itself, reading until a terminating semicolon ";".

That means that if a ":" is found inside the code of an outer colon definition, the outer ":" will call compile element with the ":" argument, which makes this process recursive, meaning that any immediate word that calls compile element will recurse if the argument it gives to it is the name of that immediate word itself, since compile element will just find that this word is immediate and execute it.

That's how or why colon definitions can be nested:

```
: outer-definition
  : inner-definition 1 2 3 ;
;
```

- Just as a note, the node repl only reads by lines so it would choke on this code being separated by newlines, in the node repl would have to be one line: ": outer-definition : inner-definition 1 2 3 ; ;" This is a strong limitation of the node repl. Note also that if we were loading an entire file, either from node or the browser, we would be creating an array of functions and read and compile all the words in the entire file, storing all the functions in the array, then after we find the end of the file, we will iterate this array and execute all the functions, mirroring what the repl does for a line and what the colon word does for a definition.

When whatever was reading the source code (the repl, the procedure that loads a file, the oh tags in the browser, etc), finds the first ":" and feeds it to compile element, the ":" word executes and starts creating a definition named "outer-definition", for this definition it creates a new environment that inherits from the current environment (in this case the root environment) and starts compiling words, by calling read word and giving that word to compile element.

The ":" word reads the word "outer-definition" and sets it as the name for the word:

```
const name = read_word()
// name = 'outer-definition'
```

Then stores a reference of whatever was the current environment

```
const previous_environment = env
// previous_environment = root
```

Then creates a new environment that will inherit from the current environment, and makes this environment become the current one.

```
env = make_env(env)
```

So now env is a temporary environment that the ":" has generated and inherits from the root environment, since make_env returns:

```
{ parent: root, word: {} }
```

This can be considered the environment of the "outer-definition" word, since it is generated to become the current environment for the compilation of that word.

Then the ":" word starts reading words until it finds a terminating semicolon ";"

```
const code = []
while ((word = read_word()) !== ";")
{
   const value = compile_element(word)
   if (value)
   {
     code.push(value)
   }
}
```

- Note that this loop will call compile element every time until a ";", but the environment in which compile element will start looking for definitions is that environment ":" has generated for that current definition, the environment of the word "outer-definition".

Then it will read the next word and find another ":", which compile_element will execute since it's an immediate 0 word, and return undefined after executing it.

The ":" word then will recurse since it's calling compile element and compile element is calling ":" and the same process will repeat.

The ":" word will read the next word, which is "inner-definition" and say:

```
const name = 'inner-definition'
```

then do the same thing with the environments, store a reference to the current environment and set the current environment to a new environment that inherits from the previous one:

```
const old_environment = env
env = make_env(env)
```

```
// env = { parent: the_environment_of_outer_definition, word: {} }
```

Now the current environment is an environment that inherits from the environment of the "outer-definition" word, and a pointer to the environment of the "outer-definition" word is stored as the "old_environment" reference, to later restore "env" to point there again after it finds the semicolon ";".

then the inner ":" will keep reading words in our example it was ": inner-definition 1 2 3 ;" so it will call compile element with those numbers and create an array like:

```
const code = [() => put(1), () => put(2), () => put(3)]
```

Then restore the "env" pointer to point back to the previous environment:

```
env = old_environment
// env = the_environment_of_the_outer_definition
```

And register the word in that current environment.

```
env.word[name] = () => { for (let fun of code) { fun() } }
// env = outer_definition_environment
```

Then it ends execution and the outer ":" definiton resumes from there.

In this case the outer definition had no code inside it.

```
: outer-definition
  : inner-definition 1 2 3 ;
;
```

The outer ":" only has read the name of the word to define, then it found a ":" inside it, so compile element executed that while the outer ":" was executed, and returned undefined to the outer ":", so the code array of "outer-definition" would be empty and it's runtime execution would be to iterate through an empty array.

Also note that both the compile time environments that both ":" generated do not exist anymore, since when a ":" finds a ";" and terminates compilation, it restores the pointer of the current environment to point back to the previous one and those environments do not get referenced by anything so javascript garbage collector frees them.

In this example it means that the inner definition was useless, since it was only visible during the compilation of the outer definition, but the outer definition didn't have more code except for the inner definition itself. Nothing has captured or frozen the inner definition and both environments of those words have been lost, so the function that inner definition generated has been lost.

It also means that outer-definition is a "do nothing" operation, because it's execution array is empty, since in it's code there was only an immediate 0 word that didn't return anything.

If the code were to be like this instead:

```
: outer-definition
  : inner-definition 1 2 3 ;
  inner-definition
;
```

Then the same procedure would apply, but after the inner definition ends execution and injects the generated definition into the environment of the outer definition, the outer definition keeps reading words and it finds the word "inner-definition" in it's code, which gives to compile element and compile element finds it, because the current environment is the environment of the outer definition and the inner definition was defined in that environment, so it gets appended to the compilation array of the outer definition.

That's when the inner-definition gets frozen into the definition of the outer and even if the environments of inner definition and outer definition disappear, the outer definition gets defined in the root environment and the inner definition gets frozen as a closure in the code array of the outer definition.

```
root.word['inner-definition'] = [() => { this is a stored reference of ou
```

even if outer_definition_env and inner_definition_env (that env = make_env(env) both ":" generated at compile time) do not exist anymore, the function that the inner definition injected into the outer definition environment has been stored into the array of code of the outer definition word and the outer definition is itself available in the root environment.

Every definition that does

```
old_environment = env
env = make_env(env)
```

will also restore that pointer later when it ends compilation with

```
env = old_environment
```

That means that the first ":" stores a pointer to the root environment and the inner ":" stores a pointer to the environment the outer one generated, then later restores env to point back to that environment so when the inner ":" ends compilation, restores env to point back to the environment that the first ":" generated and when the first ":" ends compilation will restore env to point to the root environment.

At the end, no matter how much we nest definitions, we will come back to the root environment after compiling everything.

Unless this procedure fails to restore properly the current environment pointer which will likely become a problem.

If the interpreter finds an error, it will restore env to point back to the current environment.

An error throws a javascript error, the repl will keep evaluating, but reading from files or any other way will abort and stop reading.

If an error is found at compile time while we are messing with the environments, it will lead to an unknown state, by default all the interpreter will just abort, but if we were to continue like in a repl, the compiling procedure will fail to restore the env pointer, so an error will always point to root again, which only affects the repl because reading a file or a tag would abort all evaluation and compiling anyways.

An error in the interpreter is only triggered when something tries to retrieve on the stack more elements than the stack has and when a word is not recognized at compile time.

Some words will trigger an error by themselves, for example when an immediate word reads from the source code expecting a word and read word returns undefined.

The "(" word for example will trigger an error telling it did not find a terminating ")" if read_word returns undefined when "(" was expecting to stop reading words when read_word returns ")", since read word returning undefined means the end of source code, the "(" word understands that there is no more code to read and it never found a closing parenthesis ")".

Same for any word that expects a delimiter to stop reading words and any word that expects a word name in the source code.

For example the word "bind" reads a name from the source code and creates a word that associates a value from the stack with a name.

```
24 bind twenty-four
```

The bind word will read the word twenty-four and create a word in the current environment at compile time that will push 24 on the stack.

bind is an immediate 1 word, which means that not only executes at compile time (compile element executes it instead of returning it), but also returns a function on the stack at compile time that will get appended to the execution array of whatever we were compiling (compile element just executes bind and knows bind will return a function on the stack which will return to the caller, be it the repl itself, a colon definition, etc)

bind is immediate 1 because it has to operate at compile time, since it needs to read a word name and create an entry on the current compilation environment with that name so it becomes available after bind registers it and can later be used.

After executing bind, the word "twenty-four" will be available for that compilation unit, be it the line of the repl, the whole file, etc.

In this case we are using it inside any definition so it runs in the top level. The top level is the repl execution array, the file execution array, etc.

So if we later type "twenty-four" the repl or whatever will call compile element and compile element will find twenty four in the current environment (in this case the root environment)

And twenty-four should push the number 24 on the stack when executed.

The problem is, the number 24 must come from the stack, since the job of the bind word is to associate names with word values, to avoid stack juggling, but the bind word executes at compile time.

The binding the bind word generates is that function that acted as a cell

```
const binding = () => put(binding.value)
```

It's a function that pushes a value that it's stored in itself, since functions in javascript are objects and binding.value is the value property of the function being generated.

In order to set the binding.value property to the value of the stack it has to delay the assignment to the runtime, this is why bind is an immediate 1 word that returns a function on the stack which compile element returns to the caller, because the function it returns is the setter.

The bind word does this when compile element executes it:

```
const name = read_word()
// name = "twenty-four"
const binding = () => put(binding.value)
binding.value = 0
env.word[name] = binding
stack.push(() => binding.value = stack.pop())
```

Then compile element receives that function from the stack:

```
() => binding.value = stack.pop()
```

And returns it to the caller, if we were on the repl, the caller would be the repl itself and this function would be appended to the repl_execution_array, which will later execute.

So the line in the repl:

```
24 bind twenty-four
```

will make the repl call compile_element("24") which will return that closure compile atom generates similar to: () => put(24) but 24 being actually a lexical js variable, result of parseFloat("24"), which the repl will append to its own execution array:

```
repl_execution_array = [() => put(24)]
```

Then the repl will read the next word and give it to compile element:

```
compile_element("bind")
```

And compile element will see it's an immediate 1 word, which has to execute and return a function on the stack:

the bind word calls read word and reads "twenty-four", then define an entry in the root environment with that binding, then push the setter (the () => binding.value = stack.pop()) on the stack and compile element will return that function to the repl, which the repl will add to it's own execution array:

```
repl_execution_array = [() => stack.push(24), () => binding.value = stack
// the code was: 24 bind twenty-four
```

The repl will find no more code to read so it will just iterate that array and execute each function.

The first function is just the number 24 converted to a function that pushes it on the stack when evaluated, so the repl will execute it and the stack will have that number on top.

The second function is the setter that bind returned on the stack, it takes an element from the stack and stores it to binding.value.

If we later were to use the word twenty-four, it will push whatever binding.value holds.

Let's try with a colon definition that creates a binding to use it later in a very complex operation:

```
: complex-operation bind number
   number 1 +
;
```

The word complex-operation creates a binding to associate the name "number" to a value from the stack, to use that name later to refer to that value, then it uses that binding to push that value on the stack, then it pushes 1 on the stack and calls the word "+" which will take those two numbers and sum them to return the result on the stack.

- Note that complex-operation does not really need bind here, but it's the simplest example to demonstrate what bind does. complex-operation could just be defined as: ": complex-operation 1 + ;"

In this case the top level (the repl or whatever, although the repl cannot really read multiple lines) will call read word which returns ":" and call compile_element(":") which will start a definition.

The ":" word will do that thingy with the environments, read the word "complex-operation", then start storing functions into its compilation array.

The first word in the definition of complex-operation is the word "bind", which being an immediate 1 word will execute at the compilation time of the complex operation, note that at this time the current environment points to the environment the ":" word has generated for the compilation of the complex operation (env = make_env(env)).

The word bind executes and reads the word "number" and creates that binding function

```
const binding = () => put(binding.value)
```

And registers it to the current environment at that compile time:

```
const name = read_word()
// name = "number"
env.word[name] = binding
```

then returns on the stack the setter function:

```
stack.push(() => binding.value = stack.pop())
```

which the compile element function will take from the stack and return to its caller, being the caller the ":" word that its gathering functions for the code array of the complex operation, so the code array of complex operation is:

```
code = [() => binding.value = stack.pop()]
```

But also the binding has been registered in the environment of the complex operation when bind did:

```
env.word.number = binding
```

Then the ":" word will keep reading words, remember that the definition was:

```
: complex-operation bind number
   number 1 +
;
```

So after the word bind, the word number is found and compiled, compile_element("number") returns the word that bind has just registered in the current environment and ":" appends it to the execution array of the complex operation:

```
code = [() => binding.value = stack.pop(), () => put(binding.value)]
```

then it reads the 1 which gets converted to () => put(1) by compile atom and the ":" word appends to the code array

```
code = [() => binding.value = stack.pop(), () => put(binding.value), () =
```

then it reads the word "+" which is defined in the root environment as:

```
root.word['+'] = () => { const [one, two] = stack.splice(-2); put(one + t
```

in a way, the complex-operation word which was defined like:

```
: complex-operation bind number
   number 1 +
;
```

Is internally represented by the interpreter as:

```
root.word["complex-operation"] = () =>
{
  for (let fun of [() => binding.value = stack.pop(), () => put(binding.v
    {
        fun()
    }
}
```

If we try to use that word:

```
24 complex-operation
```

The repl or whatever will evaluate the number 24, which will get pushed on the stack, then evaluate the complex-operation word, which will iterate over that array.

```
[() => binding.value = stack.pop(),
 () => put(binding.value),
 () => put(1),
 () => { const [one, two] = stack.splice(-2); put(one + two) } ]
```

The first function is the setter that bind returned on the stack when compiling, it will take that 24 from the stack and set it to the value of the binding:

```
binding.value = 24
```

The second just pushes the binding.value on the stack, in this case just returning the number again:

```
put(binding.value)
```

Then push 1 on the stack:

```
put(1)
```

Then take two elements from the stack and sum them:

```
const [one, two] = stack.splice(-2)
put(one + two)
```

And that's all.

The bind word just creates a binding at compile time, registering the word in the environment and appends the setter to the execution code to be executed at runtime.

At compile time the binding can be found even if binding.value has not been set yet (bind initially sets it to 0 anyways) and when the word bind creates in the environment gets executed, will push whatever binding.value is at that time.

Since bind creates the binding and injects the setter into the compilation array, when that code later executes will always set the value to an element from the stack, but the binding is available during the compilation process, since it's just a function pointing to it's own property and this property will get set later, whenever this function executes the setter will have been already executed.

In that example:

```
24 complex-operation
```

The complex-operation could refer to the word named "number" at compile time even if the "number" is just put(binding.value) and binding.value is not being set until runtime, because "number" is just a function that bind creates in the environment of complex-operation, then that function gets stored in the code array of complex-operation.

in the complex operation array the first function is the setter, so the first thing complex-operation will always do is to take a value from the stack and set that binding.value to whatever came from the stack. Then whenever the "number" word is used inside it's definition will just be the binding function that pushes that value on the stack again.

- Note that bind creates a dynamic binding and that sometimes is not desirable

The word bind can create multiple bindings at once if given a list instead of a name

```
1 2 3 bind (one two three)
```

Now we have three words that are bound to their respective values and we can use them:

```
one
two
three
```

And we will have [1, 2, 3] on the stack

What bind does since it's an immediate 1 word which means it has to execute at compile time and also return a function, is similar to what it does with one name, but instead of creating one binding and word in the current environment, it creates multiple.

When the repl hits the word "bind" and gives it to compile element, compile element calls it and expects a function to be on the stack after its evaluation, which will return to the repl so the repl adds it to its own compilation array.

The word bind reads a word from the source code by calling read_word, which in this case has returned the word "(".

The word bind checks if the word returned by read word is "(" instead of any other number and calls the build_list function, which is what internally the "(" word actually uses to read until a terminating ")" and returns the generated list.

That means that at compile time, when bind reads a name and the name turns to be an opening parenthesis, it will take this to understand you are giving it a list of names and call build_list, which will start reading until the closing parenthesis and return a list.

Then bind iterates that list to generate the bindings, for every name in the list it generates a binding function

```
const binding = () => put(binding.value)
```

and an entry in the current environment

```
env.word[name] = binding
```

In the case of

```
1 2 3 bind (one two three)
```

The bind word will receive the list ['one', 'two', 'three'] and iterate them, creating a binding function and an entry in the environment for every name:

```
const list_of_names = build_list()
const bindings = []

for (let name of list_of_names)
{
  const binding = () => put(binding.value)
  bindings.push(binding)
  env.word[name] = binding
}
```

Then it has to return the setter on the stack so compile element takes it and gives it to the repl and the repl will collect it for its own array execution.

The function it returns is like:

```
function ()
{
  const values = stack.splice(-list_of_names.length)
  // retrieves the same number of values than names were in the list
  for (let i = 0; i < list_of_names; i++)
  {
    bindings[list_of_names[i]].value = values[i]
  }
}
```

That's the setter that bind puts on the stack for compile element to give to its caller.

The repl will take that and append it to its own execution array.

The code:

```
1 2 3 bind (one two three)
```

In the repl execution array gets "compiled" as:

```
[() => put(1),
 () => put(2),
 () => put(3),
 () => { this is the setter bind has generated, const values = ...; for (
```

It is the same principle as with only one name, the only difference is that it generates multiple bindings and the setter gets multiple values from the stack and sets those bindings.

In a way bind is a hack to provide what could be seen as a variable, it has to be immediate 1 because it needs to delay the setter until runtime, since the values come from the stack at runtime.

If bind were not immediate, it could not generate the entry in the environment so the bindings it generates wouldn't exist after compile time, at runtime when bind executes. So it has to be immediate to be able to inject bindings at compile time.

The only purpose of bind is to help you associate values with names, to avoid stack juggling.

There are words to reorder elements from the stack, swap being the simplest one that just switches the position of the two top stack elements.

```
1 2
swap
```

The stack is [1, 2] after executing the functions 1 and 2 generated

Then swap executes and switches their positions, so now the stack is [2, 1]

This word is common in stack based languages and even with bind it is convenient.

The word "dup" creates a copy of the top element from the stack

```
1 dup
```

That will push 1, then dup will duplicate it so the stack is [1, 1]

With bind we can avoid those words since we take an element from the stack and set it to the binding, then use the binding to retrieve the value

```
1 2 swap
```

Could be also written:

```
1 2 bind (one two)
two one
```

And

```
1 dup
```

Could be:

```
1 bind one
one one
```

But obviously using swap and dup in those cases is more convenient.

Anyways if we were to use multiple values, bind will associate names with them and we will not have to care about reordering them

```
1 2 3 4 5 6 7
bind (one two three four five six seven)
```

```
one seven +
four five +
two six +
three four +
```

That would be a nightmare with stack operations

That said, stack based languages should never have more than 3 elements on the stack at a time.

One note about dup and bind itself when duplicating a value.

It's javascript and dup or bind do not make a real clone of a value. If the value is a number or string is immutable by js standards.

But if it's an object or array or whatever mutable, duplicating them will just duplicate a reference, not copy a value.

For example duplicating a list and then mutating it:

```
(1 2 3) dup pop
```

The word "pop" takes a list from the stack and calls the pop() method on that list, removing the last element of that list and pushing it on the stack. The duplicated copy created by dup will reflect the changes and in the stack the list will be shown missing the last element [[1, 2]]

dup does not prevent mutation because it does not create a copy, just duplicates a reference to the same object.

Same for bind

```
(1 2 3) bind list
list list pop
```

That's also why the "(" returns a closure that will create a copy every time, because "(" should act as syntax for a list literal

```
() => put(list.slice())
```

If not, when used in a definition for example and that definition executes multiple times it would return the original list and if it were to be mutated it would reflect the changes

```
: my-list (1 2 3) ;
my-list pop
```

```
my-list
```

In this case ":" starts compiling for that word and finds "(" which gives it to compile element and since it's immediate 1 it executes and returns something that ":" will insert.

The execution array of my-list would be:

```
[() => put(list.slice())]
```

So the my-list will return a fresh copy every time, pop will be operating with that copy instead of the original.

If the "(" instead returned simply

```
() => put(list)
```

The word "pop" would be operating on the original list, then the last call to my-list would push a list with [1, 2] instead of [1, 2, 3]

Since "(" returns a copy every time there is no problem to mutate a list returned by it, but bind and dup will not have this behavior since they do not even care what the value is.

About bind being dynamic, the dynamic nature of bind is that it's just a function acting as a cell.

Whoever has a refernence to that function can access that value and that value will propagate to anything that has a reference to it.

The binding is just a function that will push its cell value to the stack at runtime, whatever this value is.

This value will always be set to whatever value the last setter operation mutated it with.

```
1 bind number
```

Now number is that binding function that in a way could be represented as:

```
env.word.number = () => put(env.word.number.value)
```

The word "set" is a mutator word, it can take those bindings and alter their value:

```
24 set number
```

The word "set" is an immediate 1 word and does a similar trick to the one bind uses.

It executes at compile time and reads a word, then it looks that word in the current environment at compile time and stores a reference:

```
const name = read_word()
const the_binding = env.word[name]
```

Then it pushes a function on the stack for compile element to return to its caller that will set that binding at runtime:

```
put(() => the_binding.value = get())
```

Which compile element returns to the repl or whatever is calling it and the caller appends to its execution array.

So at compile time the word has been found and stored in a reference, then at runtime we use that reference to update the value.

Since bind is an immediate word that registers that binding in the current environment at compile time, any code that comes after it will be able to see that word. This includes inner definitions.

```
1 bind number

: complex-operation number 1 + ;

complex-operation
```

The top level defines number in the root environment.

When the ":" word compiles the complex-operation and finds the word named "number" it gives that to compile element.

Since the current environment at this compile time is the environment for the complex-operation word and this environment inherits from the parent, which is the root environment, it will find that binding and push it to the compilation array of the complex-operation word.

Since anything that can access this binding can mutate it, the inner definitions can use the set word to mutate it and the word will reflect this change everywhere it is used.

```
1 bind number

: mutate-the-number 24 set number ;
```

```
mutate-the-number

number
```

The word mutate-the-number will execute and the set word at runtime will perform:

```
the_binding.value = get()
```

Setting it to 24 since we just pushed that number before the setter function that set returns

So now the_binding.value = 24

Then at the top level when we use the word "number", the only thing it will do is:

```
put(the_binding.value)
```

Which was set to 24.

This is dynamic behavior, but due how the compiler and compile time environments word, it is also tied to the lexical rules of the compiling procedure.

That means that if an inner definition defines a binding with the same name, it will use that binding instead of the top level one.

```
1 bind number

: complex-operation
  24 bind number
  number 1 +
;

complex-operation
number
```

Even if the bindings are dynamic in nature, the references in this case are different, since complex-operation created a new binding.

At the top level we define the binding for number in the root environment:

```
root.word.number = () => put(root.word.number.value)
```

Then the ":" starts compiling, creates a new environment at compile time that inherits from the root environment and sets it to become the current environment.

Then the 24 gets converted to () => put(24) and appended to the compilation array of the complex-operation word.

Then the bind word executes at compile time, creating a new binding and registering it to the current environment.

At this moment the current environment is the environment for the complex-operation word, so the bind word registers it there.

The bind word also pushes the setter on the stack at compile time, which compile element will return to the ":" and the ":" will append to the complex-operation code array

```
const the_reference_of_the_binding = complex_operation_env.word.number
code = [() => put(24), the_reference_of_the_binding]
```

Note that this reference is not the same reference the root environment has, it's a different binding that exists only in the environment for the complex-operation word.

Then the ":" keeps compiling and finds the "number", the "1" token and the "+" word

So the code array of the complex-operation word is:

```
code = [() => put(24),
        the_reference_of_the_binding,
        () => put(the_reference_of_the_binding.value),
        () => put(1)
        () => { const [one, two] = get2(); put(one + two) }] // the "+" w
```

That means that complex-operation will never use the binding of the root environment, but its own binding instead.

Only children of complex-operation and complex-operation itself will be able to access to that binding. The rest will find the root env binding instead.

```
1 bind number

: complex-operation
  24 bind number
  number 1 +
;
```

```
: another-operation
  24 set number
;

complex-operation
another-operation
```

In this case complex-operation is creating its own binding like before, but another-operation is modifying the binding of the root environment.

So after evaluating complex-operation, the number binding of the top level is still 1, because complex-operation creates its own

But after evaluating another-operation, the number binding of the top level is now 24, since another-operation uses set and set took the binding from the root environment.

If we create an inner definition inside complex-operation, since complex-operation creates its own binding in its own environment and the inner definition will create its own environment and inherit from the environment of complex-operation, set at compile time will find the binding in the environment of the complex-operation binding instead of the root one.

```
1 bind number

: complex-operation
  24 bind number
  : even-more-complex-operation
    3 set number
  ;
  even-more-complex-operation
  number
;

complex-operation
```

The even-more-complex-operation word will use the binding of the complex-operation word, never the top level one since there is a binding in a closer parent than the root, the complex-operation environment.

That means that the top level number will be 1, since nothing is mutating it.

And the number of the complex-operation even if initially set to 24, gets mutated later to 3

A similar word to bind exists for when we only want to declare a binding, but not give it a value.

```
0 bind number
```

Is equivalent in a way to:

```
declare number
```

The word declare is an immediate 0 word, it does not have to set the value at runtime, it just creates a binding and initializes it to 0

It also works with a list of names like bind does:

```
declare (one two three)
```

But one two and three will be bindings set to 0

It is used when you want to set the values later, but create the binding in the current environment at compile time.

The reason bind is immediate 1 and declare immediate 0 is because bind needs to set the value with an element from the stack at runtime and has to delay that setting for the runtime while creating the binding at compile time, so it pushes the setter to the compilation array to ensure this value will be set properly in the right moment.

Since declare does not expect any value and has to initialize it to 0, it does not have to perform that hack.

Those bindings are dynamic in nature, but due to how the compilation procedure works, they achieve some sort of lexical scope. In a way they are both things at the same time.

The bindings created by bind or declare and mutated by set will usually work without problem as long as you take them as temporal helpers to associate values on the stack to names, if you try to use them as normal variables for the most part will be fine unless if you are involved in recursion. Recursive definitions will show the dynamic nature of those bindings and will not work as expected in usual lexically scoped languages since you are not creating a new binding every time the definition recurses, but reusing one that was generated at compile time.

That by default does not matter since the compiler does not even know what recursion is.

compile element will trigger an error if a word does not exist in the environments and is not syntax sugar provided by compile atom.

That means that there are no forward references by default.

Recursion if ever added will be in form of an immediate word that compiles an execution array and iterates it.

By default recursion is assumed to not even exist.

```
: some-word-name
  1 2 3 some-word-name
;
```

You might expect this to recurse, but some-word-name is not defined the moment is found in its definition and compile element takes this as an error.

If the word existed before, compile element will freeze that definition in the code array of whatever we are compiling so:

```
: some-word-name 1 2 3 ;

: some-word-name some-word-name ;
```

This is actually creating a wrapper over the old definition.

At first some-word-name is:

```
env.word['some-word-name'] = () => { for (let fun of [() => put(1), () =>
```

Then we redefine it, but in the definition we are taking a reference to the old definition

When the ":" runs the second time and starts compiling that word again, it will see in its definition the name of the word 'some-word-name', which will give to compile element and compile element will just return

```
return env.word['some-word-name']
```

Which ":" will store in the compilation array it is generating and then create the function

```
const the_frozen_definition = env.word['some-word-name']
// That was actually returned by compile element
// The frozen definition actually is: () => { for (let fun of [() => put(
// now the second ":" defines the word again:
env.word['some-word-name'] = () => { for (let fun of [the_frozen_definiti
```

I don't know if i explained it properly, but we have just created a wrapper that contains the old definition.

The old definition is lost in the root environment, but it is embedded into the code array of the new definition.

And that's why recursion will not normally work and if it's ever added will be an immediate word that compiles a definition in a different way to make it recursive.

There is a way to recurse by delaying the lookup of a word.

compile atom is the fallback of compile element for when a word is not found and it just takes this token and runs a series of checks and regular expressions to see whether this atom is special syntax.

One of the checks is for numbers, compile atom is the one converting the string "1" into a function:

```
let value = parseFloat("1")
return () => put(value)
```

If compile atom receives a token ending with a colon like in:

```
some-word:
```

It will take this as a delayed lookup form, meaning that this word will not be looked up in the environments at compile time, but instead it will be delayed until runtime.

```
if (atom.endsWith(":"))
{
  const name = atom.substr(0, atom.length -1)
  // removes the ":" at the end
  return () => { const word = find(name); if (word) { word() } else { err
}
```

The compile atom function returns a function that will perform the lookup of that word later and execute or trigger an error.

The function find is the one compile element uses to find a word in the environments.

This function is going to call find to retrieve a word from the current environment or its parents at runtime.

There is a catch though, the definitions in this language create a compile time environment that only lives during the compilation of that definition. Any definition doing this will always make the current environment point to the previous environment after it ends compilation, which means that at runtime all definitions ended compilation and none of their environments exist.

That means that with the delayed lookup we will never found the environment of a definition.

At runtime by default the current environment will almost always point to the root environment because all definitions make the env pointer point back to the previous environment and any top level definition will end by restoring the current environment pointer to the root environment since that was the previous environment when they ran.

In the case of a top level word that we want to call with the delayed lookup, it will work because at runtime the current environment points to the root and the delayed lookup will search words at runtime.

```
some-definition:
: some-definition 1 2 3 ;
```

This code will work and call some-definition even if we are using it before defining it.

If we didn't use the delayed lookup then compile element will trigger an error because that word was not defined yet.

But since it has that ":" compile atom returns that function to the caller (the repl or whatever) and the caller will append that function to its own execution array.

That means that when we type

```
some-definition:
```

the repl or whatever calls compile element which fallbacks to compile atom and compile atom returns that function to the repl, which the repl appends to it's execution array.

```
const the_repl_execution_array = [() => { const word = find("some-definit
```

Then the repl will keep compiling stuff and find the ":" word which is an immediate 0 word that executes at compile time and returns nothing, so the repl execution array will keep having only that function, but the function is defined at compile time by ":".

Then the repl will hit end of input and iterate the array calling all the functions, executing the function returned by compile atom.

That function will start searching in the root environment and will find the word defined there so it will execute it.

This implies also that a word can recurse using that method.

```
: run-for-ever run-for-ever: ;
run-for-ever
```

But it's expensive since every time the word executes will have to look at the chain of environments to find the word.

Here is where the friction on the goals of the compile element procedure starts.

compile element and the definitions themselves do not expect any environments at runtime.

Runtime is supposed to be only an array of functions being iterated.

compile element and compile atom try their best into taking decisions at compile time to allow the runtime to be just calling functions in an array, at the end compile element is the whole interpreter.

The delayed lookup mechanism breaks this assumption and also expects environments at runtime.

By default at runtime the only environment expected to exist is the root environment.

The design of the language assumes that runtime environments do not exist and that the only environment that should survive until runtime is the root environment.

I have three words that are actually a hack and break all of this design.

The first and simplest one is the word "block"

The word block tries to create something that behaves like true lexical scope at runtime.

The word block is an immediate 1 word, which means that executes at compile time and returns a function that will get appended to whatever we were compiling.

It just compiles a block until it sees the word "end" and pushes a function that will create a new environment at runtime, set it as the current environment, evaluate the code and restore the current environment to the previous one.

This together with the delayed lookup provides some fake lexical scope, since if we produce code that has a delayed lookup and execute this code inside the block wrapper, the lookup will be performed in the environment that has been created at runtime.

This interface is experimental and has to evolve, because there is a mismatch with the rest of the interpreter with this concept.

The main reason is that we wanted to avoid lookup at runtime and now we are breaking this goal.

Also the block feature is a hack because we are relying on a delayed lookup and nothing really guarantees that by just delaying a lookup it will be done in the place we expect.

It is also not correct and not good for performance, since every time the delayed lookup executes will have to search through the chain of parents.

It is mainly an afterthought and the current implementation did not expect that to exist.

That's also the reason i keep rewriting it from scratch, since this is already conflicting with the core of the language.

The core of the language is compile element, the rest is just immediate words.

A new experiment or feature can be added without touching the core, because an immediate word can do whatever it wants.

For example if we wanted to provide recursion we just make an immediate word that compiles stuff in a different way.

We do not need to touch the core, compile element will find our word, execute it and there we do whatever we have to do.

The runtime environments though are an afterthought and might end changing the design of the interpreter or not.

The main conflict is the interpreter did not want to provide runtime environments and we are hacking our way through delayed lookups.

This concept has to grow and i have to experiment with that but the main problem is everything assumed we will precompute and cache all the decisions at compile time.

And that is no longer true

compile atom provides 3 kind of syntax for delayed lookup and delayed creation of a word.

```
:name
```

This would append to the execution array a function that will create a word in the current environment at runtime associated with a value from the stack.

At runtime it will take a value from the stack and create a word named "name" in the current environment that when executed will push that value on the stack:

```
if (atom.startsWith(':'))
{
   const name = atom.substr(1)
   return () => { const value = get(); env.word[name] = put(value) }
}
```

That returned function will execute at runtime.

That means the word it creates will not exist until this function executes at runtime, so compile element will never find it at compile time unless compile element is being called at runtime

The other syntax is when a word has a colon in both the end and the start

```
:name:
```

It's just a variation of :name but will also push the value on the stack at runtime

```
() => { const value = get(); env.word[name] = put(value); put(value) }
```

It's just a shorthand to avoid

```
:name name:
```

Which at runtime will take a value from the stack, generate the word "name" that will push that value on the stack, then name: will perform the lookup of that word at runtime and execute the word.

This is the main interface for delayed lookups.

The delayed lookup mechanism does not really bother me, what bothers me is abusing that or having to use that at all to provide runtime environment lookups with the block word and the fact that right now is the only way to make the block word work.

Since the block word wraps a block of code and at runtime before evaluating this code creates a new environment, the delayed lookup is the only way to access this environment, since this environment only exists at runtime and compile element has no way to access it unless it delays the lookup of the word.

It has to evolve to avoid relying on that, but the current implementation does not adress this issue.

The main problem i have is that delaying a lookup and wrapping that delayed lookup in a block that will create an environment at runtime is not safe at all.

Nothing guarantees that the lookup will execute in the right moment.

Obviously if we do:

```
env = make_env(env)
delayed_lookup()
env = previous_env
```

We are kind of sure that delayed_lookup() will execute in that new environment.

The block word does the same thing, but does not know what a delayed lookup is, it just wraps an already compiled block of code

The block word executes at compile time, when executing it first captures a reference to the current environment.

Then it compiles a block like the word ":" does, but with the word "end" as delimiter instead of ";"

```
block
 some random code
end
```

The words some random code get compiled at compile time and block stores a reference to this code.

```
w.block = () =>
{
  const e = make_env(env)
  const code = env_block(e)
  put(() => { const old = env; env = make_env(e); code(); env = old })
}
```

That's the definition for the block word, the env_block(environment) javascript function is different from the block word, the javascript function is a helper that unifies the creation of a definition and also takes an environment as argument.
The, the word ":" uses that function internally and any other thing that creates a definition too.
There is another function that does not take an environment as argument and instead uses the current environment to compile a block of code which is named block(), but has nothing to do with the word defined in the language named "block".

the block() javascript function is that loop we saw in the code of the word ":"

```
const previous_environment = env
env = make_env(env)
const code = []
while ((word = read_word() !== delimiter))
{
   value = compile_element(word)
   if (value)
   {
     code.push(value)
   }
}
env = previous_environment
return () => { for (let fun of code) { fun() } }
```

The env_block() function does the same but env is the argument it receives instead of creating a new one.

Almost any definition does the same since they call the block() or env_block() function.

So when the block word calls the env_block() function, it starts reading until delimiter (which has a default value of "end") and receives a function representing the compiled code:

```
block
some random code
end
```

The block word receives the code compiled for the words "some random code" at compile time, then since its immediate 1 it returns a function which was:

```
() => { const old = env; env = make_env(e); code(); env = old }
```

Since this function is what gets appended to the compilation array, it will execute at runtime.

At runtime the compiled code for "some random code" will execute inside a runtime environment that inherits from the environment that code was defined in.

the word block executes at compile time and creates a new environment that inherits from the current one and stores it.

`js
const e = make_env(env)

Then it calls env_block(e) which will start compiling code until it finds the token "end" with that environment the block word generated.

So any definition or whatever in that code between block and end will be compiled in that environment.

Then the block word stores that compiled code and returns a function on the stack, since it's immediate 1.

The function when executed at runtime will create a new environment, set it as the current environment and execute the compiled code there.

Then it will restore the current environment to point back to whatever it was before the execution of the code.

That means that while the compiler has no way to access that environment because it only exists during runtime, delayed lookup and word generation with name: :name will run in that environment.

~~~~
```oh
block
24 :name
name: 1 +
end
~~~~

This compiles a delayed word generation, the :name makes compile atom return a function to the execution array that will generate a word in the current environment at runtime and take an element from the stack, to associate it to a value.

So at runtime when the function returned by compile atom when it found :name it will take the number 24 from the stack and create the word:

```
env.word.name = () => put(24)
```

then the function that compile atom returned for name: will execute

```
() => { const word = find('name'); if (word) { word() } }
```

and the rest is put(1) and the + word

this code was inside the block word, so the block word wraps it into a function that creates a new environment and sets the current environment to point to that environment, so when the function returned by compile atom for :name executes it will create the word in that environment and when the function returned for name: executes it will call find which will start searching the word from that environment.

my dislike with all of this is using the delayed lookup for something is not meant to and the fact is the only way to access a runtime environment.

But mainly that nothing really guarantees that when :name or name: execute, the environment will actually be what we were expecting.

So block has to evolve to something better, or a new word will have to be created or the core of the language has to change to accomodate this.

The main failure is to initially assume that no runtime environments exist and then realize they are convenient and introduce them.

The current design did not care about runtime environments since the purpose was to avoid runtime lookups.

The cool part of this language is that i'm the author so i can redefine it whenever i want.

I just have to decide on which abstraction or solution i want to create.

But to do that i have to play more with the language and test the limitations to see where it fails.

If the solution can be created in terms of an immediate word, it is just to create a js function or even a colon word in the language itself and mark it as immediate.

If not, then the core has to change, that's when i rewrite the whole thing usually.

Still it is likely that the ideas will be tested initially as some immediate word that compiles stuff in a different way or wraps that compilation or whatever.

I like that the core of the interpreter is tiny, since it's actually just one big function if we take compile element and compile atom as one function.

The main problem it has is it's very hard to do even the most simple task with it.

I might need to get used to it though, but it's true that it will force you to emulate the program in your mind when you are reading the code.

It has some cool abstractions, but by default it uses rpn and every token is an action.

You need to keep a mental track of the stack at every step, although the bind word simplifies this a lot.

It has some simple trace mechanism to tell you what's on the stack every time it evaluates a word.

The trace actually overwrites the javascript function that compiles elements into a list.

We have seen how everything, the repl, loading a file, creating a definition, a block, etc, has the same procedure.

Read words, call compile element on them, if compile element returns something store it into an array

Then executing the code is to iterate that array and call all those functions.

The process of checking whether compile element returned a function and storing it into that array is abstracted away by one function that takes the word to give to compile_element and the list to push the value

```
function compile_in_list (element, list)
{
  const value = compile_element(element)
  if (value)
  {
    list.push(value)
  }
}
```

The functions block() and env_block(environment) use that function internally
the repl and loading files or tags also use that function.

The trace word overwrites compile_in_list for

```
function trace_into_list (element, list)
{
  const code = compile_element(element)
  if (code)
  {
    list.push(() => { code(); console.log('trace:', element, ...stack) })
  }
}
```

hijacking the compiling procedure of everything in the interpreter to wrap it in that function.

```
trace
1 2 +
```

the output would be:

```
trace: 1 1
trace: 2 1 2
trace: + 3
```

The word no-trace just restores compile_in_list again so the interpreter comes back to normal.

The trace mechanism is quite simple and does nothing interesting.

The cool part is that you only pay the performance hit when you actually use trace because it hijacks the whole compiling procedure.

If you do not use it the compiler just stores functions.

While a hack (this language is full of them anyways) i like that i don't have to pay that performance hit if i don't want to, because the moment i make a debugger (which i will need soon) the debugger will perform something similar and any kind of metadata and logging information will only be added and used when the debugger hijacks the compiler like trace does.

It's just switching to a different way of "compiling" which is just to store functions into an array.

When adding metadata in the interpreter you slow it down a lot, which this approach or a cleaner one you only pay that price when you really need it and do not have to make a fast implementation and a slow one.

In the case of trace we are not even modifying the compiling procedure, just wrapping the results.

Another place were a similar kind of hack appears is in the auto await mode of the interpreter.

I have been showing several times how we iterate through an array of precomputed functions and evaluate them.

We gather them into a list, then later we execute them.

The tracing thingy hijacks the gathering into the list.

The autoawait mode hijacks the generation of the function that iterates that list.

I have been showing this several times:

```
for (let fun of code) { fun() }
```

An that always has been the runtime. The moment something executes.

Turns out that is such a common pattern in this interpreter that a function handles this by taking a list and returning that function:

```
function make_sub (list)
{
  const compiled_unit = () => { for (let element of list) { element() } }
  return compiled_unit
}
```

So anything that defines something and uses compile_into_list, will later give that list to make_sub to turn it into a function that iterates that list and calls all the functions in it.

The word "wait" overwrites that function and changes it to this one:

```
function make_async_sub (list)
{
  const asynchronous_compiled_unit = async () =>
  {
    for (let element of list)
    {
      element();
      if (stack[stack.length - 1] instanceof Promise)
      {
        const promise = stack.pop()
        try
        {
          const result = await promise
          if (result !== undefined)
          {
            put(result)
          }
        }
        catch (e)
        {
          catch_code(e)
        }
      }
    }
  }
  return asynchronous_compiled_unit
}
```

It is the latest hack and the most experimental of all together with the runtime environments.

It generates a javascript async function that when executed will iterate that list of functions, but every time it executes one of them it checks on the stack whether there is a promise on top.

If there is a promise it awaits it and pushes the result of awaiting it on the stack.

Since it's an async function in javascript when it uses await on the promise it will stop execution until this promise resolves or rejects.

This means that the moment you execute the immediate 0 word named "wait" it changes the way the compiler works and not it starts generating async functions that will wait for promises on the stack, so every function inside that list that pushes a promise on the stack as the last result will make the execution at runtime wait for that promise.

The most obvious example is using fetch to make a query.

The word "fetch" is just the javascript function fetch accepting one argument (the url) and pushing a promise on the stack

```
'https://www.oh.com/oh.js fetch
```

That url does not exist but it will return a promise.

We could manage that promise and call a then method or whatever, but we can just use the auto await mode instead.

```
wait

'https://www.oh.com/oh.js fetch -text log
```

The "-text" thingy is a method call on whatever is on the stack. It is provided as syntax sugar for compile atom

In this case is

```
put(get().text())
```

So it is calling the text() method without arguments on the result that the promise of fetch gave after awaiting it.

Since fetch pushed a promise on the stack with auto await mode, that async function has awaited that promise and pushed the response object on the stack (if it were to be a correct url)

Then we are calling the text() method on that response object and pushing the result of that method on the stack.

The text() method of a response object returns another promise, so we have just pushed another promise on the stack and made the async function await it and push the awaited result on the stack which we give to the "log" word which is just

```
env.word.log = () => console.log(get())
```

So if the fetch were to give an ok response object it would print the text on the console.

This is experimental in the sense that is a new feature that has to be pushed in different situations to evolve, exactly like the runtime environments which currently only allow access through the delayed binding hack.

There is the implication that if we load an entire file and say "wait" on the top level, it affects the entire file.

The problem is that those are immediate words and execute at compile time, changing the compile mode to auto await on the fly and there is no proper boundaries for that and the fact that we load an entire file is not helping.

I mean, the "no-wait" word turns back to normal compilation in the same sense "no-trace" does.

But when compiling any input, even if it's the repl or loading a file we will not generate that function until we find the end of input.

That means that:

```
wait

some words are getting compiled here

no-wait
```

You might think those words are getting compiled in auto await mode and after that we switch back to normal.

But this is not true because while those words are getting compiled and stored into an array, the async_make_sub auto await thingy will not be called until we reach the end of the input, the moment we generate that function that iterates the array.

Same for any definition when they reach the end delimiter.

Even if wait and no-wait are immediate 0 and execute at compile time and that's when they replace the function that generates that array iteration function, that function will not be called until the end of the compilation.

So those words got compiled into the normal mode because we switched back to normal just before calling the function that generates the array iterator.

The same mistake would be in a colon definition if we do:

```
: some-word
  wait
  1 2 3
  no-wait
;
```

Because the set it to auto await, compile some words into the array, then set it to no-wait back and after the ";" semicolon is when we actually use the overwritten function.

Note that will work properly if we just put the no-wait after the semicolon, then when ":" calls the function to generate the iterator it will still be the auto await function.

```
: some-word
  wait
  1 2 3
;
no-wait
```

The env.word['some-word'] function is now the one generated by auto await mode.

Also note that after compilation no one restores the auto await mode to no-wait mode for you.

You trigger that at will and that's it.

Right now i have to play with this more and see how uncomfortable it feels.

For the most part it is quite cool as long as you do not switch it constantly or at all.

But since it's experimental it needs to evolve to something better.

The only way this feature and the runtime environments or any other feature in this language will evolve is by pushing their current implementation, solution or hack to the limits and see how they explode to know the direction that must be taken.

One of the things that pushes the limits a bit is the dom abstraction feature.

For the browser you can create document object elements and append them to the document.

You have words like document, body, window, head, etc that when evaluated they push the document object, the document.body object the window object, the document.head object, etc.

You have the "element" word which is document.createElement

```
env.word.element = () => put(document.createElement(get()))
```

```
'p element
```

That would create a

element and push it on the stack, you can use append which is:

```
env.word.append = () => { const [child, parent] = get2(); parent.appendCh
```

Although a bit different because it works with lists of children also

```
p element body append
```

That would append a

element to the document.body

You can just say "to-body" which is a shorthand for "body append"

```
p element to-body
```

But there is a cooler abstraction to create elements using lists with the word "dom"

The word "dom" is not immediate, it's just a normal word that executes at runtime.

At runtime it takes a list from the stack and iterates it.

It does a lot of stuff, so the function is a bit big to put it here, but it mainly takes a list or nested lists as some sort of declarative html s-expressions.

```
(p "oh...") dom to-body
```

That would create a p element, set its textContent to "oh..." and append it to the body.

You can set ids, class names, arbitrary properties, event handlers and associate properties of that element and create a word that will push this element on the stack.

You can also nest elements and those elements will become children of the outer elements.

```
(main
 (h1 #the-title .heading "This is a title with the id the-title and the c
 (p "Please type your feelings here")
 (input -type text -placeholder "I am feeling fine.")) dom to-body
```

That would create a

element with the h1, p and input elements as children.

The h1 will have the id set to "the-title" since the #the-title is a directive to set an id.

It will also have the class "heading" since the .heading is a directive to add a class

The strings in quotes will be appended to their textContent

The directives starting with a dash like "-type" and "-placeholder" will set a property on that element with the value being the next element.

So the element will have a type and placeholder property like:

```
<input type="text" placeholder="I am feeling fine.">
```

There is a way to create or attach handlers to events.

```
(button "Please do not click me" @click ("I told you not to click" log))
```

That will register an eventListener on that button element that will execute that code in the list.

The @ directive takes the next element and it expects it to be a list or a function.

If it is a function is just attaches it to the listener like

```
button.addEventListener('click', that_function)
```

If it is a list it will compile that list as a sequence of words.

Before compiling it will do something like:

```
let event
const e = make_env(env)
e.word.event = () => put(event)
```

Now it compiles that list iterating it and calling compile element, but before doing that it sets e to be the current environment for that compilation so every element in that list gets compiled in that "e" environment.

Note how it's creating a lexical variable in js named "event" and then registering a word in that "e" environment.

This means that when compiling the elements of that list inside that environment if that list contains the word "event" it will find it and compile it properly.

Then it registers the event handler like:

```
button.addEventListener('click', (ev) => { event = ev; for (let fun of li
```

Note how it sets "event" to the event object it receives as argument. That's why it was registering an event word in the environment before compiling the list, so now if the event word executes, since that word was just to push that lexical variable, it will push the event object on the stack.

That means that if you give a list to the @ directive, you can use the event object.

```
(input -type text @change (event.target.value log))
```

That would print on the console the value of the input element when the onchange event fires.

That dot notation "event.target.value" is fancy syntax sugar provided by compile atom

It's a bridge for using javascript objects.

Shouldn't be explained in the middle of the explanation of the "dom" word, but i just introduced it by accident so i should explain it.

The "event.target.value" gets separated in two parts: "event" and "target.value"

The first part gets compiled and later evaluated and it should push an object on the stack.

In this case event was compiled in that list and the @ directive injected the event word in the environment the list was compiled so the event word is pushing the event object on the stack.

After that the object gets iterated with the chain of properties "target.value"

And if target.value is a property of that object the value will be pushed on the stack.

The dot notation has more variations which i'll have to explain once i end the explanation of the "dom" word.

The dom word has two more features and both involve generating new words at runtime.

The simplest one is ":name"

If you recall from before ":name" was syntax sugar compile atom provided for delaying a word being generated on the stack.

compile atom would return a function that at runtime will take a value from the stack and create a word named "name" that will push that value on the stack when executed.

The dom word reuses the syntax because it helps to remember that dom is a runtime word and if it generates a word it will do so at runtime.

That means that at compile time you will not be able to use that word unless you use the delayed lookup "name:"

Instead of associating a stack value at runtime, the dom word associates this name with the element it is generating.

```
(button :my-button "Hi, I am a button") dom to-body
```

That would generate a word named "my-button" at runtime, that when executed will push the button dom object on the stack.

But since dom cannot do that until runtime because it is not an immediate word, the compiler has no way to compile that word unless we use the delayed lookup.

So if we want to use that my-button word it has to be with "my-button:" to make compile atom delay the lookup and execution of that word until runtime

The last feature of the dom word is the reader and writer directives.

They are similar to the :name thingy but instead of associating a word to the dom object, they create a getter or setter for a property of that element.

```
(p writer to-p text-content "I initially have some boring text") dom to-k

"But now look how cool my text is :D" to-p:
```

The writer directive reads a name for the word to generate and a property for the dom to associate it with.

In this case it is creating a word named "to-p" that when executed will get a value from the stack and set the textContent of that element with it.

So the last line is pushing a string on the stack and then calling "to-p:" (note the delayed lookup) and that word just updates the textContent of the p element with that string.

Since again the dom is a runtime word not an immediate one, there is no way for the dom word to generate words at compile time.

Any word dom or any other abstraction generates at runtime will be accessed by delayed lookup unless we are compiling things at runtime.

The reader directive is similar but is the getter part, instead of setting the value it pushes the value of that property on the stack when evaluated.

Those words are going to be created in the current environment at the time the dom word executes.

Since that will always be at runtime, the block word can provide a runtime environment where the dom word can generate words.

```
block
(p :the-p-element "Hi...") dom to-body
end
```

The word "the-p-element" will then be created in the new environment block generates at runtime.

That means that if we use the delayed lookup for that word outside that block we will never find it.

It also means that if the @ directive receives a list it will compile that list in an environment that inherits from the block environment.

The word defun is quite similar to the block word, it does the same but instead of executing that code it creates a word at compile time that does the same block does.

```
defun some-name
1 2 3
end
some-name
```

Is equivalent to:

```
block
1 2 3
end
```

The only difference is that a word will be available to reuse that block.

That word when executed will create a new environment at runtime every time is executed, then execute the code inside that environment.

The dom word is the only reason the block and defun words exist and also the only reason delayed lookup actually exists.

The dom word generates javascript document objects and is an abstraction over the document object model api of javascript.

When using that api you usually want to create components.

The word "component" should be an immediate word that does not exist yet.

The component word is where my current understanding when writing this file has not reached yet.

It is the proper bridge between compile time and runtime fresh environments.

Right now i have block and the delayed lookups and this is why they are a hack.

The dom word asks me for runtime environments because when creating those objects you want sometimes state bubbles.

A component that can be generated at runtime as result of some computation with code that can access that runtime environment.

The component word will be the evolution of the block and defun words, but i do not want to create it without a proper bridge, so i have those hacks to make them explode and realize better how the component word should actually be and whether should or not affect the core or just be an immediate word.

If something affects the core will push me towards rewriting the whole thing, which happens like every month.

Rewriting for me is patching for you.

I just hate patching code because i fix some part and break 3 things, then i fix those things and break more and so on.

I have a short threshold that when reaches the limit forces me to rewrite.

Immediate words are extensions for compile element so i can test random stuff without affecting the core.

They can read words, compile them normally or in a weird way, do whatever and also inject behavior for the runtime to do something later if they are immediate 1

They extend the language in any way and never push me towards rewriting the code because the core does not care about any of those words.

The component word is where my understanding ends and if it's an immediate word the core will not change.

But it's the solution of my initial mistake when making this version of the interpreter assuming runtime environments will not exist.

The block and delayed thingy is me telling to the dom word:

```
We have runtime environments at home
```

The runtime evironments at home:

```
  block
  (button "If you press me i will explode"
   @click (this code gets compiled in the runtime environment of block)) dc
  end
```

If the dom word finds a list for the @click handler it will compile the list using the current environment.

Since the dom word is not immediate and executes at runtime, taking a list from the stack, anything that defines or compiles will be using the current environment and the block word is creating a new environment at runtime and executing the dom word inside that environment.

So every word the dom word defines with writer, reader or the :name syntax, and any list compiled with the @ directive will be defined in that environment.

The dom word is a runtime word that can suddenly start compiling random stuff at runtime.

That's why the component word should be a proper bridge or controlled way to do this.

It will exist eventually or the language itself will change to provide this in a better way.

Right now i have to play with compilation at runtime in random weird environments and see how they explode, to find a general abstraction to provide.

Which means using block and the delayed lookups and push them to see how confusing they can get i guess.

But that's where my current understanding is i guess.

I find it confusing to start compiling things at runtime.

It wasn't until the block word.

Now you can compile stuff but not know in what environment.

Initially at runtime the only environment expected was the root environment.

So compiling something at runtime meant the same as compiling at the top level.

The block word changes this assumption and now you can compile stuff inside a block and any word compiled will be compiled in an environment that only exists at runtime and you can use to save state.

But the block does not enforce anything and does not guarantee that your code will actually compile definitions in the current environment.

While it might be cool, i would like an alternative that ensures i'm going to use the exact environment i was expecting to use.

Also the delayed lookup is the only way to access a block or defun runtime environment, which is not cool.

The compiler did not expect runtime environments and the delayed lookup is a hotfix for that.

An immediate word could create a definition or new syntax for bridging this, but the solution is not in my mind yet.

So that's the current state of the language in a way.

The limitation of the current implementation is the initial assumption of compile time only environments, that while is a cool goal, the dom word shows the need for runtime environments and there is no proper interface to compile code for those environments.

A word could do that, but does not exist yet since i do not know what the proper interface should be at the time of this writing.

The cool part of this interpreter is that once i take a decision it is just to add a function in js or a word in the language.

An alternative for bind/declare for runtime environment scopes would also be cool.

Recursion is not implemented yet, if it ever comes it will in the form of an immediate word

Recursion will also need a solution for runtime environments.

the bind word should not be used for recursion or loops unless you understand the dynamic feature and why those bindings shouldn't be used in those cases and why those bindings are not real variables.

There is a word named "module" that compiles code inside an environment.

The module word reads a name for a word that will associate with this environment.

The module word is an immediate 0 word that reads a name and a block of code.

```
module some-name
 some definitions here
end
```

It creates a new environment that inherits from the current environment at compile time.
Then compiles the block until "end" inside that environment.
Then it creates the word it has read (in this case "some-name") and creates a word in the current environment that when executed will push that environment on the stack.
The word "some-name" generated by the module word will be set also as immediate 0

That means that using some-name in any code will push an environment on the stack at compile time.

The reason for that is for the words import and import-all

The words import and import-all are immediate 0

They execute at compile time and expect an environment on the stack.

They are meant to be used with the words created by the module word.

They take an environment from the stack at compile time and inject words from this environment to the current environment.

import-all injects all the words of that environment into the current one.

import reads a name or a list of names from the source code (like bind or declare do) and injects those names instead.

They are meant to load definitions into the current definition.

```
module my-utilities
 : complex-operation 1 + ;
 : sum-two-numbers + ;
 : do-something-important "Hi, I am doing something very important!" log
end
```

Now the word "my-utilities" is an immediate 0 word that when executed will push the environment where those words where defined.

That means that the environment my-utilities will push on the stack at compile time has in it's word: {} slot the definitions of complex-operation, sum-two-numbers and do-something-important words.

We can load them into another definition with import or import-all and they will be injected into the current compile time environment.

```
: do-many-things
  my-utilities import-all
  24 complex-operation 3 sum-two-numbers
  do-something-important
;
do-many-things
```

The do-many-things word has loaded all those definitions in it's own compile time environment and the compile element function was able to find them for this definition.

Those words are only loaded in the environment that the ":" word created for the definition of the do-many-things word.

In the repl or any other place they are not accessible unless we load them into that environment.

The import word does the same as import-all but only imports the words you tell it.

```
: do-something
  my-utilities import do-something-important
  do-something-important
;
do-something
```

In this example the import word reads the word "do-something-important" and takes the environment the my-utilities word pushed on the stack at compile time and used it to inject that word into the environment for the do-something word.

If we need more names we make import read a list

```
: do-more-stuff
  my-utilities import (do-something-important complex-operation)
  do-something-important
  24 complex-operation
;
do-more-stuff
```

With module and import we can in theory create domain specific languages by creating a set of words and loading them.

The cool part is the fact that after this definition ends compilation, the environment used for that definition will be discarded.

That together with loading immediate words allows us to load sets of words that provide different syntax and utilities only for the duration of the compilation of this definition.

Since import and import-all and the module and the word module creates are immediate and the module word compiles a block of code, we can import words from one module to another module.

```
module a-lot-of-words
  : one 1 ;
  : two 2 ;
  : three 3 ;
  : four 4 ;
  : five 5 ;
end
```

```
module fewer-words
 a-lot-of-words import (one two three)
end

: some-definition
  fewer-words import-all
  one two + three +
;
some-definition
```

We could create subsets of a module by just importing fewer-words than the other module provides like in that example.

But we can also import utilities in a module to help create more definitions or whatever.

The cool part is when we create immediate words that implement random syntax for a specific abstraction i guess.

I had to explain the dot notation:

The dot notation has a setter and getter sugar provided by compile atom

If there is a token with dots in it, compile atom takes it as dot notation.

The setter is when you append a "!" character at the end of a token with dot notation.

It is just some syntax sugar to use object properties and set them, a layer for js interoperation.

```
(1 2 3) .0
```

".0" is equivalent to:

```
put(get()[0])
```

When compile atom finds a token that starts with a dot, it returns a function that will take an element from the stack and access a property on it, pushing that value on the stack.

So in this case it will take the list from the stack and return the first element, the number 1

If instead we add a "!" character we will be mutating the value.

```
24 (1 2 3) .0!
```

That will replace the 1 for 24 in that list.

The list will be [24 2 3] instead of [1 2 3]

The dot notation can contain multiple properties

```
.style.color
```

That will take an object from the stack and try to access the style.color property of that object

The dot notation has a variation for when the token has dots but does not start with a dot.

```
body.style.color
```

It will evaluate the word "body" by compiling it first and returning a function that will execute it then access the property of whatever this word pushes on the stack.

The words document, body, head, window, etc, are already defined and they push the javascript associated objects on the stack.

This allows us to set the properties of an element.

```
'red document.body.style.color!
```

```
'p element bind p
p to-body
"Hi, i am a <p> element, how are you?" p.text-content!
```

Note the text-content instead of textContent

The dot notation will translate kebab case to camel case

It only does that for the properties, not the first word, since the first word is evaluated as a word in the language.

```
: some-word
  'p element
  dup to-body
;
"Hi" some-word.text-content!
```

The word set can also work with dot notation since it is a mutator.

```
mutator('increment-by-one', (slot, property) => slot[property]++)
mutator('decrement-by-one', (slot, property) => slot[property]--)
mutator('increment', (slot, property) => slot[property] += get())
mutator('decrement', (slot, property) => slot[property] -= get())
mutator('set', (slot, property) => slot[property] = get())
```

A mutator is just the unification of multiple ways to set a value.

```
24 set some-binding
24 set some-binding:
24 set object.property.subproperty
24 object set .property.subproperty
```

The mutator function handles all those possibilities and executes the provided code which it calls giving the object and property.

When compile element calls the immediate 1 word set, or any other word registered as a mutator since the mutator function in js generates that word and sets it as immediate 1, the mutator function executes, which reads the next token calling read_word() and determines what function to push on the stack for compile element to give to its caller.

What compile element calls when executes the set word is a function the mutator function has registered that will check what the token is and dispatch a function depending on whether is dot notation or binding or delayed binding.

for example in

```
24 set some-binding
```

The mutator registered function executes at compile time and reads "some-binding" it checks at compile time that this word exists and is in fact a binding (has a value property) then returns a function on the stack that will call the code provided as argument

```
mutator('set', (slot, property) => slot[property] = get())
```

The mutator captures this function provided for set and in the case of "set some-binding" it will call this function with:

```
const the_set_provided_function = (slot, property) => slot[property] = ge
the_set_provided_function(binding, 'value')
```

Being the binding, the "some-binding" word that found at compile time.

The code registered for set will receive (binding, 'value) as arguments

And it will just do

```
binding.value = get()
```

Which is the same function bind returns as the setter.

If the mutator function registered for set reads "some-name:" with the ":" at the end it compiles a delayed binding.

The function returned by set in this case will search that word at runtime and set it with a value on the stack.

The code that set provided will still see (object, property) as arguments.

The object and property values will be also (binding, 'value'), the binding just was searched at runtime instead of compile time.

For dot notation the mutator function will return a function that will traverse the chain of properties and give the code provided for the set function the last object and the last property.

For example:

```
'red set body.style.color
```

```
the_set_provided_function(style, 'color')
// style.color = get()
```

If the token starts with a dot then the object will be taken from the stack exactly like the dot notation provided by compile atom since it's the same interface.

```
'red body set .style.color
```

A mutator is just a way to unify all those four possibilities

It means that any mutator word will have the same interface:

```
24 increment some-value
24 increment player.x
24 player increment .x
```

```
increment-by-one some-value
```

Note increment-by-one does not need a value from the stack except for the object when the token starts with a dot.

```
player increment-by-one .x
```

compile atom uses the same function the mutator uses for dot notation so the interface is the same.

But compile atom only provides a way to access and set a property while the mutators can be a specialized setter.

The word "set" is equivalent to the dot notation syntax compile atom provides when it ends with a "!"

```
24 set player.x
24 player.x!

'red body.style.color!
'red set body.style.color
```

It has the same effect.

But compile atom does not provide "+!" or similar, that is provided by increment and decrement words.

The if word is an immediate 1 word.

It reads 3 blocks of code. One for the test code, other for the true branch and an optional one for the false branch

```
if 1 then
 do-something
else
 do-something-else
end
```

The code between "if" and "then" is the test code.

You can put arbitrary code there, it should leave a value on the stack that the if will use as the flag for true or false.

The code between "then" and "end" is the code that will execute.

By default the code is for the true branch unless you put the "else" token which will make the if word start compiling the code for the false branch.

All three blocks can be empty including the test code.

If the test code is empty the if word will execute it anyways at runtime and take a value from the stack.

That means that if is empty the value will come from the stack.

```
: is-true?
  if then
   "is true"
  else
   "is false"
  end
  log
;

1 is-true?
```

The else token is optional

```
if 1 then "this always executes" end
```

The if word will read until the "then" token and then until the "end" token.

If the else token is found it will also compile code for the else branch.

It returns a function that when executed will execute the code for the test (the code between if and then), then take an element from the stack and use that to check for truth, then evaluate the true branch code or the else branch code depending on that value.

So the if word is an immediate 1 word that returns this function to compile element by pushing it on the stack after compiling those three blocks.

```
put(() => { test(); if (get()) { true_code() } else { false_code() } })
```

And that's what ends into the compilation array.