Design and Implementation of a LISP Dialect & Interpreter

AQA NEA Comp. Sci.

Samuel F. D. Knutsen

Last Compiled: April 10, 2019

Beauchamps High School and Sixth Form,
Centre №16137,
Candidate №2248.

Contents

Abstract — Throughout this paper I will be looking at use cases, motivations, implementation, documentation and usage of building an interpreter for our new Lisp Dialect.

1 Analysis

In this section I will be investigating the usage and implementation of a Lisp Interpreter of my own variety, i.e. the language will be inspired by the conventions and harbour the basic properties of a Lisp. These variations upon the language are called Lisp Dialects, one of which will be my own, which I will be implementing, and outlining in this paper.

LISP – is a style of programming language, and stands for LISt Processor.
because of the fact that the entire language itself is simply constructed from data, which are just lists of *datum*, but more on that later.

Designing and implementing a language are two distinct steps, but I need to consider both while working on each of the steps. That's to say, that when designing the language, I need to consider what would be realistic in implementing the language afterwards. The features of a language are limited to what I will be able to actually write the code for, as well as not allowing for ambiguity in the language. In the second stage, when implementing it, I also need to stay true to our original design, and not stray from what I originally intended (with the exception of allowing expansion upon the language), unless I had set unrealistic design goals.

1.1 Motivation

Programming languages are not only naturally very important for programming and programmers themselves, but also a very fascinating topic, bridging the gap between what is human language, and what is a computer language.

My motivations for making a Lisp interpreter, however, are many. Lisp, for one, is quite a powerful language, and I mean that in the sense of how much you can do and manipulate the language and your program themselves. The major advantage of Lisp is indeed that it can be manipulated the very same way data can be manipulated, as the language is data itself.

Lisp macros are really what sets Lisp apart. If I asked you to implement a while loop statement or an if statement in Python, could you do it? If you tried, you'd end up with an ugly mess of lambdas or perhaps even pieces of your program written inside strings that get parsed *after* run time, and such. In Lisp such a matter is trivial, thanks to its powerful macro system that is capable of building parts of the program in the program itself.

Speaking of the `while` loop, it is indeed very often implemented in the Lisp language itself, as opposed to being built into the interpreter. Here is my example, which is quite trivial, and will become more understandable, after having read this paper through:

```
(define macro (while condition body)
  (iterate
    (if (eval condition)
      (eval body)
      break)))
```

Another motivation is the potential that I may implement some features into the language that other Lisp dialects don't have, in addition, I can also control certain features that I don't want in the language, that other Lisp dialects almost almost religiously include (for example confusing naming on `car`, `cdr` and such, when `head` and `tail` (or `rest`) would suffice).

1.2 Research

To begin this project, a certain amount of research needs to be covered. First we need to familiarise ourselves with the general concept of what defines and encompasses a Lisp and what I'd like my dialect to be inspired by.

1.2.1 What makes a Lisp?

Lisp is a *family* of scripting/programming languages belonging typically to the declarative/functional paradigm of programming languages. It is characterised by its heavy use of parentheses ('`(`' and '`)`'). The reason for this, is because the entire syntax itself is also a data structure.[1] More on the specifics on the syntax in the design section.

1.2.2 Making our language executable

Essentially, this is how to make a programming language be evaluated and how it can become executable on a computer, by a computer.

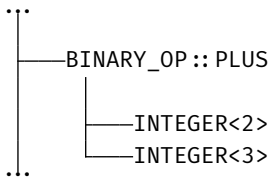
Almost every implementation of the most common programming languages start with what is known as a 'lexer' or 'tokeniser' that performs lexical analysis. As one might gather from the name, this stage in the implementation tokenises the program. Every program starts of as just a long string of characters stored in a file, these individual characters aren't useful, we want to gather them up in to groups of characters, e.g. the program "`print("Hello World")`" is understood by the computer as just a string of characters: `['p', 'r', 'i', 'n', 't', '(', '"', ...]`, this is not useful. We want to group the individual components of the language together. As such:

```
[[IDENTIFIER, 'print'], [L_PAREN, '('], [STRING, 'Hello World'], [R_PAREN, ')']]
```

After this we proceed to the parsing stage. This essentially forms an abstract syntax tree from the tokens. This let's us define some sort of separation and concept of nesting between statements, grouping them together in a logical order.[2] e.g.

```
"2 + 3" ⇒ [[NUMERIC, '2'], [OP, '+'], [NUMERIC, '3']]
```

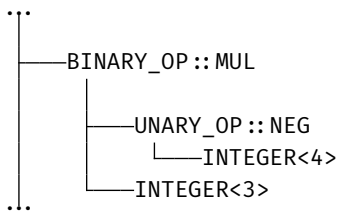
becomes



As a branch on a bigger syntax tree.

Another example:

```
"-4 * 3" ⇒ [[OP, '-'], [NUMERIC, '4'], [OP, '*'], [NUMERIC, '3']]
```



Now that we have our syntax tree, I will have to implement some sort of a visitor/walker, in order to execute and evaluate our instructions. Usually we would start with the leaves of the tree and work ourself up, eventually evaluating the whole expression, having evaluated the sub-expressions first.

Luckily for me, I needn't consider operator precedence, as 'operators' are just function names which can be called, which forces prefix/polish notation to be used, where precedence is implied by the order of the operators and their bracketing. Another pro of parsing Lisps is that the Abstract Syntax Tree ('AST' from here on) already has a lot of similarity to the language itself, over all parsing is much easier than it would be for some of today's more 'human oriented', interpreted programming languages.

2 Design of the language

There are many ways of implementing a Lisp, the earliest Lisps had a very limited amount of key/reserved words (about 4), which from the early versions of MacLisp and Common Lisp has increased to about a minimum of about 9 or 10.

One of the most important and powerful features of Lisp is that its syntax is practically its own syntax tree. These nested lists that the Lisp grammar is composed from are called s-expressions,[3] meaning "symbolic-expressions".

2.1 Symbolic Expressions

In many (but not all) Lisp implementations and dialects, one of the most fundamental datatypes is the cons cell. The cons cell is a way of constructing trees, and thus lists alike.

Lisp ASTs are essentially linked-lists, capable of containing pointers to other nested lists (i.e. a 'tree'). Take the list `(x y z)` for example, to represent this in the form of cons cells, we need to consider the fundamental cons function. The cons function takes in a fundamental datum as first argument (can also be a pointer to another list) and a pointer to the next cons cell, and to end the list, simply a NULL pointer (an empty list or NIL is often used in the place of NULL).

Hence, `(x y z)` becomes `(cons x (cons y (cons z NULL)))`.
But this way of writing is verbose, so the preferred notation is the dot-pair notation, so, `(cons x (cons y (cons z NULL)))` is written as `(x . (y . (z . NULL)))`

A linked-list/n-ary tree as such can implemented very plainly in C:

```c
struct cons_cell {
  atom data;
  struct cons_cell *next;
};

typedef struct cons_cell *cons;

// Thus, (x y z) will be:

atom x = Symbol('x');
atom y = Symbol('y');
atom z = Symbol('z');

cons make_cons() {
  cons parent = malloc(sizeof(struct cons_cell));
  parent->next = NULL;

  return parent;
}

/*  I could simply write a quick little function for what
 *  I'm about to do, but I'll write out the construction of the
 *  cons list explicitly, such that it's more obvious what we're doing.
 */
cons list = make_cons();
list->data = x;
list->next = make_cons();
list->next->data = y;
list->next->next = make_cons();
list->next->next->data = z;
list->next->next->next = NULL;  // Same as (x . (y . (z . NULL)))
```

Which can be visualised as:

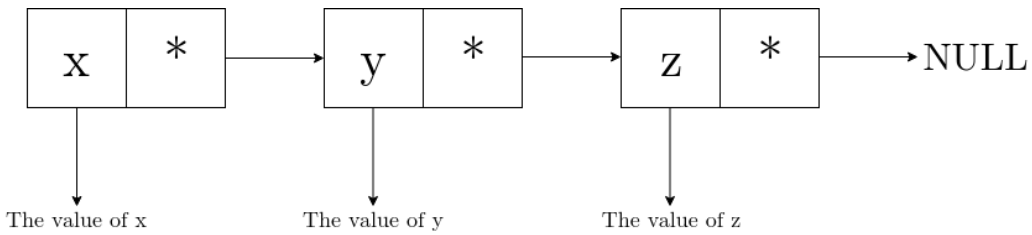


Figure 1: Visual representation of cons cells (i.e. a linked list)

However, I will not be implementing my s-exps in this manner, nor will my Lisp dialect even contain the function `cons` (although `cons` can be very easily implemented as a macro). This is because I will not be writing this in C, but rather in a higher level language (Python >3.7). I'll be writing this in Python as it is what my school has available and it is what is taught there.

2.2 Syntax

Lisp consists of lists, and lists within lists, as I discussed earlier. Every list is denoted by a set of parentheses where the first element is generally a function name, and the following being its arguments. Some lists get expanded from their macro at the pre-compilation/preprocessing stage, and eventually all lists get evaluated as a function call. One might wonder how one makes a normal list, if Lisp treats all lists as function calls. Lisp has a somewhat unique feature, and that is the 'unevaluate'/'quote' "operator" (if you will) which is denoted by putting a `'` (single single-quotation mark) in front of any expression, to stop the interpreter from evaluating it. So, your standard list or array is denoted:

`'(a b c)`
And when we want to return it back to its former unquoted self,
we call the `eval` function on it:
`(eval '(a b c))` is the same as `(a b c)`

Here we see that list items are separated by spaces, rather than the more common comma (`,`) in other languages, making white-space significant to some degree. However, other than that, white-space is quite insignificant and the language has no need for things such as terminators (terminators are semi-colons in C and a new line in Python for example), and that's because everything is neatly wrapped up in parentheses and therefore every statement is automatically separated from each other, anyway.

2.2.1 Polish Notation / Prefix notation

Lisp could be said to be a language using Polish notation, that's to say that the language places its 'operators' (which we'll discuss later) as prefixes, meaning they're put in front of operands. Traditionally operators are placed in between their operands, this is called 'infix' notation ($1 + 2$ for example).

Prefix notation has some advantages and disadvantages, really, the only disadvantage to prefix notation is the fact that it may be less readable. The *sole* reason for the fact that it is less readable is simply that we are not used to it, because we're used to infix notation, instead. We could just as easily be using prefix notation, but things didn't turn out that way, but that doesn't prevent us from learning and quickly getting used to it.

On the other hand, the advantages really do make up for the disadvantages. One major advantage is the fact that precedence is *implied*, if and only if the exact amount of

operands an operator can have are two; when that is true there is absolutely no need for things such as parentheses. However, Lisp *does* make heavy use of parentheses and thus suffers from another formidable advantage, 'n-arity' of operators. N-arity, Polyadic or Variadic functions/operators are simply operators that can take any number of operands. How would you add up five numbers together using infix notation? You'd have to use the operator four times over (e.g. $1 + 2 + 3 + 4 + 5$ which bracketed would become $1 + (2 + (3 + (4 + 5)))$ (because really infix operators can only take two arguments)) as opposed to in Lisp, where you only need to use the + operator once, (e.g. (+ 1 2 3 4 5), far fewer characters, use of operator was only once and no implicit brackets).

The fact that Lisp uses prefix notation was not necessarily a choice for the language per se, but rather a consequence of the languages structure and a symptom of the language's aim to, itself, *be data*.

2.2.2 Unique Data-types

Most Lisp dialects have about 7 fundamental datatypes:

1. Integers
2. Floats
3. Cons
4. Symbols (including Keywords)
5. Strings
6. Functions
7. Null

However as mentioned before, because of the way I'll be implementing this dialect, I will be excluding `cons` as a part of the language and will also be introducing a new one, which I'm calling the Atom.

My new list of datatypes which will be implemented in my dialect is as following:

- Numerics (an umbrella for integers and floats)
- Symbols
- Atoms
- Strings
- Defintions (impure functions)
- Nil

My atoms are not at all your standard Lisp atom, in most Lisp contexts 'atom' just means a fundamental datum, but this is not what I'll be calling an atom.

Atoms are a useful construct in my experience. The Ruby programming language has them, and calls the 'Symbols', however that means something totally different in the context of a Lisp, the Erlang and Elixir programming languages have them, and many Lisps have something that heavily resembles atoms (both syntactically and practically) called 'Keywords', but my 'Atom' is slightly different.

Atoms simply exist for their name. No atoms ever need to be created/initialised (manually) as one could simply imagine that all possible atoms already exist at once.

When an atom is first used, it is assigned a location in memory based on some hash, and any future use of that same atom references the same exact location in memory.

Atoms are not strings, you can't change anything about them, to do that, simply use another atom. Two separate identical strings will take up and be declared in different locations in memory, this is not true for atoms. Atoms will always use the same location in memory, there is no reason for them to do otherwise, (unlike with strings)

Atoms must start with a colon, then an identifier string, just like symbols (but with a colon in front). One may think of atoms as being used the same way as `enums` are, existing only for the purpose of having a way to identify something.

e.g. `:this-is-an-atom`

2.2.3 Operators

Technically in Lisp, any function could be called an operator. It is also true that all the operators we traditionally call operators are also operators in Lisp (i.e. `+, -, *, /, %,` &c.). Hence, transitively, all (traditional) operators are functions!

The fact that all operators are just functions, I see as a hugh advantage to the language, because, not only does that allow the programmer to extend the meaning of an operator, but it also allows them to create their own operators just as easily as defining any function. Another consequence of this that we may use operators in our function names / symbols. Just as `+` is a function, you could have a function named `++` or `**`, and that can be used with alphanumeric characters too, to have a function `*hello*` for example, and even `hello-world`, which is a common mistake amongst beginner programers, trying to use hyphens in variables names, but the language (most likely C or Python, &c.), understands `hello-world` as `hello` *minus* `world`, as opposed to what the programmer *actually* meant. Perhaps another advantage of Lisp…

2.2.4 Key/Reserved Words & Symbols

Every language has a certain amount of reserved words and symbols that actually form, comprise and compose the syntax of a language, some languages will allow you to redefine these key words, but most won't, as it would essentially break the language. Basic examples of these key words, in Python for example, are words such as: `return`, `while` `def` and `True` (also including operators such as `=`).

My language should have a few defined reserved words, but my aim is to keep the number of them as low as possible, because, like in many Lisps, many things can be defined as macros in a standard/prelude library anyway. Some words that I plan to reserve are the following:

- Yield statement, similar to the return-statement, `(yield ... )`.
- Unevaluator, this is the single single-quote, `'some-name`, `'(some list of symbols)`.
- Nil, this is the only kind of its type,
  and will be entirely reserved as its own lexeme, `nil`.
- All symbols such as `(` and `)`, and `"`, as they exist to describe other constructs. (i.e. lists and strings)

Other than that, the semi-colons (`;`) will describe EON comments, Atoms and Symbols will be matched through certain expressions, and anything else will be completely ignored by the language.

Other items, that are not understood by the Lexer and Parser, but are still technically part of the list of reserved words are the internal macros, which will be identified and dealt with at the evaluation stage are called 'internal macros', because their behaviour is technically like macros, but they are built internally into the interpreter.

2.2.5 Internal Macros

Internal macros in a language are also not usually allowed to be reassigned in most languages, however a surprisingly large amount of Lisp dialects and implementations do allow this, however I will not be allowing this myself.

My initial list of internal macros will be as following, (but I am certain it will expand when I actually implement the language):

- `define` – Probably the most important macro, it takes in a name, arguments for a function and a function body, and creates a function in the current scope with that name. Essentially the equivalent of `def` in languages such as Ruby, Python, Scala, &c.
- `lambda` – Same as define, but the function is nameless. This is also called an anonymous function.
- `let` – Would bind a value to a symbol, say $x = 3$, in Lisp we'd write `(let (x 3))`.
- `print` – Print the operands of the print statement to `STDOUT`.
- `if` – `if` is a three way statement, the first operand is the condition, the second is the consequence, the third is the alternative, if-this-then-that-else-that.
- `<` – Check if each operand is less than the next.
- `>` – Check if each operand is greater than the next.
- `=` – Check if all operands are the equal to each other.
- `/=` – Check if at least one operand is not the same as the rest.
- `type` – Returns the type of its evaluated operand.
- `list` – Creates an unevaluated list, but all the operands get evaluated at the construction of the list.
- `+` – Addition macro will add all its operands.
- `-` – Subtraction macro will subtract all its operands from each other.
- `*` – Multiplication macro will multiply all its operands.
- `/` – Division macro will divide all its operands by each other.
- `%` – Mod macro will give the remainder of the division of all its operands.
- `eval` – Takes an unevaluated list or a string, and evaluates it as normal code, a very important macro in any Lisp dialect.

2.3 Grammar

Any Lisp's grammar is naturally very similar, and for the most part quite simple, but has indeed steadily been increasing in complexity over the years. The Original Lisp had a very simple grammar and syntax, and had only the symbols `DEFINE`, `LAMBDA`, `LABEL`, `COND`, `COMBINE`, `FIRST`, `REST`, `NULL`, `ATOM`, `EQ`, `NIL`, `T`, and `QUOTE` predefined.[3] All symbol names were converted to upper-case and thus symbol names became case insensitive in

early Lisps, this is still a tradition upheld in many Lisps today (because it's said to be less error prone), however this is not a tradition I will be upholding.

Many modern Lisps have not only introduced the shortened quote syntax (`' ...` as a shorting of `(quote ...)`), but have also introduced another form of quote, the 'quasiquote'.

The quasiquote works just as the normal quote but with added functionality. It can "unquote" selective operands through use of the `(unquote ...)` macro inside of the `(quasiquote ...)` macro. This essentially means you can choose to evaluate certain operands, unlike quote, which keeps all operands unevaluated until you choose to evaluate them individually after its initial construction, when accessing the unquoted list.

In modern Lisps, some further syntactic sugar has been introduced for the quasiquote:

`(quasiquote ...)`	has been aliased to	`` `... `` (a backquote/backtick)
`(unquote datum)`	has been aliased to	`,datum`
`(unquote-splicing data)`	has been aliased to	`,@data`

(Where `unquote-splicing` unquotes an entire list and merges it with the parent list)

Some examples of behaviour include:

```
`(1 2  (+ 3 4) 5)      ⇒    '(1 2 (+ 3 4) 5)
`(1 2 ,(+ 3 4) 5)      ⇒    '(1 2 7 5)
`(1 2 ,@(list 3 4) 5)  ⇒    '(1 2 3 4 5)
```

Lisp syntax is always expanding, and the sharp-sign (`#`), is actually a syntax element, purely meant for expanding the syntax itself, for example, in Common Lisp, `#\c` is a character, `#(...)` is a *vector* and `#xff` is the hexadecimal for 255_{10}. At this point I personally think the syntax is getting out of hand, and I doubt I'll be implementing this in my Lisp dialect.

2.3.1 Generating a Token Stream

The lexical analyser is responsible for reading through our program string and splitting up the program in to sensible tokens representing a single datum, that comprises and represents something in the languages syntax. Take for example the program:
"`(+ 12 (* 3 4))`", our brains have already started tokenising and parsing the text, as soon as we look at it, and we can identify quickly some basic components, brackets, operators, numerics and spaces, but, for a computer these are all characters of no particular separation. A lexers job is to separate them in to parts, such as say
`List [ Sym +, Num 12, List [ Sym *, Num 3, Num, 4 ] ]`, and ensure we dont get any non-sense tokens such as '`(+`' or '`* 3`'.

Very often, when parsing regular languages, a lexical analyser may use simple regualar expressions to parse the language, say for example, a very primitive numeric matcher (for ints and floats), may look something like:

```
/[0-9]+(\.[0-9]+)?/
```

In pseudo-code, a basic lexer may look something like:

```
TokenStream = List

lexer :: String → TokenStream
lexer (program) ⇒
  char_pointer = 0
  stream = new TokenStream

  partial = program[0..]

  while partial[0] ≠ EOF ⇒
    condition if
      | (partial[0] == '(') ⇒
          stream.push (new Token L_PAREN)
          char_pointer = char_pointer + 1
      | (partial[0] == ')') ⇒
          stream.push (new Token R_PAREN)
          char_pointer = char_pointer + 1
      | (partial[0..2] == 'nil') ⇒
          stream.push (new Token NIL)
          char_pointer = char_pointer + 3
      | (match /[0-9]+(\.[0-9]+)?/ partial[0..]) ⇒
          stream.push (new Token NUMERIC matched)
          char_pointer = char_pointer + length(matched)
      | (match /[a-zA-Z_]+[a-zA-Z_0-9]*/ partial[0..]) ⇒
          stream.push (new Token IDENTIFIER matched)
          char_pointer = char_pointer + length(matched)
      | otherwise ⇒
          char_pointer = char_pointer + 1

    partial = program[char_pointer..]

  return stream
```

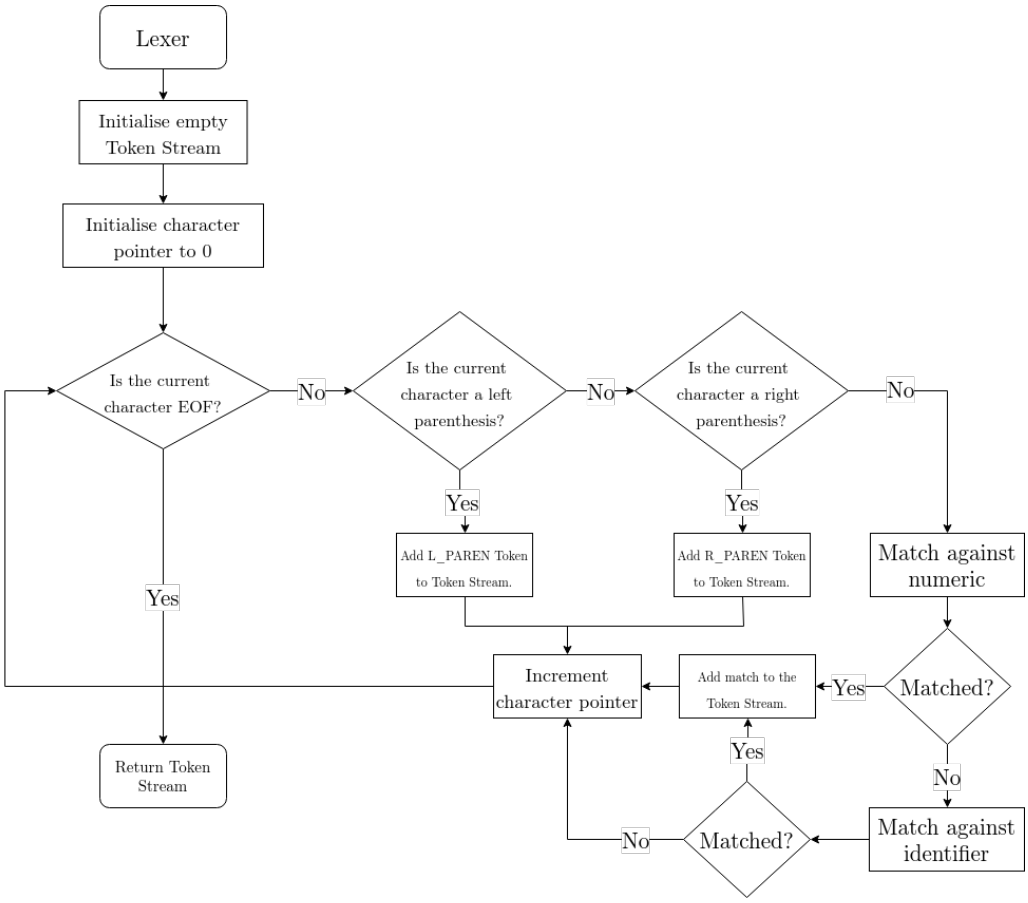


Figure 2: The lexer also represented by a flow-chart.

This basic function `lexer` does most of the works, it takes in a `String` (the program string itself) and gives a `TokenStream` (which is really just an alias for a list). The lexer initialises some variables, very notably, the `char_pointer`, which very simply keeps a track of how many characters through the program code we are. Then the `partial` variable is just a section of the program string, it begins at the location of `char_pointer` and all the way to the end of the string.

A while-loop will run all the way until we hit a null (`\0`) EOF string terminator, and checks various conditions as well as incrementing the `char_pointer` and updating the `partial` variable. The conditions check whether we've matched a certain token out of the beginning of the `partial` string, if so, we push a new `Token` to the `stream` list, with the appropriate type and matched string supplied with it.

2.3.2 Generating an Abstract Syntax Tree

Now that we have our list of tokens, the next step is to take those tokens and reconstruct the program as a tree. The tree introduces important concepts back into the structure of the program. Things such as nesting and precedence are clearly represented through the use of a tree. A tree will also help us give appropriate scoping within certain nests at the evaluation stage.

The parsing pattern we'll be implementing for generating our tree will likely be a bottom-up parser (or a shift-reduce parser). A shift-reduce parser does exactly what its name says it does: it shifts tokens off the token stack, and parses the *leaves* of the tree, before it generates their super-branches (hence the name bottom-up). It is often implemented recursively (as with many parsers) and is generally a lot more effective than a top-down parser, which requires a lot of guess-work.

A general implementation of such a parser written in pseudo-code, could look something this:

```
# -- Get a our stream of tokens by calling the lexer
stream : TokenStream = Lexer::lex PROGRAM_STRING

# -- Define various AST datatypes
DataType Tree ⇒
  self.children = []

DataType Call (values) ⇒
  self.values = values
  get self.operator ⇒
    return head (self.values)
  get self.operands ⇒
    return tail (self.values)

# --- Generalised datatype for an atomic AST datum
Abstract DataType Atomic (value) ⇒
  self.value = value

DataType Numeric inherits Atomic  # Atomic types represent a single datum.
DataType Symbol  inherits Atomic

# -- Actually implement the parser
parse :: TokenStream → Tree  # (Type annotation)
parse (stream) ⇒
  tree = new Tree # Create an empty tree-root
  until empty? stream ⇒
    tree.children.push (atomic (stream)) # Deals with parsing individual datum
    stream.shift # Now shift off of the TokenStream stack
  return tree # Return the super-tree we've built up.

atomic :: TokenStream → (Call | Atomic)
atomic (stream) ⇒
  condition if ⇒
    | (stream[0].type is L_PAREN) ⇒
        call = new Call []  # An open left-parentheses means a function call.
        until stream[0] is R_PAREN ⇒ # Serch for a maching right-parentheses.
          stream.shift # shift off L_PAREN from stack (initially).
          call.values.push (atomic (stream))   # Push and recursively call
        stream.shift # Shift the R_PAREN off   # atomic on shifted stack.
        return call
    | (stream[0].type is NUMERIC) ⇒ return new Numeric (stream[0].string)
    | (stream[0].type is IDENTIFIER) ⇒ return new Symbol (stream[0].string)
    | otherwise ⇒
        Throw (UnknownTokenType,
          "Token type" ++ stream[0].type ++ "is unknown.")
        # If our lexer was implemented properly, we wouldn't have to throw
        # an error, so we'll hopefully never reach the `Throw`.
```

The above code, may seem weird at first glance, so let's go through both of the functions defined above.

`parse` — The function `parse`, takes in a stream variable of type `TokenStream` and returns a tree of type `Tree` (which is basically a wrapper for a list). Somewhat ironically, it doesn't implement any of the parsing algorithm itself, but rather calls a delegated function (`atomic`) to handle the parsing of atomic data. `parse` will run a loop, which runs until the stream of tokens has been completely emptied. In the loop body, we call the delegated `atomic` function on the stream, whose return value will get pushed as a child/branch onto the parent/root-tree. After that we shift one token off the stack and repeat until all tokens have been shifted off, (note: `atomic` may shift tokens off the stack as well, when parsing sub-expressions for a bigger expression).

`atomic` — The `atomic` function takes a (probably incomplete, due to shifting) token-`stream` variable of type `TokenStream` and returns either a `Call` node or a derived `Atomic` node such as `Numeric` or `Symbol` (i.e. `(Call | Atomic)`). `atomic`, is just one big conditional, that operates depending on which token happens to be on top of the stack at that point in time. The first condition checks whether we have a function call being opened/started in our code (i.e. we've seen a left-parenthesis).

> If we do spot an `L_PAREN` atop of our stack, we do indeed have a function call, and hence we must continue shifting off the stack until we reach a closing right-parenthesis. But, remember, being a bottom-up parser, we must first parse all sub-expressions, so we recursively call `atomic`, to deal with all the expressions contained within the parentheses. Doing this, we also have implemented and permitted nesting into our language, by allowing other function calls to be parsed within parent function calls. When we've finally reached that corresponding right-parenthesis, and have pushed all our parsed sub-expression to the function call, we return the `Call` node and we end up back in the `parse` function, where the `call` node is pushed to the root-tree (or it may indeed be pushed to another parent call-node).

2.3.3 Macro Expansion Phase

After we've generated the most basic syntax tree, we should start the macro expansion phase. Macros, you may be familiar with from other languages (especially compiled languages), they're part of the preprocessor stage, and thus do not exist at evaluation time.

Let's demonstrate through an example. Take the macro, *and* the function:

```
(define macro    (add-3-macro x) (+ x 3))
(define function (add-3-func  x) (+ x 3))

(print (add-3-macro 2) "\n") ⇒ 5
(print (add-3-func  2) "\n") ⇒ 5
```

Both *eventually* end up evaluating to `5`, but did so by different methods. `add-3-func` had it's own Symbol Table made, that symbol table was pushed to the call stack, and the symbol `x` was bound in it. `x` was bound to the value `2` when the function was called, the function returned the evaluation of its body, with it's own bound symbols and then had it's symbol table wiped clean, and popped of the interpreter's call stack. Wew! All that for something that could have just been written as `(+ 2 3)`…Well, that's exactly what that macro did!

The macro, is essentially just text replacement (although what the macro accomplishes is actually done through manipulating the syntax tree). So, when the macro-expander sees your making a call with a macro-name as the caller, it essentially copy-pastes the arguments into the macro body, and replaces all instances of `x` with `2` (in this example).

So essentially,

`(print (add-3-macro 2) "\n")` expands to: `(print (+ 2 3) "\n")`

2.4 Interpreter

Finally, to what's probably the most important part of the implementation. The interpreter or *evaluator* is what will be doing the computation, by traversing the syntax tree that we've just generated.

The interpreter will essentially consist of three important parts/aspects.

1. Symbol Tables, and usage of them in Scoping.
2. Defining internal macros/functions.
3. Running through the sub-trees of the AST recursively and evaluating appropriately certain expressions, by first evaluating the leaves of the tree, slightly similar to how we parsed the program.

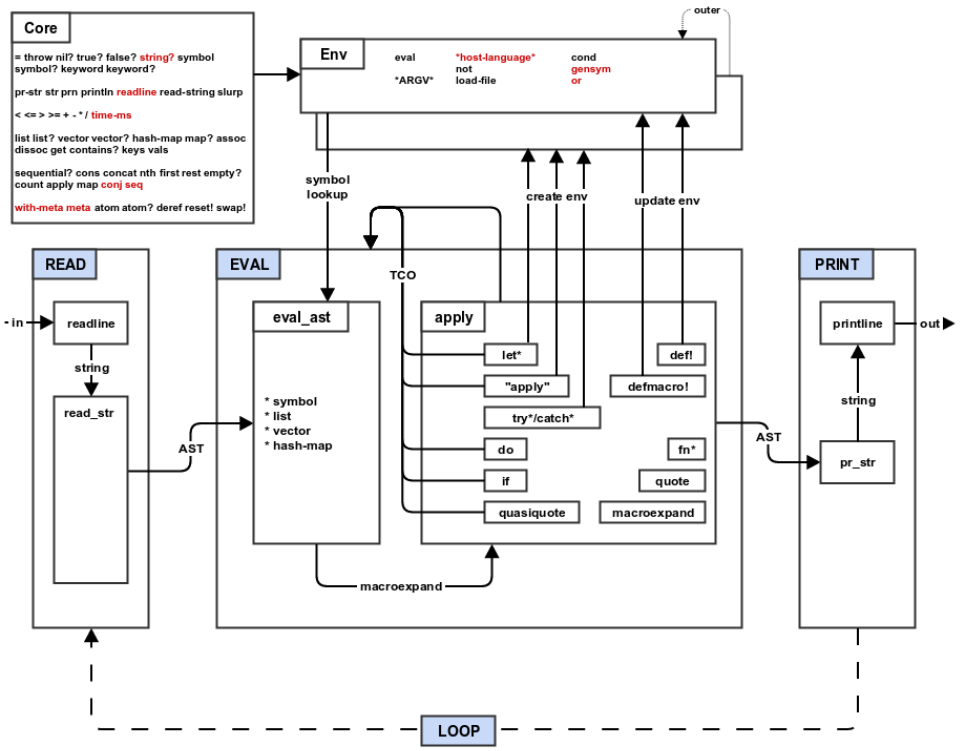


Figure 3: Here is an example of how a typical fully functional REPL-based Lisp interpreter might function.

2.4.1 Scoping & Binding Symbols in Symbol Tables

Let's start off by defining various symbol-table stacks (such as call-stacks and *frozen*-stacks and a list of current, parent scopes)

```
DataType SymbolTable (scope_id) ⇒
  self.id = scope_id # A unique ID for each scope.
  self.local = %{} # A hash-table of all local variables to this scope.
  self.frozen = :false # Whether the symbol table can be modified
                       # (lambdas will have frozen super-scopes).
  self.bind (symbol, value) ⇒
    self.local[symbol] = value
  self.clean ⇒
    self.local = %{} # Empty the local variables.

# --- Define all the scope stack we need to keep track of:

ALL_SCOPES    = new Stack of SymbolTable
PARENT_SCOPES = new Stack of Integer  # Integer IDs (i.e. unique scope IDs)

CALL_STACK    = new Stack of SymbolTable
GLOBAL_FROZEN = new Stack of SymbolTable where .frozen ← :true
# ---

# A type for function definitions.
DataType FuncDef (scope, name, args, subtree) ⇒
  self.name = name
  self.subtree = subtree
  self.scope = scope
  self.table = find(scope, :id, ALL_SCOPES)
  self.args = args

  self.call (operands) ⇒  # What happens when we call a
    PARENT_SCOPES.push (self.scope) #    user defined function.
    CALL_STACK.push (self.table) # Push appropriate scopes.
    for i in length (self.args) ⇒
      self.table.bind(self.args[i], operands[i]) # Bind arguments,
                                 # to the call stack's latest scope.
    evaluate(self.subtree) # Finally evaluate the body.

    PARENT_SCOPES.pop
    CALL_STACK.pop # Pop the symbol-table off again.
    self.table.clean # Get rid of old bindings.
```

This outlines the tools we'd be using, for dealing with scoping things in our language.

2.4.2 Dealing with Internal Macros

Internal macros are really important. They provide a way to interface with the computer directly (or in this case, it'd be the implementation language (i.e. Python)), which introduces very useful functionality into the language. Take for example, the add `+' function/internal macro. It is basically required that this is not implemented in the language we're making, itself, as it is one of the most basic operations we can perform, and there'd be no way of implementing it without direct help from our host language. Others include `define` and `-` and so on. I've already run through numerous macros earlier in this paper.

Say for example, our interpreter encounters a symbol within our defined list internal macros, perhaps layed out as a hash-map like so:

```
INTERNALS = {
  :define => __DEFINITION_MACRO__,
  :+      => __ADDITION_MACRO__,
  :-      => __SUBTRACTION_MACRO__,

  ⋮

  etc.
}
```

where `__DEFINITION_MACRO__` and such are functions that deal with the computation of that internal macro. So in the evaluation function, we'd deal with it something like this.

```
# Inside the evaluation function:
evaluate :: (Call | Atomic) → Anything
evaluate (node) ⇒
  # ...
  if typeof (node) is Symbol ⇒
    if node.name is in INTERNALS ⇒
      return INTERNALS[node.name]
    # Otherwise, look in the Symbol Tables
    return lookup_symbol (PARENT_SCOPES, node.name)
  # ...
  if typeof (node) is Call ⇒
    caller = evaluate (node.caller)
    if typeof (caller) is FuncDef ⇒
      return caller.call (caller.operands)
    if caller is INTERNAL_DEFINITION ⇒
      return caller (node)
    else ⇒
      Throw (UncallableCallerError, "Can't make call to this type ... ")
  # ... etc.
```

What would an internal representation of one of those macros look like? Let's take the + internal macro, perhaps it would look something like this (in pseudo-code):

```
__ADDITION_MACRO__ :: Call → Atomic
__ADDITION_MACRO__ (call_node) ⇒
  args = map(evaluate, call_node.operands) # Evaluate every operand first.
  sum = 0
  for arg in args ⇒
    if typeof (arg) is not Numeric ⇒
      Throw (TypeError, arg ++ " is not a Numeric type.")
    sum = sum + arg
  return sum
  # An internal iterative solution is much more efficient
  #   than a more idiomatic recursive solution.
```

Really, theres nothing fancy going on here, it is simply a way to build the most basic components of the language, and interface with features (such as basic addition) from the implementation language.

2.4.3 Interpretation Pattern

As briefly seen above, there are two basic functions I plan to be implementing. A `visit` function and the very critical `evaluate` function.

Let's start by describing the `visit` function. It's a simple to implement, all it needs to do is visit each root-branch of the `Tree` and evaluate each branch (which will subsequently evaluate all its subtrees recursively) in a simple loop. It may be implemented as such in pseudocode:

```
visit :: Tree → Nothing
visit (AST) ⇒
  for child in AST ⇒
    evaluate (child)
  # The most basic idea of the visitor, although it may
  #   do error handling an other things as well.
```

Naturally, we must have an implementation of the evaluate function too. We had partially implemented the `evaluate` function above, but let's now complete that implementation.

```
evaluate :: (Call | Atomic) -> Anything
evaluate (node) =>
  case typeof(node)
  when Numeric =>
    return literal_eval (node.value) # Doesn't really get evaluated, per se,
                                     # as its really an 'atomic' datum.
  when String =>
    return node # Already the most basic datatype.
  when Atom =>
    # if the atom already exists in memory, just return it.
    if node.value is in ATOMS_HASHMAP =>
      return ATOMS_HASHMAP[node.value]
    # Otherwise, add it to the hashmap, giving it a unique location
    #   in memory, which will be referenced whenever the same atom is
    #   used again.
    ATOMS_HASHMAP[node.value] = new Atomise(node.value)
    return ATOMS_HASHMAP[node.value]
  when Symbol =>
    if node.name is in INTERNALS =>
      return INTERNALS[node.name]
    return lookup_symbol (PARENT_SCOPES, node.name)
    #      ^^^^^^^^^^^^^ Simple to implement, obvious what it does.
  when Call =>
    caller = evaluate (node.caller)
    if typeof (caller) is FuncDef =>
      return caller.call (caller.operands) # Make call to user defined function.
    if caller is INTERNAL_DEFINITION =>
      return caller (node)
    else =>
      Throw (UncallableCallerError, "Can't make call to this type ... ")

  default =>
    Throw (UnknownTreeNode, "I do not recognise the type of data you've
                             passed to be evaluated.")
```

2.4.4 Including a REPL

It'd be nice with a REPL as well. Quite simply a REPL is a READ-EVAL-PRINT-LOOP, and its implementation is in its name.

```
(loop (print (eval (read))))  ;; Just one line, implements the whole thing.
```

This is quite a naïve implementation, but it still works just fine, all we need to do, is execute the Lisp code using our interpreter.

3 Technical Solution & Implementation

In this section we'll be walking through the code (Python code) that implements the entire Lisp language that we've designed. Somewhere around the beginning of writing the implementation of the language I realised I would need a name for the language, and I settled on Lispy, which is essentially just 'Lisp' and 'Python' put together. Python files do end in the extension `.py`, and hence the file extension we'll be using for Lispy program files is `.lispy`. It's perhaps not the most creative name, and no doubt a name that's (probably) been used before, by someone else writing their Lisp in Python.

Any absolute file paths presented are relative to the root directory of the repository for the code.

The file structure of the repository is easy to follow and looks like this:

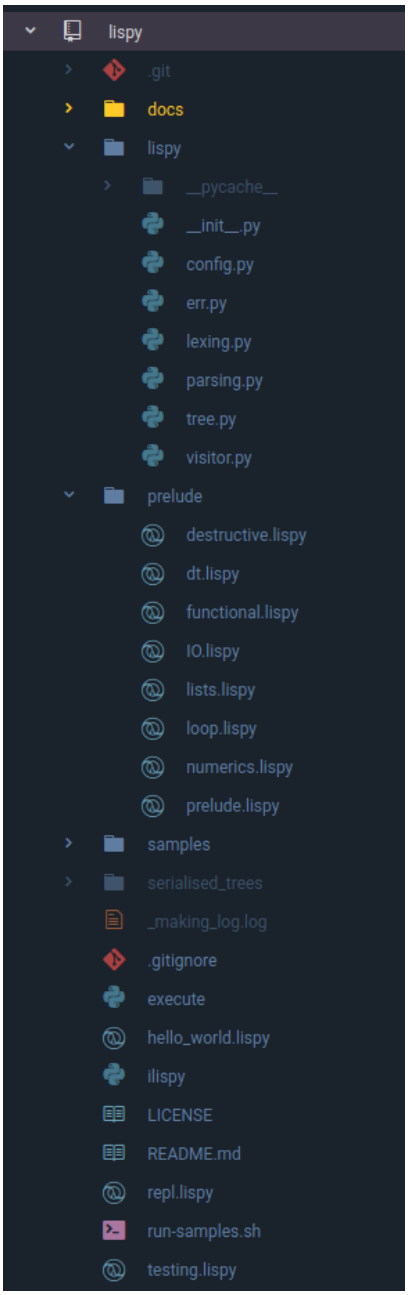


Figure 4: Repo. file structure.

The `lispy` folder contains everything from the lexer, to the evaluator, with some extra files for tweaking it's behaviour and tackling errors. The `__init__.py` file ties all the files in the folder together into a neatly wrapped Python module, such that externally, in another Python file (say the `./execute` or the `./ilispy` file) you can simply do `import lispy` and access all the language implementation from there.

```python
#!/usr/bin/env python3

import lispy
import sys

def argument_error(msg):
    print('Argument Error:')
    print('    >>>  ' + '\n\t'.join(msg.split('\n')))

if len(sys.argv) > 1:
    files = filter(lambda e: e[-6:] == '.lispy', sys.argv)
    files = list(files)
    if len(files) >= 1:
        for file in files:
            lispy.run(file)
    else:
        argument_error(
        'At least one filename needs to be supplied to'
        + '\nthe LISPY interpreter. Filename needs to end in `.lispy\'.')
else:
    argument_error('Please supply at least 1 argument to the interpreter.')
```

Figure 5: The `./execute` file, is the main file that runs Lispy programs.

The main file responsible for executing code found in actual `.lispy` files is the `execute` file, and takes in arguments from the command line, which needs to be one or more Lispy program files. It does this by accessing the ARGV variable, containing the passed arguments, similar to how you pass `argc` and `argv` to the `main` function in C. e.g.

```c
int main(int argc, char **argv)
{
   ...
}
```

```python
#!/usr/bin/env python3

# ═══ Interactive LISPY (ilispy) ═══ #
#|  This file, will run the REPL file interactively.
#|  That means it won't exit on errors, and
#|  gives a shell like feel, with history and such.

import lispy
lispy.conf.EXIT_ON_ERROR = False

import signal
import sys


def _(sig, frame):
    lispy.visitor.EX.throw(lispy.visitor.CURRENT_LOCATION,
        "Halted execution, due to KILL SIGINT!")
    sys.exit(0)

signal.signal(signal.SIGINT, _)

try:
    lispy.run('./repl.lispy')
except EOFError:
    print("\n\nCtrl+D --- EOF")
    print("Bye-bye.")
```

Figure 6: Allows us to run a REPL, in a way such that it will recover from any errors, instead of just exiting.

The `./ilspy` file just provides us with another way of writing Lispy code. Instead of reading files and executing them, this allows us to write our code interactively, executing and evaluating the code on the fly, statement by statement, always observing the return value for each line you write. This is excellent for quickly getting used to the language and debugging code. (This file also runs in the command line).

```
;; REPL Implementation for LISPY

;; A REPL is easy to implement in Lisps, in fact
;;   the very name REPL comes from something that
;;   can be very easily done in lisp:
;;     It stands for a READ-EVAL-PRINT-LOOP
;;
;; A REPL is implemented very naively as such:
;   (loop (print (eval (read))))

;                  vvvv - use `repr` to give an unescaped print-out.
(loop (print "=> " (repr (eval (read "[lispy]> "))) "\n"))
;           ^^ print return value.   ^^^^^^^^ prompt input, where you type.
;^^^^  ^^^^^             ^^^^  ^^^^
;(loop (print           (eval (read))))
```

Figure 7: This is the actual Lispy files that implements the REPL.

The behaviour of a REPL and it's implementation was already discussed in 2.4.4.

3.1 Lexerical Analysis & Tokenisation

The core concepts of lexical analysis have been outlined in 2.3.1, so we'll only be discussing it's implementation here.

3.1.1 Matching through Regular Expressions

The first thing we need to lay out in our lexer, is precisely what kind of tokens are allowed to exist (and will subsequently understood by the parser) and how we match them, i.e. how we identify them and collect them.

This can be done through the application of regular expression (again discussed in 2.3.1). The regular expressions we'll be using are outlined at the top of out `lexing.py` file:

```
# Alias the regex compiler
exp = re.compile

# Identifiers are matched as such:
#   (Atoms and Symbols are the only identifiers)
SYMS = r"_a-zA-Zα-ωA-Ω\+\-\=\<\>\*\/\%\^\&\:\$\£\#\~\`\|\\\¬\,\.\?\!\@"
IDENT_STR = r"[{syms}][0-9\'{syms}]*".format(syms=SYMS)
                                        # e.g.
L_PAREN    = exp(r"\A\(")               # '('
R_PAREN    = exp(r"\A\)")               # ')'
NIL        = exp(r"\Anil")              # 'nil'
SYMBOL     = exp(r"\A" + IDENT_STR)     # 'hello-world'
UNEVAL     = exp(r"\A\'")               # '
ATOM       = exp(r"\A\:+[0-9" + IDENT_STR[1:])  # ':good-bye'
NUMERIC    = exp(r"\A[0-9]+(\.[0-9]+)?([xob][0-9]+)?(e[\+\-]?)?[0-9a-fA-f]*")
TERMINATOR = exp(r"\A\n")
STRING     = exp(r"\A([\"'])((\\{2})*|(.*?[^\\](\\{2})*))\1")
# `Token` object is a chunk of the code we're interpreting;
#         it holds the type of thing it is as well as what
#         exactly the writer has written and at what line and
#         column it was written as
# i.e.  (a 3)     ⟹     <Token[L_PAREN] '(' (1:1)>,
#                        <Token[SYMBOL]  'a' (1:2)>,
#                        <Token[NUMERIC] '3' (1:4)>,
#                        <Token[R_PAREN] ')' (1:5)>
```

Figure 8: The set of regex matchable tokens. (`/lispy/lexing.py`)

Some of these really don't require regular expressions, and are in fact not matched using them, but I left them there so in order to outline what sort if tokens exist.

Most of the regular expressions are quite self explanatory, but I'd like to through a few of them. The first thing I want to bring your attention to are the "`\A`" at the beginning of each regex, these simply tell the regex compiler that we'll only like to be matching things from the *very beginning* of the string, as opposed to anywhere in the string or at the beginning of each line or something else.

You may notice that both atoms and symbols use the same 'identifier' string for matching, this makes sense as they are indeed both identifiers, and will both need to match against things such as letters, underscores, dashes and other special symbols (an extended word type) listed in the string. You can also see that numerals are only allowed in identifiers if an only if preceded by an extended word type. Atoms may also have a number anywhere in them, given that they start with a colon of course.

Numerics are matched with what may be a surprisingly long expression, and that is because it allows numerals with different bases: hexadecimal, octal and binary; and also allows for exponent notation, e.g.

`4e10`	$\iff$	4×10^{10}
`5.3e+22`	$\iff$	5.3×10^{22}
`345e-61`	$\iff$	345×10^{-61}
`0b00101`	$\iff$	101_2 or 5_{10}
`0×ff`	$\iff$	ff_{16} or 255_{10}
`0o771`	$\iff$	771_8 or 505_{10}

3.1.2 Tokens and Token Streams

```
# `Token` object is a chunk of the code we're interpreting;
#         it holds the type of thing it is as well as what
#         exactly the writer has written and at what line and
#         column it was written as
# i.e.  (a 3)    ⟹    <Token[L_PAREN] '(' (1:1)>,
#                      <Token[SYMBOL]  'a' (1:2)>,
#                      <Token[NUMERIC] '3' (1:4)>,
#                      <Token[R_PAREN] ')' (1:5)>
impl_loc = {'line':-1,'column':-1, 'filename':'IMPLICIT'}
class Token(object):
    def __init__(self, token_type, string, loc=impl_loc):
        self.type = token_type
        self.string = string
        self.location = loc
        self.location['span'] = len(string) + [0, 2][self.type == 'STRING']

    def __str__(self):
        return "<Token({}) '{}'>".format(
            self.type,
            self.string
        )

EOF_TOKEN = Token('EOF', EOF)
```

Figure 9: The quite simple token object. (in `/lispy/lexing.py`)

The token object takes in: A string identifying what kind of token it is; another string of what the actual value of the token is (i.e. what we matched); and a hash-map of the location that the token finds itself lexically (a line number and column number, the filename, as well as the span (the length of the matched string) that's computed by the

constructor function itself). The `__str__` method also provides a string representation of the token, which will be useful for debugging.

Example usage might look something like:

```
Token('NUMERIC', "310e-5", {
    'line': 12,
    'column': 4,
    'filename': '~/scripts/some-code.lispy'
})
```

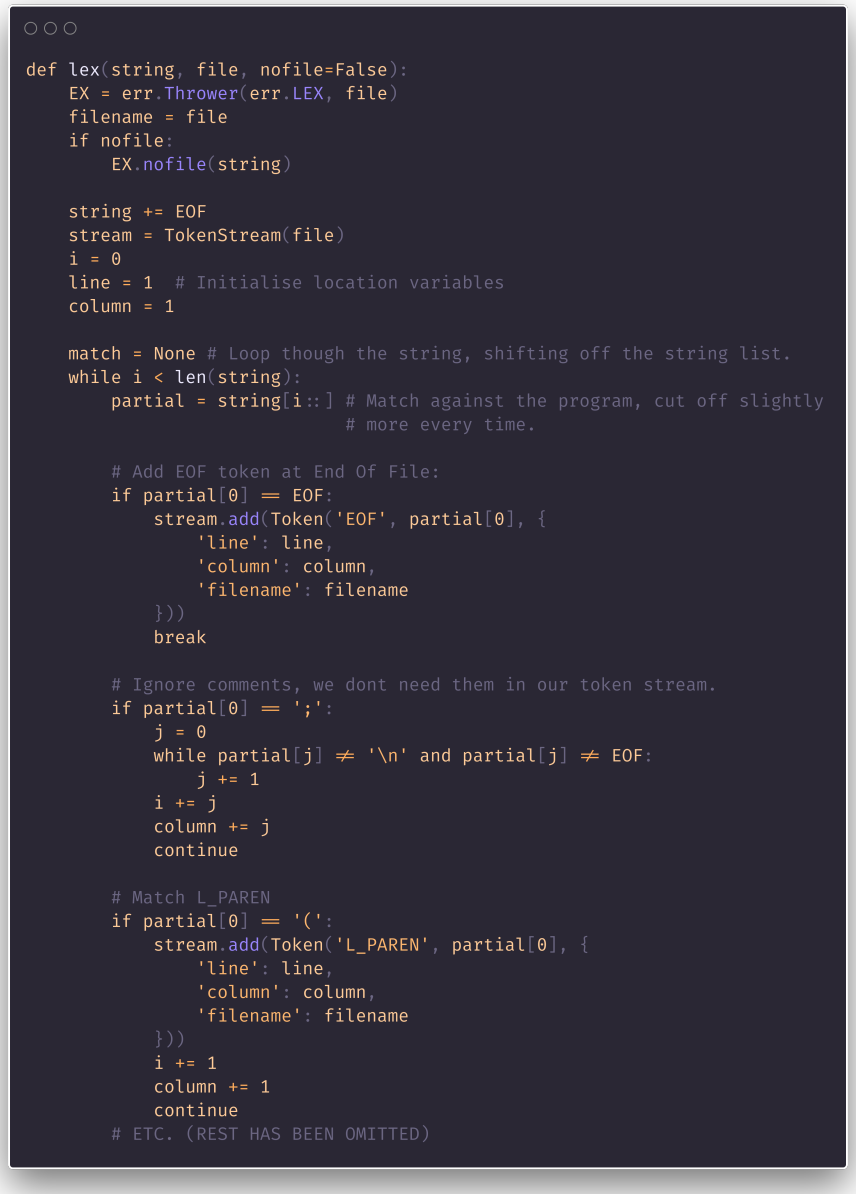
```python
# `TokenStream` is a wrapper for a list of tokens
#               to make it easier to manage what token we're
#               currently focused on.
class TokenStream(object):
    def __init__(self, file, tokens = None):
        self.file = file
        self.tokens = tokens or []
        self.i = 0

    # Simply returns the token that is at `self.i`,
    #   the streams current focused token.
    def current(self):
        if self.i >= self.size():
            return EOF_TOKEN
        return self.tokens[self.i]

    # Returns the amount of tokens in the stream.
    def size(self):
        # !! OMITTED !! #

    # Pushes on top of the token stream stack.
    def push(self, token):
        # !! OMITTED !! #
    add = push

    # Pops off the top of the token stream stack.
    def pop(self, j = -1):
        # !! OMITTED !! #

    # `next` will step forward in the token stream, and set
    #        the current token to the one after the current one.
    def next(self, j = 1):
        self.i += j
        if self.i >= self.size():
            return EOF_TOKEN
        return self.tokens[self.i]

    # `ahead` will peak ahead in the token stream, and return the token
    #         right after the current token.
    def ahead(self, j = 1):
        # !! OMITTED !! #

    # `back` takes a step back and sets the current token to the one before
now.
    def back(self, j = 1):
        # !! OMITTED !! #

    # `behind` simply returns the token before the current.
    def behind(self, j = 1):
        # !! OMITTED !! #

    # `purge` will remove all tokens of a certain kind.
    def purge(self, type):
        # !! OMITTED !! #

    def __str__(self):
        # !! OMITTED !! #
```

Figure 10: The token stream, essentially a specialised list. Some method bodies have been omitted for the sake of brevity.

The `TokenStream` object allows us to manage a vector of tokens, a bit like a Python iterator, by incrementing an internal instance variable (`self.i`) keeping track of where

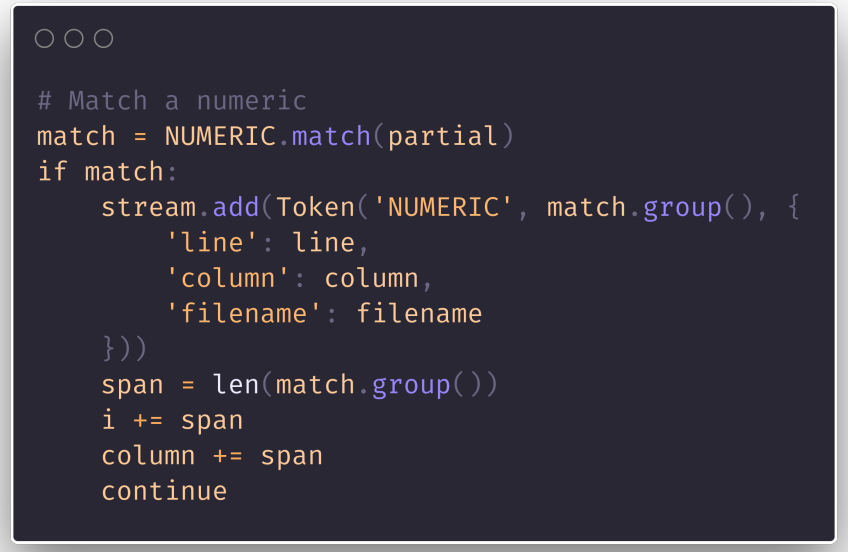
we are in the stream, and is controlled by methods such as `TokenStream#next` and `TokenStream#back` and we can get the current token through `TokenStream#current`.

The Lexer itself is implemented in the `lex` function, it takes a program string and returns a `TokenStream` type:



```
def lex(string, file, nofile=False):
    EX = err.Thrower(err.LEX, file)
    filename = file
    if nofile:
        EX.nofile(string)

    string += EOF
    stream = TokenStream(file)
    i = 0
    line = 1  # Initialise location variables
    column = 1

    match = None # Loop though the string, shifting off the string list.
    while i < len(string):
        partial = string[i::] # Match against the program, cut off slightly
                              # more every time.

        # Add EOF token at End Of File:
        if partial[0] == EOF:
            stream.add(Token('EOF', partial[0], {
                'line': line,
                'column': column,
                'filename': filename
            }))
            break

        # Ignore comments, we dont need them in our token stream.
        if partial[0] == ';':
            j = 0
            while partial[j] ≠ '\n' and partial[j] ≠ EOF:
                j += 1
            i += j
            column += j
            continue

        # Match L_PAREN
        if partial[0] == '(':
            stream.add(Token('L_PAREN', partial[0], {
                'line': line,
                'column': column,
                'filename': filename
            }))
            i += 1
            column += 1
            continue
        # ETC. (REST HAS BEEN OMITTED)
```

Figure 11: The `lex` method, some lexer rules have been omitted to fit in the picture.



```
# Match a numeric
match = NUMERIC.match(partial)
if match:
    stream.add(Token('NUMERIC', match.group(), {
        'line': line,
        'column': column,
        'filename': filename
    }))
    span = len(match.group())
    i += span
    column += span
    continue
```

Figure 12: Using the Regular Expressions to match a numeric in the Lex function.

Above, we can see we assign our match variable, to a potential match between a numeric, and the current stage our code is at. If that match happened to return a valid match, and not `None`, we'll create a new numeric token, with type `'NUMERIC'` and the matched string, which is returned by `match.group()`. Location of the token is given by the current `line` and `column` variables, `filename` is also passed into the dictionary. We need to advance the character pointer (`i`) by the size of the matched string, and increment the `column` variable by the same amount too.

3.1.3 Parentheses Balancer

```
# Simple stack-based algo for checking the amount of
#  opening parens to the amount of closing, and giving a
#  helpful error message
def paren_balancer(stream):
    stream = copy.copy(stream)  # Create a shallow copy of the stream
    stack = []                  # in order to dereference the original stream.
    balanced = True
    location = None

    while stream.current() ≠ EOF_TOKEN:
        location = stream.current().location
        if stream.current().type == 'L_PAREN':
            stack.append(0)
        elif stream.current().type == 'R_PAREN':
            if len(stack) == 0:
                balanced = False
                break
            else:
                stack.pop()

        stream.next()

    opens = 0
    close = 0
    for t in stream.tokens: # Keep track of opens vs. closes
        if t.type == 'L_PAREN': opens += 1
        if t.type == 'R_PAREN': close += 1

    # If the stack is empty, we've too many, otherwise too little closing
parens.
    message = ('Unbalanced amount of parentheses,\n'
        + 'consider removing {} of them...'.format(close - opens))
    if len(stack) ≠ 0:
        location = stream.tokens[-2].location
        message = 'Missing {} closing parentheses...'.format(len(stack))
    return {
        'balanced': balanced and len(stack) == 0,
        'location': location,
        'message': message
    }
```

Figure 13: In such a parenthesis heavy language as Lisp, a parentheses balancer can be very useful.

The paren-balancer essentially counts the amount of open parens, and tells you if you have too many or too few parentheses. If you have enough opens vs. closing parentheses, but have arranged them in an illogical manner (e.g. “)(”), it will end up with a negative open vs. close count, and tell you, you have an issue with how you’ve arranged your parentheses.

3.2 Parser

The parser for many Lisps is a relatively simple stage when compared to other programming languages. This fact is due to Lisp's use of parentheses and Polish notation, eliminating the need for some of the *harder* parts of parsing more natural languages, such as operator precedence and such.

Everything in Lisp is either atomic, i.e. a datum, or a list, which makes constructing the parse tree, even simpler.

The parser when I was writing the code was only about 42 lines of code for a long while, and was entirely functional, complete with error handling and everything. Now the parser is ca. 240 lines of code, and that was because at some point, I finally added macros to the language, adding a lot of search and replace complexity to the parsing stage when expanding macros.

The main components that constitute the parse-tree data structure and the token to parse-tree converter (i.e. 'the parser') are defined across two files: `/lispy/tree.py` and `/lispy/parsing.py`.

3.2.1 Abstract Syntax Tree

The syntax tree ('Abstract Syntax Tree' ⇒ 'AST'), is described in terms of its components/nodes/branches/leaves in the `/lisp/tree.py` file, containing a number of classes and abstract classes describing it. The classes have variables used to hold the node's value and node-type, but also methods such as `__str__` which also provide a visual representation of the tree and its nodes, which has actually been quite helpful through debugging.

The first class is `Tree` which is essentially a wrapper for a generic Python list, in fact, the class inherits the list type/class, so it inherits all your normal Python list functionality, with the addition of a new `__str__` method; a variable called `self.file` that keeps track of which file that AST comes from, and an alias for Python's `self.append` called `self.push`, just so I can keep a consistent naming of methods. The code for it looks like this:

```
class Tree(list):
    def __init__(self, file):
        self.file = file
        list.__init__(self)
    def push(self, *arg):
        return self.append(*arg)
    def __str__(self):
        rep = ''
        for e in self:
            rep += u'\u2503\n\u2523\u2501' + str(e) +'\n'
        return rep
```

Figure 14: Defined at the top of `/lispy/tree.py`

Most of the code here is trivial except from perhaps the the string representation method, which uses special Unicode characters (denoted by the `'\u####'` strings) to draw special box-drawing characters[1] which help draw the branches and trunk of the AST, in character represented form.

[1] `https://en.wikipedia.org/wiki/Box-drawing_character`

The next three classes / node definitions are three abstract classes, not meant to be used directly, but to be inherited from. The first one is actually a doubly abstract class, as the next two abstract classes inherit from that one. The first class is just called `Node`, and it is the mother-class of all subsequent possible AST elements, it just describes/outlines the basic behaviour of any node. It is described as such:

```
TAB = ' ' * 3
class Node(object):
    scope = None
    def __init__(self, value, loc):
        self.value = value
        self.type = self.__class__
        self.name = str(self.type).split('.')[-1][:-2]
    def __str__(self, depth=0):
        return "{}<AST::Node[{}] ({})>".format(
            '' if depth == 0 else TAB * 2,
            self.name,
            [str, repr][type(self.value) is str](self.value)
        )
```

Figure 15: All nodes have at least: a type; a value; and a name.

The `TAB = ' ' * 3` at the top just means that a tab will be the equivalent of three spaces, when representing the AST/Nodes.

We can see the node also implements a string representation method, using the same box-drawing characters to denote belonging of certain values.

The `self.value` variable will hold the value for the node, say for example `5` for a Numeric node, or `"Hello!"` for a String node. `self.type` is the classe's own type, which is in this case `Node`, and that type is Python given, and accessed though the inherent `self.__class__` variable that exist in every class. The next variable (`self.name`) is related, but provides a string representation of the type (in this case `'Node'`). This is done by calling `str( ... )` on the class-type variable, but, this will include super-classes, and namespaces, separated by dots, so the `str` method actually yields something like: `"<class 'lispy.tree.Node'>"`, so we just split the string by the dots, and pick the last element, and the ignore the two last characters (the "`'>`"), to give us `'Node'`.

The `Operator` class is the second abstract class and inherits from `Node`, it is probably the most complicated looking thing in the `/lispy/tree.py` file, but that's just because of its complicated string representation. The `Operator` class describes all nodes in the language that can have multiple children (i.e. function calls). The `Operator` class looks like this:

```
class Operator(Node):
    def __init__(self, value, loc, *operands):
        self.location = loc
        self.operands = list(operands)
        self.value = value
        self.type = self.__class__
        self.name = str(self.type).split('.')[-1][:-2]
        self.shorthand = None
    def __str__(self, depth=0):  # Don't even try to understand this.
        operands = '\n'.join(
            map(lambda e: (u'\u2503{}'.format(TAB * ([1, 2][depth>0] + depth) +
(e.__str__(depth + 1) if issubclass(type(e), Node) else str(e)))), self.operands)
        )
        caller_lines = self.value.__str__().split('\n')
        caller = '\n{}'.format(TAB*(depth+2) + ' ').join(caller_lines)
        return u"{ind}
<AST::Node[{}]\n\u2503{ind1}\u2523\u2501caller\n\u2503{ind1}\u2503
\u2517\u2501{}\n\u2503{ind1}\u2517\u2501operands=[".format(
            self.name,
            caller,
            ind = '' if depth == 0 else TAB * 2,
            ind1 = TAB * (depth+2) if depth == 0 else TAB * (depth + 3)
        ) + (']>' if len(self.operands) == 0 else ('\n' + operands + u'\n\u2503
{}]>'.format(TAB * (depth + [2,3][depth>0]))))
```

Figure 16: Inherits from `Node`.

`Operator` still has the normal `Node` variables, but with a few extra. First of all, the `self.value` now represents the head of the list, thats to say, the callee of the function (the operator itself) if this were a function call (i.e. it's not unevaluated). The operands to the call, (i.e. the tail of the list) is, well, `self.operands`. `self.shorthand` is set to `None`, and basically indicates whether this call is a shorthand-lambda, that's to say it's arguments are implicit and listed by numerals, more on this concept later. Finally, we have a new `self.location` variable which just hold information on where this node corresponds to in the file (line number, span, column, &c.), which will be very helpful if we need to throw errors.

The `__str__` method was put together very quickly, as I didn't want to spend too long working on just the small step of the parser, that is trying to represent the AST as a string, it works by, again, using a number of box-drawing characters and calling the string methods on all its children, and creating something resembling a tree. If I had more time to implement the visualisation this is certainly code that I would improve/re-write. It made sense when I wrote it, not so much anymore, more than anything its messy.

Finally, onto our last abstract class, the `Data` class:

```
class Data(Node):
    def __init__(self, value, loc):
        self.location = loc
        self.value = value
        self.type = self.__class__
        self.name = str(self.type).split('.')[-1][:-2]
```

Figure 17: Abstract node class for all atomic datatypes (any datum)

`Data` is almost no different from `Node`, except now it's also got a location variable. Let's now look at the child classes that use these two abstact classes:

```
class Yield(Data):
    pass

class Call(Operator):
    pass

class Symbol(Data):
    pass

class Atom(Data):
    pass

class Numeric(Data):
    pass
 class Uneval(Data):
    def __hash__(self):
        return hash(self.value)

class String(Data):
    def __hash__(self):
        return hash(self.value)
```

Figure 18: All child nodes all either inherit from `Operator` or `Data`.

The only class that inherits from `Operator` is `Call`, this could change in future versions of the language (?), but really, this is the only one we need in a relatively standard Lisp-style language. We've implemented the nodes for the basic LISPY datatypes (`Numeric`, `Symbol`, `Atom`, `String`, and the slightly more special, `Uneval`, `Yield`). However, we're missing nil, and since `Nil` doesn't really hold any value other than just being nil (nil just represents the absence of a value), it is not inherited from `Data`, nor `Operator`. Nil is just defined as:

```
class Nil(Node):
    def __init__(self, loc):
        self.location = loc
        self.value = 'nil'
        self.type = self.__class__
        self.name = str(self.type).split('.')[-1][:-2]
    def __hash__(self):
        return hash(self.value)
```

Figure 19: `Nil` type AST node.

That concludes the `/lispy/tree.py` file, these components will be vitally important in not only parsing, but when visiting the AST for evaluation of the tree.

In the `/lispy/tree.py` we will actually be assembling the AST together

3.2.2 Bottom-up / Shift-reduce Pattern

Our parser will be implemented as a bottom-up parser, generating the leaves of the AST and working our way inwards towards the roots. This has been discussed more in depth in section 2.3.2.

We need to import all our older files, in order to use the functionality we've so far implemented, and define some useful variables, like what an EOF is and the error handler, which I'll go through later and a simple numerical parser, which converts strings to actual Python integers. These are all just meta-utilities:

```
from . import lexing
from . import tree

from . import err
from . import config as conf

import sys, copy, ast, re

EOF = '\0'

EX = None

def numeric(string, location):
    try:
        return ast.literal_eval(string)
    except:
        return EX.throw(location,
            'Could not parse `{}\' as a numeric, malformed
literal.'.format(string))
```

Figure 20: Header of the `/lispy/parsing.py` file.

Let's look at the bottom of the file, where the parser is actually defined:

```
def parse(stream, string=None):
    global EX
    AST = None
    if EX is None :
        EX = err.Thrower(err.PARSE, stream.file)
        if string is not None:
            EX.nofile(string)
    if AST is None:
        AST = tree.Tree(stream.file)
        stream.purge('TERMINATOR')

    def parse_loop(AST):
        if conf.DEBUG: print("TOP LEVEL PARSE: ", stream.current())
        branch = atom(stream.current(), stream)
        if branch is not -1:
            if conf.DEBUG: print('Adding branch: ', branch.type)
            AST.push(branch)
        if stream.ahead().type == 'EOF':
            return AST

        stream.next()
        return parse_loop(AST)

    AST = parse_loop(AST)
    if conf.DEBUG: print("Size of AST:", sys.getsizeof(AST))
    return AST
```

Figure 21: The simple parser loop.

The `parse` function takes in a stream and starts out by defining the `EX` variable for throwing exceptions in case of bad code, and purges useless tokens from the stream. If we have no AST variable defined, `parse` will define it with an empty tree.

We define the `parse_loop` function, which steps through the stream, using `stream.next()` and at each time it encounters a parsable token, it calls `atom` on it. If it has reached an EOF token, it will know its reached the end of the file. The function calls itself recursively to advance to the next token. The function ultimately returns the fully-grown AST.

The `atom` function is where all the shift-reduce functionality of the parsing actually lies. No, the name has nothing to do with the datatype in our language called atoms, it simply means it parses atomic data, however this is now untrue, after I added the `Call` type parsing capability to it, I just never came up with another name. It is defined as such:

```
SHORTHAND = None
def atom(token, stream):
    global SHORTHAND
    if conf.DEBUG: print('Atomic token type: ', token.type)
    loc = token.location

    if token.type == 'L_PAREN':
        SHORTHAND = 0
        caller = None
        if stream.ahead().type ≠ 'R_PAREN':
            caller = atom(stream.next(), stream)
        else:
            stream.next() # Go past the R_PAREN, so outer calls don't get closed
            return tree.Call(caller, loc)
        operands = []
        while stream.ahead().type ≠ 'R_PAREN':
            if stream.current().type == 'EOF':
                return EX.throw(
                    stream.current().location,
                    'Unexcpected EOF, missing closing parenthesis')
            operands.append(atom(stream.next(), stream))
        stream.next()  # Skip the R_PAREN we just spotted ahead.
        if caller.value == 'yield':
            if len(operands) == 0:
                operands.append(tree.Nil(loc))
            return tree.Yield(operands[0], loc)
        call = tree.Call(caller, loc, *operands)
        if (caller.type is tree.Symbol
        and caller.value == '→'):
            call.shorthand = SHORTHAND
            SHORTHAND = None
        return call
    if token.type == 'NUMERIC':
        return tree.Numeric(numeric(token.string, loc), loc)
    if token.type == 'SYMBOL':
        if SHORTHAND is not None and re.match(r"\%[1-9]+", token.string):
            SHORTHAND += 1
        return tree.Symbol(token.string, loc)
    if token.type == 'ATOM':
        return tree.Atom(token.string, loc)
    if token.type == 'UNEVAL':
        return tree.Uneval(atom(stream.next(), stream), loc)
    if token.type == 'STRING':
        return tree.String(token.string, loc)
    if token.type == 'NIL':
        return tree.Nil(loc)

    return -1
```

Figure 22: Parses atomic data and function calls, line 190 in `/lispy/parsing.py`.

The `atom` function is relatively simple, if it finds a number, it will return a Numeric Node, and so on for all the other types, if it finds an `L_PAREN` token, it will search for the corresponding `R_PAREN` token. While going looking for the `R_PAREN` it will recursively call `atom` on all the tokens in between, pushing them all to an array. When it finds the `R_PAREN`, it will create a `Call` node and add all its children to it, finally returning the

`Call` node.

3.2.3 Preprocessing & Macro Expansion Phase

This phase was actually added *after* I had finished the interpreter, and a lot of the Prelude library. It is responsible for simply finding macro definitions (macro definitions are very similar to C's `#define` preprocessor directive) and searches for calls with the same symbol as the definition. The code for this is as follows:

```
def preprocess(AST, macros={}):
    global MACROS
    MACROS = macros  # Clear the known macros, such that
                     #   any subsequent instances of the parser
                     #   don't use old macros, irrelevent to them
                     #   (and to avoid name clashes ... )

    for i in range(len(AST)):
        AST[i] = macro_expansion(AST, i)
    return AST
```

Figure 23: The preprocessor function, it tries to expand any macros found in each branch of the syntax tree.

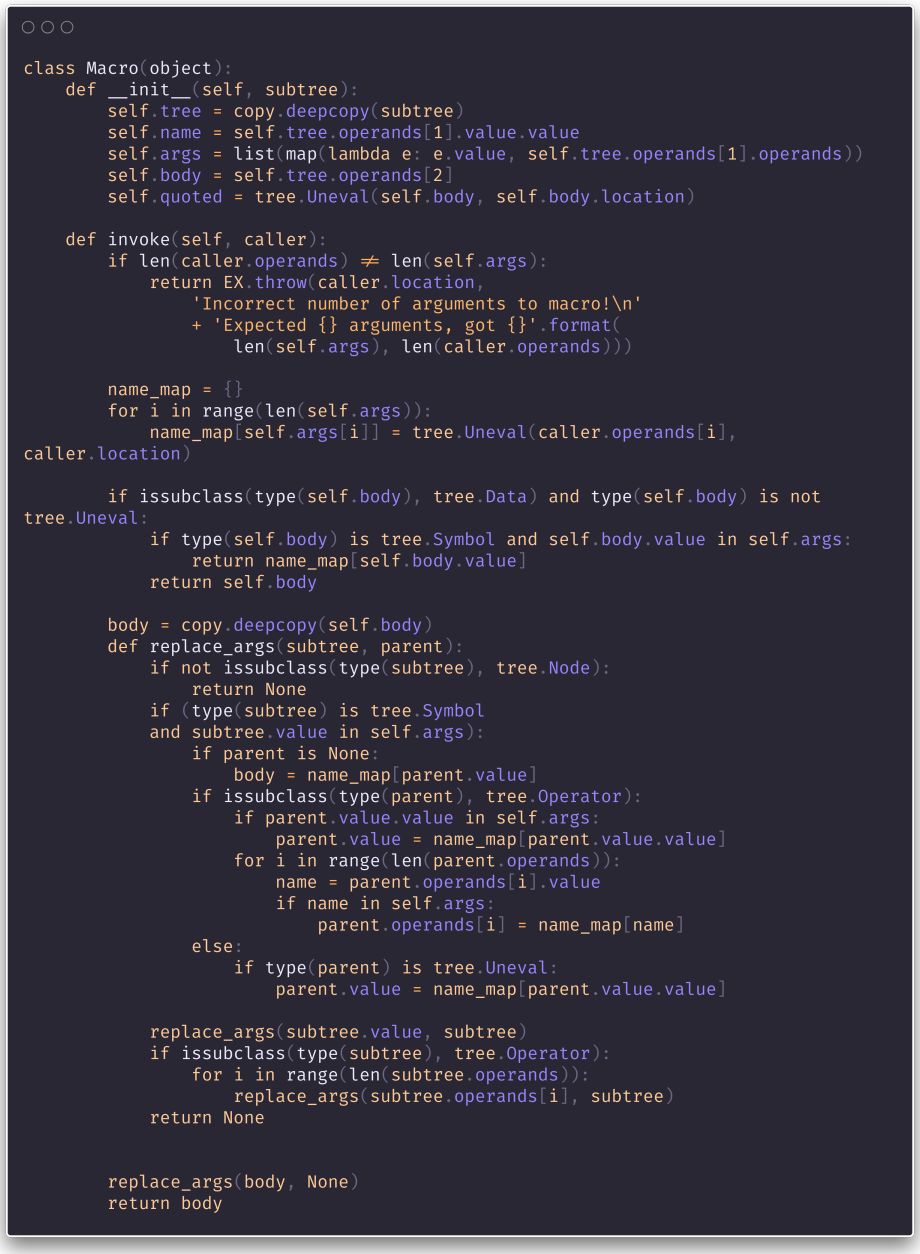
```
def macro_expansion(ast, i):
    def search_brach(subtree, parent=None):
        t = type(subtree)

        if issubclass(t, tree.Operator):
            found = False
            operands = subtree.operands
            if (issubclass(type(subtree.value), tree.Node)
            and subtree.value.value == 'define'
            and len(subtree.operands) > 0
            and type(subtree.operands[0]) is tree.Symbol
            and subtree.operands[0].value == 'macro'):
                caller = subtree.operands[1]
                found = type(caller.value.value) is str
                operands = subtree.operands[2:]
            else:
                if type(subtree.value) is tree.Symbol:
                    if subtree.value.value in MACROS:
                        replacement = MACROS[subtree.value.value].invoke(subtree)
                        if parent is None:
                            ast[i] = replacement
                        else:
                            if (type(parent.value.value) is tree.Symbol
                            and parent.value == subtree):
                                parent.value = replacement
                            else:
                                for j in range(len(parent.operands)):
                                    if parent.operands[j] == subtree:
                                        parent.operands[j] = replacement
                                        break

            search_brach(subtree.value, parent=subtree)
            for operand in operands:
                search_brach(operand, parent=subtree)

            if found:
                MACROS[caller.value.value] = Macro(subtree)
                empty_branch = tree.Nil(subtree.location)
                if parent is None:
                    ast[i] = empty_branch
                else:
                    if parent.value == subtree:
                        parent.value = empty_branch
                    else:
                        for j in range(len(parent.operands)):
                            if parent.operands[j] == subtree:
                                parent.operands[j] = empty_branch
                                break
            return None

        if issubclass(t, tree.Data):
            if subtree.value in MACROS and type(subtree) is tree.Symbol:
                if parent is None:
                    ast[i] = MACROS[subtree.value].quoted
                else:
                    if issubclass(type(parent), tree.Operator):
                        if (type(parent.value) is tree.Symbol
                        and parent.value.value == subtree.value):
                            replacement = MACROS[subtree.value].invoke(parent)
                            if issubclass(type(replacement), tree.Operator):
                                parent.value = replacement.value
                                parent.operands = replacement.operands
                            else:
                                parent = replacement
                        else:
                            for j in range(len(parent.operands)):
                                if parent.operands[j].value == subtree.value:
                                    parent.operands[j] =
MACROS[subtree.value].quoted
                else:
                    search_brach(subtree.value, parent=subtree)
            return None

    search_brach(ast[i])
    return ast[i]
```

Figure 24: The macro expansion function defines a way of searching the tree and replacing any invocation of a macro with a proper expansion

The method seems pretty heavy, but really all it is, is a search and replace. When a macro is being called upon (e.g. `(my-macro 3 4)`), we look through our macro dictionary

(`MACROS`) and if we find that it is indeed a macro, we replace it with an invocation of that macro (e.g. `MACRO['my-macro'].invoke(3, 4)`, you can see this happening on line 134 of `parsing.py`). Of course, when we have a macro called `my-macro`, that takes two arguments, say `x` and `y`, we need to, again, search and replace all instances of `x` and `y` with their invoker's parameters, (3 and 4 respectively in our last example).

So far we've been talking about the macros in the `MACROS` dictionary as some unknown object with methods such as `.invoke( ... )` implemented in them and all sorts. Let's actually have a look at the definition of that `Macro` object, at the top of our `parsing.py` file:



```
class Macro(object):
    def __init__(self, subtree):
        self.tree = copy.deepcopy(subtree)
        self.name = self.tree.operands[1].value.value
        self.args = list(map(lambda e: e.value, self.tree.operands[1].operands))
        self.body = self.tree.operands[2]
        self.quoted = tree.Uneval(self.body, self.body.location)

    def invoke(self, caller):
        if len(caller.operands) =/= len(self.args):
            return EX.throw(caller.location,
                'Incorrect number of arguments to macro!\n'
                + 'Expected {} arguments, got {}'.format(
                    len(self.args), len(caller.operands)))

        name_map = {}
        for i in range(len(self.args)):
            name_map[self.args[i]] = tree.Uneval(caller.operands[i],
caller.location)

        if issubclass(type(self.body), tree.Data) and type(self.body) is not
tree.Uneval:
            if type(self.body) is tree.Symbol and self.body.value in self.args:
                return name_map[self.body.value]
            return self.body

        body = copy.deepcopy(self.body)
        def replace_args(subtree, parent):
            if not issubclass(type(subtree), tree.Node):
                return None
            if (type(subtree) is tree.Symbol
            and subtree.value in self.args):
                if parent is None:
                    body = name_map[parent.value]
                if issubclass(type(parent), tree.Operator):
                    if parent.value.value in self.args:
                        parent.value = name_map[parent.value.value]
                    for i in range(len(parent.operands)):
                        name = parent.operands[i].value
                        if name in self.args:
                            parent.operands[i] = name_map[name]
                else:
                    if type(parent) is tree.Uneval:
                        parent.value = name_map[parent.value.value]

            replace_args(subtree.value, subtree)
            if issubclass(type(subtree), tree.Operator):
                for i in range(len(subtree.operands)):
                    replace_args(subtree.operands[i], subtree)
            return None


        replace_args(body, None)
        return body
```

Figure 25: The macro object class and it's methods.

The `Macro` object has a few important instance variables, and an invoke method, which we discussed earlier. How the invoke function works, is it takes in the parent call object of the actual invocation, checks through the argument list of the call, and will use

those arguments to replace their corresponding usage in the body of the macro. First of all we will check the length of the argument list of the parent call, and check the length of its internal list of arguments, and check if they have the same length. If they do have the same length, we create a name map of the arguments, and have them map to their corresponding invoker arguments, so that we can then go through the macro body and replace those symbols with the correct expressions we passed in.

An example of this could be a macro define as such:

```
(define macro (sum-and-add-one a b)
  (+ a b 1))

(print (sum-and-add-one (+ 3 1) 5))
```

after expansion, this code would be directly equivalent to:

```
(print (+ (+ 3 1) 5 1))
```

Hopefully this shows how macros are really just text replacement.

3.3 Interpreter/Evaluator

Finally onto the interpreter, the thing that will actually evaluate and reduce our code into our desired results.

There are many options I could have gone with here, first of all, you might wonder, why an interpreter and not a compiler for example? Well, the first reason for that is that my Lisp dialect is dynamically typed, making it hard to compile, as usually compiled languages do all their type checking at compile time as opposed to at runtime, with dynamically typed languages. Unless we secretly implement static typing through use of type inference, but that would still limit our language in other ways.

The second reason for making it not compiled, is that I would have spend a lot of time, reducing the AST into a stream of CPU-architecture specific machine-code, which would not only be a large task, but would include me having to learn x86-64 assembly language, and even then, it would only work on *some* CPUs.

Hence, I chose to go interpreted, in which case I had (in general) one of two options, an AST traversing interpreter or a bytecode assembler with a corresponding virtual machine.

My interpreter ended up being a single pass AST visitor, which executes as it walks down the tree. This is slower then compiling for a virtual machine, but having to build a virtual machine, a load of instructions and reducing the AST to the VM specific bytecode instructions would be a much larger task, and likely not something I could complete in time for this NEA.

3.3.1 Recursive Decent Evaluation

The phrase 'recursive decent' is usually used to describe a certain type of top-down parser, but I feel it also describes, and works in an analogous fashion to my interpreter.

The interpreter is defined as a set of mutually recursive functions, all starting from a little loop which calls the evaluate function on each branch of the AST, which itself calls a load of other functions which help evaluate certain things (mostly these are the internal macros) which then *again* call the evaluate function, evaluating other sub-expressions. This results in the evaluator evaluating the innermost expressions first before it can fully evaluate the parent-expressions.

3.3.2 Visiting & Walking the Tree

First let's have a look at the entry-point functions of the interpreter:

```
# All evaluation starts here:
def visit(AST, pc=0, string=None):
    AST = parsing.preprocess(AST)
    global LAST_RETURNED, LAST_EVALUATED
    if conf.DEBUG: print("\nVisiting (`{}\' branch: {}):\n".format(AST.file, pc),
AST, sep="")
    global EX
    if string is not None:
        EX.nofile(string)
    ret = tree.Nil({'line': 1, 'column': 1, 'filename': AST.file})
    if len(AST) == 0:
        return ret
    try:
        ret = evaluate(AST[pc])
        LAST_RETURNED = ret
        LAST_EVALUATED = LAST_RETURNED
    except RecursionError:
        return EX.throw(CURRENT_LOCATION,
            'Recursion level too deep!\n'
            + 'You might have an infinite loop somewhere,\n'
            + 'or your recursing over something too many times.\n\n'
            + 'python      call-stack depth:  {},\n'.format(conf.RECURSION_LIMIT)
            + 'interpreter call-stack depth:  {}.'  .format(len(CALL_STACK)))
    if conf.RECOVERING_FROM_ERROR:
        conf.RECOVERING_FROM_ERROR = False
    if pc + 1 >= len(AST):
        return ret
    return visit(AST, pc + 1, string)
```

Figure 26: The `visit` function, goes through each branch and evaluates each of them. Defined in `/lispy/visitor.py` on line `1126`.

The first thing we see, is that the AST has any potential macros expanded before we

do anything with it. We make sure all our global variables are set up for the execution process, and then we evaluate the branch of the AST found at index location '`pc`'. The `evaluate` call is wrapped in `try` statement which is set up to catch any `RecursionError` that may be thrown, due to the maximum recursion depth being exceed (LISPY doesn't have any tail-call optimisation and every recursive function call in LISPY is equivalent to a recursive call in Python, both of which are unfortunate scenarios and something I'd fix in future iterations of the language).

```
def walk(AST):
    global EX, LAST_RETURNED, LAST_EVALUATED
    EX = err.Thrower(err.EXEC, AST.file)
    if not symbol_declared(CURRENT_SCOPES, '$PRELUDE_LOADING'):
        main_table = lookup_table(0x0)
        main_table.bind('$PRELUDE_LOADING', ATOMS[':true'])

        here = os.path.dirname(os.path.abspath(__file__))
        load_file(here + '/../prelude/prelude.lispy')

        main_table.bind('$PRELUDE_LOADED', ATOMS[':true'])

        if conf.DEBUG: print('\n\nAUTOMATICALLY LOADED PRELUDE\n\n')

    return visit(AST)
```

Figure 27: The `walk` method calls `visit`, while also doing some extra work, such as loading in the prelude library for the first time

The `evaluate` method is at the centre of all execution in the interpreter.

```
def evaluate(node):
    global TABLES, CURRENT_SCOPES, FROZEN_TABLES, CALL_STACK
    global EX, CURRENT_LOCATION, LAST_EVALUATED, LAST_RETURNED, ATOMS
    # All tables and scope/call stacks need to be
    # able to be modified by this method.

    if conf.RECOVERING_FROM_ERROR:
        return err.NIL_ERROR

    if not is_node(node):
        LAST_EVALUATED = node
        return LAST_EVALUATED
                    # Anything that isn't an AST node will simply
                    #   be retuned as itself, as that means it's already
                    #   an evaluated fundemental datatype attempting to be
                    #   evaluated with no reason. e.g. (eval 2)...

    CURRENT_LOCATION = node.location

    if node.type is tree.Yield:
        LAST_EVALUATED = node
        return LAST_EVALUATED  # Doesn't get evaluated per se.
    if node.type is tree.Nil:
        LAST_EVALUATED = node
        return LAST_EVALUATED  # Doesn't get evaluated per se.
    if node.type is tree.Atom:
        if node.value in ATOMS:
            LAST_EVALUATED = ATOMS[node.value]
            return LAST_EVALUATED
        ATOMS[node.value] = Atomise(node.value)
        LAST_EVALUATED = ATOMS[node.value]
        return LAST_EVALUATED
    if node.type is tree.Numeric:
        LAST_EVALUATED = node.value
        return LAST_EVALUATED
    if node.type is tree.Symbol:
        if node.value == '_':
            return LAST_RETURNED
        if node.value in ['break', 'next']:
            return node
        if node.value in MACROS:
            LAST_EVALUATED = MACROS[node.value]
            return LAST_EVALUATED
        LAST_EVALUATED = search_tables(CURRENT_SCOPES, node.value)
        return LAST_EVALUATED
    if node.type is tree.Uneval:
        LAST_EVALUATED = node
        return LAST_EVALUATED  # Doesn't get evaluated per se.
    if node.type is tree.String:
        LAST_EVALUATED = node.value
        return LAST_EVALUATED
    if node.type is tree.Call:
        if node.value is None:
            return EX.throw(node.location,
                'Cannot make empty call. Evaluating an\n'+
                'empty list does not make sense.')
        LAST_EVALUATED = execute_method(node)
        return LAST_EVALUATED

    raise Exception("Don't know what to do with %s, this is a bug" % str(node))
```

Figure 28: The `evaluate` method takes any AST node / subtree and evaluates it, or, in the case of a fully evaluated atomic datum being passed in, it will simply return it back again.

The method simply takes in a node, and runs through a plethora of if-statements and returns an evaluation depending on what type of data you passed in, often calling itself recursively.

When evaluating a symbol, it first looks through the dictionary of known internal macros, if it isn't an internal macro, it will look through the known symbol tables of the current scopes and return the value bound to it in the table it was found in. A similar thing happens when it tries to evaluate an atom, if the atom already exists, it'll return the uniquely hashed `Atomise` object, otherwise, it'll add a new one.

If the evaluator encounters a `Call` type object, it will call the '`execute_method`' method:

```
def execute_method(node, args=None):
    global LAST_EVALUATED, LAST_RETURNED, FROZEN_TABLES, CALL_STACK
    definition = node
    if not isinstance(node, (Definition, function)):
        definition = evaluate(node.value)

    if type(definition) is function:
        return definition(node)

    if type(definition) is not Definition:
        loc = None
        if is_node(definition):
            loc = node.value.location
        else:
            loc = CURRENT_LOCATION

        return EX.throw(loc,
            'Cannot make call to to type of `{}\''.format(
                to_type(definition)))

    if definition is None:
        return EX.throw(CURRENT_LOCATION,
            'Cannot make empty call.')

    if args is None:
        args = list(map(evaluate, node.operands))
    else:
        print("Artificial args")
    FROZEN_TABLES = definition.frozen

    added = definition.table.scope == CURRENT_SCOPES[-1]
    if not added:
        CURRENT_SCOPES.append(definition.table.scope)

    call_table =  clone(definition.table)
    call_table.clean()
    call_table.give_args(args)
    CALL_STACK.append(call_table)

    # Aaaaand, then we finally, make the call ...
    result = definition.call()
    # Back to cleaning up our Symbol tables.

    definition.table.clean()
    FROZEN_TABLES = []
    if not added:
        CURRENT_SCOPES.pop()
    CALL_STACK[-1].clean()
    CALL_STACK.pop()
    del call_table

    LAST_EVALUATED = result
    LAST_RETURNED = LAST_EVALUATED
    return LAST_RETURNED
```

Figure 29: `execute_method` is responsible for managing the symbol tables of a method call and the call stack, before and after evaluation.

This method has many edge cases to consider, like the method called being an internal macro for example, but ultimately it is meant to manage the calls symbol tables before

and after calling. First it evaluates the caller, if the caller evaluated to a `Definition` object, that means it is a valid function call (a call to, say, the number three `(3)`, would throw an error, as we cannot call a number). The `Definition` object has the methods corresponding symbol table, which will be pushed onto the call stack, so that any local variables and arguments bound inside the function scope can be accessed. Once this is done we call the `Definition` object's `call` method, which clones the function body and calls, once again, the `evaluate` method on it. Once this is finished, we pop off the call stack, and any frozen super-scope tables we might have added, and return the result of the evaluation, i.e. the result of `definition.call()`.

So far we've mentioned a number of objects and classes needed for our above, code. Namely, the `SymbolTable`, `Definition` and `Atomise` class.

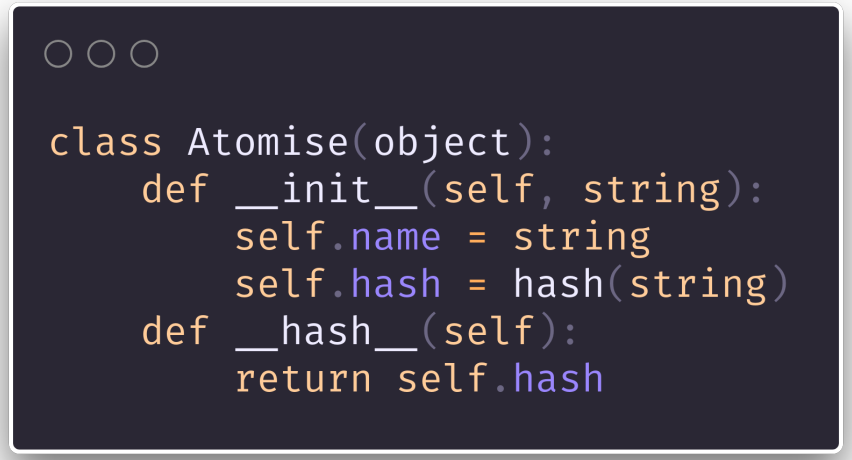
```
class Definition(object):
    def __init__(self, branch, table, taking, frozen):
        self.tree = recursive_clone(branch)
        self.frozen = frozen
        self.table = table
        self.args = taking
    def call(self):
        result = evaluate(recursive_clone(self.tree))
        self.table.clean()
        return result
```

Figure 30: The `Definition` class.

The class responsible for packaging method definitions must hold:

- The method's parameter list.
- The method body.
- A refrence to the symbol table

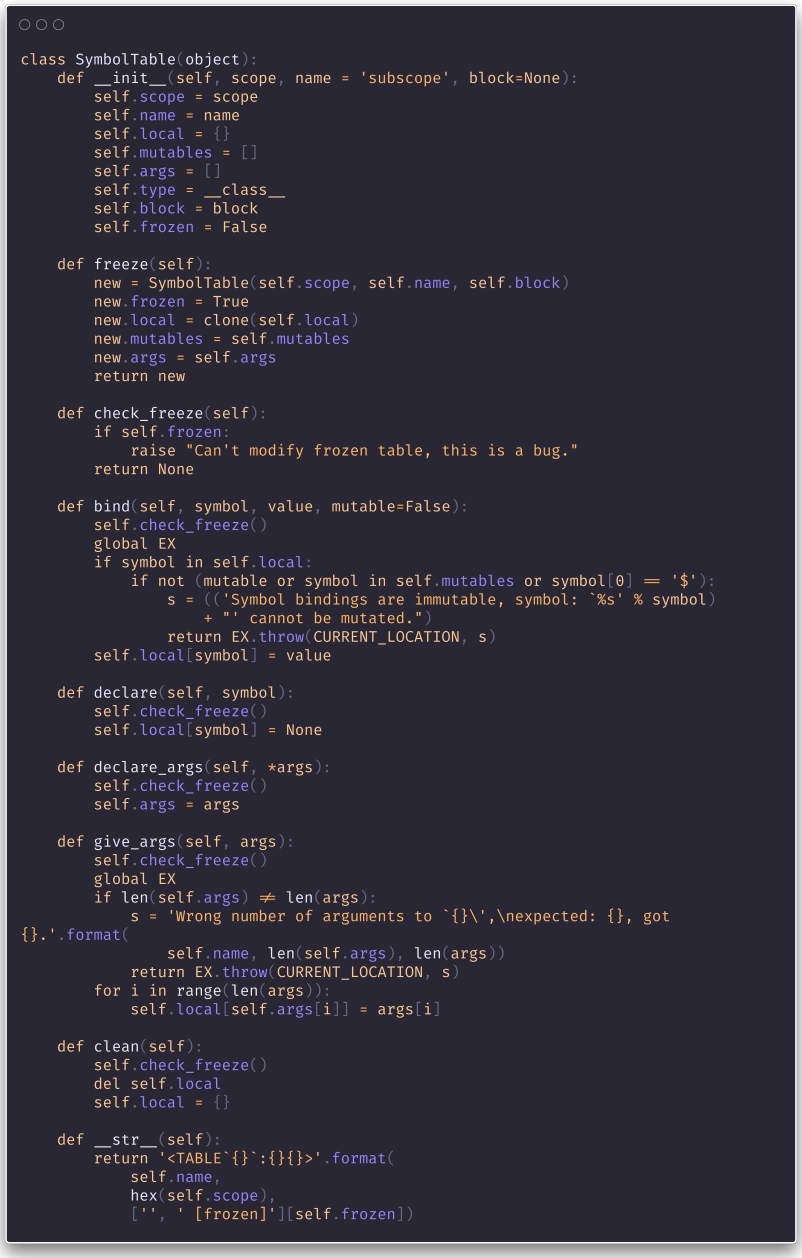
The class also has a method, which clones the method body and evaluates it.



```
class Atomise(object):
    def __init__(self, string):
        self.name = string
        self.hash = hash(string)
    def __hash__(self):
        return self.hash
```

Figure 31: `Atomise` class.

The `Atomise` class has little functionality, it is simply a wrapper. It ensures that all atoms in LISPY have the same memory reference, and importantly the same hash.



```
class SymbolTable(object):
    def __init__(self, scope, name = 'subscope', block=None):
        self.scope = scope
        self.name = name
        self.local = {}
        self.mutables = []
        self.args = []
        self.type = __class__
        self.block = block
        self.frozen = False

    def freeze(self):
        new = SymbolTable(self.scope, self.name, self.block)
        new.frozen = True
        new.local = clone(self.local)
        new.mutables = self.mutables
        new.args = self.args
        return new

    def check_freeze(self):
        if self.frozen:
            raise "Can't modify frozen table, this is a bug."
        return None

    def bind(self, symbol, value, mutable=False):
        self.check_freeze()
        global EX
        if symbol in self.local:
            if not (mutable or symbol in self.mutables or symbol[0] == '$'):
                s = (('Symbol bindings are immutable, symbol: `%s' % symbol)
                    + "' cannot be mutated.")
                return EX.throw(CURRENT_LOCATION, s)
        self.local[symbol] = value

    def declare(self, symbol):
        self.check_freeze()
        self.local[symbol] = None

    def declare_args(self, *args):
        self.check_freeze()
        self.args = args

    def give_args(self, args):
        self.check_freeze()
        global EX
        if len(self.args) =/= len(args):
            s = 'Wrong number of arguments to `{}\',\nexpected: {}, got
{}.'.format(
                self.name, len(self.args), len(args))
            return EX.throw(CURRENT_LOCATION, s)
        for i in range(len(args)):
            self.local[self.args[i]] = args[i]

    def clean(self):
        self.check_freeze()
        del self.local
        self.local = {}

    def __str__(self):
        return '<TABLE`{}`:{}{}>'.format(
            self.name,
            hex(self.scope),
            ['', ' [frozen]'][self.frozen])
```

Figure 32: The `SymbolTable` class, holds any local variables existing within a specific scope.

This class is one of the more complex ones, yet quite straight forward in its functionality. It holds a `scope` instance variable, which is an integer, which is a way to facilitate

looking up tables, and they are linked to the hash of the parent node the scope encompasses in the AST. It has a `name`, linked to the name of the function the scope is for; a `local` variable, which is a dictionary of local variables; `mutables` is a list of variable names which are allowed to be mutated; `args` is a list of which locals are exist as function arguments; `frozen` is a Boolean, signifying whether the table is allowed to be altered in any way.

Description of notable methods:

- `bind` — Sets a variable in the table, takes name and a corresponding value.
- `declare` — Tells the table that a certain variable exists, and will soon be bound.
- `declare_args` — Informs the table of what arguments its corresponding function will have.
- `give_args` — Gives the table the function arguments upon an invocation. Here it will also check for the correct number of arguments.
- `clean` — Will empty the table of variables, and get it restore it to how it was formaly, ready for second use.

3.3.3 Loading External Files / Require Statement

LISPY has a `(require :folder/name-of-file)` statement for importing, parsing, and executing files other than the current file being interpreted.

This requiring of other files is actually done secretly every time you run the interpreter, by requiring the prelude library, for all the basic language functionality. You can imagine this as a secret `(require :$LISPY_INSTALL/prelude/prelude)` happening at the top of the file your executing. This is similar to the `std` (standard) library in C or C++.

This require statement is implemented as:

```
def load_file(name):
    abspath = os.path.abspath(name)
    if abspath in LOADED_FILES:
        EX.warn(CURRENT_LOCATION,
            ('File at absolute location: `{}\'\n'
            + 'has already been loaded/required.\n'
            + 'This will almost certainly cause'
            + 'immutability errors ... ').format(abspath))
    LOADED_FILES.append(abspath)
    PROGRAM_STRING = None
    with open(name, 'r') as file:
        PROGRAM_STRING = file.read()
    stream = lexing.lex(PROGRAM_STRING, name)
    AST = parsing.parse(stream)
    visit(AST)
```

Figure 33: Loads external `.lispy` files.

which is then used by the `require` (`_require_macro`) internal macro:

```
def _require_macro(node):
    files = list(map(evaluate, node.operands))

    i = 0
    for f in files:
        if not name_node(f):
            t = str(type(f))
            if is_node(f):
                if f.type is tree.Uneval:
                    t = 'Unevaluation of ' + f.value.name
                else:
                    t = f.name
            else:
                if type(f) is int or type(f) is float:
                    t = 'Numeric'
            return EX.throw(node.operands[i].location,
                'Can\'t interpret type `{}\' as a name for anything'.format(t))

        file_name = None
        if type(f) is Atomise:
            file_name = f.name[1:]
        elif type(f) is str:
            file_name = f
        elif type(f) is tree.Uneval:
            file_name = f.value.value
        if file_name is None:
            return EX.throw(node.operands[i].location,
                'Was not able to deduce a filename from `require\n`'+
                + 'argument supplied...')

        curren_path = os.path.dirname(node.location['filename'])
        file_name = curren_path + '/' + file_name
        if not os.path.isfile(file_name):
            file_name = file_name + '.lispy'
            if not os.path.isfile(file_name):
                return EX.throw(node.operands[i].location,
                    'Cannot find file `{s}\' or `{s}.lispy\''.format(
                        s=file_name
                    ))

        # Start reading the actual file:
        if conf.DEBUG: print("\n\nLOADING FILE: ", file_name)
        load_file(file_name)
        # Finished the walk...

        i += 1
    return ATOMS[':true']
```

Figure 34: Parses file-paths and calls `load_file` on them.

3.4 All Internal Macros

An internal macro is just like any LISPY macro, except it's written in and interfaces with the implementation language (Python).

All the internal macros are listed in a global dictionary in the `visitor.py` file, where each key is the macro name and the corresponding value is a reference to the Python function that implements the macro.

The list of macros is found at line `968` and is, as of current, the following:

```
MACROS = {
    'do': _do_macro,
    'prog': _do_macro,
    'yield': _yield_macro,
    'require': _require_macro,
    'eval': _eval_macro,
    'scope': _scope_macro, # < Mostly for debugging.
    'type': _type_macro,
    'name': _name_macro,
    'if': _if_macro,
    'unless': _unless_macro,
    'list': _list_macro,
    'size': _size_macro,
    'index': _index_macro,
    'iterate': _iterate_macro,
    'push': _push_macro,
    'unshift': _unshift_macro,
    'concat': _concat_macro,
    'concat!': _concat_des_macro,
    'pop': _pop_macro,
    'shift': _shift_macro,
    '<>': _composition_macro, # < Neat, isn't it?
    '+': _add_macro,
    '-': _sub_macro,
    '*': _mul_macro,
    '/': _div_macro,
    '%': _mod_macro,
    '=': _eq_macro,
    '/=':_nq_macro,
    '!': _ne_macro,
    '&&': _and_macro,
    '||': _or_macro,
    '^^': _xor_macro, # < Not a common one.
    '<': _lt_macro,
    '>': _gt_macro,
    '⩽':_le_macro,
```

```
    '⩾':_ge_macro,
    'string': _string_macro,
    'repr': _repr_macro,
    'ast': _ast_macro, # < Mostly for debugging.
    'out': _out_macro,
    'read': _read_macro,
    'puts': _puts_macro,
    'let': _let_macro,
    'delete': _delete_macro,
    'mutate': lambda node: _let_macro(node, mutable=True),
    'λ': _lambda_macro,
    '→': _shorthand_macro,
    'define': _define_macro,
}
```

The implementation of each internal macro will take a long time to go over, so I'll leave their implementations and their helper functions out of this document, so if you need to refer to their implementation, see `/lispy/visitor.py` on line `343` to line `966`.

An outline of what each internal macro does will be outlined in the following subsections of this subsection:

3.4.1 `do` & `prog`

Takes a variadic amount of arguments, each argument is executed subsequently, essentially allowing you to just chain multiple statements together as a block.

3.4.2 `yield`

Allows you to *return* or *yield* from a function. This is only useful if you're inside a `do` block and want to return early.

3.4.3 `require`

Takes in any number of *name nodes*, which is either:

- An Atom
- A string
- An unevaluated symbol.

The name node will we read as a file path, and loaded in to the program.

3.4.4 `eval`

Takes a single argument. It will evaluate anything that was previously unevaluated. If already evaluated, it will do nothing and return its argument.

3.4.5 `scope`

Takes exactly one symbol, will return a list containing the name of the scope, and the scope hash for the scope that the given symbol finds itself in.

3.4.6 `type`

Takes in one argument of any type, and returns an atom with the name of the type of the evaluated argument that was passed in.

3.4.7 `name`

Converts any name node into an atom with the same *name*.

3.4.8 `if`

An if statement takes 2 or 3 arguments. The first argument is the condition, the second the consequence, and the third (if given) is the alternative (if-this-then-that-otherwise-that). The first argument must evaluate to something true or false, if the first argument is true the second argument gets evaluated, if not, the third argument gets evaluated (if it exists).

3.4.9 `unless`

The very same as the if statement, except it negates the condition, and proceeds as normal.

```
;; The following statements are equivalent:

(if (/= a b)
  (puts "a and b are different")
  (puts "a and b are the same"))

(unless (= a b)
  (puts "a and b are different")
  (puts "a and b are the same"))
```

3.4.10 `list`

Creates a list of all it's arguments, this is similar to the quote, but here all the arguments get evaluated.

3.4.11 `size`

Takes in a single list or string, returns the length of the elements.

3.4.12 `index`

Takes exactly two arguments, the first argument is a signed integer that represents an index number of the array starting from zero, negative indices represent accessing elements from the end of the list.

3.4.13 `iterate`

Takes a single statement and executes it over and over again, forever. The loop can only be broken if a the statement evaluates to a `break`.

3.4.14 `push`

Takes two arguments, the first one can be anything, and the second one needs to be a list. The fist argument will be appended to the the end of the list.

3.4.15 `unshift`

Same as push, instead it prepends to the list as oppsed to appending.

3.4.16 `concat`

Takes any number of lists or strings and merges them together as a new list or string.

3.4.17 `concat!`

The exclamation mark at the end signifies that the method is destructive. This works the same as `concat` but the first argument must be a symbol, and that symbol will be mutated to the result of the concatenation.

3.4.18 `pop`

Removes the last element from the list that is passed in.

3.4.19 `shift`

Removes the first element from the list that is passed in.

3.4.20 `<>`

Takes in any number of arguments, all of which *must* evaluate to *functions*, these functions will be composed together.

For example:

```
(f (g (h (i (j (k x y)))))) ;; read: f of g of h of i of j of k of x and y.
;; This can be better written as a composition.
((<> f g h i j k) x y) ;; Far fewer parentheses, easier to read, and shorter.
```

3.4.21 `+`

Takes any number of all numbers or all strings and adds them all together.

3.4.22 `-`

Takes any number of numbers and finds their difference.

3.4.23 `*`

Takes any number of numbers and finds their product.

3.4.24 `/`

Takes any number of numbers and finds their quotient. Will always return a float no matter what type the arguments are, unless the numbers divide each other evenly.

3.4.25 `%`

Takes any number of numbers and returns their modulo.

3.4.26 `=`

Will check if all its arguments are equal to each other.

3.4.27 `/=`

Will check if at least one of its arguments are not equal to the rest.

3.4.28 `!`

Negates the *one* argument given.

3.4.29 `&&`

Checks if all arguments evaluated to something *truthy*.

3.4.30 `||`

Checks if at least one argument or more are truthy.

3.4.31 `^`

Checks if an even number of arguments are truthy (XOR — Exclusive Or).

3.4.32 `<`

Checks if each subsequent argument is less than the last.

3.4.33 `>`

Checks if each subsequent argument is greater than the last.

3.4.34 `<=`

Checks if each subsequent argument is less than or equal the last.

3.4.35 `>=`

Checks if each subsequent argument is greater than or equal the last.

3.4.36 `string`

Tries it best to convert its argument into a string representation.

3.4.37 `repr`

Converts a string to its unescaped version.

e.g. It would return the string backslash 'n' ("\n") when given a string with a new-line in it.

3.4.38 `ast`

Returns a string representation of the unevaluated syntax tree of the argument you passed in.

3.4.39 `out`

Will print the string representation of any number of arguments passed into it, with no seperation between the strings and no trailing new line appended implicitly to STDOUT.

3.4.40 `read`

Will wait for user input from STDIN, and takes it when the user gives a new line.

3.4.41 `puts`

Does the same as `out`, but each string is separated by a space, and adds a trailing new line.

3.4.42 `let`

Takes in any number of symbol-value pairs (e.g. `(sym val)`; `(x 3)`) where the first value is a symbol which the value will get bound to.

For example:

```
(let (x 3) (y 7))  ;; Read as: `let x = 3 and y = 7.'
```

3.4.43 `delete`

Takes any number of symbols and deletes them from their symbol table thereby removing their bindings.

3.4.44 `mutate`

Same as let, except it allows the mutation of its symbols.

3.4.45 `λ` & `lambda`

Takes two arguments. The first one is a list of symbols and the second can be any expression. The first list represents the arguments being given to the anonymous function

/ closure / lambda and the second argument is the lambda body, which may use the arguments given to it listed in the argument list.

`λ` has been aliased to `lambda` in the prelude library.

3.4.46 →

This is a shorthand lambda. It works the same as normal lambda but it doesn't need an argument list. This is because the number of arguments are deduced at the parse stage, this is possible as arguments are simply listed numerically with a percentage sign in front.

For example:

```
(→ (+ 5 %1 %2))  ;; A function that takes two arguments and
                  ;; returns their sum plus five.
;; This is equivalent to:

(lambda (n m) (+ 5 n m))
```

3.4.47 define

This is similar to the lambda, except it gives the function a name. The first argument must be a list, the list must contain first the function name, and the rest are the function parameters. The second argument is the function body.

For example:

```
(define (f x y)
  (+ 5 x y))
;; Which is equivalent to
(let (f (lambda (x y) (+ 5 x y))))
```

3.5 Debugging, Configuration & Verbose Mode

In the file `/lispy/config.py` it is possible to alter the behaviour of the execution slightly.

The first number of variables listed are recursion depth limits, which allow Python's call stack to grow to a certain size by the line of code:

```
RECURSION_LIMIT = YOUR_LIMIT_HERE
sys.setrecursionlimit(RECURSION_LIMIT)
```

Right now the recursion limit is very high, which *could* lead to some overflows.

The following constants: `DEBUG, EXIT_ON_ERROR, RECOVERING_FROM_ERROR` change slightly the behaviour / output of the program.

`DEBUG` — Will make the interpreter output information about just about anything it is doing. You may have seen many lines of code saying something similar to: `if conf.DEBUG: print( ... )`, this is what they're for. The debug mode has been very useful while developing the components of the interpreter.

`EXIT_ON_ERROR` — Is another Boolean, it dictates whether the program should continue executing after encountering an error. This is useful of REPLs as the REPL should still continue working even if you write a faulty line of code.

`RECOVERING_FROM_ERROR` — Is just a global that keeps track of whether or not the program has recovered form an error yet.

3.6 Prelude / Standard Library for the Language

The Prelude is simply a set of files that run before *your* program begins to execute. The library outlines some simple functions, globals, and macros which are useful for writing your code. Examples of things the prelude implements are: The `while` statement; the `for` loop; functions from functional programming such as `map`, `reduce` and `filter`; &c.

The current files in the prelude are (in load order):

- `functional.lispy` — Basic functional functions, this file implements:
  - `lambda`, which is aliased to `λ`.
  - `apply` which is a macro that takes a function and applies it to a list of arguments as that functions argument list.
- `destructive.lispy` — All destructive operations, this file implements:
  - `incr!`, which takes a symbol and a number, and increments the symbol by that number.
  - `decr!`, which does the same as `incr!` except it decrements instead.

- `+1`, which increments a symbol by exactly one.
- `-1`, which decrements a symbol by exactly one.

- `dt.lispy` — Defines a number of functions for workin with datatypes, this file implements:
  - `to_string`, which converts its argument to a string.
  - `to_atom`, which converts its argument to an atom.
  - `type?`, which checks if its first arguments type matches the type of the second argument.
  - `nil?`, which checks if its argument is of type NIL.
  - `atom?`, which checks if its argument is of type ATOM.

- `loop.lispy` — Defines a number of different styles of loop, this file implements:
  - `while`, which will repeat its second argument, as long as its first argument keeps being true.
  - `until`, which will repeat its second argument, as long as its first argument keeps being false.
  - `loop`, which is a a loop that will run forever.
  - `from`, which will run a function a given amount of times, from a given range, where the argument supplied to that function is the current position through that range.
  - `times`, will run a function a given amount of times.

- `lists.lispy` — A collections of functions for list operations, this file implements:
  - `quote`, which simply quotes an expression, just like the single quote operator, however here it's named.
  - `push, unshift, concat, concat!` have all been aliased to: `append, prepend, merge, merge!`, respectively.
  - `empty?`, which takes a list and returns a Boolean depending on whether the list is empty.
  - `first`, which returns the first item of a list.
  - `last`, which returns the last item of a list.
  - `head` is an alias of `first`.
  - `tail`, returns a list of all elements of the given list excluding the first element of the list.
  - `for`, which implements a for loop with the syntax: `(for some-symbol in some-list (do execute this body))`.
  - `reverse`, which reverses a given list.
  - `fill`, which fills a given array with integers between a specified range.
  - `range`, which returns an array filled with integers from a given lowe bound and upper bound.

- `map`, which takes a function and a list and applies that function to every element of that list.
- `reduce`, which will apply an operation (given by a function) between every element of the given list, reducing it down to one value.
- `filter`, which takes a function which is applied to every element of a supplied list, and if that function evaluates to true, it is appended to a new list, which is then ultimately returned.

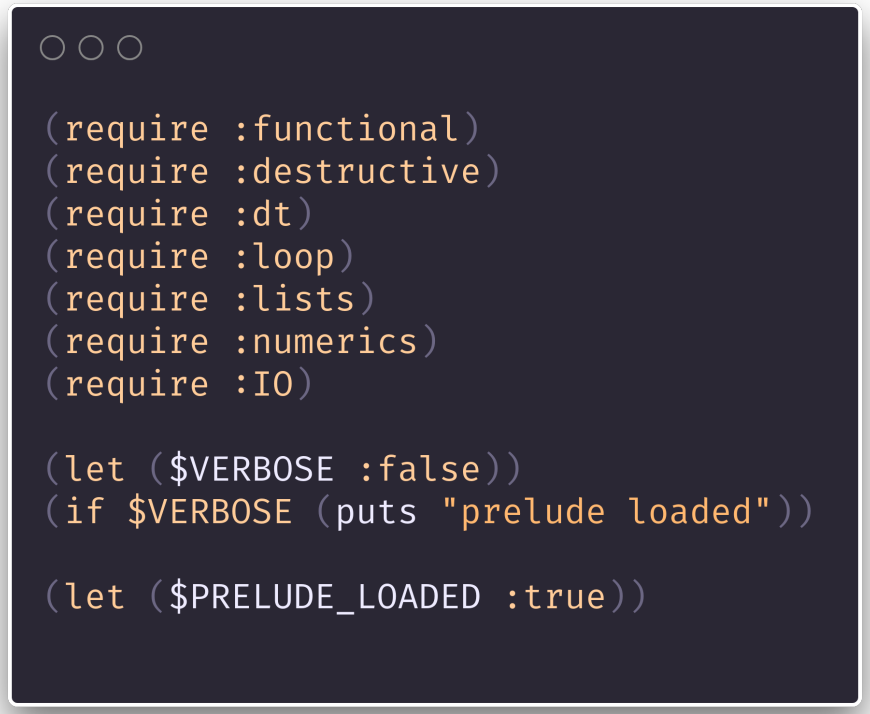
• `numerics.lispy` — Defines a set of functions for used for doing work on numbers, this file implements:

- `zero?`, which takes a number and returns whether the number is zero or not.
- `divisible?`, checks whether your first argument is divisible by the second argument.
- `sum`, which returns the sum of a list. It is defined as `(define (sum l) (apply + l))`.

• `IO.lispy` — Functions relating to input and output, this file implements:

- `EOF`, which is a variable that has value of: `"\0"`.
- `out, read` have been aliased to `print, input` respectively.

The file `/prelude/prelude.lispy` requires all the files above, and looks like this.



```
(require :functional)
(require :destructive)
(require :dt)
(require :loop)
(require :lists)
(require :numerics)
(require :IO)

(let ($VERBOSE :false))
(if $VERBOSE (puts "prelude loaded"))

(let ($PRELUDE_LOADED :true))
```

Figure 35: The prelude file that gets loaded at the beginning of every interpreter spawn.

It is important to note that the Prelude is entirely implemented in LISPY itself, this demonstrates that the language is not only quite usable, but also extendable in its syntax, mostly through the use of macros.

4 Testing of Implementation

I have been continuously the implementation of the language as I've been developing it, many of the sample code, and all of the debug/verbose prints found throughout the code were done long before the language implementation was finished.

4.1 Testing Discrete Sub-Components

Each stage of the interpreter has many small print statements that will print if the `DEBUG` constant is set to `True` in the `config.py` file, these allow me to see what every stage of the interpreter is doing.

It will be impossible to replicate and document the hundreds of tests I've run throughout the development of the interpreter, but I will run through an example of how I tested these sub-components.

Most of my testing has been done in a file found in the root directory of the repository called '`/testing.lispy`', which currently happens to have something appropriate to use as an example here.

`testing.lispy` looks like this:

```
(define macro (ten-times sym body) (do
  (let (sym 0))
  (iterate (do
    (mutate (sym (+ (eval sym) 1)))
    (if (> (eval sym) 10) break)
    (eval body)))
  (delete sym)))


(ten-times counter (do
  (puts counter)))
                    ;;       vvv  --- UNDERSCORE
(→ (+ %1 %2 %3 %4)) ;; Here, `_' just means the return value of the
(_ 5 5 5 5)         ;;   last statement that has been evaluated.
(puts (string "5*4 =" _))
```

Let's now also write a little Python script which will print out the exact information we need. It is called `/debug-stages.py` and looks like this:

```
import lispy


# Let's first read the contents of the file
#   as a string into a variable.
TESTING_FILE = './testing.lispy'

PROGRAM_STRING = None
with open(TESTING_FILE, 'r') as file:
    PROGRAM_STRING = file.read()  # Read from the file.


# ═══ Lexing ═══ #
#   The lex method will reduce the program string
#   into a TokenStream.
stream = lispy.lexing.lex(PROGRAM_STRING, TESTING_FILE)

print("\n\n═══ Token Stream ═══\n")
print(stream)  # Here we print the stream out.


# ═══ Parsing ═══ #
#   We call the parse method on the stream
#   and it will return an AST (parse tree).
ast = lispy.parsing.parse(stream)

print("\n\n═══ Abstract Syntax Tree ═══\n")
print(ast)  # Print a visualisation of the tree.

# ═══ Macro Expansion ═══ #
#   Here we search and invoke and replace macros
#   in the parse tree.
expanded = lispy.parsing.preprocess(ast)

print("\n\n═══ Macro Expanded ═══\n")
print(expanded)
```

Figure 36: Our debugging script.

4.1.1 Lexical Analysis

The lexer has been described in detail previously, and produces the expected token stream. This is what the above program's token stream looks like:

```
<Token(L_PAREN) ........... [1, 1] ... '('>
<Token(SYMBOL) ............ [1, 2] ... 'define'>
<Token(SYMBOL) ............ [1, 9] ... 'macro'>
<Token(L_PAREN) .......... [1, 15] ... '('>
<Token(SYMBOL) ........... [1, 16] ... 'ten-times'>
<Token(SYMBOL) ........... [1, 26] ... 'sym'>
<Token(SYMBOL) ........... [1, 30] ... 'body'>
<Token(R_PAREN) .......... [1, 34] ... ')'>
<Token(L_PAREN) .......... [1, 36] ... '('>
<Token(SYMBOL) ........... [1, 37] ... 'do'>
<Token(TERMINATOR) ....... [1, 39] ... '\n'>
<Token(L_PAREN) ........... [2, 3] ... '('>
<Token(SYMBOL) ............ [2, 4] ... 'let'>
<Token(L_PAREN) ........... [2, 8] ... '('>
<Token(SYMBOL) ............ [2, 9] ... 'sym'>
<Token(NUMERIC) .......... [2, 13] ... '0'>
<Token(R_PAREN) .......... [2, 14] ... ')'>
<Token(R_PAREN) .......... [2, 15] ... ')'>
<Token(TERMINATOR) ....... [2, 16] ... '\n'>
<Token(L_PAREN) ........... [3, 3] ... '('>
<Token(SYMBOL) ............ [3, 4] ... 'iterate'>
<Token(L_PAREN) .......... [3, 12] ... '('>
<Token(SYMBOL) ........... [3, 13] ... 'do'>
<Token(TERMINATOR) ....... [3, 15] ... '\n'>
<Token(L_PAREN) ........... [4, 5] ... '('>
<Token(SYMBOL) ............ [4, 6] ... 'mutate'>
<Token(L_PAREN) .......... [4, 13] ... '('>
<Token(SYMBOL) ........... [4, 14] ... 'sym'>
<Token(L_PAREN) .......... [4, 18] ... '('>
<Token(SYMBOL) ........... [4, 19] ... '+'>
<Token(L_PAREN) .......... [4, 21] ... '('>
<Token(SYMBOL) ........... [4, 22] ... 'eval'>
<Token(SYMBOL) ........... [4, 27] ... 'sym'>
<Token(R_PAREN) .......... [4, 30] ... ')'>
<Token(NUMERIC) .......... [4, 32] ... '1'>
<Token(R_PAREN) .......... [4, 33] ... ')'>
<Token(R_PAREN) .......... [4, 34] ... ')'>
<Token(R_PAREN) .......... [4, 35] ... ')'>
<Token(TERMINATOR) ....... [4, 36] ... '\n'>
<Token(L_PAREN) ........... [5, 5] ... '('>
<Token(SYMBOL) ............ [5, 6] ... 'if'>
<Token(L_PAREN) ........... [5, 9] ... '('>
<Token(SYMBOL) ........... [5, 10] ... '>'>
<Token(L_PAREN) .......... [5, 12] ... '('>
<Token(SYMBOL) ........... [5, 13] ... 'eval'>
<Token(SYMBOL) ........... [5, 18] ... 'sym'>
<Token(R_PAREN) .......... [5, 21] ... ')'>
<Token(NUMERIC) .......... [5, 23] ... '10'>
<Token(R_PAREN) .......... [5, 25] ... ')'>
<Token(SYMBOL) ........... [5, 27] ... 'break'>
<Token(R_PAREN) .......... [5, 32] ... ')'>
<Token(TERMINATOR) ....... [5, 33] ... '\n'>
<Token(L_PAREN) ........... [6, 5] ... '('>
<Token(SYMBOL) ............ [6, 6] ... 'eval'>
<Token(SYMBOL) ........... [6, 11] ... 'body'>
<Token(R_PAREN) .......... [6, 15] ... ')'>
```

```
<Token(R_PAREN) ......... [6, 16] ... ')'>
<Token(R_PAREN) ......... [6, 17] ... ')'>
<Token(TERMINATOR) ....... [6, 18] ... '\n'>
<Token(L_PAREN) ........... [7, 3] ... '('>
<Token(SYMBOL) ............ [7, 4] ... 'delete'>
<Token(SYMBOL) ........... [7, 11] ... 'sym'>
<Token(R_PAREN) ......... [7, 14] ... ')'>
<Token(R_PAREN) ......... [7, 15] ... ')'>
<Token(R_PAREN) ......... [7, 16] ... ')'>
<Token(TERMINATOR) ....... [7, 17] ... '\n'>
<Token(TERMINATOR) ........ [8, 1] ... '\n'>
<Token(TERMINATOR) ........ [9, 1] ... '\n'>
<Token(L_PAREN) ......... [10, 1] ... '('>
<Token(SYMBOL) ........... [10, 2] ... 'ten-times'>
<Token(SYMBOL) ......... [10, 12] ... 'counter'>
<Token(L_PAREN) ......... [10, 20] ... '('>
<Token(SYMBOL) ......... [10, 21] ... 'do'>
<Token(TERMINATOR) ...... [10, 23] ... '\n'>
<Token(L_PAREN) ......... [11, 3] ... '('>
<Token(SYMBOL) ........... [11, 4] ... 'puts'>
<Token(SYMBOL) ........... [11, 9] ... 'counter'>
<Token(R_PAREN) ......... [11, 16] ... ')'>
<Token(R_PAREN) ......... [11, 17] ... ')'>
<Token(R_PAREN) ......... [11, 18] ... ')'>
<Token(TERMINATOR) ...... [11, 19] ... '\n'>
<Token(TERMINATOR) ...... [12, 50] ... '\n'>
<Token(L_PAREN) ......... [13, 1] ... '('>
<Token(SYMBOL) ........... [13, 2] ... '->'>
<Token(L_PAREN) ......... [13, 5] ... '('>
<Token(SYMBOL) ........... [13, 6] ... '+'>
<Token(SYMBOL) ........... [13, 8] ... '%1'>
<Token(SYMBOL) ......... [13, 11] ... '%2'>
<Token(SYMBOL) ......... [13, 14] ... '%3'>
<Token(SYMBOL) ......... [13, 17] ... '%4'>
<Token(R_PAREN) ......... [13, 19] ... ')'>
<Token(R_PAREN) ......... [13, 20] ... ')'>
<Token(TERMINATOR) ...... [13, 69] ... '\n'>
<Token(L_PAREN) ......... [14, 1] ... '('>
<Token(SYMBOL) ........... [14, 2] ... '_'>
<Token(NUMERIC) ......... [14, 4] ... '5'>
<Token(NUMERIC) ......... [14, 6] ... '5'>
<Token(NUMERIC) ......... [14, 8] ... '5'>
<Token(NUMERIC) ......... [14, 10] ... '5'>
<Token(R_PAREN) ......... [14, 11] ... ')'>
<Token(TERMINATOR) ...... [14, 66] ... '\n'>
<Token(L_PAREN) ......... [15, 1] ... '('>
<Token(SYMBOL) ........... [15, 2] ... 'puts'>
<Token(L_PAREN) ......... [15, 7] ... '('>
<Token(SYMBOL) ........... [15, 8] ... 'string'>
<Token(STRING) ......... [15, 15] ... '5*4 ='>
<Token(SYMBOL) ......... [15, 23] ... '_'>
<Token(R_PAREN) ......... [15, 24] ... ')'>
<Token(R_PAREN) ......... [15, 25] ... ')'>
<Token(TERMINATOR) ...... [15, 26] ... '\n'>
<Token(EOF) ............. [16, 1] ... '\x00'>
```

Here we can see every token labelled with all the correct annotation, and accurate line and column numbers. The stream also provides readable strings which are snippets of the code they correspond to. Every token stream ends in an EOF token.

4.1.2 Initial Parse Tree

The script then prints out the initial parse tree, which if you read carefully, you can quite clearly see how it corresponds to the original code. Indentation in the source file corresponds vaguely to indentation in the tree.

The tree looks like this:

```
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('define')>
|       └─operands=[
|          <AST::Node[Symbol] ('macro')>
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('ten-times')>
|             └─operands=[
|                <AST::Node[Symbol] ('sym')>
|                <AST::Node[Symbol] ('body')>
|               ]>
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('do')>
|             └─operands=[
|                <AST::Node[Call]
|                ├─caller
|                |  └─<AST::Node[Symbol] ('let')>
|                └─operands=[
|                   <AST::Node[Call]
|                   ├─caller
|                   |  └─<AST::Node[Symbol] ('sym')>
|                   └─operands=[
|                      <AST::Node[Numeric] (0)>
|                     ]>
|                  ]>
|                <AST::Node[Call]
|                ├─caller
|                |  └─<AST::Node[Symbol] ('iterate')>
|                └─operands=[
|                   <AST::Node[Call]
|                   ├─caller
|                   |  └─<AST::Node[Symbol] ('do')>
|                   └─operands=[
|                      <AST::Node[Call]
|                      ├─caller
|                      |  └─<AST::Node[Symbol] ('mutate')>
|                      └─operands=[
|                         <AST::Node[Call]
|                         ├─caller
|                         |  └─<AST::Node[Symbol] ('sym')>
|                         └─operands=[
|                            <AST::Node[Call]
|                            ├─caller
|                            |  └─<AST::Node[Symbol] ('+')>
|                            └─operands=[
|                               <AST::Node[Call]
```


├─caller
│ └─<AST::Node[Symbol] ('eval')>
└─operands=[
<AST::Node[Symbol] ('sym')>
]>
<AST::Node[Numeric] (1)>
]>
]>
]>
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('if')>
└─operands=[
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('>')>
└─operands=[
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('eval')>
└─operands=[
<AST::Node[Symbol] ('sym')>
]>
<AST::Node[Numeric] (10)>
]>
<AST::Node[Symbol] ('break')>
]>
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('eval')>
└─operands=[
<AST::Node[Symbol] ('body')>
]>
]>
]>
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('delete')>
└─operands=[
<AST::Node[Symbol] ('sym')>
]>
]>
]>
├─<AST::Node[Call]
│ ├─caller
│ │ └─<AST::Node[Symbol] ('ten-times')>
│ └─operands=[
<AST::Node[Symbol] ('counter')>
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('do')>
└─operands=[
<AST::Node[Call]
├─caller
│ └─<AST::Node[Symbol] ('puts')>
└─operands=[
<AST::Node[Symbol] ('counter')>
]>

```
|              ]>
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('->')>
|       └─operands=[
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('+')>
|             └─operands=[
|                <AST::Node[Symbol] ('%1')>
|                <AST::Node[Symbol] ('%2')>
|                <AST::Node[Symbol] ('%3')>
|                <AST::Node[Symbol] ('%4')>
|               ]>
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('_')>
|       └─operands=[
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('puts')>
|       └─operands=[
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('string')>
|             └─operands=[
|                <AST::Node[String] ('5*4 =')>
|                <AST::Node[Symbol] ('_')>
|               ]>
|         ]>
```

4.1.3 Macro Expanded Tree

Every line beginning with '`(define macro ( ...`' needs to defines a macro, and macros are only defined under the context of the preprocessor part of the parser, and the evaluator would have no idea what to do with it. Hence every occurrence of a macro definition is simply replaced with a nil node, as you'll see in the macro expanded tree, as well as the expansion of the above macro.

The final, macro expanded, abstract syntax tree looks like this:

```
│
├─<AST::Node[Nil] ('nil')>
│
├─<AST::Node[Call]
│       ├─caller
│       │  └─<AST::Node[Symbol] ('do')>
│       └─operands=[
│          <AST::Node[Call]
│             ├─caller
│             │  └─<AST::Node[Symbol] ('let')>
│             └─operands=[
│                <AST::Node[Call]
│                ├─caller
│                │  └─<AST::Node[Uneval] (<AST::Node[Symbol] ('counter')>)>
│                └─operands=[
│                   <AST::Node[Numeric] (0)>
│                  ]>
│              ]>
│          <AST::Node[Call]
│             ├─caller
│             │  └─<AST::Node[Symbol] ('iterate')>
│             └─operands=[
│                <AST::Node[Call]
│                ├─caller
│                │  └─<AST::Node[Symbol] ('do')>
│                └─operands=[
│                   <AST::Node[Call]
│                   ├─caller
│                   │  └─<AST::Node[Symbol] ('mutate')>
│                   └─operands=[
│                      <AST::Node[Call]
│                      ├─caller
│                      │  └─<AST::Node[Uneval] (<AST::Node[Symbol] ('counter')>)>
│                      └─operands=[
│                         <AST::Node[Call]
│                         ├─caller
│                         │  └─<AST::Node[Symbol] ('+')>
│                         └─operands=[
│                            <AST::Node[Call]
│                            ├─caller
│                            │  └─<AST::Node[Symbol] ('eval')>
│                            └─operands=[
│                               <AST::Node[Uneval] (<AST::Node[Symbol] ('counter')>)>
│                              ]>
│                            <AST::Node[Numeric] (1)>
│                           ]>
│                        ]>
│                     ]>
│                   <AST::Node[Call]
│                   ├─caller
│                   │  └─<AST::Node[Symbol] ('if')>
│                   └─operands=[
│                      <AST::Node[Call]
│                      ├─caller
│                      │  └─<AST::Node[Symbol] ('>')>
│                      └─operands=[
│                         <AST::Node[Call]
```

```
|                        ├─caller
|                        |  └─<AST::Node[Symbol] ('eval')>
|                        └─operands=[
|                           <AST::Node[Uneval] (<AST::Node[Symbol] ('counter')>)>
|                          ]>
|                       <AST::Node[Numeric] (10)>
|                      ]>
|                    <AST::Node[Symbol] ('break')>
|                   ]>
|                 <AST::Node[Call]
|                 ├─caller
|                 |  └─<AST::Node[Symbol] ('eval')>
|                 └─operands=[
|                    <AST::Node[Uneval] (<AST::Node[Call]
|                                        ├─caller
|                                        |  └─<AST::Node[Symbol] ('do')>
|                                        └─operands=[
|                                           <AST::Node[Call]
|                                              ├─caller
|                                              |  └─<AST::Node[Symbol] ('puts')>
|                                              └─operands=[
|                                                 <AST::Node[Symbol] ('counter')>
|                                                ]>
|                                          ])>
|                   ]>
|                 ]>
|              ]>
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('delete')>
|             └─operands=[
|                <AST::Node[Uneval] (<AST::Node[Symbol] ('counter')>)>
|               ]>
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('->')>
|       └─operands=[
|          <AST::Node[Call]
|             ├─caller
|             |  └─<AST::Node[Symbol] ('+')>
|             └─operands=[
|                <AST::Node[Symbol] ('%1')>
|                <AST::Node[Symbol] ('%2')>
|                <AST::Node[Symbol] ('%3')>
|                <AST::Node[Symbol] ('%4')>
|               ]>
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |  └─<AST::Node[Symbol] ('_')>
|       └─operands=[
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
|          <AST::Node[Numeric] (5)>
```

```
|         ]>
|
├─<AST::Node[Call]
|       ├─caller
|       |   └─<AST::Node[Symbol] ('puts')>
|       └─operands=[
|          <AST::Node[Call]
|              ├─caller
|              |   └─<AST::Node[Symbol] ('string')>
|              └─operands=[
|                 <AST::Node[String] ('5*4 =')>
|                 <AST::Node[Symbol] ('_')>
|                ]>
|         ]>
```

4.1.4 Scoping & Tables

4.2 Testing the Prelude Library

4.3 Testing through Sample Code

4.4 Testing in the REPL

5 Appraisal

5.1 Comparison against Objectives

5.2 Future Improvements, Potential & Ideas

5.3 Users' Usage & Feedback

ASDAWSDWEASDXA

6 References

[1] Edwin D. Reilly. "Milestones in computer science & information technology". In: Greenwood Publishing Group, 2003, pp. 156, 157. ISBN: 978-1-57356-521-9. URL: https://books.google.co.uk/books?id=JTYPKxug49IC&pg=PA157.

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Inc, 1986. ISBN: 0-201-10088-6.

[3] J. McCarthy. "Recursive Functions of Symbolic Expressions & Their Computation by Machine, Part I". In: *Artificial intelligence Project — RLE & MIT Computation Center* (March 1959). URL: https://www.informatimago.com/develop/lisp/com/informatimago/small-cl-pgms/aim-8/aim-8.html.