Design and Implementation of a LISP Dialect & Interpreter

AQA NEA Comp. Sci.

Samuel F. D. Knutsen

Last Compiled: March 25, 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.

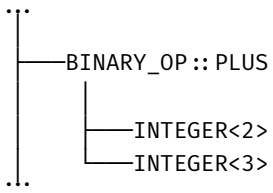
Almost every implementation of the most common programming languages start with what is known as a 'lexer' or 'tokeniser' that performs lexical analysis. As one might gather from the name, this stage in the implementation tokenises the program. Every program starts of as just a long string of characters 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']]
```

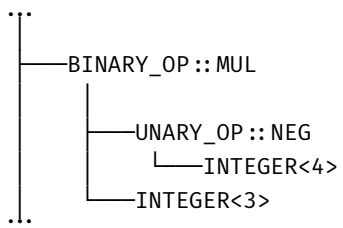
becomes



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:

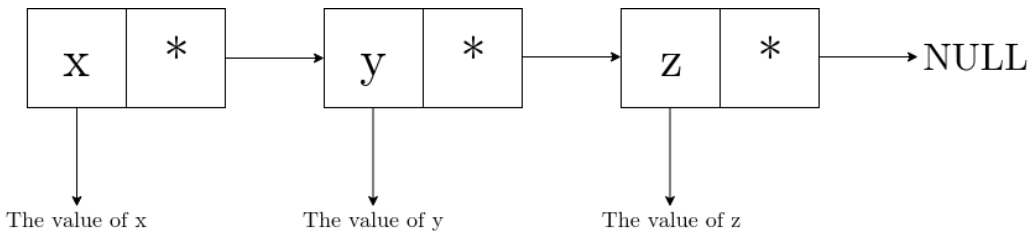


Figure 1: Visual representation of cons cells (i.e. a linked list)

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. `+, -, *, /, %,` &ct.). 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, &ct.), 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, &ct.
- `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
```

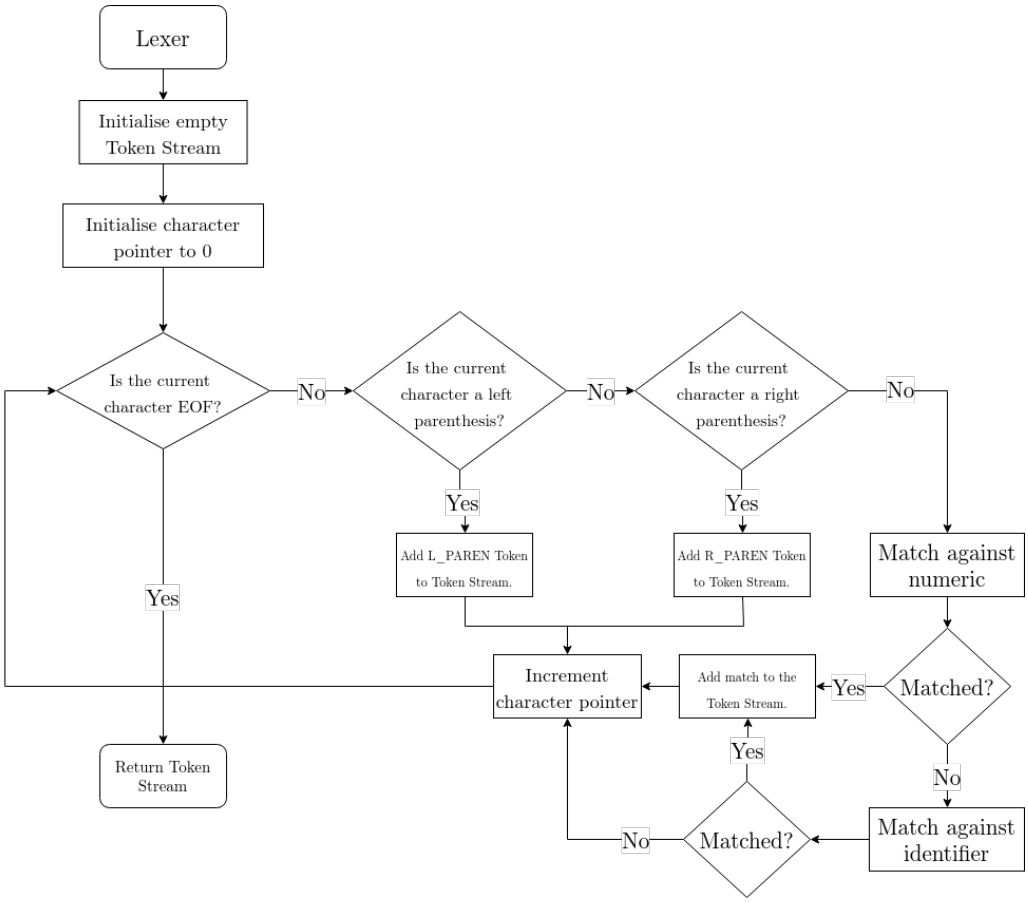


Figure 2: The lexer also represented by a flow-chart.

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.

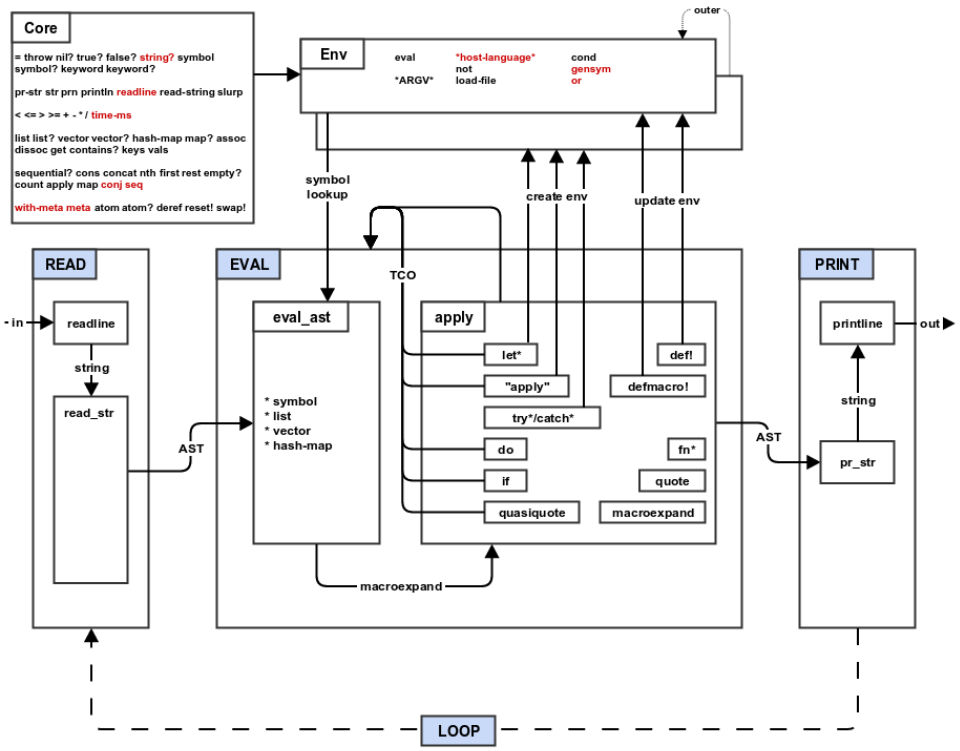


Figure 3: Here is an example of how a typical fully functional REPL-based Lisp interpreter might function.

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.

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

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

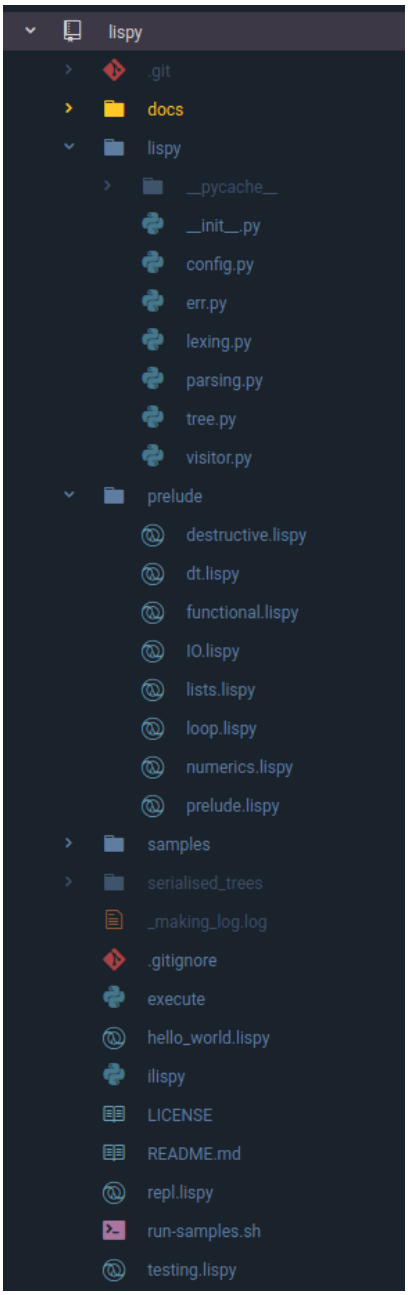


Figure 4: Repo. file structure.

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'
})
```

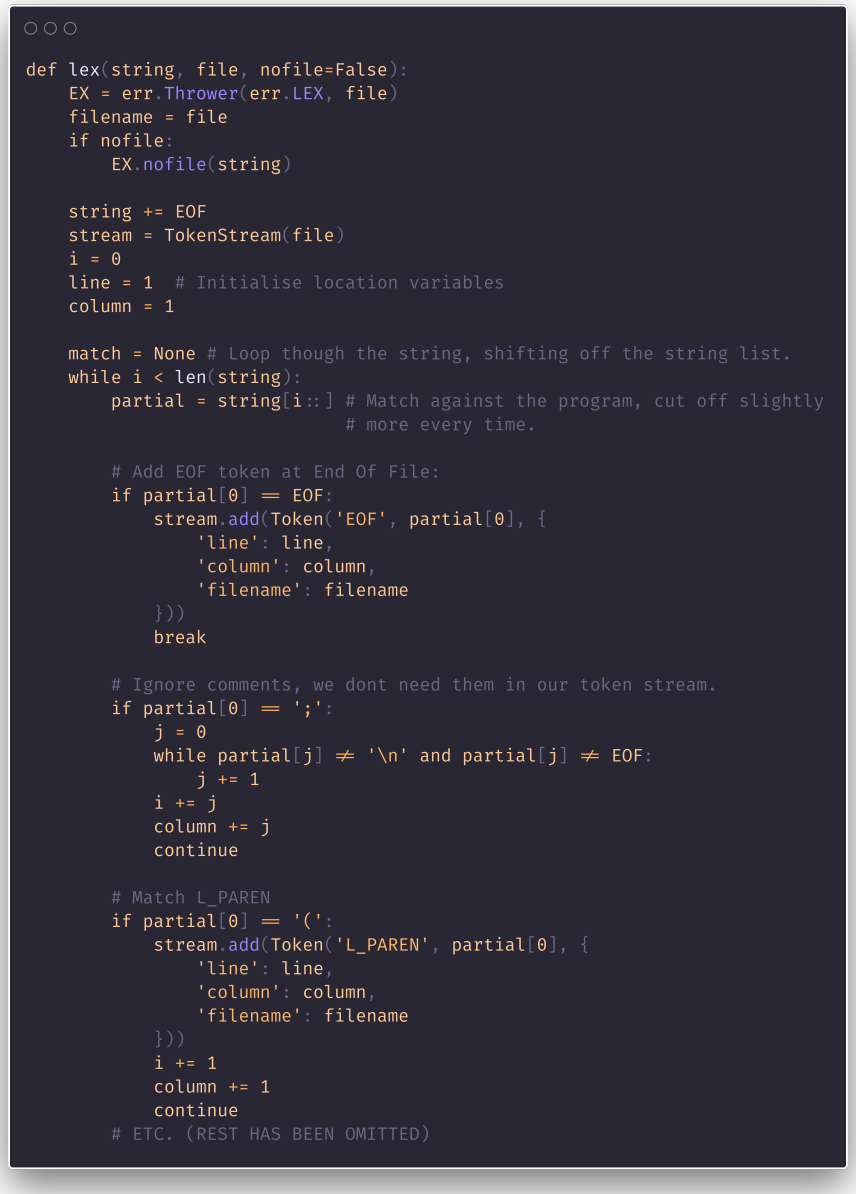
```python
# `TokenStream` is a wrapper for a list of tokens
#               to make it easier to manage what token we're
#               currently focused on.
class TokenStream(object):
    def __init__(self, file, tokens = None):
        self.file = file
        self.tokens = tokens or []
        self.i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

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.

7 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.