Let's write a LISP lexer together

Coding dojo by @meisterluk

2024-02-06

pyGRAZ

Vocabulary

- An *interpreter* reads source code and applies semantics immediately
- A formal *grammar* specifies the set of admissible source codes for the interpreter
- An interpreter can include a lexical analysis / tokenization (component “*lexer*”) and a semantic analysis / parsing (component “*parser*”)

The simplest formal grammar

Input source code:

W

The simplest formal grammar

Input source code:

W

Output:

Hello world!

The simplest formal grammar

Input source code:

W

Output:

Hello world!

Admissible source codes:

W

The simplest formal grammar

Input source code:

P

Output:

Hello pygraz!

The simplest formal grammar

Input source code:

Output:

Admissible source codes:

W
P

The simplest formal grammar

Input source code:

Admissible source codes:

W
P

Output:

Implementation:

```
if src == "W":
    print("Hello world!")
else:
    print("Hello pygraz!")
```

The simplest formal grammar

Input source code:

Admissible source codes:

W
P

Output:

Implementation:

```
if src.strip() == "W":
    print("Hello world!")
else:
    print("Hello pygraz!")
```

Boooring – can we do something non-static?

A non-static formal grammar

Input source code:

put Hello world!

Admissible source codes:

put <some-string>

Output:

Hello world!

A non-static formal grammar

Input source code:

put Hello world!

Admissible source codes:

put <*some-string*>

Output:

Hello world!

Implementation:

```
assert(src[0:4] == "put ")
print(src[4:])
```

But what if we need to compute
the output string beforehand?

An expressive formal grammar

Input source code:

put Hello sum(4, 5)!

Admissible source codes:

put *<some-string>*
and expression *sum(<args>)*

Output:

Hello 9!

Implementation:

An expressive formal grammar

Input source code:

put Hello sum(4, 5)!

Admissible source codes:

put *<some-string>*
and expression *sum(<args>)*

Output:

Hello 9!

Implementation:

?

An expressive formal grammar

```
put Hello sum(4, 5)!
```

identifiers

arguments

operators

An expressive formal grammar

```
put Hello sum(4, 5)!
```

identifiers

arguments

operators

Why are identifiers invoked so differently?

Where do I need commas between arguments?

What happens if I use "sum" as argument?

The simplest nested formal grammar

```
(put Hello (sum 4 5) !)
```

identifiers

arguments

operators

The simplest nested formal grammar

```
(put Hello (sum 4 5) !)
```

identifiers

arguments

operators

parenthesized prefix notation

The simplest nested formal grammar

```
(put Hello (sum 4 5) !)
```

identifiers

arguments

operators

parenthesized prefix notation = LISP?

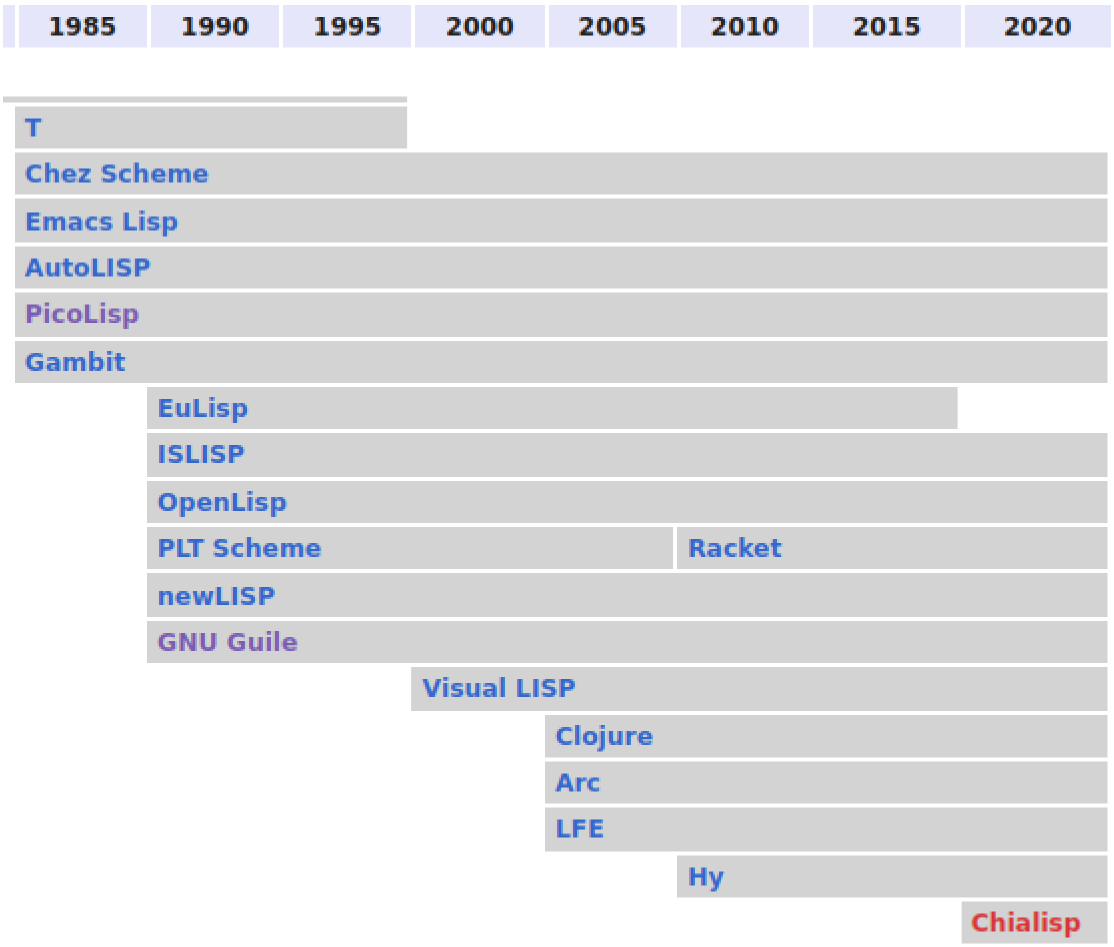
LISP

Timeline [edit]

Timeline of Lisp dialects

1958	1960	1965	1970	1975	1980	1985	1990	1995
LISP 1, 1.5, LISP 2(abandoned)								
		Maclisp						
			Interlisp					
			MDL					
				Lisp Machine Lisp				
				Scheme				R5RS
				NIL				
					ZIL (Zork Implementation Language)			
					Franz Lisp			
					Common Lisp		ANSI standard	
					Le Lisp			
					MIT Scheme			
					XLISP			

Wikipedia EN: Lisp (programming language)



Wikipedia EN: Lisp (programming language)

```
#lang racket/base

(require (for-syntax racket/base
                     racket/list
                     syntax/parse/pre))

(provide ~> ~>> and~> and~>> _
         lambda~> lambda~>> lambda~>* lambda~>>*
         lambda-and~> lambda-and~>> lambda-and~>* lambda-and~>>*
         (rename-out [lambda~> λ~>] [lambda~>> λ~>>]
                     [lambda~>* λ~>*] [lambda~>>* λ~>>*]
                     [lambda-and~> λ-and~>] [lambda-and~>> λ-and~>>]
                     [lambda-and~>* λ-and~>*] [lambda-and~>>* λ-and~>>*]))
```

```
=> (print "Hy!")
Hy!
=> (defn salutationsnm [name] (print (+ "Hy " name "!")))
=> (salutationsnm "YourName")
Hy YourName!
```

```
(defn- alias-help
  "Returns a string containing help for an alias, or nil if the string is not an
  alias."
  [aliases task-name]
  (if (aliases task-name)
    (let [alias-expansion (aliases task-name)
          explanation (-> alias-expansion meta :doc)]
      (cond explanation (str task-name ": " explanation)
            (string? alias-expansion) (str
                                       (format
                                        (str "'%s' is an alias for '%s',"
                                             " which has following help doc:\n")
                                        task-name alias-expansion)
                                       (help-for alias-expansion))
            :no-explanation-or-string (str task-name " is an alias, expands to "
                                           alias-expansion)))))
```

```
(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
          ((get-balance) get-balance)
          ((deposit) deposit)
          ((withdraw) withdraw)
          (else (error "Invalid method!")))
        (cdr args)))))
```

In my opinion

- *LISP* is a programming language with syntax (defining a formal grammar) and semantics
- The *LISP family/dialects* is a set of programming languages following the style of LISP 1.0 or LISP 1.5
- The syntax of LISP is called *S-expressions*.
- *S*-expressions is a form of *parenthesized prefix notation*

S-expressions

```
GNU Guile 3.0.7
Copyright (C) 1995-2021 Free Software Foundation, Inc.

Guile comes with ABSOLUTELY NO WARRANTY; for details type `,show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type `,show c' for details.

Enter `,help' for help.
scheme@(guile-user)> (cons 1 3)
$1 = (1 . 3)
scheme@(guile-user)> (cons 1 (cons 3 '()))
$2 = (1 3)
scheme@(guile-user)>
```

Characteristics [edit]

In the usual parenthesized syntax of Lisp, an S-expression is classically defined[1] as

1. an atom of the form `x`, or
2. an expression of the form `(x . y)` where *x* and *y* are S-expressions.

This definition reflects LISP's representation of a list as a series of "cells", each one an ordered pair. In plain lists, *y* points to the next cell (if any), thus forming a list. The recursive clause of the

Coding Dojo

Task:
Let us read a file written in parenthesized prefix notation.

Coding Dojo

Task:
Let us read a file written in parenthesized prefix notation.

Funfacts:

- The standard library code module provides a REPL to parse python code: https://bernsteinbear.com/blog/simple-python-repl/
- Peter Norvig documented our task in a blog post: https://norvig.com/lispy.html
  “The beauty of Scheme is that the full language only needs 5 keywords and 8 syntactic forms. In comparison, Python has 33 keywords and 110 syntactic forms, and Java has 50 keywords and 133 syntactic forms.”

Approach

1) Identify individual characters of the formal grammar. Give them names.
2) Define an INIT state. Which characters are admissible?
3) Reiterate to identify the lexing state diagram.
4) Yield tokens as you read character by character
5) A parser fetches these tokens to put them into a nested structure
6) The nested structure is interpreted.

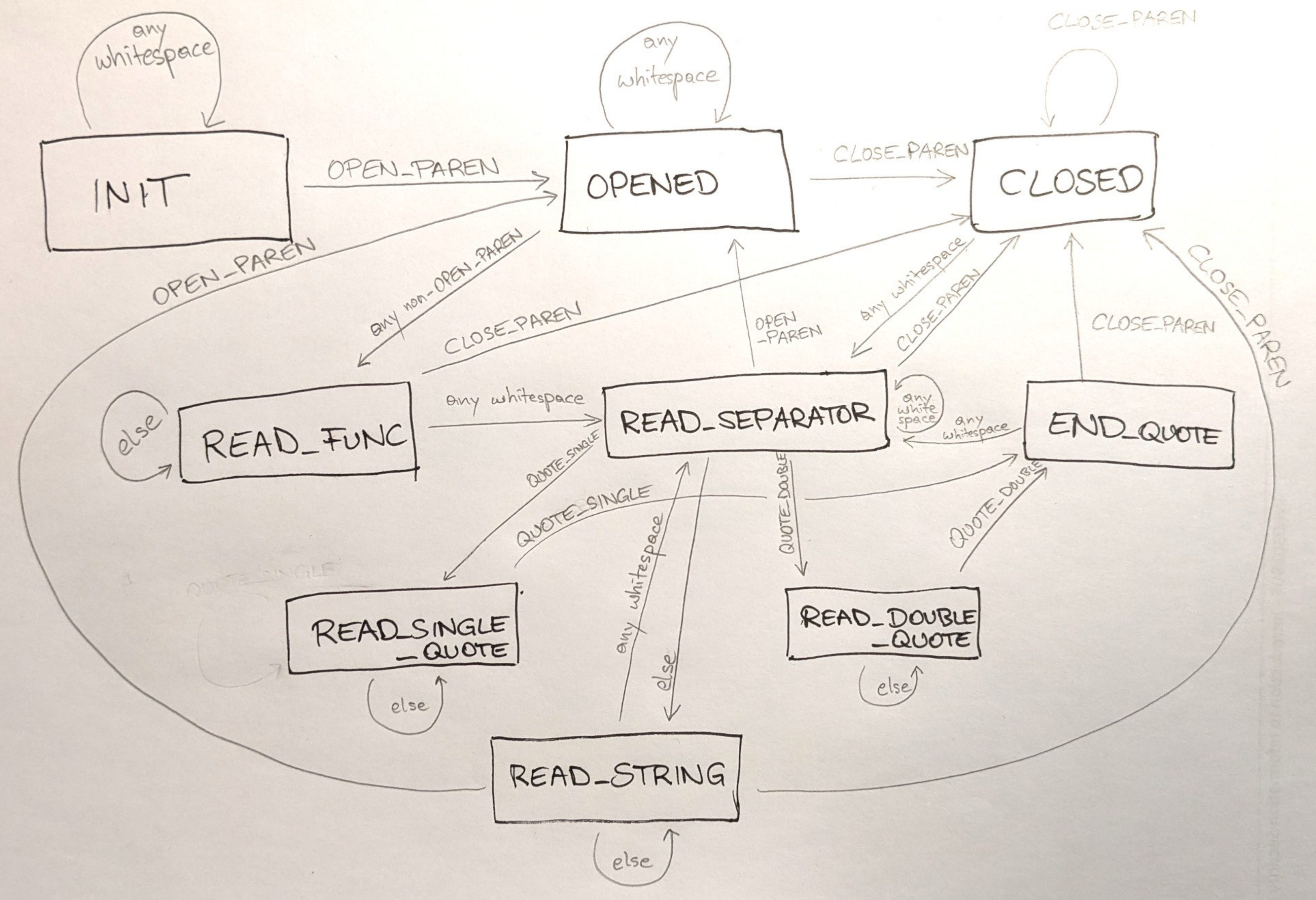
Approach

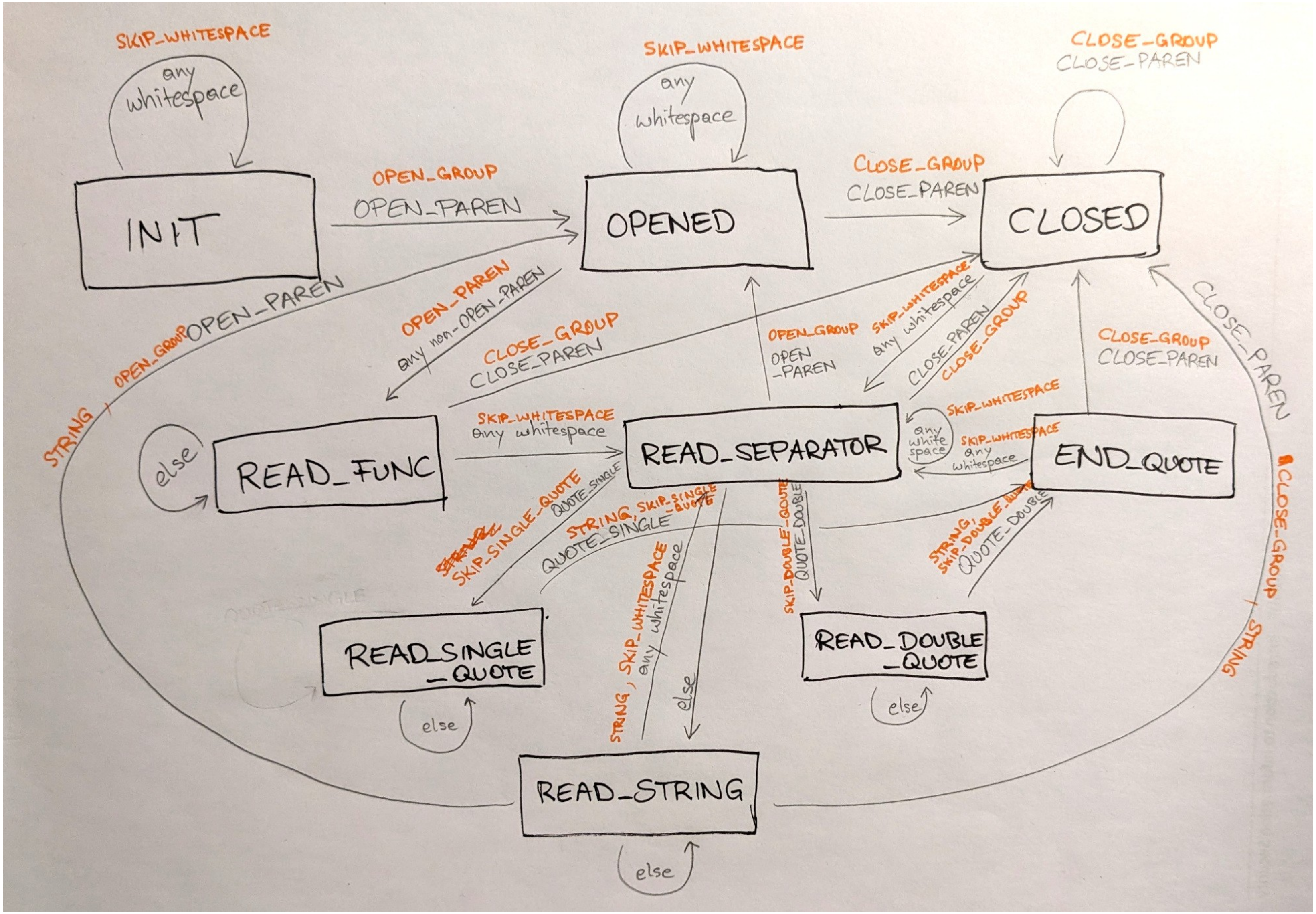
1) Identify individual characters of the formal grammar. Give them names.
2) Define an INIT state. Which characters are admissible?
3) Reiterate to identify the lexing state diagram.
4) Yield tokens as you read character by character

lexer

5) A parser fetches these tokens to put them into a nested structure
6) The nested structure is interpreted.

parser


INIT
OPENED
CLOSED
READ_FUNC
READ_SEPARATOR
END_QUOTE
READ_SINGLE _QUOTE
READ_DOUBLE _QUOTE
READ_STRING
any whitespace
OPEN_PAREN
CLOSE_PAREN
any non-OPEN_PAREN
else
QUOTE_SINGLE
QUOTE_DOUBLE
any white space



INIT
OPENED
CLOSED
READ_FUNC
READ_SEPARATOR
END_QUOTE
READ_SINGLE_QUOTE
READ_DOUBLE_QUOTE
READ_STRING
SKIP_WHITESPACE
any whitespace
OPEN_GROUP
OPEN_PAREN
CLOSE_GROUP
CLOSE_PAREN
OPEN_PAREN
any non-OPEN_PAREN
STRING, OPEN_GROUP
OPEN_PAREN
CLOSE_GROUP
CLOSE_PAREN
OPEN_GROUP
OPEN -PAREN
SKIP_WHITESPACE
any whitespace
CLOSE_PAREN
CLOSE_GROUP
CLOSE_GROUP
CLOSE_PAREN
CLOSE_PAREN
else
SKIP_WHITESPACE
any whitespace
SKIP_WHITESPACE
any white space
SKIP_WHITESPACE
any whitespace
SKIP_SINGLE_QUOTE
QUOTE_SINGLE
STRING, SKIP_SINGLE_QUOTE
QUOTE_SINGLE
STRING, SKIP_WHITESPACE
any whitespace
else
SKIP_DOUBLE_QUOTE
QUOTE_DOUBLE
STRING, SKIP_DOUBLE_QUOTE
QUOTE_DOUBLE
CLOSE_GROUP / STRING
else
else
else

```python
# raw characters of the syntax, I want to match
OPEN_PAREN = '('
CLOSE_PAREN = ')'
QUOTE_SINGLE = "'"
QUOTE_DOUBLE = '"'
```

```python
class LexingState(enum.Enum):
    INIT = 1
    OPENED = 2
    CLOSED = 3
    READ_FUNC = 4
    READ_SEPARATOR = 5
    READ_STRING = 6
    READ_SINGLE_QUOTE = 7
    READ_DOUBLE_QUOTE = 8
    END_QUOTE = 9
```

```python
def main(spec_file):
    """Read the specification file and represent it in an arbitrary way
    so the user can verify that the file is interpreted appropriately.
    """
    lex = LispLexer(spec_file)

    with open(spec_file) as fd:
        content = fd.read()
        par = LispParser(content)

        for token in lex.feed(content):
            tok, start, end = token
            print(tok, repr(content[start:end]))

            par.consume_token(token)

        spec_tree = par.finalize()

    print()
    print(repr(spec_tree))
```

```python
class LispLexer:
    def __init__(self, source):
        self.state = LexingState.INIT
        self.scalar_id = 0
        self.column_id = 0
        self.line_id = 0
        self.start_string = None
        self.source = source
```

```python
class LexedToken(enum.Enum):
    OPEN_GROUP = 1  # start a new group
    STRING = 2  # gives a new string argument
    CLOSE_GROUP = 3  # terminates a group
    SKIP_WHITESPACE = 4  # whitespace not required to understand semantics
    SKIP_SINGLE_QUOTE = 5  # single quote not required to understand semantics
    SKIP_DOUBLE_QUOTE = 6  # double quote not required to understand semantics
```

Let's go!