SPLIWACA Regular Expressions

A regular expression or regex is a sequence of characters used to scan for substrings of a longer string that fit a certain pattern [[1]](#_Bibliography). Regular expressions are often used to define programming languages [[2]](#_Bibliography), as they can easily define a finite string to fit a pattern. The regular expressions for a language can then be converted to a non-deterministic finite automaton and then a deterministic finite automaton [[3]](#_Bibliography) which can be used to generate a lexical analyser (lexer). Otherwise, regular expressions can also be used during looping through a string to construct the tokens in a hand-coded lexer.

Based on the definition of SPLIWACA at <https://github.com/Starwort/SPLIWACA/blob/master/README.md>, it is theoretically possible to create regular expressions that fully define the language at this point. For example, we can define a comment as:

/\\*(.|\n)\*?\\*/|//.\*

Breaking it down character by character by character:

/ means match literally the character ‘/’

\ escapes the next character, meaning \* means match literally the character ‘\*’

( begins a group

. means match any normal character

| indicates the union of the previous and the next patterns, so both will be matched

\n indicates a new line

) ends the group

\*? means match the previous group zero or more times, as few times as possible to match the pattern. This is known as lazy matching.

\\* means match ‘\*’ literally

/ means match ‘/’ literally

| indicates the union between previous and next patterns

// means match the string ‘//’ literally

. means match any normal character

And finally \* means match the previous pattern zero or more times [[4]](#_Bibliography)

So we can create a full set of regular expressions to check a) if a string is valid code in Spliwaca and b) construct the tokens.

The final function regex that I have created took me a little while to debug and introduced me to something called catastrophic backtracking. Catastrophic backtracking occurs when a regex takes a section of a string, then retracts back over time as it releases the end to subsequent parts of the regex [[5]](#_Bibliography).

Pass through once and construct tokens. For all expression blocks, construct a single token. Then, pass over all expression block tokens and construct tokens. If there is no match in the regex pass for a given block, then the remainder must be an identifier and can be checked to return an identifier token or a lexer error if there is a space in it. I had originally copied my regex for a string literal into the function regular expression, but I forgot that part of my string regex referred to an earlier part. This meant the regex interpreter I am using to test my regular expressions [[6]](#_Bibliography) started backtracking too much and ended up throwing a catastrophic backtracking error. I figured out what the problem was in the end, and pointed it at the correct location. My final regex for functions is below, with the location catastrophic backtracking was occurring at highlighted:

(FUNCTION|FUNC) [a-zA-Z\_][a-zA-Z0-9\_]\* (TAKES ((STR|Str|str|STRING|String|string|INT|Int|int|INTEGER|Integer|integer|FLOAT|Float|float|REAL|Real|real|NUMBER|Number|number|BOOL|Bool|bool|BOOLEAN|Boolean|boolean|COMPLEX|Complex|complex|LIST|List|list|ARRAY|Array|array|TUPLE|Tuple|tuple|DICT|Dict|dict|DICTIONARY|Dictionary|dictionary|MAP|Map|map|MAPPING|Mapping|mapping) .+?)+?)?(\->|RETURNS) (STR|Str|str|STRING|String|string|INT|Int|int|INTEGER|Integer|integer|FLOAT|Float|float|REAL|Real|real|NUMBER|Number|number|BOOL|Bool|bool|BOOLEAN|Boolean|boolean|COMPLEX|Complex|complex|LIST|List|list|ARRAY|Array|array|TUPLE|Tuple|tuple|DICT|Dict|dict|DICTIONARY|Dictionary|dictionary|MAP|Map|map|MAPPING|Mapping|mapping) AS \*?\n+((\t| )\*.+|\n\*)+?\n+ \*?\n\*?(\t| )\*RETURN (([\"']).\*?(?<!\\)\11|(RAW|PLAINTEXT) (.\*)|(\+|\-)?(\d+(\.\d+)?)+?((\+|\-)?(\d+(\.\d+)?)?i)?|((\+|\-)(\d+(\.\d+)?)?i)+) \*?\n\*?(\t| )\*?END (FUNCTION|FUNC)

The \11 at that point indicates that the text there must be the same as in the 11th capturing group – ([\”’]) – so the quotes are matched. After iterating over it a little more, I came to the much simpler

List of all characters on my keyboard:

`\-= \[\]\\;’,./~!@#\$%^&\\*\(\)\_\+ \{\}\|:” <>\?

qwertyuiop asdfghjkl zxcvbnm QWERTYUIOP ASDFGHJKL ZXCVBNM1234567890

# Bibliography

|  |  |
| --- | --- |
| [1] | [Online]. Available: https://en.wikipedia.org/wiki/Regular\_expression. |
| [2] | [Online]. Available: https://en.wikipedia.org/wiki/Lexical\_analysis#Lexical\_grammar. |
| [3] | T. Æ. Mogensen, Basics of Compiler Design. |
| [4] | [Online]. Available: www.regular-expressions.info. |
| [5] | [Online]. Available: https://www.regular-expressions.info/catastrophic.html. |
| [6] | [Online]. Available: regex101.com. |