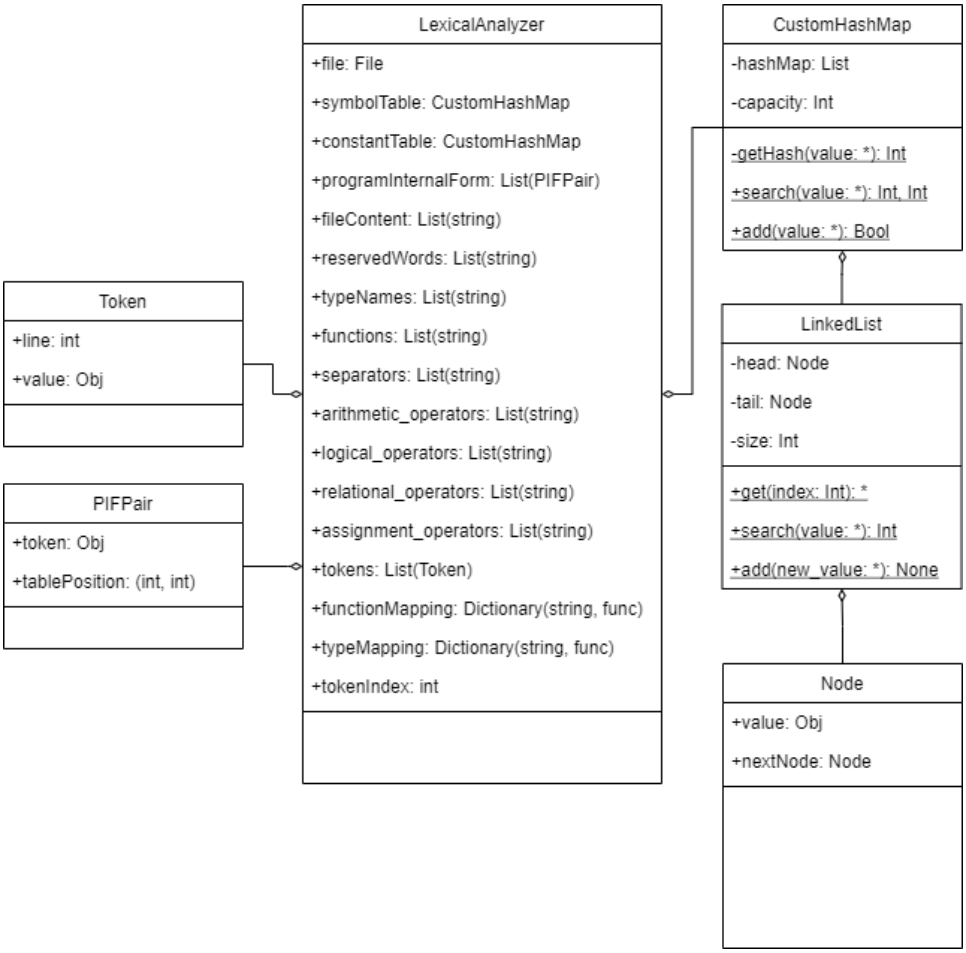
https://github.com/Bizso96/Formal-Languages-Analyzer

Lexical analyzer documentation



Reading inputs:

- token.in should have several lines of input words, separated by commas, in the following order: reservedWords, typeNames, functions, separators, arithmetic_operators, logical_operators, relational_operators, assignment_operators. The name of the line inputs can also be specified, followed by a colon
- read_tokens_input(filename) - reads the token information from the give file. The information will be stored as a list of strings in the corresponding lists
    - ```
      def read_tokens_input(self, filename):
          token_file = open("input/" + filename, "r")
          if self.file.closed:
      ```
      ```

```
            return False

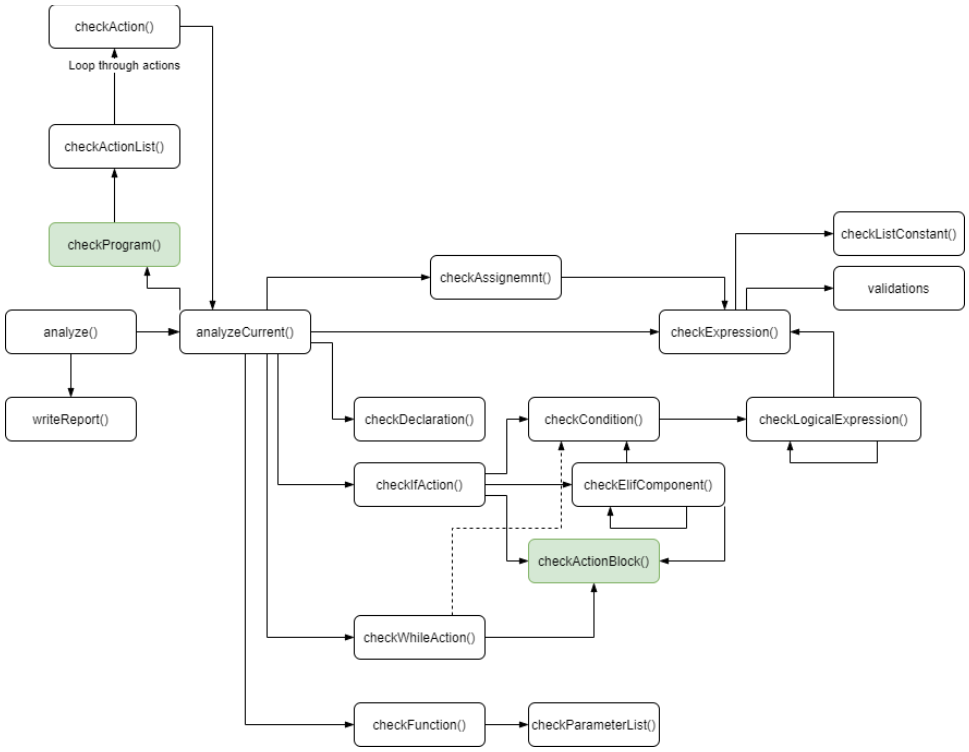
        self.reservedWords = token_file.readline().split(":")[1].replace(' ',
'').strip().split(",")
        self.typeNames = token_file.readline().split(":")[1].replace(' ',
'').strip().split(",")
        self.functions = token_file.readline().split(":")[1].replace(' ',
'').strip().split(",")
        self.separators = token_file.readline().split(":")[1].replace('"',
'').strip().split(", ")
        self.arithmetic_operators =
token_file.readline().split(":")[1].replace('"', '').replace(' ',
'').strip().split(",")
        self.logical_operators =
token_file.readline().split(":")[1].replace('"', '').replace(' ',
'').strip().split(",")
        self.relational_operators =
token_file.readline().split(":")[1].replace('"', '').replace(' ',
'').strip().split(",")
        self.assignment_operators =
token_file.readline().split(":")[1].replace('"', '').replace(' ',
'').strip().split(",")
```

- open_file(filename), close_file(), read_file_content() are used to read the input program
- tokenize() - tokenizes the input program after reading it from a file. The tokenization occurs by splitting the input by whitespaces and separators read from the token input, and also keeping the separators. In order to keep strings as a single token, the expression "..." is also considered a separator
    - ```
      def tokenize(self):
          rePattern = re.compile('(' + '\".*\"|' + '[' + r'\s' +
      '\\'.join(self.separators) + '])')
          l = 0
          for line in self.fileContent:
              l += 1
              initialSplit = rePattern.split(line)
              for i in initialSplit:
                  if i and not i.isspace():
                      self.tokens.append(Token(l, i))
      ```

Lexical and syntactic analysis:

- The lexical and syntactic analysis is initiated by calling the analyze() function. The analysis is done token by token, with tokenIndex marking the token that is currently being resolved
- The reserved words have their own functions, knowing what is expected from the upcoming tokens. Each check function will go through its scope (e.g., checkWhileAction will start with a while keyword, and will stop when reaching the end of its last action block). The analyzeCurrent() functions calls the appropriate functions based on the current token. The functions whose name starts with check... will advance the tokenIndex after checking for
      ```

expected tokens (with the exception of checkList, they will also step out of their scope, so the next function called should presume that it is starting in its own scope)

- During the analysis a symbol table, a constant table and a program internal form is constructed
- There are several functions that are used to validate certain elements (validateVariable, validateTypeName, validateIntegerConstant, validateBooleanConstant, etc.). These do not advance the tokenIndex
- When a lexical is encountered, an error message will be displayed. If the end of the program is reached without encountering errors, the message “Lexically correct” is displayed

Report:

- After the analyis is complete, or a lexical error is encountered, a report can be written by calling the writeReport() function. This will write in PIF, the symbol table and the constant table in separate output files



Function flow

Examples

```
enter
    integer a;
    a <- 45;

    integer b;
    b <- 90;

    a <- a + b;

    if(a % 2 == 0){
    write(a);
    };
exit
```

```
Lexically correct
```

In case of a program without errors, the message “Lexically correct” is displayed.

```
enter
    integer k5$;
exit
```

```
Lexical error on line 2: Wrong variable format
```

For wrong variable formats, the error message “Wrong variable format” is displayed

```
enter
    integer i;
    i <- 45g;
exit
```

```
Lexical error on line 3: Unknown element in expression: 45g
```

For unknown constant formats the message “Unknown element in expression: ...” is displayed. This can generally happen in expressions.

Syntactic errors also have their messages based on the expected tokens, but syntactic analysis is not in the scope of the exercise.