جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

CSC 3315

Project - Part 2: Lexer

Prepared by:

DRIOUECH Saad

EL AMRANI Omar

LAMIRI Fatima Zahrae

Submitted by team captain:

Lamiri Fatima Zahrae

Supervised by:

Dr. Violetta Cavalli Sforza

Spring 2021

I. Team Dynamics:

Learning from the experiences of the past project parts, our team has developed much needed synergy and learned how to work more effectively. This part of the project, however, didn't have a clear task distribution from the get-go, as most of the work is concentrated in creating the lexer, which is a monolithic task that is hard to divide. As such, we met up on both Tuesday and Wednesday (20th and 21th of April), and worked and discussed as a group our implementation choices for the Lexer. For the miscellaneous tasks, such as creating user-generated code and writing the project's documentation, they were distributed evenly between the team members. Omar El Amrani and Imane Iraoui were tasked with writing the documentation for the Lexer, while Saad Driouech and Fatima Zahrae Lamiri worked on the user-generated code.

Overall, the challenge with this part for our group was mostly the monolithic aspect of the project which left little room for task distribution. It was hard for us to adapt to this new style as there were some clashes when it came to the minute details of the implementation of the Lexer. However, we were able to overcome these challenges for the sake of the overall success of the team.

II. Changes made compared to the previous phase:

- We added delimiters to the strings in order to differentiate between them and other user identifiers. The delimiter in this case is a "$" sign. We also allow for the use of spaces in strings.
  Modified regEx: $[a-zA-Z_0-9]+$

- We added delimiters to characters for the same reason. The delimiters in this case is a single quote, so the character starts and ends with a single quote.
  Modified regEx: '[a-zA-Z_0-9]'

- We changed the reserved word equivalent to 'else' from "ila ghalate" to "ilaghalate" attached.

- We added a mandatory space after any operator in order to differentiate between a sign and an operator (which causes a problem in the case of "+" and "-").
  Modified BNF:
  <expression> ::= <expression> <operator><space> <expression>| <operand> | (<expression>)
  <space> ::= ' '

III. Lexer Documentation

- **Choices of Implementation and Justification:**

We chose to work with Python, mostly because all the team members have worked with Python in the past and are familiar with it compared to the other choices offered. We thought it would be best to go for it to optimize the time spent on this phase. Moreover, it has a built-in module (re) that handles regular expressions. As such, it is heavily documented and explained in online resources, which will be of much help for us when it comes to running into roadblocks while using it.

- **Lexer Design:**

General description:
The job of the lexer is to scan the code and turn the characters into a sequence of tokens. To illustrate this point, let's take the example of "437+756". The lexer's job is to understand that 437 are all part of the same lexeme which is a numeric literal, it then looks at the symbol + and understands that it's a different lexeme that represents an operand, and finally it considers 756 as a numeric literal as well. The lexer also has the job of building the symbol and literal table. These two are used so that the lexer is able to recognise both user-defined identifiers and values used in the p.

Data structures:

→ Lists:

- literalTable: it implements the literal table, which contains literals encountered in the code.

→ Dictionaries:

- symbolTable: it implements the symbol table, which contains all the symbols used in our program (i.e identifiers; variable names, function names…) along with their type.
- tokensDict: it maps every lexeme in the language to its corresponding token.

Functions:

→ scanInt: keeps scanning characters in the file until it finds a character that's not a digit. This function is used to read numbers in our program. It returns a string which holds all the consecutive digits (i.e one integer). This function takes as an input arguments the input file pointer (to access the file and read from it), a character which is the current character, and temp which is a string used to store the whole numeric literal.

→ scanString: keeps scanning characters in the file until it finds the delimitors that we have set to "$". This function is used to read strings in our program. It returns a string which holds all the consecutive alphanumeric and allowed characters. It returns a string that holds all these consecutive characters (i.e the string). This function takes as input arguments the string delimiter and the file pointer (to read from the file).

→ scanComment: keep scanning characters in the file until it finds the comment delimiter which is set to "!!". This function is used to read comments in our program, and it's called after we encounter "!!" which indicates the start of a comment. The function doesn't return anything and it's mainly used to **get past the comments in our program.** This function takes as input argument the file pointer in order to read from the file.

- **User Manual:**

The program is written in Python 3, specifically Python 3.8.2. The code is organized into 3 different files:

- main.py: This file contains the lexer code, along with all the necessary functions.
- code.txt: This file contains the program code (the input of the program). The format of the input is just a code written in our language (specified in the previous phase).
- output.txt: This file is generated after running the lexer. It contains the output of our code, which is formatted as follows:

Line 'lineNumber' Token '#token': 'lexeme'

The text files are assumed to be all located in the same folder, which is the same folder that contains the main.py file.

To run the lexer, the main.py file needs to be run. We personally used repl.it to write and test our code, but any other IDE should work as long as Python 3 is installed.

- **Requirements Check:**

Element	How it is matched
White space	`re.match(" ", c)`: This function checks if the current character is a white space
New line	`re.match("\n", c)`: This function checks if the current character is a new line, if that's the case then it prints that a new line was encountered.
Punctuation	`re.match("[;|,|:]", c)`:This function checks if the current character is a punctuation character, and if it is the case it finds its token in the dictionary and it prints it with the current line of code and the character.
Numeric literals	`re.match("[0-9]", c)`: using this function, we first check if the current character is a numeric

	literal. Then reads the next character and calls the function scanInt to get the rest of the number if it exists, and we print the line number, the token, and the numeric literal.
Operators	re.match("[=|<|>|!|+|*|/|%|-]", c): This function checks if the current character matches any of the operators. If yes, then we handle the following cases: - If it's "+" or a "-": we check whether there is a space after it. If yes, then that means it's an operand. Otherwise, it means that it's a sign so we read the whole thing as a numeric literal by calling scanInt(). - If it's "=": We check whether it's followed by ">" which signifies an assignment, otherwise we tokenize it as equal. - If it's "!": We check whether it's followed by another "!", if so then it designates the start of a comment so we read it by calling scanComment. Otherwise, if it's followed by "=", we tokenize it as not equal. - If it's "<" or ">": We check whether it's followed by "=", if so we read the whole thing as "<=" and ">=" respectively. Otherwise we consider them just as "<" and ">". In all cases, we print the line number, the token, and the character.
Identifiers and reserved words and logical operations.	re.match("[a-zA-Z_]", c): This function checks if the character is an alphabet(either capital or small letter) or an underscore. If that is the case, we keep reading characters and appending them until we reach a character that is not an alphabet, underscore, or number. Then we check for three cases: - if the string constructed by appending those characters is in the logical operators list, we find its special token matched by the dictionary, and we print it. - if it is in the reserved word list, we find its special token in the dictionary then we print. - if it is none of the above, we assume it is an identifier, so we print the token of IDs.
Character literals	We check if the current character is a single quote (the delimiter of character literals). If yes, we read the next character, and we check if it is a single character literal using

	`re.match("[a-zA-Z_0-9]", c)`. Then we read another character and we check if it is a single quote. If yes, we tokenize the character between single quotes as a character literal. We print the line number, the token, and the character
String literals	We check if the current character is a "$" sign. If so, then this designates the start of a string literal. We then scan the characters using `scanString` and we print the scanned string in case a delimiter, i.e "$", is found at the end, along with the line number and the token.
Parentheses and brackets	We check if the current character is in the list "(", ")", "[", "]". If it is the case, we match the character by its own token, and we print the line number, the token, and the character.