FLCD – Lab 4 – Documentation

Lung Alin-Sebastian

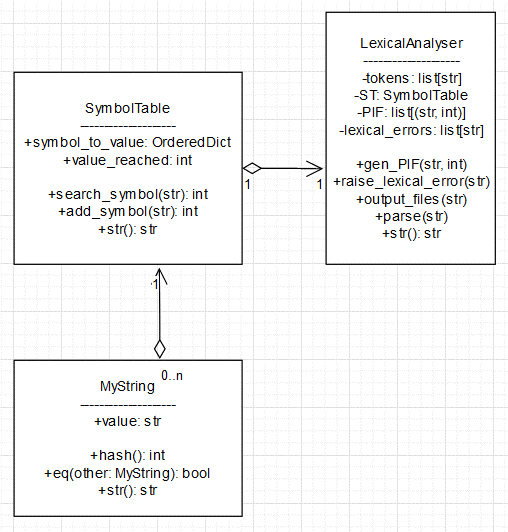
Gr. 934/2

Source code: [https://github.com/IcerOut/FLCD](https://github.com/IcerOut/FLCD/releases/tag/Lab-2)

Problem statement:

Implement a symbol table and a scanner (lexical analyzer) for your minilanguage.

Application Design:



Implementation details:

SymbolTable:

The function “search\_symbol” takes in the symbol and returns the value associated to it or “-1” if the symbol doesn’t appear in the SymbolTable.

The function “add\_symbol” takes in the symbol and, if it already exists, returns its associated value. If it does not, it adds it to the hashtable and returns its newly associates value.

The implementation uses, internally, a Python dict object. It has a complexity for search and add of O(1)

Python’s dict uses the \_\_hash\_\_() method of whichever object it hashes. MyString’s hash function uses the sum of ASCII values % 65536 as the hash value.  
In case of hash collisions, Python’s dict uses open addressing using probing. Initially it has 8 free position and it gets resized when it reaches 2/3 of its maximum capacity

LexicalAnalyser:

The LexicalAnalyser() class contains the list of tokens (read from token.in), a Symbol Table of type SymbolTable(), a PIF (a list) and a list of encountered lexical errors (a list).

When parsing, we open the program file and iterate through it line by line.

We remove all CR (\r) characters to simplify parsing.

Then we split the line into tokens using re.split(). For the pattern, we use '([ \t\n()\\[\\]{},;#\'\"])'

We basically split the string by each delimiter, but keep those delimiters in the tokens list as well (using regex capture groups)

Then we iterate through the tokens and clasify them as follows:

* If we are currently in a string constant and the token we encountered is not the string delimiter, we add it to the string constant we are building
* If we encounter a ‘#’, the rest of the line is a comment so we skip to the next line (So the content of the comments is ignored)
* If we encounter a single or double quote:
  + If we are not in a string constant, we start one
  + If we are in a string constant and it is the same character that started our current string, it means we just reached the end so we add the built string to the ST and PIF and then mark that we are no longer in a string constant
  + Otherwise, it’s a single quote inside of a double-quote string (or viceversa), so we just add it to the string we are building as normal
* If we encounter a space or tab, we skip it (and we do not add it to the PIF)
* If we encounter a newline (LN; ‘\n’):
  + If we are in a string constant, we raise a Lexical Error because it means the string was unfinished
  + Otherwise, we skip it (and do not add it to the PIF)
* If it’s in the token list, it means it’s a keyword, operator or separator, so we add it to the PIF with index 0
* If it mathches the pattern '^0|(-?[1-9][0**-**9]\*)$' then it’s a numerical constant. This pattern should match all positive or negative whole numbers. We add it to the ST and the PIF
* If it matches the pattern '^[a**-**zA**-**Z][\_a**-**zA**-**Z0**-**9]\*$' then it’s an identifier. This pattern should match all sequences of letters, numbers or underscores that start with a letter. We add it to the ST and the PIF
* If it doesn’t match any of these branches, it’s an unexpected token so we raise a Lexical Error

Test cases:

Input: “p1.Z”  
Output: “p1.Z PIF.out” and “p1.Z ST.out”

Input: “p2.Z”  
Output: “p2.Z PIF.out” and “p2.Z ST.out”

Input: “p3.Z”  
Output: “p3.Z PIF.out” and “p3.Z ST.out”

Input: “p1err.Z”  
Output: “p1err.Z PIF.out” and “p1err.Z ST.out”