Lab3

https://github.com/AnastasiaSusciuc/UBB/tree/main/Anul 3/Sem5/FLCD/Labs/Lab2 (both the ST and the scanner are here)

SCANNER:

- scan → scans the source code, builds the PIF and the ST's (one separate ST for constants and identifiers) and writes them in three files: "pif.txt", "st_identifier.txt", "st_constant.txt; it also prints the errors, if there are some lexical errors; The scanning algorithm splits each line of the program into tokens, and for each token has a specific alogrithm for eg if it's a constant or identifier, it looks up its position in the ST, if it's an operator/separator/reserved word, its position is (-1,-1). Also, if it's a constant or identifier, instead of keeping the variable name/constant value, it will be added into the PIF with the code "const" or "id". If the token is none of the above, that means we have a lexical error at that line, and the error is appended to the message.
- <u>_is_constant</u> → checks if a token is a constant using regex
- <u>_is_identifier</u> → checks if a token is an identifier using regex
- add exception message → adds a new lexical error to the list of all lexical errors
- _write_scan_output → writes to file the output of the scanning
- <u>__inside_operator</u> → checks if a character is part of an operator
- <u>__get_operator_token</u> → gets the operator token
- <u>__get_string_token</u> → gets the token that forms a string
- tokenize → transforms a string in tokens; The tokenizing algorithm goes character
 by character on each line and checks whether the current character is part of an
 operator is a separator, begins a string or is building a constant or identifier, and
 then appends the tokens to a list which is returned.
- <u>__get_tokens</u> → reads the tokens from a file (token.in)

Lab3

PIF

Is implemented using a list, which has elements of the form (token, position), where position is a pair representing the position from the ST (for constants and identifiers) or (-1, -1) for the other valid tokens.

SymbolTable

- is based on a Hash Table
- add/ remove has O(1) amortized complexity
- uses simple chaining (each bucket has a list that solves the collisions)
- I used the hash function from https://cp-algorithms.com/string/string-hashing.html, where p and m are some constants

$$\operatorname{hash}(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m$$

$$= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,$$

• in my implementation, two symbol tables will be instantiated, one for constants, one for identifiers

add(self, key):

```
adds an element into the hashtable
:param key: the value of the element
:return: the hash value and the position of the element in its list
"""
```

remove(self, key):

```
removes the key from hashtable
```

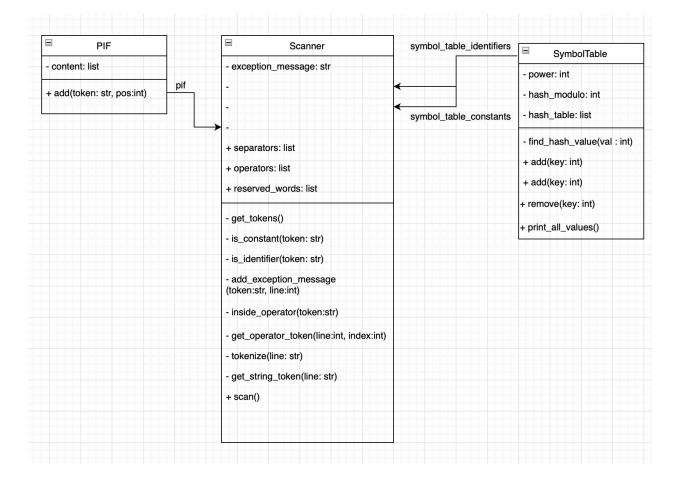
Lab3 2

```
:param key: the element to be removed
:return: the hash value
```

exists(self, key):

```
"""
:param key: the element we are looking for
:return: True if key is inside the hash table
"""
```

UML DIAGRAM



Lab3