https://github.com/the-coding-cloud/FLCD/tree/main/Lab3 - GitHub repository

Lab 3 – Documentation

Purpose

The purpose of this lab was to implement a scanning algorithm for the language specified during Lab1, using the Symbol Table from Lab 2

Implementation

The algorithm goes through the program code line by line and does the following steps:

1. Tokenize
2. Classify
3. Codify

Each line is split by spaces, separators and operators. We have to take into consideration the distinction between +/- as binary operators and unary operators.

In the classification step, I check if each token falls into one the following categories: ***separator***, ***operator***, ***keyword***, ***constant*** or ***identifier***. If the token cannot be classified, we have a lexical error and it is printed to the console which token led to the error and the line.

Then, if the token is a constant or identifier, we add it to the symbol table (if it is not already there) and then add the corresponding code of the token + its position in the symbol table to the PIF.

When a Scanner object is instantiated, the separators, operators and keywords of the language are loaded into the "tokenCodes" field. At the end, the PIF and ST will be written to a file of the form *program_name.out*, in order to easily distinguish between the outputs after scanning multiple input programs.

In my implementation, the Scanner class has 6 private fields:

- st: a Hashtable object, representing the symbol table
- operators: a list containing all the operators from the language
- separators: a list containing all the separators from the language
- keywords: a list containing all the keywords from the language
- tokenCodes: a dictionary containing the codification of the tokens from the language
- pif: a list containing all the PIF entries, which are of the form [code, STposition]

In addition, the class has 1 public method named ***scan***, that receives as a parameter a *fileName* of type *string* – this is the function that is called in order to do the lexical analysis of the program with the given *fileName*. The other methods are private and represent smaller chunks from the scanning algorithm. **writeScanningOutput** method is used to provide the output file containing the ST and PIF.

Interface:

- **scan:**
  - input: fileName - string
  - output: PIF – list, ST – Hastable object
  - preconditions: fileName must be a valid filename from where the program is read
  - postconditions: the content of the PIF and ST will be written to a file ending in .out
- **writeScanningOutput**
  - input: fileName – string, message – string
  - output: -
  - preconditions: fileName should be the filename from where the program is read, while message should be None in case no lexical error occurs or a string indicating the line and the token where the error occured
  - postconditions: the content of the PIF and ST will be written to a file ending in .out
- **tokenize**
  - input: string – string
  - output: tokens – list of strings
  - preconditions: the input parameter should be a string
  - postconditions: tokens will contain a list of tokens obtained after splitting string into tokens by separators, operators and " "; separators and operators will also be included in the list
- **classify**
  - input: token - string
  - output: tokenType - integer
  - preconditions: the input parameter should be a string
  - postconditions: returns -1 if it's a separator/operator/keyword, 0 if it's an identifier, 1 if it's a constant and None if it cannot be classified into the previous categories
- **codify**
  - input: token – string, tokenType – integer
  - output: -
  - preconditions: the token should be a string and the tokenType should be -1, 0 or 1
  - postconditions: the token will be added to the PIF as a list of the form [tokenCode, STposition]
- **isOperator**
  - input: token – string
  - output: True/False
  - preconditions: the token should be a string
  - postconditions: the function returns True if the token is an operator and False otherwise
- **isSeparator**
  - input: token – string
  - output: True/False
  - preconditions: the token should be a string
  - postconditions: the function returns True if the token is a separator and False otherwise

- **isKeyword**
    - input: token – string
    - output: True/False
    - preconditions: the token should be a string
    - postconditions: the function returns True if the token is a keyword and False otherwise
- **isInt**
    - input: token – string
    - output: True/False
    - preconditions: the token should be a string
    - postconditions: the function returns True if the token can be converted to an integer and False otherwise
- **isConstant**
    - intput: token – string
    - output: True/False
    - preconditions: the token should be a string
    - postconditions: the function returns True if the token is a valid constant in the mini language and false otherwise
- **isIdentifier**
    - input: token – string
    - output: True/False
    - preconditions: the token should be a string
    - postconditions: the function returns True if the token is a valid identifier in the mini language and false otherwise
- **processTokens**
    - input: fileName – string
    - output: -
    - preconditions: fileName should be a string representing a valid file containing the tokens from the language and their code
    - postconditions: the tokens and their codes should be stored in the private fields of the Scanner class according to their category

Scanner Class Diagram