Documentation

The github link to my repository:

https://github.com/VescanAntonia/University-Computer-Science/tree/main/Semester%205/Formal%20languages%20and%20compiler%20design

I implemented a SymbolTable with a given capacity, a fixed prime number. It is designed to work on a collection of strings, divided in separate buckets. Each bucket is an ArrayList and the SymbolTable provides methods to add, look for an element’s position and two hash functions for an element that we want to add.

The hash function for the String elements receives an element and returns the position of the bucket that the element is supposed to go into. It first sets the value of the hash to 0, then for each character in the key, we multiply the current hash by 31 (a prime number) and then add the ASCII value of the character. After that, we take the absolute value of hash to ensure it's non-negative, and then calculate the modulo with the capacity to determine the bucket index. The hash function for the Integer elements has the same purpose, the only difference is made by the way the position is calculated, which in this situation is the key%capacity.

The add function receives an element and has the purpose to add the element to the SymbolTable to the position given by the hash function. The function returns false if the element is already in the SymbolTable. If the element is not already in the SymbolTable we get the position of the element by applying the hash function on it. We get the bucket that corresponds to the optained position and add the given element. Then we return true if the element was added.

The lookUp function receives an element and has the purpose to look for the given element in the SymbolTable. Firstly, we apply the hash function on the element to get the bucket id in which it is supposed to be. Then we get the bucket corresponding to the position we got. We iterate through all the elements of the bucket and check if the given element equals the current element in the bucket list. When we find the element we return the position in the bucket list. If we dont find the element we return -1.

ProgramInternalForm is represented as a ArrayList<List<Object> to store the mapping between tokens and its position or -1 in case of reserved words. I have a getProgramInternalForm() which returns the PIF and addToPif(K token, V pos) which adds the given key and value to the PIF. This class also has a toString method which transforms the PIF into a StringBuilder to be printed.

*Scanner*

public class Scanner {

private String program; //the source code

private final List<String> tokens;

private final List<String> reservedWords;

private SymbolTable<String> identifiersST;

private SymbolTable<Object> constantsST;

private int index = 0;

private ProgramInternalForm<String,Integer> PIF;

private int currentLine = 1;

* initializeTokens(String pathFile) – loads and classifies the tokens into reserved words or tokens after reading them from file;
* isReservedWord(String token) - checks if a token is in the reserved words set
* writeOutputToFile(String fileName, String resultedOutput) - writes output to a given file;
* setProgram(String program) - sets the source code program content;
* loadProgram(String programFileName) - loads the program from a file and calls setProgram to set it;
* writeResultsToFiles(String programFileName) - writes contents of PIF,identifiersST, constantsST to corresponding output files;
* scan(String programFileName) – loads the program from the given file and performs lexical analysis on it;
* nextToken() – skips the whitespaces, then identifies and processes the next token in the program and raise an error if an invalid token was found;
* addConstantToST(String constant) - adds a constant to the constants symbol table and return its position;
* addIdentifierToIdentifierST(String identifier) - adds an identifier to the identifier symbol table and return its position;
* checkIntConstant() – identifies and processes integer constants. It uses a regular expression to match integer constants in the program text: ^([+-]?[1-9][0-9]\*|0) which matches the patterns:
* + or - (optional sign)
* a non-zero digit (1-9)
* zero or more digits (0-9)

this method then checks if a match is found using matched.find() and stores it in intConstant. Then increments the index in order to continue the scanning, add the intConstant to the constantsST and the mapping to its position in the PIF.

* checkStringConstant() – identifies and processes string constants. It uses a regular expression to match string constants in the program text: ^\"[a-zA-z0-9\_ ?:\*^+=.!]\*\" which matches the patterns:
* a double quote character (") to indicate the start of a string constant
* zero or more characters, including letters (both lowercase and uppercase), digits, and various symbols
* another double quote character (") to indicate the end of the string constant.

this method then checks if a match is found using matched.find() and stores it in stringConstant. Then increments the index in order to continue the scanning, add the stringConstant to the constantsST and the mapping to its position in the PIF.

* checkIdentifier() – identifies and processes identifiers during lexical analysis. It uses a regular expression to match identifier in the program text: ^([a-zA-Z\_][a-zA-Z0-9\_]\*) which matches the patterns:
* an identifier must start with a letter (either uppercase or lowercase) or an underscore (\_);
* the remaining characters can include letters (uppercase or lowercase), digits, and underscores

this method then checks if a match is found using matched.find() and stores it in identifir variable. It calls the method checkIfValidToken() and checks the validity of the identifier. Updates the index in order to continue the scanning, then adds the identifier to the identifiersST and maps the corresponding position in the PIF.

* checkFromTokenList() – identifies and processes tokens that are recognized from predefined lists, reservedWords and tokens, during lexical analysis. It extracts the substring of the program text starting from the current index and splits it by spaces to obtain the first word/token in the substring. This possibleToken represents a potential token to be matched. For each reserved token in the list checks if the possible token starts with it and if it is tha case, creates a regex with which to check whether the possible token matched the regex and has anything before/after it. If it matches the regex it means that it is an invalid token and return false. Otherwise it maps the reservedToken to position -1 in the PIF and continues the iteration. After this it iterates through the token list and check if the possibleToken matches the current token in the list or if it starts with it. If it does it modifies the index in order to continue and mapps the token to -1 in the PIF.
* skipAndNextLine() - skips over the whitespaces and the new lines iterating currentLine and index
* checkIfValidToken() – checks if a given token is a valid token. It first checks it to not be a reserved word and then it uses a regex to see if it can be a valid token. The regex "^(?![0-9])[a-zA-Z\_][a-zA-Z0-9\_]\*" which represents:
* ^: Start of the string.
* (?![0-9]): A negative lookahead assertion that ensures the identifier does not start with a digit.
* [a-zA-Z\_]: The first character of the identifier must be a letter (uppercase or lowercase) or an underscore.
* [a-zA-Z0-9\_]\*: Zero or more characters following the initial character, which can be letters (uppercase or lowercase), digits, or underscores. Otherwise it uses the lookUp method and returns true/false if it finds the token in the identifiers list.

