

Github Repo : <https://github.com/ciprianturcu/University-Projects/tree/main/Semester5/Formal%20Languages%20and%20Compiler%20Design/lab4/src>
Output can be found under folder outpifst.

Documentation – Symbol Table:

The Symbol Table is composed as a Hash Table. Both the Symbol Table and Hash Table are generic implementations such that they could be used for integer, string, or any other types. The Hash Table uses open addressing with linear probing to solve common collision problems and uses a rehash when the load gets over a threshold therefore, we are not constrained to a specific capacity. From the implementation of the Hash Table the Symbol Table's elements are stored as only keys representing the given symbol and the index in which they are placed by the hash function as the position.

Operations:

Hash Table :

- hashFunction(K key) – computes the position in the hash table in which the given key should be placed
- rehashTable() – rehashes the entire table to increase capacity
- insertNode(K key) – insert a key into the hash table and return its position when the operation is successful
- getPosition(K key) – get the index in the hash table where the key is located
- getByposition(int position) – get the key from the hash table at the given position
- getCapacity() – get table capacity.

Symbol Table:

- getHashTable () – get the table
- add() – add a symbol to the table and return its position
- getPosition(K symbol) – return the position in the table of the symbol
- getByPosition(int position) – return the symbol from the given position

Documentation – Scanner Exception:

The Scanner Exception class is a wrapper of RuntimeException.

Documentation – Custom Scanner:

The Custom Scanner class is building the program internal form and the 2 symbol tables, one for identifiers, one for constants; while checking if the input program is lexically correct or not. The program internal form allows us to track the program tokens (int constants, string constants, char constants, identifiers, separators, operators or reserved words), it is formed by a list containing pairs of the token and that token's position in their respective symbol table. For tokens that do not fall under the requirements for an identifier or constant the symbol table position is considered to be -1. The scanner also maintains a current line read from the program file as well as a linePosition and a lineCounter such that when encountering an error, it can point to that error throwing a new ScannerException. Also the scanner has 3 lists containing the elements for reserved words, separators, and operators.

Operations:

- scanProgram(String filename) – starts the scanning process of a (given by filename) program.
- processLine() – sets the new values for line counter and position and starts the search for the tokens within that line.
- findNextToken() – makes the passing through all cases and if no case is met throws an exception for lexical error.
- treatSpaces() – skips spaces.
- intConstClassifier() : Boolean – checks if we have an int constant in the current program line. If the number represents a valid int constant it adds it to the constants Symbol Table (if it wasn't added before) and it adds it to the program internal form, updates the linePosition and returns true. For an invalid int constant we return false;
- stringConstClassifier() : Boolean - checks if we have a string constant in the current program line. If the number represents a valid string constant it adds it to the constants Symbol Table (if it wasn't added before) and it adds it to the program internal form, updates the linePosition and returns true. For an invalid string constant, we return false;
- charConstClassifier() : Boolean - checks if we have a char constant in the current program line. If the number represents a valid char constant it adds it to the constants Symbol Table (if it wasn't added before) and it adds it to the program internal form, updates the linePosition and returns true. For an invalid char constant, we return false;
- identifierClassifier() : Boolean – checks if we have an identifier in the current program line. If the identifier is valid it is added to the identifier Symbol Table (if it wasn't added before) and the program internal form, updates the linePosition and returns true. For an invalid identifier, we return false.
- tokenClassifier() : Boolean - returns true if the next token is a reserved word, operator or separator.
- checkTokenForOperators(String token) : Boolean – checks if the token is found within the defined operator list or if it starts with an operator from the list, in case we find it we add the length of the operator to the linePosition, add the operator to the program internal form and return true. If the token cannot be classified as an operator from the defined list, we return false.
- checkTokenForSeparators(String token) : Boolean – checks if the token is found within the defined separator list or if it starts with a separator from the list, in case we find it we add the length of the separator to the linePosition, add the separator to the program internal form and return true. If the token cannot be classified as a separator from the defined list, we return false.
- checkTokenForReservedWords(String token) : Boolean - checks if the token starts with a reserved word from the list, in case we find it we check if it is a valid reserved word, then we add the length of the reserved word to the linePosition, add the reserved word to the program internal form and return true. If the token cannot be classified as a reserved word from the defined list, or is not valid, we return false.
- createOutputOfScan() – creates the output files.
- createPaddedString(String firstString, String secondString) – used in creating the output text of scanning the program.

