Github Repo : https://github.com/ciprianturcu/University-Projects/tree/main/Semester5/Formal%20Languages%20and%20Compiler%20Design/week6-7-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:

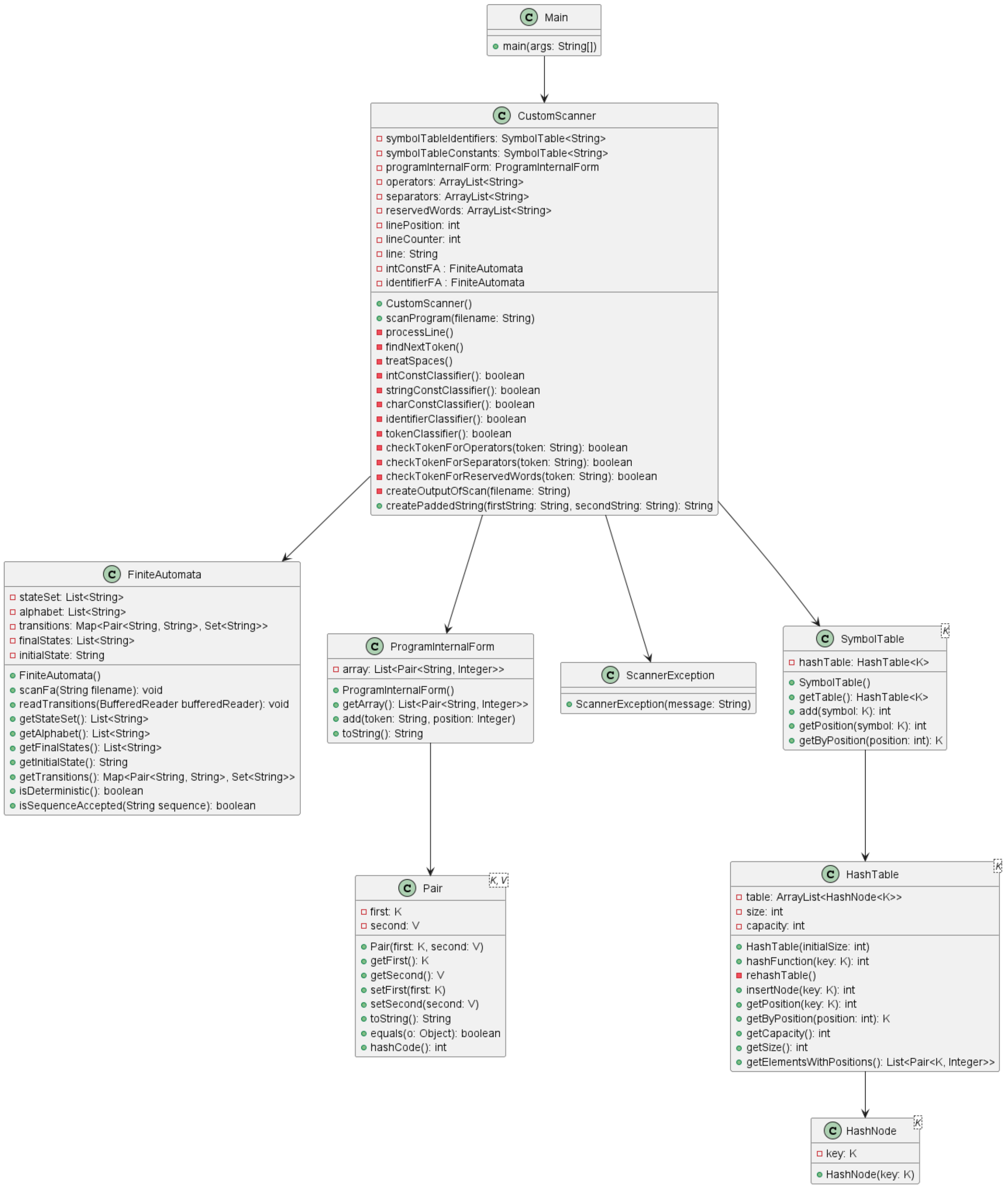
The Scanner Exception class is a wrapper of RunTimeException.

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 an 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 an 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.


Main
main(args: String[])
CustomScanner
symbolTableIdentifiers: SymbolTable<String>
symbolTableConstants: SymbolTable<String>
programInternalForm: ProgramInternalForm
operators: ArrayList<String>
separators: ArrayList<String>
reservedWords: ArrayList<String>
linePosition: int
lineCounter: int
line: String
intConstFA : FiniteAutomata
identifierFA : FiniteAutomata
CustomScanner()
scanProgram(filename: String)
processLine()
findNextToken()
treatSpaces()
intConstClassifier(): boolean
stringConstClassifier(): boolean
charConstClassifier(): boolean
identifierClassifier(): boolean
tokenClassifier(): boolean
checkTokenForOperators(token: String): boolean
checkTokenForSeparators(token: String): boolean
checkTokenForReservedWords(token: String): boolean
createOutputOfScan(filename: String)
createPaddedString(firstString: String, secondString: String): String
FiniteAutomata
stateSet: List<String>
alphabet: List<String>
transitions: Map<Pair<String, String>, Set<String>>
finalStates: List<String>
initialState: String
FiniteAutomata()
scanFa(String filename): void
readTransitions(BufferedReader bufferedReader): void
getStateSet(): List<String>
getAlphabet(): List<String>
getFinalStates(): List<String>
getInitialState(): String
getTransitions(): Map<Pair<String, String>, Set<String>>
isDeterministic(): boolean
isSequenceAccepted(String sequence): boolean
ProgramInternalForm
array: List<Pair<String, Integer>>
ProgramInternalForm()
getArray(): List<Pair<String, Integer>>
add(token: String, position: Integer)
toString(): String
ScannerException
ScannerException(message: String)
SymbolTable
K
hashTable: HashTable<K>
SymbolTable()
getTable(): HashTable<K>
add(symbol: K): int
getPosition(symbol: K): int
getByPosition(position: int): K
Pair
K, V
first: K
second: V
Pair(first: K, second: V)
getFirst(): K
getSecond(): V
setFirst(first: K)
setSecond(second: V)
toString(): String
equals(o: Object): boolean
hashCode(): int
HashTable
K
table: ArrayList<HashNode<K>>
size: int
capacity: int
HashTable(initialSize: int)
hashFunction(key: K): int
rehashTable()
insertNode(key: K): int
getPosition(key: K): int
getByPosition(position: int): K
getCapacity(): int
getSize(): int
getElementsWithPositions(): List<Pair<K, Integer>>
HashNode
K
key: K
HashNode(key: K)

Documentation – Finite Automata

This FiniteAutomata class is a finite automation which implements reading and checking functions on an automaton, that is read from a file with specific input. The input file has a specific structure and contains the states, alphabet, initial state, final states and transitions.

Operations:

- scanFa(String filename) – constructs all the necessary data by reading the input file and identifying the states, alphabet, initial state, final states and transitions.
- readTransitions(BufferedReader bufferedReader) – used to read all the transitions from the file (transitions are kept on different lines at the end of the file).
- getStateSet() : List<String> - returns all the states of the automaton.
- getAlphabet() : List<String> - returns the alphabet of the automaton.
- getInitialState() : String – returns the initial state of the automaton.
- getTransitions() : Map<Pair<String, String>, Set<String>> - returns the transition of the automaton, as a map between the start state the path and all the end states that respect that start,path pair.
- isDeterministic() : Boolean – returns true if for any given start state, path pair there is no more than 1 end state meaning for that path we only have a transition. Else false.
- isSequenceAccepted(String sequence) – first verifies that the automaton is deterministic using the isDeterministic() function, then if the sequence can be parsed using the transitions starting from the initial state and ending in one of the final states then the function returns true. Else false. (sequence needs to be represented as an continuous character sequence that only has characters from the given alphabet).

Input File Format :

<file> ::= <states>\n<alphabet>\n<initialstate>\n<finalstates>\n<transitions>

<states> ::= ~states\n<statelist>

<statelist> ::= <state> | <state> <statelist>

<state> ::= A | B | ... | Z

<alphabet> ::= ~alphabet\n<pathlist>

<pathlist> ::= <path> | <path> <pathlist>

<path> ::= a | b | ... | z

<initialstate> ::= ~initialState\n<state>

<finalstate> ::= ~finalStates\n<statelist>

<transitions> ::= ~transitions\n<transitionlist>

<transitionlist> ::= <transition> | <transition>\n<transitionlist>

<transition> ::= <state> <path> <state>