COMPILERS PROJECT PHASE 1 REPORT

DATA STRUCTURES & ALGORITHMS USED

STATE CLASS

A class implementation of a state in the state machine.

MEMBERS

- `m_Id`: a unique identifier for each state
- `m_Transitions`: a map of transitions to other states given a certain character as input

METHODS

- `addTransition`: a method to add a transition from this state to another given an input
- `addTransitions`: a method to add a transition from this state to a list of states given an input

NFA CLASS

A class implementation of a non-deterministic finite state machine.

MEMBERS

- `m_StartState`: the starting state of the NFA
- `m_TerminalStates`: the set of terminal states of the NFA

METHODS

- `concatenate`: a method to concatenate an NFA to this NFA using Thompson's method where all the transitions from the second NFA's start state are set as valid transition to the first NFA's combined terminal state.
- `unionize`: a method to create a union of an NFA with this NFA using Thompson's method where a new starting state is created with an epsilon transition to both NFAs start states and a new terminal state is created with an epsilon transition from both NFAs terminal states.

- **`kleeneClosure`**: a method to create a kleene closure of this NFA using Thompson's method where new starting and terminal states are created. An epsilon transition is created from the new start state to the original start state and the new terminal state. Another epsilon transition is created from the original terminal state to the original start state.
- **`positiveClosure`**: a method to create a positive closure of this NFA using Thompson's method where new starting and terminal states are created. An epsilon transition is created from the new start state to the original start state. Another epsilon transition is created from the original terminal state to the original start state.

DFA CLASS

A class implementation of a deterministic finite state machine.

MEMBERS

- **`m_StartClosure`**: the epsilon closure of the starting state of the corresponding NFA
- **`m_TerminalClosures`**: the epsilon closure of the terminal states of the corresponding NFA
- **`m_Table`**: the table of transitions of each epsilon closure in the DFA
- **`m_NFATerminalStates`**: the set of terminal states of the corresponding NFA

METHODS

- **`getEpsilonClosure`**: a method to get the epsilon closure of a given state using depth first search on the transition graph adding to a set the states that can be reached using epsilon transitions.
- **`subsetConstruction`**: a method to convert an NFA to a DFA using the subset construction algorithm. A depth first search is performed starting from the epsilon closure of the start state of the NFA where given a certain input, all reachable states (epsilon closures) from this epsilon closure is added to a transition table and then DFS is performed again from the reachable states until all possible epsilon closures are visited.
- **`initializePartitions`**: a method that initialises the partitions in the minimisation of the DFA where the starting partitions contain a partition for all non-terminal states and a partition for each terminal state in the **`m_NFATerminalStates`**.
- **`splitPartition`**: a method that splits each partition according to the transitions of the states given a certain input. For each state in the partition a set of transitions is created. The set of states corresponding to each unique set of transitions are split from the original partition and put in a new partition.
- **`minimize`**: a method that minimises the DFA into a corresponding minimal DFA. The DFA states are partitioned into initial partitions using **`initializePartitions`** method. The partitions are continuously attempted to be split into smaller partitions using **`splitPartition`** method until no partition can be split. The new partitions are now the new states of the minimised DFA, and their transitions are copied over to **`m_Table`**.

REGEX HANDLER

A class designed for managing and manipulating regular expressions, specifically for converting expressions from infix notation to postfix notation.

METHODS

- **`infixToPostfix`**: a method to convert a regular expression written in infix notation to postfix notation.
- **`isOperator`**: a method to check if a given character is an operator. A character is an operator if it is either: '*' (kleene closure), '+' (positive closure), '.' (concatenation), '|' (union), '(', or ')'.
- **`isOperand`**: a method to check if a given character is an operand. A character is an operand if it's not an operator.
- **`getPrecedence`**: a method to get the precedence of a given operator.

LEXICAL ANALYZER GENERATOR CLASS

A class designed for parsing the rules of a programming language, including definitions, expressions, keywords, and punctuations, and subsequently constructing a Deterministic Finite Automaton (DFA) based on these rules.

MEMBERS

- **`m_DFA`**: the DFA constructed from the given rules
- **`m_TokensPrecedence`**: the tokens precedence based on the order of their definitions
- **`m_RegexEpsilonClosures`**: the terminal states for every regex on the constructed DFA
- **`m_FilePath`**: the relative path of the text file where the language rules are defined
- **`m_RegularDefinitions`**: the language regular definitions constructed from the rules file
- **`m_RegularExpressions`**: the language regular expressions constructed from the rules file
- **`m_Keywords`**: the language keywords constructed from the rules file
- **`m_Punctuations`**: the language punctuations constructed from the rules file

METHODS

- **`readRulesFromFile`**: a method to read the rules from the file line by line.
- **`processRegularDefinition`**: a method to process the language regular definitions and add it to the **`m_RegularDefinitions`** class member.
- **`processRegularExpression`**: a method to process the language regular expressions and add it to the **`m_RegularExpressions`** class member. Also used to determine the tokens precedence by filling out the **`m_TokensPrecedence`** class member.
- **`processKeywords`**: a method to process the language keywords and add it to the **`m_Keywords`** class member.
- **`processPunctuations`**: a method to process the language punctuations and add it to the **`m_Punctuations`** class member.
- **`combineNFAs`**: a method that takes a list of NFAs and combine them in a single NFA.

- `convertDefToNFA:` a method to convert a regular definition to its corresponding NFA.
- `convertRegexToNFA:` a method to convert a regular expression to its corresponding NFA.
- `convertSymbolToNFA:` a method to convert a terminal symbol to its corresponding NFA. A terminal symbol is neither a definition like letter or digit, nor an operation like union or concatenation.
- `generateDFA:` a method to generate the DFA which represents the language rules.

LEXICAL ANALYZER CLASS

MEMBERS

- `m_CurrentPosition:` the start position of the next token processing
- `m_InputText:` the program to be compiled
- `m_StartClosure:` the start state of the DFA
- `m_TerminalClosures:` the terminal states of the DFA
- `m_Table:` the transition table of the DFA
- `m_TokensPrecedence:` the tokens precedence based on the order of their definitions
- `m_RegexEpsilonClosures:` the terminal states for every regex on the constructed DFA
- `m_Keywords:` the language keywords constructed from the rules file
- `m_Punctuations:` the language punctuations constructed from the rules file

METHODS

- `skipWhitespaces:` a method to skip all white spaces before the next token.
- `getNextToken:` a method to get the next token of the processed program.

RESULTANT TRANSITION TABLE FOR THE MINIMAL DFA

```
──────────── Transition Table ────────────
A (start):
-(!)→ D
-(*)→ E (terminal)
-(+)→ F (terminal)
-(-)→ F (terminal)
-(/)→ E (terminal)
-(0)→ C (terminal)
-(1)→ C (terminal)
-(2)→ C (terminal)
-(3)→ C (terminal)
-(4)→ C (terminal)
-(5)→ C (terminal)
-(6)→ C (terminal)
-(7)→ C (terminal)
-(8)→ C (terminal)
-(9)→ C (terminal)
-(<)→ G (terminal)
-(=)→ H (terminal)
-(>)→ G (terminal)
-(A)→ B (terminal)
-(B)→ B (terminal)
-(C)→ B (terminal)
-(D)→ B (terminal)
-(E)→ B (terminal)
-(F)→ B (terminal)
-(G)→ B (terminal)
-(H)→ B (terminal)
-(I)→ B (terminal)
-(J)→ B (terminal)
-(K)→ B (terminal)
-(L)→ B (terminal)
-(M)→ B (terminal)
-(N)→ B (terminal)
-(O)→ B (terminal)
-(P)→ B (terminal)
-(Q)→ B (terminal)
-(R)→ B (terminal)
-(S)→ B (terminal)
-(T)→ B (terminal)
-(U)→ B (terminal)
-(V)→ B (terminal)
-(W)→ B (terminal)
-(X)→ B (terminal)
-(Y)→ B (terminal)
-(Z)→ B (terminal)
-(a)→ B (terminal)
-(b)→ B (terminal)
-(c)→ B (terminal)
-(d)→ B (terminal)
-(e)→ B (terminal)
-(f)→ B (terminal)
-(g)→ B (terminal)
```

```
-(f)→ B (terminal)
-(g)→ B (terminal)
-(h)→ B (terminal)
-(i)→ B (terminal)
-(j)→ B (terminal)
-(k)→ B (terminal)
-(l)→ B (terminal)
-(m)→ B (terminal)
-(n)→ B (terminal)
-(o)→ B (terminal)
-(p)→ B (terminal)
-(q)→ B (terminal)
-(r)→ B (terminal)
-(s)→ B (terminal)
-(t)→ B (terminal)
-(u)→ B (terminal)
-(v)→ B (terminal)
-(w)→ B (terminal)
-(x)→ B (terminal)
-(y)→ B (terminal)
-(z)→ B (terminal)
──────────

B (terminal):
-(0)→ B (terminal)
-(1)→ B (terminal)
-(2)→ B (terminal)
-(3)→ B (terminal)
-(4)→ B (terminal)
-(5)→ B (terminal)
-(6)→ B (terminal)
-(7)→ B (terminal)
-(8)→ B (terminal)
-(9)→ B (terminal)
-(A)→ B (terminal)
-(B)→ B (terminal)
-(C)→ B (terminal)
-(D)→ B (terminal)
-(E)→ B (terminal)
-(F)→ B (terminal)
-(G)→ B (terminal)
-(H)→ B (terminal)
-(I)→ B (terminal)
-(J)→ B (terminal)
-(K)→ B (terminal)
-(L)→ B (terminal)
-(M)→ B (terminal)
-(N)→ B (terminal)
-(O)→ B (terminal)
-(P)→ B (terminal)
-(Q)→ B (terminal)
-(R)→ B (terminal)
-(S)→ B (terminal)
-(T)→ B (terminal)
```

```
-(S)→ B (terminal)
-(T)→ B (terminal)
-(U)→ B (terminal)
-(V)→ B (terminal)
-(W)→ B (terminal)
-(X)→ B (terminal)
-(Y)→ B (terminal)
-(Z)→ B (terminal)
-(a)→ B (terminal)
-(b)→ B (terminal)
-(c)→ B (terminal)
-(d)→ B (terminal)
-(e)→ B (terminal)
-(f)→ B (terminal)
-(g)→ B (terminal)
-(h)→ B (terminal)
-(i)→ B (terminal)
-(j)→ B (terminal)
-(k)→ B (terminal)
-(l)→ B (terminal)
-(m)→ B (terminal)
-(n)→ B (terminal)
-(o)→ B (terminal)
-(p)→ B (terminal)
-(q)→ B (terminal)
-(r)→ B (terminal)
-(s)→ B (terminal)
-(t)→ B (terminal)
-(u)→ B (terminal)
-(v)→ B (terminal)
-(w)→ B (terminal)
-(x)→ B (terminal)
-(y)→ B (terminal)
-(z)→ B (terminal)
──────────────────
H (terminal):
-(=)→ I (terminal)
──────────────────
C (terminal):
-(.)→ J
-(0)→ C (terminal)
-(1)→ C (terminal)
-(2)→ C (terminal)
-(3)→ C (terminal)
-(4)→ C (terminal)
-(5)→ C (terminal)
-(6)→ C (terminal)
-(7)→ C (terminal)
-(8)→ C (terminal)
-(9)→ C (terminal)
──────────────────
G (terminal):
-(=)→ I (terminal)
```

```
-(8)→ C (terminal)
-(9)→ C (terminal)
──────────────────
G (terminal):
-(=)→ I (terminal)
──────────────────
K:
-(0)→ L (terminal)
-(1)→ L (terminal)
-(2)→ L (terminal)
-(3)→ L (terminal)
-(4)→ L (terminal)
-(5)→ L (terminal)
-(6)→ L (terminal)
-(7)→ L (terminal)
-(8)→ L (terminal)
-(9)→ L (terminal)
──────────────────
D:
-(=)→ I (terminal)
──────────────────
M (terminal):
-(0)→ M (terminal)
-(1)→ M (terminal)
-(2)→ M (terminal)
-(3)→ M (terminal)
-(4)→ M (terminal)
-(5)→ M (terminal)
-(6)→ M (terminal)
-(7)→ M (terminal)
-(8)→ M (terminal)
-(9)→ M (terminal)
-(E)→ K
──────────────────
J:
-(0)→ M (terminal)
-(1)→ M (terminal)
-(2)→ M (terminal)
-(3)→ M (terminal)
-(4)→ M (terminal)
-(5)→ M (terminal)
-(6)→ M (terminal)
-(7)→ M (terminal)
-(8)→ M (terminal)
-(9)→ M (terminal)
──────────────────
L (terminal):
-(0)→ L (terminal)
-(1)→ L (terminal)
-(2)→ L (terminal)
-(3)→ L (terminal)
-(4)→ L (terminal)
-(5)→ L (terminal)
```

RESULTANT STREAM OF TOKENS FOR THE EXAMPLE TEST PROGRAM