Karim Mohamed Ali (34) *Mazen Ahmed Khater (36)*
Michael Ramy Ramsis (37) *Youssef Abdallah Youssef (65)*

Programming Languages and Compilers
Assignment 1
Phase1: Lexical Analyzer Generator

Problem Statement:

- Your task in this phase of the assignment is to design and implement a lexical analyzer generator tool.
- The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens. The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.
- The generated lexical analyzer has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer should produce the token class and the attribute value.
- If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens.

Modules of Lexical Analyzer Generator:

We have decomposed the Lexical Analyzer into 4 main modules:

1. Scanner Module:

- Scanner Class: responsible for parsing input files and extracting all inputs (regular definitions, regular expressions, keywords, ...etc) into suitable format to be used in the next modules. Its main functions are:
- Read the lexical rules input file.
- Extract all regular definitions to be used in the regular expressions.
  RegularDefinition:
  A class holding information about the regular definition (the name of the definition and the parsed value).
  For example: (letter = a - z | A - Z) will parsed as:
  Definition Type = letter
  Definition Value = ((a|b|c|...|z) | (A|b|C|...|Z))

- Extract all regular expressions and replace any regular definition by its value. This will be used to convert the infix expressions to a postfix expressions.
  RegularExpression:
  A class holding information about the regular expression. (expression name, infix expression value and an infix to postfix converter function).
- Any '+' operator is transformed into '*'. Ex: (digit+ → digit digit*)
- Keywords and punctuations are treated as special expressions to have higher priority during the recognition phase. (Ex: "int" must be recognized as Keyword not id).
- Read the input program file and send each token to the minimized DFA.
- Member variables of Scanner Class:
  - vector<RegularDefinition> reg_definitions; that contains all regular definitions.
  - vector<RegularExpression> reg_expressions; that contains all regular expressions.
  - vector<string> keywords, punctuations;

2. NFA Module:

- NFAState:
  A struct that holds the information for each NFA state that is created.
  It also holds the transitions to other NFA states under each input symbol.

  Member variables:
  -static int stateCounter: assigned to the stateId and incremented each time the constructor of NFA state is called.
  -int stateId: unique value assigned to each created NFA state
  -unordered_map<char, vector<reference_wrapper<NFAState>>> transitions: holds all the transitions from this NFAState to other NFA states. Note: that it holds a reference to each state, rather than holding them by value.
  -vector<reference_wrapper<NFAState>> epsilonTransitions: a vector that holds all the epsilon transitions of a NFA state.
  -vector<reference_wrapper<NFAState>> epsilonClosure: a vector that holds the epsilon closure of a NFA state.
  -bool acceptState: A boolean that determines whether this state is an accept state or not.
  -string acceptStateToken: a string representing the token held by each accept state.

- StateMachine:
  class holding the initial state and final state for each state machine (NFA).

- NFABuilder:
  The main class that constructs the NFA of each regular expression by applying *Thompson's construction algorithm*.
  - static StateMachine& stateMachineOfSymbol(char): constructs a machine for a single symbol transition.
  - static StateMachine& concatenateTwoMachines(StateMachine &fsm1, StateMachine &fsm2): constructs a machine that represents the concatenation of two state machines.

- static StateMachine& unifyTwoMachines(StateMachine &fsm1, StateMachine &fsm2); constructs a machine that represents the union of two state machines.
- static StateMachine& getTheMachineClosure(StateMachine &fsm): constructs a machine that represents the closure of a state machine.

- NFASimulator:
  A class that simulates the transitions of a NFA state machine under a particular input string.

- LexicalAnalyzer:
  The main class that acts as a facade by combining all the functionality of the lexical analyzer.
- *static void execute():*
  Collects the lexical rules provided by the Scanner as regular expressions.
  Builds NFA from these regular expressions.
  Combines all the NFA machines of all the regular expressions into a single NFA machine.
  Calls the DFA builder to convert the NFA combined machine into a DFA machine using *subset construction algorithm*.
  Calls the minimization module to get the minimized DFA machine.
  Calls the machine simulator on the minimized state machine.

3. DFA Module:

DFAState:

- A class that represents the DFA states after converting the NFA to DFA.
- It holds in it:
  1. The combined NFA states which were converted to one DFA state.
  2. An index.
  3. If it contains any accept state it saves the accept state token of the highest priority accept state (if there are many).
  4. It also holds all the transitions of this state (in an unordered map).

DFABuilder:

- The class that converts from NFA to DFA
- It first calculates the epsilon closure of every NFA State in the given NFA to use in the conversion.
- Then it starts with the initial DFA state (which is formed by the epsilon closure of the NFA's initial state) then it computes each transition in its row which corresponds to the input character.
- If in any transition forms a new DFA state we check if it's a new state, if so we push it in the DFA states vector to compute it's transitions' row and we also check whether it is a final state or not.
- The result is a vector containing all DFA States.

4. Minimization Module:

- Minimize:
    - A class that minimizes DFA States, by doing the following steps:
        1. Extract accept tokens from all accept states.
        2. Separate Non-accept states in separated groups.
        3. Separate each accept state according to accept token, and put all states that belong to the same token in a separated group.
        4. Compare between states in the same group by creating for each state a String contains output colors in specific order.
        5. Then put all similar states “having the same string which was created in step 4” in the same group.
        6. Repeat step 3, 4 and 5 until all states have the same previous group.
        7. At the end remove all Dead states and replace them in the transition table.

  Main Functions:

  1. *ExtractTokens ()*: Extract accept tokens from accept states.
  2. *Separate ():* Separate Non-Accept states and each Accept Token in Accept states in separated groups.
  3. *Difference ()*: Compare between states in the same group by creating for each state a String contains output colors in specific order.
  4. *UpdateColor ():* put all similar states in the same group.
  5. *RemoveDeadState ():* create minimization transition table by removing dead states.

  Classes needed in the implementation:

  1. ColorNode: For save previous and Current Color “group” for each state.
  2. Node: for save state_id, Current_Color “Current Group” and String which are used in step 4.

Assumptions:

1. We assumed that the character ‘~’ is the concatenation operator and it was added between any 2 concatenated definitions in the lexical rules input file.
2. In case that tokens in the input program are not space separated, the minimized DFA will return the last accepted state and then continue starting from the first character after the accepted regular expression. (EX: int 2sum → int num id)
3. We represent each group by a specific Color. (in Module 4)
4. Each color is represented by an integer. (in Module 4)