Project Phase #1 Report

Compilers

No	Name	ID
1	Zeyad Zidan	19015709
2	Abdelrahman Gad	19015894
3	Abdelaziz Mohamed	19015941
4	Omar Khairat	19016063

1. Problem Statement

This phase of the assignment aims to practice techniques for building automatic lexical analyzer generator tools. The 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 automaton (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 must 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.
- The lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java. Use the given simple program to test the generated lexical analyzer.

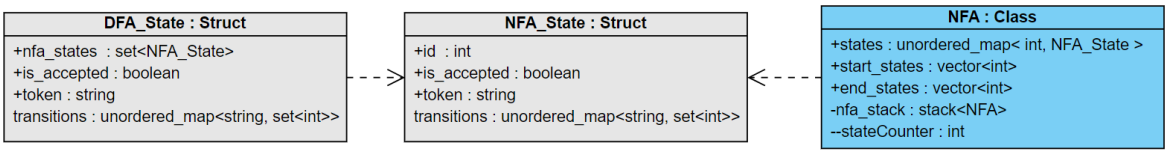
- The generated lexical analyzer will integrate with a generated parser which you should implement in phase 2 of the assignment such that the lexical analyzer is to be called by the parser to find the next token.

2. Data Structures

In this phase and the upcoming phases, already implemented C++ data structures is (and will be) used to help achieving the goal set upon the project. These data structures are –

- Vector
- Unordered Map
- Set
- Stack
- Queue
- Pair

In addition, application-specific data structures were implemented and are shown in the upcoming figure.



2.1 DFA State Structure (State)

Struct Type

nfaStates	Set of NFA states represented by the DFA state.
isAcceptance	Indicates if the DFA state is an acceptance state.
token	Token associated with the DFA state.
transition	Map representing transitions from the DFA state to other states.

3. Procedures

3.1 Parsing

Parser::get_rules_lines method

Purpose	Reads the lexical rules file path to get the rules lines based on the newline between each rule line.

Input	The lexical rules file path.
Output	Returns vector of string of the lexical rules lines.

Parser::get_keywords_lines method

Purpose	Detects the keywords lines from the rest of lexical rules based on the format that they start with ‘{’ and end with ‘}’.
Input	Vector of string of lexical rules lines.
Output	Returns vector of string of keywords lines.

Parser::get_punctuation_lines method

Purpose	Detects the punctuation lines from the rest of lexical rules based on the format that they start with ‘[’ and end with ‘]’.
Input	Vector of string of lexical rules lines.
Output	Returns vector of string of punctuation lines.

Parser::get_regular_def_lines method

Purpose	Identifies regular expression lines within a set of lexical rules by detecting the presence of '=' and ':' in a specific format. It further parses these lines into the left-hand side (LHS) representing the regular definition name and the right-hand side (RHS) representing the regular definition name.
Input	Vector of string of lexical rules lines.
Output	Returns a vector containing strings and string pairs representing regular definition lines.

Parser::get_regular_expr_lines method

Purpose	Identifies regular expression lines within a set of lexical rules by detecting the presence of ':' and '=' in a specific format. It further parses these lines into the left-hand side (LHS) representing the regular expression name and the right-hand side (RHS) representing the expression.
Input	Vector of string of lexical rules lines.
Output	Returns a vector containing strings and string pairs representing regular expression lines.

Parser::parse_keywords method

Purpose	Parses keywords in keywords lines bases on that there is at least on space between each keyword.
Input	Vector of string of keywords lines.
Output	Returns a vector containing strings denoting keywords.

Parser::parse_punctuation method

Purpose	Parses keywords in keywords lines bases on that there is at least on space between each punctuation token.
Input	Vector of string of punctuation lines.
Output	Returns a vector containing strings denoting punctuation tokens.

Parser::is_special_char method

Purpose	Checks whether a given character is a special character, specifically one of the characters '+', '-', '|', '(', ')', or '*'.
Input	The character to be checked for being a special character.
Output	Returns true if the input character is a special character, and false otherwise.

Parser::replace_dashes method

Purpose	This method processes a vector of parsed tokens, where certain tokens represent a range of characters denoted by a dash ('-'). It replaces these dash-delimited ranges with individual tokens representing each character within the range where the characters in this range are put as there is whether each of them is used.
Input	A vector of parsed tokens, where some tokens may represent character ranges using dashes.
Output	Returns a new vector of tokens with dash-delimited character ranges replaced by individual tokens representing each character within the range.

Parser::parse_rhs method

Purpose	Takes a string representing the right-hand side (RHS) of a lexical rule and parses it into a vector of individual tokens. It handles special characters and modifies the tokenization to account for specific cases, such as dot insertion for concatenation and dash-delimited character ranges.

Input	A string of the right-hand side of a lexical rule to be parsed.
Output	Returns a vector of parsed tokens representing the RHS of the lexical rule, with appropriate modifications to handle special characters and concatenation.

Pseudocode:

```
function parse_rhs(rhs_line):
    parsed_tokens = empty vector of strings
    i = 0
    temp = empty string
    dummy = empty string
    rhs_line = " " + rhs_line + " "
    char_found = false

    while i < length of rhs_line:
        if is_special_char(rhs_line[i]) and i != 0 and rhs_line[i - 1] != '\\':
            char_found = true
            if temp is not empty:
                parsed_tokens.push_back(modify_dot_token(temp))
                if rhs_line[i] == '(':
                    parsed_tokens.push_back(".")
                temp = empty string
            parsed_tokens.push_back(rhs_line[i] + dummy)
        else if rhs_line[i] == ' ' and char_found and parsed_tokens is not empty:
            last_temp = last element of parsed_tokens
            if i == length of rhs_line - 1:
                if temp is not empty:
                    parsed_tokens.push_back(modify_dot_token(temp))
            else if rhs_line[i + 1] != ' ':
                if temp is not empty:
                    parsed_tokens.push_back(modify_dot_token(temp))
                if (last_temp is not "|" and last_temp is not "-" and last_temp is not "(")
or not temp.empty()
                    and rhs_line[i + 1] is not '|' and rhs_line[i + 1] is not '-' and rhs_line[i +
1] is not '+'
                    and rhs_line[i + 1] is not '*' and rhs_line[i + 1] is not ')':
                    parsed_tokens.push_back(".")
            temp = empty string
        else if i == length of rhs_line - 1 and rhs_line[i] == ' ' and parsed_tokens is
empty and not temp.empty():
            parsed_tokens.push_back(modify_dot_token(temp))
        else if rhs_line[i] is not ' ':
```

```
        char_found = true
        temp += rhs_line[i]
    i++

return replace_dashes(parsed_tokens)
```

Parser::parse_defs method

Purpose	Processes the regular definition string and parses it into tokens of signs and characters and replace the nested pre-found with tokens corresponding to it.
Input	A vector of pairs, where each pair consists of a string representing the LHS (left-hand side) the definition name and another string representing the RHS (right-hand side) of a regular definition.
Output	Returns a vector containing unordered maps, where each map uses the name of the regular definition as the key and its parsed tokens on the right-hand side as the corresponding value in vector of string.

Parser::parse_expr method

Purpose	Parses expression lines, converting them into a structured format. It replaces any references to previously defined regular definitions with their corresponding parsed tokens from the provided unordered map.
Input	A vector of pairs where the first element represents the left-hand side (LHS) the name of the expression, and the second element is the right-hand side (RHS) is the regular expression line. In addition to, An unordered map containing previously defined regular definations and their parsed tokens.
Output	Returns a vector of pairs, where each pair consists of the LHS expression name and its corresponding parsed tokens on the RHS. The output represents the parsed expressions with resolved references to regular expressions.

Parser::infixtoPos method

Purpose	Convert an infix expression represented as a vector of strings into its corresponding postfix (reverse Polish notation) form. It takes into account the precedence of operators and ensures the correct order of operations.
Input	A vector of strings representing the infix expression to be converted.
Output	Returns a vector of strings representing the postfix expression derived from the input infix expression.

Pseudocode

procedure Parser::infixToPostfix(infix: vector<string>) → vector<string>	
1.	pos := empty vector of strings
2.	stck := empty stack of strings
3.	special_chars := empty unordered_map<string, int>
4.	special_chars["*"] := 5
5.	special_chars["+"] := 4
6.	special_chars["."] := 3
7.	special_chars["|"] := 2
8.	special_chars["("] := special_chars[")"] := 0
9.	for each token in infix:
10.	if token is "(":
11.	push token to stck
12.	else if token is ")":
13.	while stck is not empty and stck.top() is not "(":
14.	append stck.top() to pos
15.	pop from stck
16.	if stck is not empty:
17.	pop from stck // Remove the "(" from the stack
18.	else if token is "*", "+", "|", or ".":
19.	while stck is not empty and special_chars[token] ← special_chars[stck.top()]:
20.	append stck.top() to pos
21.	pop from stck
22.	push token to stck
23.	else:
24.	// Operand
25.	append token to pos
26.	// Pop any remaining operators from the stack
27.	while stck is not empty:
28.	append stck.top() to pos
29.	pop from stck
30.	return pos

Parser::convert_exprs_to_pos method

Purpose	Converts a vector of regular expressions, where each expression is represented as a pair with a expression name and a vector of infix tokens, to

	a vector of expressions in postfix notation represented as pair with an expression name and a vector of infix tokens denoting the expression.
Input	Vector of pairs, where each pair consists of an expression name and a vector of infix tokens representing the expression.
Output	Returns a vector of pairs, where each pair contains the expression name and the corresponding vector of tokens in postfix notation obtained by converting the infix expressions to postfix one.

3.2 Constructing NFA

procedure NFA::concatenate(void)	
1.	nfa2 = nfa_stack.pop() , nfa1 = nfa_stack.pop()
2.	add nfa2 states to nfa1
3.	nfa1 add epsilon transition from nfa1 end states to nfa2 start states
4.	nfa1.end_states = nfa2.end_states
5.	nfa_stack.push(nfa1)

procedure NFA::or(void)	
1.	nfa2 = nfa_stack.pop() , nfa1 = nfa_stack.pop()
2.	create NFA: new_nfa
3.	new_start_state_id = stateCounter
4.	new_nfa add state with (id= new_start_state_id, is start state)
5.	stateCounter = stateCounter + 1
6.	new_end_state_id = stateCounter
7.	add state to new_nfa with (id= new_end_state_id, is acceptance state)
8.	stateCounter = stateCounter + 1
9.	add all nfa1 states to new_nfa
10.	add all nfa2 states to new_nfa
11.	new_nfa add epsilon transitions:
12.	from new_start_state to nfa1 and nfa2 start states
13.	from nfa1 and nfa2 end states to new_end_state
14.	nfa_stack.push(new_nfa)

procedure NFA::kleeneStar(void)	
1.	nfa2 = nfa_stack.pop() , nfa1 = nfa_stack.pop()
2.	create NFA: new_nfa
3.	new_start_state_id = stateCounter
4.	new_nfa add state with (id= new_start_state_id, is start state)
5.	stateCounter = stateCounter + 1
6.	new_end_state_id = stateCounter
7.	add state to new_nfa with (id= new_end_state_id, is acceptance state)

```
8.      stateCounter = stateCounter + 1
9.      add all nfa1 states to new_nfa
10.     add all nfa2 states to new_nfa
11.     new_nfa add epsilon transitions:
12.         from new_start_state to nfa1 and nfa2 start states
13.         from nfa1 and nfa2 end states to new_end_state
14.     nfa_stack.push(new_nfa)
```

procedure NFA::positiveClosure(void)

```
1.      nfa = nfa_stack.pop()
2.      create NFA: new_nfa
3.      new_nfa add epsilon transitions from nfa end states to nfa start states
4.      nfa_stack.push(nfa)
```

procedure NFA::processSymbol(void)

```
1.      create NFA: new_nfa
2.      new_start_state_id = stateCounter
3.      new_nfa add state with (id= new_start_state_id, is start state)
4.      stateCounter = stateCounter + 1
5.      for each character c in symbol:
6.          new_end_state_id = stateCounter
7.          add state to new_nfa with (id= new_end_state_id)
8.          stateCounter = stateCounter + 1
9.          if c = "\":
10.             new_nfa add transition with (c+the next character)
11.                     from new_start_state to new_end_state_id
12.             skip the next character
13.         else:
14.             new_nfa add transition with (c) from new_start_state to
15.     new_end_state_id
16.         new_start_state = new_end_state
17.     make the last new_end_state is acceptance state
18.     new_nfa.end_state.add(new_end_state)
19.     nfa_stack.push(new_nfa)
```

procedure NFA::concatenateAllStack(void)

```
1.      If nfa_stack is empty: return
2.      If nfa_stack.size :
3.          nfa = nfa_stack.pop
4.
```

```
5.      states = nfa.state, start_states = nfa.start_states, end_states =
6.  nfa.end_states
7.      return
8.  clear start_states, end_states, states
9.  new_start_state_id = stateCounter
10. add state with (id= new_start_state_id, is start state)
11. stateCounter = stateCounter + 1
12. while nfa_stack is not empty:
13.     nfa = nfa_stack.pop
14.     add nfa.states to states
15.     add epsilon transitions from new_start_state to nfa start states
        add nfa end states to end_states
```

procedure NFA::getEpsilonClosureSetMap(void)

```
1.  epsilon_closure_set = map (int -> set (int)) // state id to its epsilon closure
2.  for each state in states:
3.      epsilon_closure_i = set (int), q = queue (int)
4.      q.push(state.id)
5.      while q is not empty:
6.          current_state = q.pop
7.          add current_state to epsilon_closure_i
8.          for each next_state in epsilon transitions of current_state:
9.                  if next_state is in epsilon_closure_set:
10.                         add all of epsilon_closure_set[next_state] to
11. epsilon_closure_i
12.                 else:
13.                         q.push(next_state)
14.     epsilon_closure_set[state.id] = epsilon_closure_i
15. return epsilon_closure_set
```

procedure NFA::convert_exprs_postfix_to_NFA()
@Params
exprs: vector(pair(string, vector(string))),
keywords:vector(string),
punctuations: vector(string)

```
1.  epsilon_closure_set = map (int -> set (int)) // state id to its epsilon closure
2.  for each state in states:
3.      epsilon_closure_i = set (int), q = queue (int)
4.      q.push(state.id)
5.      while q is not empty:
6.          current_state = q.pop
7.          add current_state to epsilon_closure_i
```

```
8.                  for each next_state in epsilon transitions of current_state:
9.                      if next_state is in epsilon_closure_set:
10.                         add all of epsilon_closure_set[next_state] to
11.         epsilon_closure_i
12.                     else:
13.                         q.push(next_state)
14.             epsilon_closure_set[state.id] = epsilon_closure_i
15.         return epsilon_closure_set
```

3.3 Constructing DFA

getDFAStateIDFromNFAStates Function

Purpose	Finds the ID of the DFA state containing a specified target NFA state.
Input	unordered_map<int, DFA::State> representing DFA states, and the target NFA state.
Output	Returns the ID of the DFA state containing the target NFA state, or -1 if not found.

constructDFA Function

Purpose	Constructs a DFA from a given NFA and a set of priority values for token identification.
Input	NFA object (nfa), priority map for tokens (priority).
Output	Returns an unordered_map<int, DFA::State> representing the constructed DFA.

processTransitions Function

Purpose	Processes transitions of the DFA states based on the provided DFA state map.
Input	unordered_map<int, DFA::State> representing DFA states.
Output	Returns an updated unordered_map<int, DFA::State> with processed transitions.

3.4 Minimize DFA

findIndex Function

Purpose	Finds the index of a target set within a vector of unordered sets.
Input	Vector of equivalence classes (equivalenceClasses) and the target set (targetSet).

Output	Returns the index of the target set if found, or -1 if not found.

minimizeDFA Function

Purpose	Minimizes a DFA by merging equivalent states.
Input	Original DFA states (dfa_states) and token priorities (priority).
Output	Returns a minimized DFA represented by an unordered_map<int, DFA::State>.

3.5 Implementation Details

- Identifies acceptance and non-acceptance states in the original DFA.
- Introduces a dead state to handle transitions to non-existing states.
- Utilizes equivalence checking functions for acceptance and non-acceptance states.
- Merges equivalent states into equivalence classes for both acceptance and non-acceptance states.
- Determines the most prioritized token within each equivalence class of acceptance states.
- Constructs the minimized DFA by mapping old states to new equivalence class IDs.
- Handles transitions by identifying the target state in the corresponding equivalence class.

3.5.1 Equivalence Checking Functions

- areEquivalent: Checks if two states are equivalent in terms of acceptance and transitions.
- areEquivalentWithToken: Extends equivalence check to include token priorities.

3.5.2 Resulting Structure

The minimized DFA is represented as an unordered map with new state IDs and corresponding state objects.

3.5.3 Overall Approach

- Iteratively builds equivalence classes for acceptance and non-acceptance states.
- Determines the most prioritized token within each acceptance class.
- Constructs the minimized DFA by mapping transitions to new equivalence class IDs.

3.6 Pattern Matching

Procedure **matchExpression**(***expression***):

1. Let ***pattern*** = empty string, ***symbol_table*** = vector of pairs, ***current_state*** = ***DFA*** [1]
2. For each char ***c*** in ***expression***
3. If ***current_state*** is an accepting state → save it as ***acceptor***

4. If ***current_state*** has a transition that ***c*** can take
5. Take the transition and set ***current_state*** = DFA [***next***]
6. Else
7. *// End of pattern or an error is encountered. (Tokenizing Phase)*
8. If an acceptor is present
9. Accept the pattern that matches the latest acceptor,
10. and report the string of errors.
11. If not present, just report the string of errors.
12. Reset all variables and counters after this phase.

End procedure.

4. Resultant Transition Table from Minimal DFA

A preview of the transition table is available here.

Example of: id:a(b|c) a*:

DFA State Minimzed ID	Is Acceptance	Token	a	b	c
-2	FALSE				
0	FALSE		2	0	0
1	FALSE		0	-2	-2
2	TRUE	id	2	-2	-2

5. Resultant Stream of Tokens in The Test Program

5.1 Test Program

```
int sum , count , pass , mnt; while (pass !\\=\
        10)
        {
        pass = pass \\+ 1 ;
}
```

```
TOKEN = int --- MATCHED PATTERN = int
TOKEN = id --- MATCHED PATTERN = sum
TOKEN = , --- MATCHED PATTERN = ,
TOKEN = id --- MATCHED PATTERN = count
TOKEN = , --- MATCHED PATTERN = ,
TOKEN = id --- MATCHED PATTERN = pass
TOKEN = , --- MATCHED PATTERN = ,
TOKEN = id --- MATCHED PATTERN = mnt
TOKEN = ; --- MATCHED PATTERN = ;
TOKEN = while --- MATCHED PATTERN = while
TOKEN = ( --- MATCHED PATTERN = (
TOKEN = id --- MATCHED PATTERN = pass
TOKEN = relop --- MATCHED PATTERN = !\

TOKEN = num --- MATCHED PATTERN = 10
TOKEN = ) --- MATCHED PATTERN = )
TOKEN = { --- MATCHED PATTERN = {
TOKEN = id --- MATCHED PATTERN = pass
TOKEN = assign --- MATCHED PATTERN = =
TOKEN = id --- MATCHED PATTERN = pass
TOKEN = addop --- MATCHED PATTERN = \
TOKEN = num --- MATCHED PATTERN = 1
TOKEN = ; --- MATCHED PATTERN = ;
TOKEN = } --- MATCHED PATTERN = }
```

5.2 Example #1

abc@123

```
TOKEN = id --- MATCHED PATTERN = abc
TOKEN = error --- MATCHED PATTERN = @
TOKEN = num --- MATCHED PATTERN = 123
```

6. Assumptions

- Tokens are considered separate when there is either a space between them or they are delimited by the characters +, *, (,), or |.
- If a token consists of a group of characters, they are parsed as a single entity unless they represent the name of a regular definition. In the latter case, the token is replaced with the corresponding definition.
- Any dead state is issued an id of -2. This particular value is issued since -1 is already used to check for conditional errors and other checks.
- The start state in the minimized DFA unordered map is always issued the **index #1**.
- Epsilon closure sets are precomputed for NFA states.
- The DFA is constructed using a set of unmarked states and epsilon closures.
- Token priorities are considered when determining acceptance states.
- The ***processTransitions*** function updates transitions using DFA state IDs.