21AIE301 FORMAL LANGUAGES AND AUTOMATA

CSE-AI

Dated: 07/01/2023

Submitted By: Team - 05

PROJECT TOPIC: LEXICAL ANALYZER

Team Members:

AMEEN ASHADULLAH M - (CB.EN.U4AIE20004)
GHAYATHRI DEVI K - (CB.EN.U4AIE20017)
GORANTLA V N S L VISHNU VARDHAN - (CB.EN.U4AIE20019)
MENTA SAI AASHISH - (CB.EN.U4AIE20039)
SAI ARAVIND V - (CB.EN.U4AIE20062)

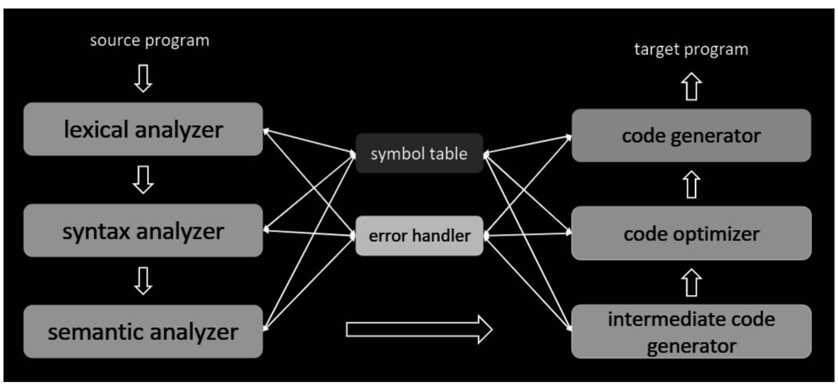
ABSTRACT

Automaton refers to "automation", denoting automatic processes carrying out the production of specific processes. The automata theory deals with the logic of computation with respect to simple machines, known as automata. Through automata, computer scientists are able to understand how machines compute functions and solve problems and more importantly, what it means for a function to be defined as computable or for a question to be described as decidable .

These automatons are abstract models of machines that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input. As a whole, automata theory is used to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems, in which signals are sampled periodically. In this project we have implemented a lexical analyzer using finite automaton, which performs lexical analysis that is the first phase of the compiler and converts a high level input program into a sequence of tokens.

INTRODUCTION

The process of converting source code written in any programming language, usually a mid- or high-level language, into a machine-level language that is understandable by the computer is known as Compilation.



1. First stage of a three-part frontend to help understand the source program

- Processes every character in the input program
- If a word is valid, then it is assigned to a syntactic category

This is similar to identifying the part of speech of an English word

Let us see the description of the lexical analyzer/

Input:

A high level language program, such as a C or Java program, in the form of a sequence of ASCII characters

Output:

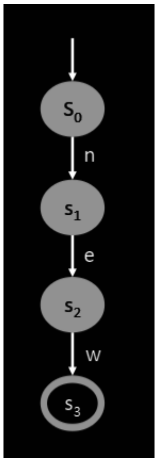
A sequence of tokens along with attributes corresponding to different syntactic categories that is forwarded to the parser for syntax analysis

Functionality:

- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers
- Performs some preprocessor functions in languages

Recognizing Word *"new"*

```
c = getNextChar();
if (c == 'n')
c = getNextChar();
if (c == 'e')
c = getNextChar();
if (c == 'w')
report success;
else
// Other logic
else
// Other logic
else
// Other logic
```



Formalism for Scanners

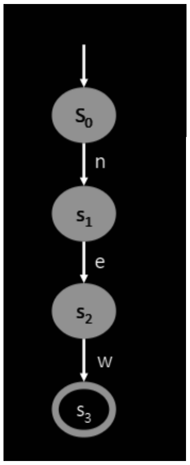
An alphabet is a finite set of symbols.

- Typical symbols are letters, digits, and punctuations
- ASCII and UNICODE are examples of alphabets
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet
- A language is any countable set of strings over a fixed alphabet

Finite State Automaton

- A finite state automaton (FSA) is a five-tuple or quintuple (s,Σ,**δ,s_0,s_F**)
- is a finite set of states
- Σ is the alphabet or character set. It is the union of all edge labels in the FSA, and is finite.
- **δ(s,c)** represents the transition from state on input
- $s_0 \in S$ is the designated start state
- $s_F \subseteq s$ is the set of final states
- A FSA accepts a string if and only if:
    1. FSA starts in **s_0.**
    2. Execute transitions for the sequence of characters in x
    3. Final state is an accepting state $\in s_F$ after it has been consumed.

FSA for recognizing *“new”*



- FSA = $(S, \Sigma, \delta, s_0, S_F)$
- $S = (s_0, s_1, s_2, s_3)$
- $\Sigma = \{n, e, w\}$
- $\delta = \{s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{e} s_2, s_2 \xrightarrow{w} s_3\}$
- $s_0 = s_0$
- $S_F = \{s_3\}$

Terminologies for lexical analyzer

Token

- A string of characters which logically belong together in a syntactic category
- Sentences consist of a string of tokens
- For example, float, identifier, equal, minus, int, num, semicolon
- Tokens are treated as terminal symbols of the grammar specifying the source language
- May have an optional attribute

Pattern

- The rule describing the set of strings for which the same token is produced
- The pattern is said to match each string in the set
- float, letter(letter|digit|_)*, =, -, digit + , ;

Lexeme

- The sequence of characters matched by a pattern to form the corresponding token
- “float”, “abs_zero”, “=”, “-”, “273”, “;”

An attribute of a token is a value that the scanner extracts from the corresponding lexeme and supplies to the syntax analyzer.

Tokens in Programming Languages

Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols (parentheses, brackets, commas, semicolons, and colons)

Attributes for tokens (apart from the integer representing the token)

- identifier: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
- Int num: the value of the integer (similarly for float num, etc.)
- string: the string itself

The exact set of attributes are dependent on the compiler designer

Role of a Lexical Analyzer

- Identify tokens and corresponding lexemes
- Construct constants: for example, convert a number to token num and pass the value as its attribute.
    1. 31 becomes <num, 31>
- Recognize keyword and identifiers
    1. counter = counter + increment becomes id = id + id
    2. Check that id here is not a keyword
- Discard whatever does not contribute to parsing
    1. White spaces (blanks, tabs, newlines) and comments

Specifying and Recognizing Patterns and Tokens

- Patterns are denoted with regular expressions, and recognized with finite state automata
- Regular definitions, a mechanism based on regular expressions, are popular for specification of tokens
- Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens
    1. Usually used to model LA before translating them to executable programs

Transition Diagrams

Transition diagrams (TDs) are generalized DFAs with the following differences

- Edges may be labeled by a symbol, a set of symbols, or a regular definition
- Few accepting states may be indicated as retracting states
    1. Indicates that the lexeme does not include the symbol that transitions to the accepting state
- Each accepting state has an action attached to it
    1. Action is executed when the state is reached
    2. Typically, such an action returns a token and its attribute value

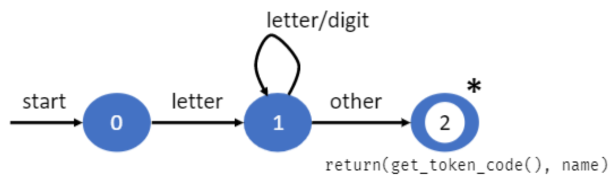
Let us create the transition diagrams for each of the tokens.

Identifiers and reserved words

Regular Expression

```
letter = [a-zA-Z]
digit = [0-9]
Identifier = letter(letter|digit)*
```

Transition diagram



- * indicates a retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code if so

- Otherwise, it returns the integer code of the IDENTIFIER token, with name containing the string of characters forming the token
    1. Name is not relevant for reserved words

Let us do a sample specification for other tokens too.

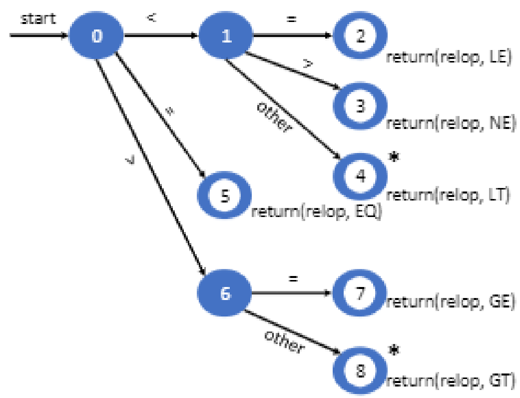
<u>Regular Expressions</u>

1. digit = [0-9]
2. digits = digit$^+$
3. number = digits(.digits)?(E[+-]? digits)?
4. letter = [A-Za-z]
5. id = letter(letter|digit)*
6. if = if
7. then = then
8. else = else
9. relop = <|>|<=|>=|=|<>
10. ws = (blank|tab|newline)$^+$

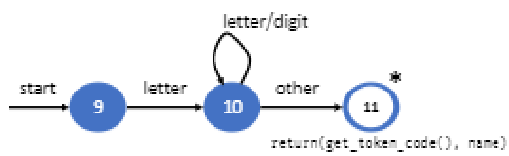
Tokens, Lexemes and attributes

Lexemes	Token Name	Attribute Value
Any *ws*	--	--
if	**if**	--
then	**then**	--
else	**else**	--
Any *id*	**id**	Pointer to symbol table entry
Any *number*	**number**	Pointer to symbol table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

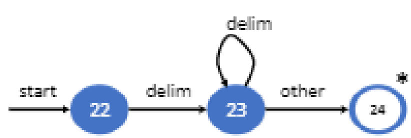
Transition diagram for relop



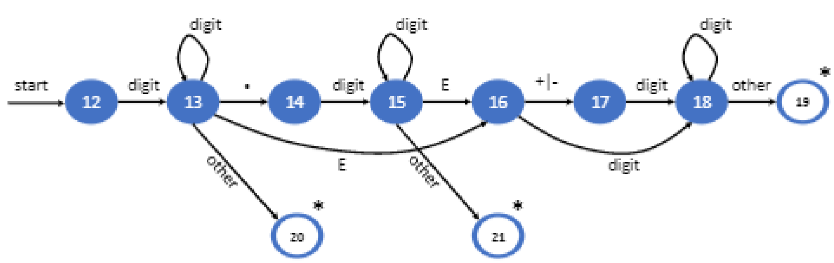
Transition diagram for ID's and keywords



Transition diagram for whitespaces



Transition diagram for unsigned numbers



Now different transition diagrams must be combined appropriately to yield a scanner or a lexical analyzer. We can try different transition diagrams one after the another. For example, transition diagrams for reserved words, constants, identifiers and operators could be tried in that order. However this does not use the “longest match” characteristic. The next would be an identifier and not reserved word then followed by identifier ext.

Challenges in Lexical Analysis

1. Some languages do not have any reserved words.
2. It is difficult for the scanner to differentiate between keywords and user defined identifiers.
3. They require arbitrary lookahead and very large buffers. In the worst case the buffers may have to be reloaded in case of wrong inferences.

Let us now implement the scanners.

1. Specify REs for each syntactic category
2. Construct an NFA for each RE
3. Join the NFAs with **ε**-transitions
4. Create the equivalent DFA
5. Minimize the DFA
6. Generate code to implement the DFA

Building a lexical analyzer using finite automata

One way to build a lexical analyzer is to use a finite state automaton (FSA). An FSA is a machine that can be in one of a finite number of states, and it transitions from one state to another based on input. In the case of a lexical analyzer, the input is a stream of characters, and the states of the FSA correspond to different lexical categories, such as keywords, identifiers, numbers, and so on.

Here is a general outline of how you might build a lexical analyzer using an FSA:

- Define the set of states for the FSA. These might include states for different types of tokens, such as keywords, identifiers, and numbers, as well as states for handling whitespace and comments.
- Define the set of input characters that the FSA will recognize. This will typically include all of the ASCII characters, as well as any other characters that are used in the language you are working with.

- Define the transitions between states. For each state, you will need to specify what the FSA should do when it receives a particular character. For example, if the FSA is in a state that represents an identifier, and it receives a letter, it should stay in the identifier state. If it receives a digit, it should also stay in the identifier state. If it receives any other character, it should transition to a different state.
- Implement the FSA using a programming language of your choice. This will typically involve writing a loop that reads characters from the input stream, looks up the transition for the current state and character, and updates the state accordingly.
- Add code to the FSA to recognize and return tokens. As the FSA processes the input stream, it should keep track of the characters that make up each token. When it transitions to a new state, it should return the current token to the calling program.

Role of FSA in building the Lexical Analyzer

The role of a Finite State Automaton (FSA) in a lexical analyzer is to recognize the tokens in the input character stream and to group them into lexemes. A lexeme is a sequence of characters that form a token.

For example, in a programming language, a lexeme may be a keyword like "while" or "for", an identifier like a variable name, a constant like a number, or a symbol like a punctuation mark. The lexer uses an FSA to identify the lexemes in the input and to determine their corresponding tokens.

The FSA is constructed from a set of rules called regular expressions, which specify the patterns that the lexeme must match. The FSA reads the input character by character and transitions between states according to the rules until it reaches an accepting state, indicating that it has recognized a valid lexeme. The lexer then returns the corresponding token for that lexeme to the parser, which is responsible for interpreting the meaning of the tokens in the context of the programming language.

Concepts used for building a lexical analyzer using FSA Deterministic finite automata

A deterministic finite automaton (DFA) is a type of finite state machine (FSM) that accepts or rejects strings of symbols. It consists of a set of states, a set of input symbols, a transition function, a start state, and a set of accept states.
Here is how a DFA works:

- The DFA starts in the start state.
- It reads the first symbol of the input string.
- Based on the current state and the symbol it has just read, the DFA uses the transition function to determine the next state.

- The DFA moves to the next state and reads the next symbol of the input string.
- The process repeats until the DFA has read the entire input string.
- If the DFA ends in an accept state, it accepts the input string. Otherwise, it rejects the input string.

DFAs are used in many different applications, including lexical analysis, pattern matching, and data validation. They are simple to design and implement, and they are relatively efficient at runtime, making them a popular choice for many tasks.

Regular expressions

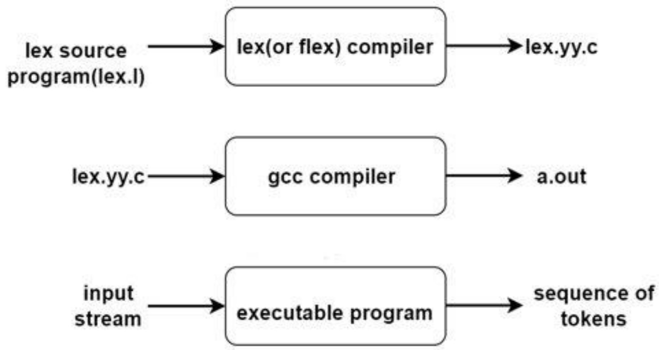
- A regular expression is a sequence of characters that forms a search pattern. It is mainly used to search for and match patterns in text. Regular expressions are often used to perform searches, replace substrings, and validate input.
- The basic syntax for regular expressions consists of a pattern enclosed within forward slashes (/) characters. For example: /cat/ would match the word "cat" in the input string.
- You can also include special characters in the pattern to match more complex patterns. For example, the pattern /ca.t/ would match any three-letter word that starts with "ca" and ends with "t", including "cat", "car", and "cut".
- There are many other special characters and syntax rules for creating regular expressions, which you can learn about through online resources or by using a regular expression reference

What is the high level idea in implementing scanners ? Let us see it below.

1. Read input character one by one
2. Look up the transition based on the current state and the input character.
3. Switch to the new state.
4. Check for termination conditions, accept and error.
5. Repeat.

Generating a lexical analyzer

A *lex* or *flex* is a program that is used to generate a lexical analyzer. These translate regular definitions into C source code for efficient lexical analysis.



A *lexical analyzer generator* systematically translates regular expressions to *NFA* which is then translated to an efficient *DFA*.

Here is an example for lexical analysis. It splits the tokens and non-tokens.

Consider the following code into the lexical analyzer.

```
#include <stdio.h>

    int maximum(int x, int y) {

        // This will compare 2 numbers

        if (x > y)

            return x;

        else {

            return y;

        }

    }
```

Examples of the tokens created:

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

Examples of non-tokens created:

Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

Lexical Errors

- A character sequence which is not possible to scan into any valid token is a lexical error. Important facts about the lexical error:
- Lexical errors are not very common, but it should be managed by a scanner
- Misspelling of identifiers, operators, keyword are considered as lexical errors
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error handling in Lexical Analysis

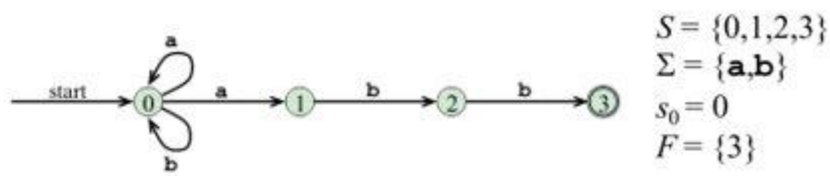
- LA cannot catch any other errors except for simple errors such as illegal symbols
- In such cases, LA skips characters in the input until a well-formed token is found

    1. This is called “panic mode” recovery
- We can think of other possible recovery strategies
    1. Delete one character from the remaining input, or insert a missing character
    2. Replace a character, or transpose two adjacent characters
    3. Idea is to see if a single (or few) transformation(s) can repair the error

Structure of the generated analyzer

The lexical analyzer comprises of a program to simulate automata and 3 components created from the lex program by the lex

1. *a transition table for the automaton*
2. *functions passed directly through lex to the output* and
3. *actions from the input program which appear as fragments of code to be invoked by the automaton simulator at the appropriate time*.



A *transition graph* is an equivalent way of representing an NFA, for each state s and an input symbol x(and ε), the set of successor states x leads to from s.
The empty set φ is used where there is no edge labeled x from s.

A transition table is the mapping of δ of an NFA.

δ(0, a) = {0, 1}

δ(0, b) = {0}

δ(1, b) = {2}

δ(2, b) = {3}

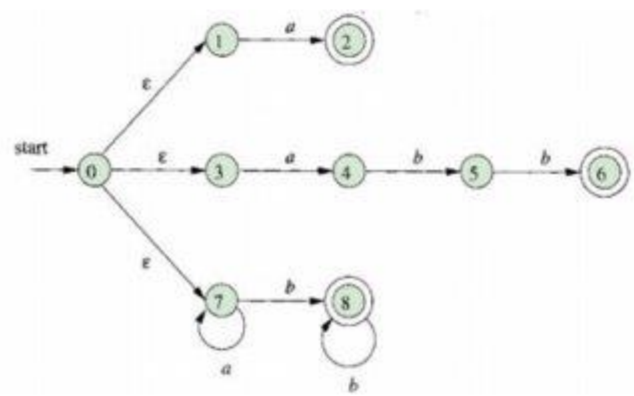
STATE	INPUT A	INPUT B
0	{0, 1}	{0}
1		{2}
2		{3}

Construction of an automaton begins by taking each regular expression pattern in the lex program and converting it to an NFA.

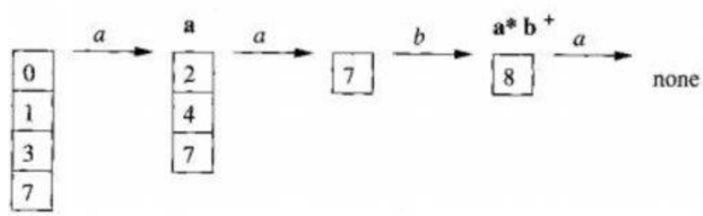
An NFA accepts an input string x if there is some path with edges labeled with symbols from x in a sequence from start to an accepting state in the transition graph.

The result is a single automaton which will recognize lexemes matching any patterns in the program so that we can combine all NFAs into one by using a new start with e-transitions for each of the start states of the NFSs.

Pattern matching based on NFAs



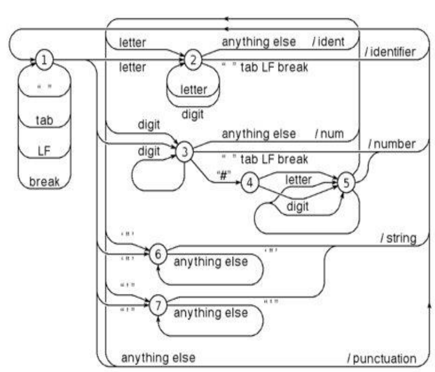
It will read input from lexemeBegin and as it moves the forward pointer ahead in the input, It will calculate the set of states it is in at each point. When the simulation reaches a point where there are no states remaining it will determine the longest prefix that is a lexeme matching a pattern. This is because there will be no longer prefix of an input that will get the NFA to an accepting state because the will be no more states at this point



To determine the longest prefix, we look backwards in the sequence of sets of states until we find a set including one or more accepting states. If there is a case where there are several such states, we select the one associated with the earliest pattern pi in the list from the lex program. To do this we move the forward pointer to the end of the lexeme and perform the action Ai associated with patterns pi.

Lexical Analyzer with melee moore machine

- Mealy and Moore state machines are types of finite state machines (FSMs) that are used to model the behavior of systems. They are named after their respective inventors, George H. Mealy and Edward F. Moore.
- A Mealy machine is a type of FSM that produces an output based on its current input and state. The output is a function of both the input and the current state. In other words, the output of a Mealy machine depends on both the current input and the current state.
- A Moore machine is a type of FSM that produces an output based only on its current state. The output is a function of the current state only, and is independent of the current input. In other words, the output of a Moore machine depends only on the current state.
- Both Mealy and Moore machines are used to model the behavior of systems, and both have their own advantages and disadvantages. Mealy machines are generally easier to implement, but may produce more outputs for a given input-state combination than a Moore machine. Moore machines are generally more predictable, but may require more states to implement the same behavior as a Mealy machine.



State table

State	Input	Next	Output
1	" "	1	
1	letter	2	
1	digit	3	
1	"'"	6	
1	'"'	7	
1	other	1	punctuation
2	letter	2	
2	digit	2	
2	" "	1	identifier
2	"'"	6	identifier
2	'"'	7	identifier
2	other	1	identifier + punctuation
3	digit	3	
3	"#"	4	
3	" "	1	number
3	letter	2	number
3	"'"	6	number
3	'"'	7	number
3	other	1	number + punctuation
4	letter	5	
4	digit	5	
5	letter	5	
5	digit	5	
5	" "	1	number
5	"'"	6	number
5	'"'	7	number
5	other	1	number + punctuation
6	"'"	1	string
6	other	6	
7	'"'	1	string
7	other	7	

Advantages of Lexical Analysis

- Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer's code to create a compiled binary executable code
- It is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
- A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task

Disadvantages of Lexical Analysis

- You need to spend significant time reading the source program and partitioning it in the form of tokens
- Some regular expressions are quite difficult to understand
- More effort is needed to develop and debug the Lexer and its token descriptions
- Additional runtime overhead is required to generate the Lexer tables and construct the tokens

APPENDIX

REGULAR EXPRESSIONS

```
import re

def lexer(text):
    # Set up the FSA
    state = "start"
    lexeme = ""
    tokens = []

    # Define the FSA transitions
    def start_state(char):
        nonlocal state, lexeme
        if char.isspace():
            state = "start"
        elif char.isalpha():
            state = "identifier"
            lexeme += char
        elif char.isdigit():
            state = "integer"
            lexeme += char
        else:
            state = "symbol"
            lexeme += char

    def identifier_state(char):
        nonlocal state, lexeme
        if char.isalpha() or char.isdigit():
            lexeme += char
        else:
            tokens.append(("identifier", lexeme))
            lexeme = ""
            start_state(char)

    def integer_state(char):
        nonlocal state, lexeme
        if char.isdigit():
            lexeme += char
        else:
            tokens.append(("integer", lexeme))
            lexeme = ""
            start_state(char)

    def symbol_state(char):
        nonlocal state, lexeme
        tokens.append(("symbol", lexeme))
        lexeme = ""
        start_state(char)

    # Define the FSA states
    states = {
        "start": start_state,
        "identifier": identifier_state,
        "integer": integer_state,
```

```
        "symbol": symbol_state
    }

    # Iterate through the characters in the input text
    for char in text:
        states[state](char)

    # Add any remaining lexemes to the tokens list
    if lexeme:
        tokens.append((state, lexeme))

    return tokens

# Test the lexer
text = "int x = 5 + 6;"
print(lexer(text))
```

OUTPUT:

```
[('identifier', 'int'), ('identifier', 'x'), ('symbol', '='), ('integer',
'5'), ('symbol', '+'), ('integer', '6'), ('symbol', ';')]
```

REGULAR EXPRESSIONS TO FINITE AUTOMATA

```
from copy import deepcopy

#Regex validation

def is_valid_regex(regex):

    return valid_brackets(regex) and valid_operations(regex)

def valid_brackets(regex):

    opened_brackets = 0

    for c in regex:

        if c == '(':

            opened_brackets += 1

        if c == ')':

            opened_brackets -= 1

        if opened_brackets < 0:

            print('ERROR missing bracket')

            return False

    if opened_brackets == 0:

        return True
```

```
    print('ERROR unclosed brackets')
    return False
def valid_operations(regex):
    for i, c in enumerate(regex):
        if c == '*':
            if i == 0:
                print('ERROR * with no argument at', i)
                return False
            if regex[i - 1] in '(|':
                print('ERROR * with no argument at', i)
                return False
        if c == '|':
            if i == 0 or i == len(regex) - 1:
                print('ERROR | with missing argument at', i)
                return False
            if regex[i - 1] in '(|':
                print('ERROR | with missing argument at', i)
                return False
            if regex[i + 1] in ')|':
                print('ERROR | with missing argument at', i)
                return False
    return True

class RegexNode:
    @staticmethod
    def trim_brackets(regex):
        while regex[0] == '(' and regex[-1] == ')' and
is_valid_regex(regex[1:-1]):
            regex = regex[1:-1]
        return regex
```

```
    @staticmethod
    def is_concat(c):
        return c == '(' or RegexNode.is_letter(c)
    @staticmethod
    def is_letter(c):
        return c in alphabet
    def __init__(self, regex):
        self.nullable = None
        self.firstpos = []
        self.lastpos = []
        self.item = None
        self.position = None
        self.children = []
        if DEBUG:
            print('Current : '+regex)
        #Check if it is leaf
        if len(regex) == 1 and self.is_letter(regex):
            #Leaf
            self.item = regex
            #Lambda checking
            if use_lambda:
                if self.item == lambda_symbol:
                    self.nullable = True
                else:
                    self.nullable = False
            else:
                self.nullable = False
            return
        #It is an internal node
        #Finding the leftmost operators in all three
```

```
        kleene = -1
        or_operator = -1
        concatenation = -1
        i = 0
        #Getting the rest of terms
        while i < len(regex):
            if regex[i] == '(':
                #Composed block
                bracketing_level = 1
                #Skipping the entire term
                i+=1
                while bracketing_level != 0 and i < len(regex):
                    if regex[i] == '(':
                        bracketing_level += 1
                    if regex[i] == ')':
                        bracketing_level -= 1
                    i+=1
            else:
                #Going to the next char
                i+=1
            #Found a concatenation in previous iteration
            #And also it was the last element check if breaking
            if i == len(regex):
                break
            #Testing if concatenation
            if self.is_concat(regex[i]):
                if concatenation == -1:
                    concatenation = i
                continue
            #Testing for kleene
```

```
            if regex[i] == '*':
                if kleene == -1:
                    kleene = i
                continue
            #Testing for or operator
            if regex[i] == '|':
                if or_operator == -1:
                    or_operator = i

        #Setting the current operation by priority
        if or_operator != -1:
            #Found an or operation
            self.item = '|'

self.children.append(RegexNode(self.trim_brackets(regex[:or_operator])))
self.children.append(RegexNode(self.trim_brackets(regex[(or_operator+1):])))
        elif concatenation != -1:
            #Found a concatenation
            self.item = '.'
self.children.append(RegexNode(self.trim_brackets(regex[:concatenation])))

self.children.append(RegexNode(self.trim_brackets(regex[concatenation:])))
        elif kleene != -1:
            #Found a kleene
            self.item = '*'

self.children.append(RegexNode(self.trim_brackets(regex[:kleene])))
    def calc_functions(self, pos, followpos):
        if self.is_letter(self.item):
            #Is a leaf
            self.firstpos = [pos]
            self.lastpos = [pos]
```

```
            self.position = pos
            #Add the position in the followpos list
            followpos.append([self.item,[]])
            return pos+1
        #Is an internal node
        for child in self.children:
            pos = child.calc_functions(pos, followpos)
        #Calculate current functions
        if self.item == '.':
            #Is concatenation
            #Firstpos
            if self.children[0].nullable:
                self.firstpos = sorted(list(set(self.children[0].firstpos +
self.children[1].firstpos)))
            else:
                self.firstpos = deepcopy(self.children[0].firstpos)
            #Lastpos
            if self.children[1].nullable:
                self.lastpos = sorted(list(set(self.children[0].lastpos +
self.children[1].lastpos)))
            else:
                self.lastpos = deepcopy(self.children[1].lastpos)
            #Nullable
            self.nullable = self.children[0].nullable and
self.children[1].nullable
            #Followpos
            for i in self.children[0].lastpos:
                for j in self.children[1].firstpos:
                    if j not in followpos[i][1]:
                        followpos[i][1] = sorted(followpos[i][1] + [j])
```

```
        elif self.item == '|':
            #Is or operator
            #Firstpos
            self.firstpos = sorted(list(set(self.children[0].firstpos +
self.children[1].firstpos)))
            #Lastpos
            self.lastpos = sorted(list(set(self.children[0].lastpos +
self.children[1].lastpos)))
            #Nullable
            self.nullable = self.children[0].nullable or
self.children[1].nullable
        elif self.item == '*':
            #Is kleene
            #Firstpos
            self.firstpos = deepcopy(self.children[0].firstpos)
            #Lastpos
            self.lastpos = deepcopy(self.children[0].lastpos)
            #Nullable
            self.nullable = True
            #Followpos
            for i in self.children[0].lastpos:
                for j in self.children[0].firstpos:
                    if j not in followpos[i][1]:
                        followpos[i][1] = sorted(followpos[i][1] + [j])
        return pos
    def write_level(self, level):
        print(str(level) + ' ' + self.item, self.firstpos, self.lastpos,
self.nullable, '' if self.position == None else self.position)
        for child in self.children:
            child.write_level(level+1)
```

```
class RegexTree:
    def __init__(self, regex):
        self.root = RegexNode(regex)
        self.followpos = []
        self.functions()
    def write(self):
        self.root.write_level(0)
    def functions(self):
        positions = self.root.calc_functions(0, self.followpos)
        if DEBUG == True:
            print(self.followpos)
    def toDfa(self):
        def contains_hashtag(q):
            for i in q:
                if self.followpos[i][0] == '#':
                    return True
            return False
        M = [] #Marked states
        Q = [] #States list in the followpos form ( array of positions )
        V = alphabet - {'#', lambda_symbol if use_lambda else ''} #Automata
alphabet
        d = [] #Delta function, an array of dictionaries d[q] = {x1:q1, x2:q2
..} where d(q,x1) = q1, d(q,x2) = q2..
        F = [] #FInal states list in the form of indexes (int)
        q0 = self.root.firstpos
        Q.append(q0)
        if contains_hashtag(q0):
            F.append(Q.index(q0))
        while len(Q) - len(M) > 0:
            #There exists one unmarked
            #We take one of those
```

```
            q = [i for i in Q if i not in M][0]
            #Generating the delta dictionary for the new state
            d.append({})
            #We mark it
            M.append(q)
            #For each letter in the automata's alphabet
            for a in V:
                # Compute destination state ( d(q,a) = U )
                U = []
                #Compute U
                #foreach position in state
                for i in q:
                    #if i has label a
                    if self.followpos[i][0] == a:
                        #We add the position to U's composition
                        U = U + self.followpos[i][1]
                U = sorted(list(set(U)))
                #Checking if this is a valid state
                if len(U) == 0:
                    #No positions, skipping, it won't produce any new states
(
also won't be final )
                    continue
                if U not in Q:
                    Q.append(U)
                    if contains_hashtag(U):
                        F.append(Q.index(U))
                #d(q,a) = U
                d[Q.index(q)][a] = Q.index(U)

        return Dfa(Q,V,d,Q.index(q0),F)
```

```
class Dfa:
    def __init__(self,Q,V,d,q0,F):
        self.Q = Q
        self.V = V
        self.d = d
        self.q0 = q0
        self.F = F
    def run(self, text):
        #Checking if the input is in the current alphabet
        if len(set(text) - self.V) != 0:
            #Not all the characters are in the language
            print('ERROR characters',(set(text)-self.V),'are not in the
automata\'s alphabet')
            exit(0)
        #Running the automata
        q = self.q0
        for i in text:
            #Check if transition exists
            if q >= len(self.d):
                print('Message NOT accepted, state has no transitions')
                exit(0)
            if i not in self.d[q].keys():
                print('Message NOT accepted, state has no transitions with
the character')
                exit(0)
            #Execute transition
            q = self.d[q][i]
        if q in self.F:
            print('Message accepted!')
        else:
            print('Message NOT accepted, stopped in an unfinal state')
```

```
    def write(self):
        for i in range(len(self.Q)):
            #Printing index, the delta fuunction for that transition and if
it's final state
            print(i,self.d[i],'F' if i in self.F else '')


#Preprocessing Functions
def preprocess(regex):
    regex = clean_kleene(regex)
    regex = regex.replace(' ','')
    regex = '(' + regex + ')' + '#'
    while '()' in regex:
        regex = regex.replace('()','')
    return regex


def clean_kleene(regex):
    for i in range(0, len(regex) - 1):
        while i < len(regex) - 1 and regex[i + 1] == regex[i] and regex[i] ==
'*':
            regex = regex[:i] + regex[i + 1:]
    return regex


def gen_alphabet(regex):
    return set(regex) - set('()|*')


#Settings
DEBUG = False
use_lambda = False
lambda_symbol = '_'
alphabet = None
```

```
#Main
regex = '(a|ba)*#|a)'
#Check
if not is_valid_regex(regex):
    exit(0)
#Preprocess regex and generate the alphabet
p_regex = preprocess(regex)
alphabet = gen_alphabet(p_regex)
#add optional letters that don't appear in the expression
extra = ''
alphabet = alphabet.union(set(extra))

#Construct
tree = RegexTree(p_regex)
if DEBUG:
    tree.write()
dfa = tree.toDfa()
#Test
message = ''
print('This is the regex : ' + regex)
print('This is the alphabet : ' + ''.join(sorted(alphabet)))
print('This is the automata : \n')
dfa.write()
print('\nTesting for : "'+message+'" : ')
dfa.run(message)
```

OUTPUT:

```
This is the regex : ((a|ba)*#|a)
This is the alphabet : #ab
This is the automata :

0 {'b': 1, 'a': 2} F
1 {'a': 3}
2 {'b': 1, 'a': 3} F
3 {'b': 1, 'a': 3} F

Testing for : "" :
Message accepted!
```

DETERMINISTIC FINITE AUTOMATA

```
import sys

# Set of all characters that can appear in the input

alphabet = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_")

# Set of keywords

keywords = set("if else while int return void")

# Set of operators

operators = set("+ - * /")

# Set of separators

separators = set("( ) { } [ ] , ;")

# Set of whitespaces

whitespaces = set(" \t\n")

# DFA states

START = 0

IDENTIFIER = 1

INTEGER = 2

OPERATOR = 3

SEPARATOR = 4
```

```
END = 5


def lex(input_str):
    """Perform lexical analysis on the input string"""
    tokens = []
    i = 0
    while i < len(input_str):
        # Initialize the DFA state to start state
        state = START
        j = i
        while True:
            # Get the current character
            c = input_str[j]

            if state == START:
                if c in alphabet:
                    state = IDENTIFIER
                elif c in "0123456789":
                    state = INTEGER
                elif c in operators:
                    state =OPERATOR
                elif c in whitespaces:
                    state = START
                else:
                    print(f"Invalid character: {c}")
                    sys.exit(1)
            elif state == IDENTIFIER:
                if c in alphabet:
                    pass
                else:
```

```
                state = END
                j -= 1
        elif state == INTEGER:
            if c in "0123456789":
                pass
            else:
                state = END
                j -= 1
        elif state == OPERATOR:
            state = END
            j -= 1
        elif state == SEPARATOR:
            state = END
            j -= 1
        elif state == END:
            # Tokenization complete
            break
        j += 1

    # Determine the token type based on the DFA state
    token_value = input_str[i:j]
    if state == IDENTIFIER:
        token_type = "IDENTIFIER"
        if token_value in keywords:
            token_type = "KEYWORD"
    elif state == INTEGER:
        token_type = "INTEGER"
    elif state == OPERATOR:
        token_type = "OPERATOR"
    elif state == SEPARATOR:
```

```
            token_type = "SEPARATOR"
        else:
            token_type="UNKNOWN"

        # Add the token to the list of tokens
        tokens.append((token_type, token_value))
        # Move to the next character
        i = j + 1
    return tokens
def main():
    # Read the input from a file
 with open("C:/Users/Aravind Vadlapudi/OneDrive - Amrita Vishwa Vidyapeetham/Desktop/Clg Documents/Semester-5/FLA Project/input.txt", "r") as f:
        input_str = f.read()


    # Perform lexical analysis
    tokens = lex(input_str)


    # Print the results for each token
    for token_type, token_value in tokens:
        #print(f"Token type: {token_type}")
        print(f"Token value: {token_value}")


if __name__ == "__main__":
    main()
```

OUTPUT:

```
Token value: AI
Token value: redirects
Token value: here
```

Token value: For

Token value: other

Token value: uses

Token value: see

Token value: AI

Token value: disambiguation

Token value: Artificial

Token value: intelligence

Token value: disambiguation

Token value: and

Token value: Intelligent

Token value: agent

Token value: Part

Token value: of

Token value: a

Token value: series

Token value: on

Token value: Artificial

Token value: intelligence

Token value: Anatomy1751201

Token value:

Major

Token value: goals

Token value: Approaches

Token value: Philosophy

Token value: History

Token value: Technology

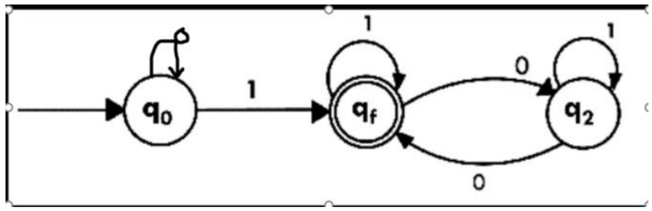
Token value: Glossary

Token value: vte

DFA TO LEXER

```
def lexical_analyzer(dfa, input_string):

    # initialize the current state to the start state of the DFA

    current_state = dfa['start']

    # initialize the list of tokens

    tokens = []

    # initialize a variable to store the current token

    current_token = ''

    # initialize a flag to indicate whether the input string is accepted by
the DFA

    accepted = False

    # iterate through the input string

    for c in input_string:

        # update the current state based on the current character and the
transition function of the DFA

        current_state = dfa['transition'][current_state][c]

        # add the character to the current token

        current_token += c

        # if the current state is an accepting state, add the current token
to the list of tokens and reset the current token

        if current_state in dfa['accept']:

            tokens.append(current_token)

            current_token = ''

    # set the accepted flag to True if the input string has been fully
processed and the current token is empty

    if not current_token:

        accepted = True

    # return the list of tokens and the accepted flag

    return tokens, accepted
```

Regular expression: 0* 1 (1* 01* 01)

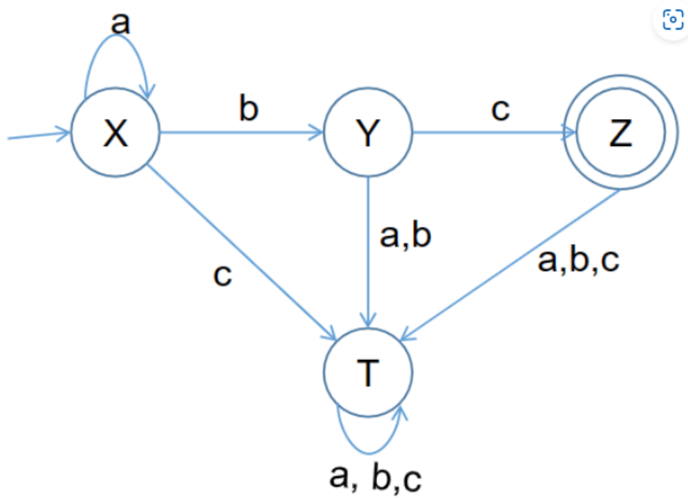



```
# define the DFA
dfa = {
    'start': 0,
    'accept': {1},
    'transition': {
        0: {'a': 0, 'b': 1},
        1: {'a': 2, 'b': 1},
        2: {'a': 1, 'b': 2},
    }
}

# define the input string
#input_string = "bbabab"
input_string = "aabaaa"
# call the lexical analyzer
tokens = lexical_analyzer(dfa, input_string)

# print the list of tokens
print(tokens)
```

OUTPUT:

```
(['aab', 'aa'], False)
```

Regular expression: a*bc




```
# define the DFA
dfa = {
    'start': 0,
    'accept': {2},
    'transition': {
        0: {'a': 0, 'b': 1, 'c': 3},
        1: {'a': 3, 'b': 3, 'c': 2},
        2: {'a': 3, 'b': 3, 'c': 3},
        3: {'a': 3, 'b': 3, 'c': 3},
    }
}

# define the input string
input_string = "aaaabc"

# call the lexical analyzer
tokens = lexical_analyzer(dfa, input_string)

# print the list of tokens
print(tokens)
```

OUTPUT:

```
(['aaaabc'], True)
```

```
# define the DFA
dfa = {
    'start': 0,
    'accept': {2},
    'transition': {
        0: {'a': 1, 'b': 1},
        1: {'a': 2, 'b': 1},
        2: {'a': 1, 'b': 1},
    }
}

# define the input string
input_string = "aaaabba"

# call the lexical analyzer
tokens = lexical_analyzer(dfa, input_string)

# print the list of tokens
print(tokens)
```

OUTPUT:

```
(['aa', 'aa', 'bba'], True)
```

Mealy and Moore Machine

```
class Token:
    def __init__(self, value, type):
        self.value = value
```

```
        self.type = type

    def __repr__(self):
        return '{}({}, {})'.format(self.__class__.__name__, repr(self.value),
repr(self.type))

class PatternError(ValueError):
    pass

class Lexer:
    def __init__(self, patterns, ignore_pattern=None):
        for pattern, _ in patterns:
            if pattern.match(''):
                raise PatternError('Pattern matches empty string')
            if ignore_pattern and ignore_pattern.match(''):
                raise PatternError('Ignore pattern matches empty string')
        self.patterns = patterns
        self.ignore_pattern = ignore_pattern

    def lex(self, string):
        cursor = 0
        line_number = 1
        column_number = 0
        while cursor < len(string):
            try:
                match = self.ignore_pattern.match(string, cursor)
            except AttributeError:
                pass
            else:
                if match is not None:
                    line_number += match.group().count('\n')
```

```
                cursor = match.end()
                line_start = string.rfind('\n', 0, cursor) + 1
                if line_start == 0:
                    column_number = 0
                else:
                    column_number = match.end() - line_start
                continue

        for pattern, type in self.patterns:
            match = pattern.match(string, cursor)
            if match is not None:
                line_number += match.group().count('\n')
                cursor = match.end()
                line_start = string.rfind('\n', 0, cursor) + 1
                if line_start == 0:
                    column_number = 0
                else:
                    column_number = match.end() - line_start
                yield Token(match.group(), type)
                break
        else:
            start = string.rfind('\n', 0, cursor) + 1
            stop = string.find('\n', cursor)
            if stop == -1:
                stop = len(string)
            print(string[start:stop])
            from sys import stderr
            raise SyntaxError('unexpected token at line {}, column {}'.format(line_number, column_number))


import re
```

```
# define the patterns and corresponding token types
patterns = [
    (re.compile(r'\d+'), 'NUMBER'),
    (re.compile(r'[a-zA-Z_][a-zA-Z0-9_]*'), 'NAME'),
    (re.compile(r'\+'), 'PLUS'),
    (re.compile(r'\-'), 'MINUS'),
    (re.compile(r'\*'), 'MULTIPLY'),
    (re.compile(r'\/'), 'DIVIDE'),
    (re.compile(r'\^'), 'EXPONENT'),
    (re.compile(r'\%'), 'MODULO'),
    (re.compile(r'\('), 'LEFT_PAREN'),
    (re.compile(r'\)'), 'RIGHT_PAREN'),
    (re.compile(r'\['), 'LEFT_BRACKET'),
    (re.compile(r'\]'), 'RIGHT_BRACKET'),
    (re.compile(r'\{'), 'LEFT_BRACE'),
    (re.compile(r'\}'), 'RIGHT_BRACE'),
    (re.compile(r'[^\d\w\s+\-*/^%()\[\]{}]+'), 'SYMBOL')
]

# define the ignore pattern
ignore_pattern = re.compile(r'\s+')

def main():
    # take input from the user
    string = input("Enter the string to be lexed: ")

    # create an instance of the Lexer class
    lexer = Lexer(patterns, ignore_pattern)
```

```
    # call the lex method on the input string
    tokens = lexer.lex(string)

    # iterate over the generator and print each token
    for token in tokens:
        print(token)

if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter the string to be lexed: automata
Token('automata', 'NAME')
```

*****