INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
CAMPUS GUADALAJARA

Development of advanced computer science applications

Group 302

Teachers

Rodolfo Julio Castello Zetina

Luis Guillermo Hernández Rojas

Salvador Miguel Hinojosa Cervantes

Activity 3.1. Lexical analyzer

Written Report

Samuel Alejandro Díaz del Guante Ochoa A01637592

Guadalajara, Jal., June 10 2023

Introduction

Summary

The purpose of this activity is to create a lexical analyzer, also known as a scanner. It is an essential part of a compiler/interpreter and it is responsible for breaking down the source code of a programming language into a sequence of meaningful tokens. Common tokens include keywords, identifiers, operators, literals, and punctuation symbols. The scanner reads the input source code character by character and groups them into tokens based on a set of rules. This paper aims to explain the design and implementation process of a lexical analyser. Expect to find the specifications and rules of the language, the modeling of the language's automata and resulting transition table as well as explanations about the code itself, its outputs and test scenarios.

Notation

The implementation of a lexical analyzer can be achieved by two approaches. One of them is code driven meaning every case and rule from the defined language has to be stated directly in the code. This might seem the logical way to implement the scanner however it has many drawbacks. The main one being that as new characters and rules are introduced to the language, the code grows dramatically in size and complexity and the chance of error skyrockets with it. Also, debugging and maintenance becomes increasingly difficult since the logic of the scanner hangs entirely on code and miniscule changes can have unintended consequences. Fortunately there exists an alternative to this way. Instead of putting all the rules of the language in code, we can generate a table that contains all the transitions between each state as the scanner reads and processes incoming characters. A scanner with the help of a transition table is the ideal way to solve this problem as it reduces considerably the workload of the scanner in the code and as new rules and characters are introduced to the language almost always the transition table is the only element that needs adjustments. In general it is far more reliable and sustainable than a code driven lexical analyzer.

Analysis

The lexical analyzer is for a programming language named C Minus Minus (C–). Hereinafter the language's rules and tokens:

Tokens

A. Keywords

a. int
b. float
c. string
d. for
e. if
f. else

g. while
h. return
i. read
j. write
k. void

B. Special symbols

a. + arithmetic addition operation
b. - arithmetic subtraction operation
c. * arithmetic multiplication operation
d. / arithmetic division operation
e. < logic operator less than
f. <= logic operator less or equal than
g. > logic operator greater than
h. >= logic operator greater or equal than
i. == logic operator equal
j. != logic operator different
k. = assignation
l. ; semicolon
m. , coma
n. " quotation mark
o. . dot
p. (open parenthesis
q.) close parenthesis
r. [open square brackets
s.] close square brackets
t. { open curly brackets
u. } close curly brackets
v. /* open comment
w. */ close comment

C. Regular expressions

a. ID
b. STRING
c. NUMBER

D. Others

a. Comments (/*...*/)
b. Strings (“...”)

E. Errors

Rules

A. Keywords

a. All keywords are reserved words and they are NOT case sensitive, i.e., they can be written in lowercase and/or capital letters.

B. Special symbols

a. Special symbols are considered delimiters along with most tokens.

C. Regular expressions

a. letter = [a-zA-Z]
b. digit = [0-9]

c. STRING = "•*"
d. ID = letter (letter | digit)*
    i. Identifiers are NOT case sensitive, i.e., they can be written in lowercase and/or capital letters.
e. NUMBER = digit+ (. digit+)?

D. Others

a. Comments (/*...*/)
    i. Comments are enclosed by quotation marks and may include any character but must be enclosed before the end of the file.
b. Strings (“...”)
    i. Strings constants are enclosed by quotation marks and may include any character but must be enclosed before a new line or the end of the file.
c. White space consists of blanks, newlines, and tabs. White space is ignored, but it MUST be recognized. White space together with IDs, STRINGs, NUMBERs, and keywords are considered as delimiters.

E. Errors

a. Consists of unknown characters and bad token formation such as a “**!**” not being followed by a “**=**”.
b. The scanner must recognize the type of error, stop analysis and output what it has scanned up to that point.

The automata of the language

The first step to create a scanner based on a transition table is to map all the states and transitions between them. For this scanner we are employing the graphic representation of a Deterministic Finite Automaton (Figure 1). A DFA is a type of finite state machine that recognizes and accepts or rejects strings of symbols based on a set of predefined rules. Below is a description of each of the parts of the DFA in more detail.

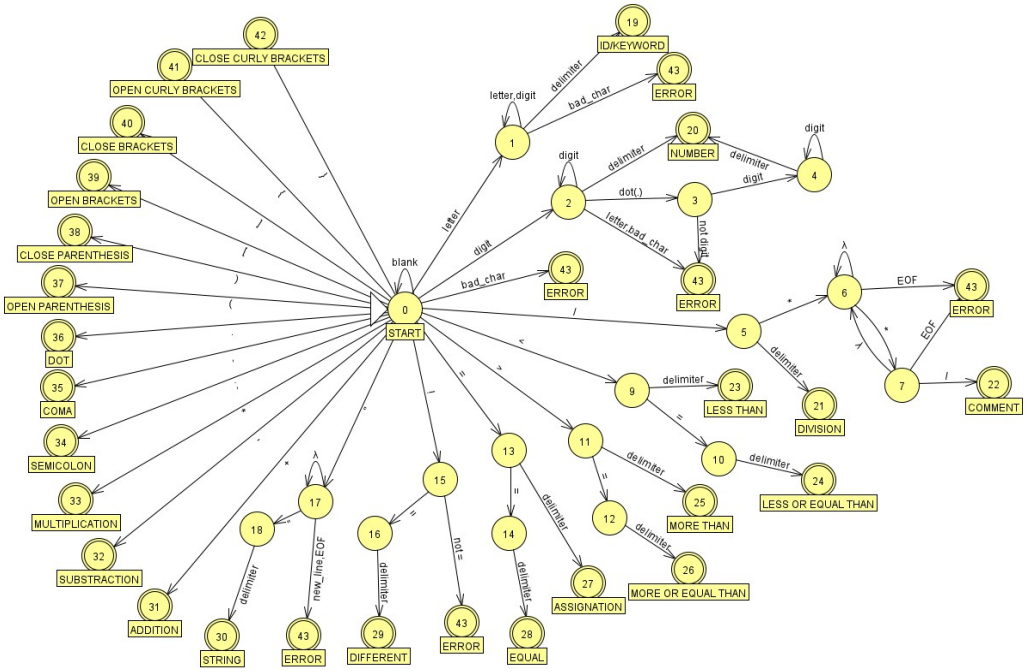

Figure 1. Whole DFA of the language

If the first character read by the scanner is a known letter then it is almost certain that the token will yield to either a keyword or an identifier (Figure 2). The scanner then has the task to continue to append letters or numbers until a delimiter is reached, then the code handles the logic to determine if the accepted token is a keyword or an identifier. The only case where a token can become bad is if an unknown character is reached while processing the input in the first state.

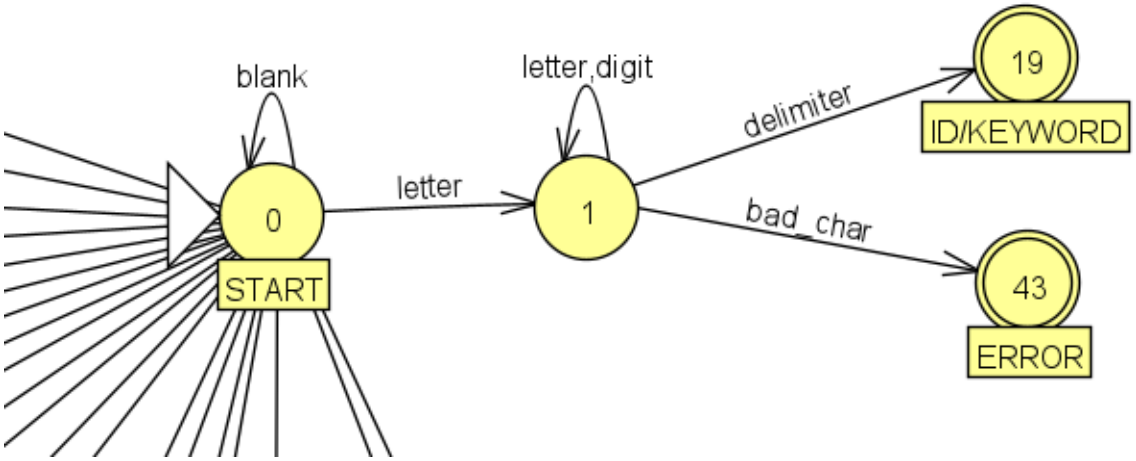

Figure 2. DFA section for Identifiers and Keywords

Numbers are next (Figure 4). By the definition of the language the scanner must be able to parse integers and float numbers, though the distinction between them doesn't come into play in this stage but later on inside the code logic. A number can only receive digits and if it is tailed by a letter or bad character it must result in an error final state. In case the special character dot (.) is reached before a valid delimiter then it must be followed by digit otherwise the scanner must output the error state.

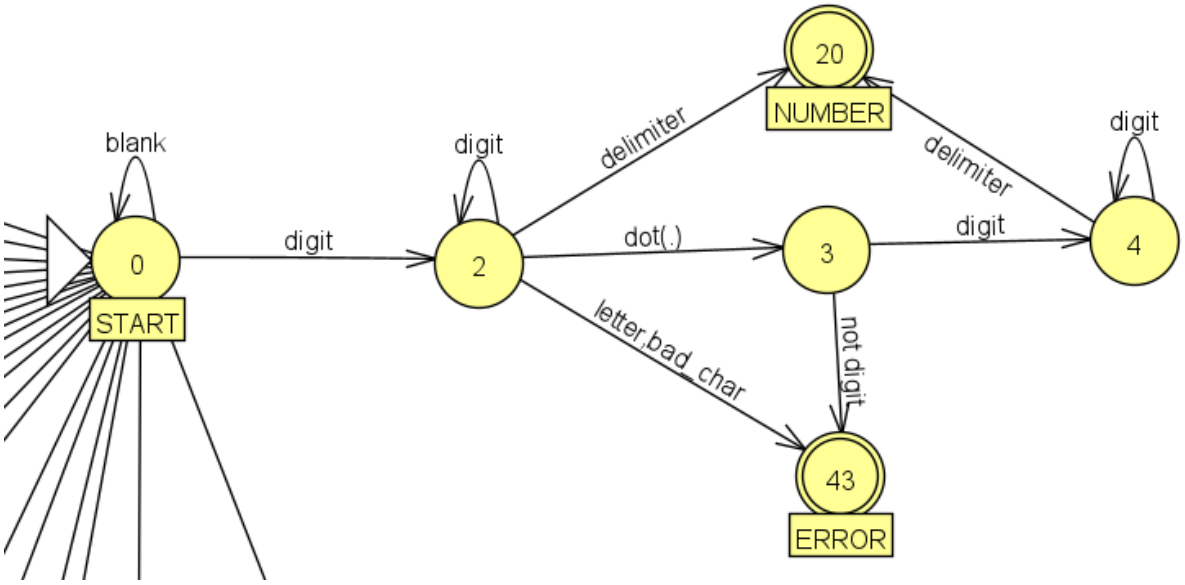

Figure 3. DFA section for Numbers

The division operator (/) can be interpreted as the special character if the next character is any delimiter but a multiplication operator (*). If these two operator are next to each other (/*) then a comment has been opened and it must be close with another set of multiplication and division operators (*/) before reaching the end of the file. Comments are special in the sense that they can contain any sort of character, those outside the definition of the language and blanks such as white spaces, tabs and newlines (Figure 4).

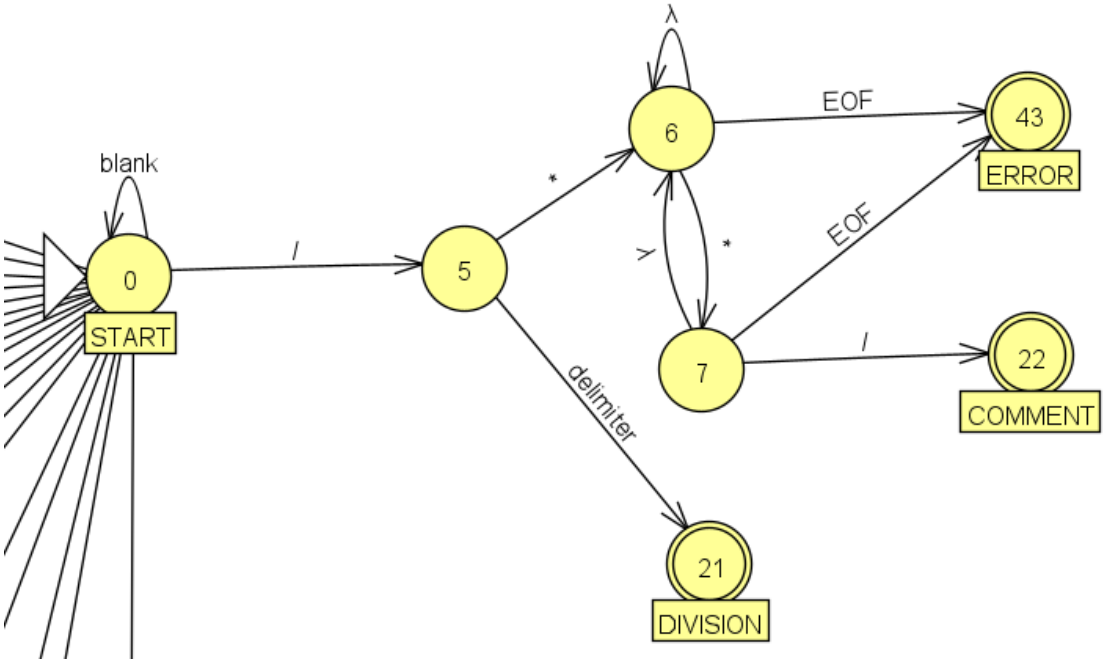

Figure 4. DFA section for comments and division operator

The consequent batch of special characters share the special characteristic of potentially ending with one or two characters (Figure 5). Almost all the logic operators like less (<) and greater (>) than have an ending state if a delimiter is next to it unless that character is another assignment sign (=). The only two exceptions to this rule in this category are the logic operator different (!=) and string constants. In the definition of the language there is not a special character that consists of an exclamation mark (!), it always has to be next to the assignment sign (=). The other special case has to do with string constants and double quotations (“) as they cannot stand on their own and must be enclosed by another quotation before the end of a line or the end of a file. Similar to comments, string constants can have any type of character within it including white spaces and tabs.

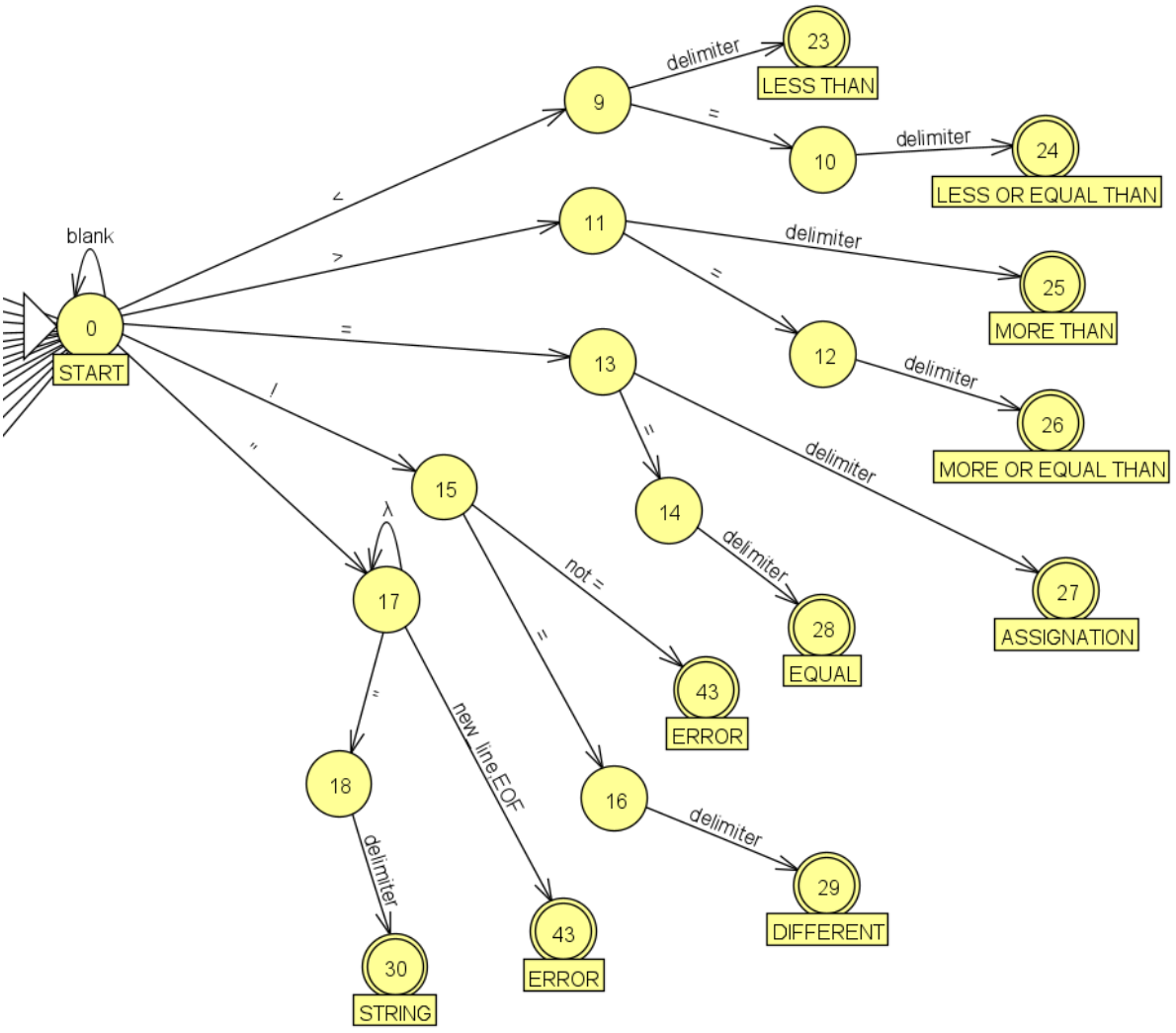

Figure 5. DFA section for special characters with two possible final states and string constants

As soon as the rest of the special characters are read by the scanner they can be interpreted as final states and converted into tokens since there does not exist further transitions to other states (Figure 6).

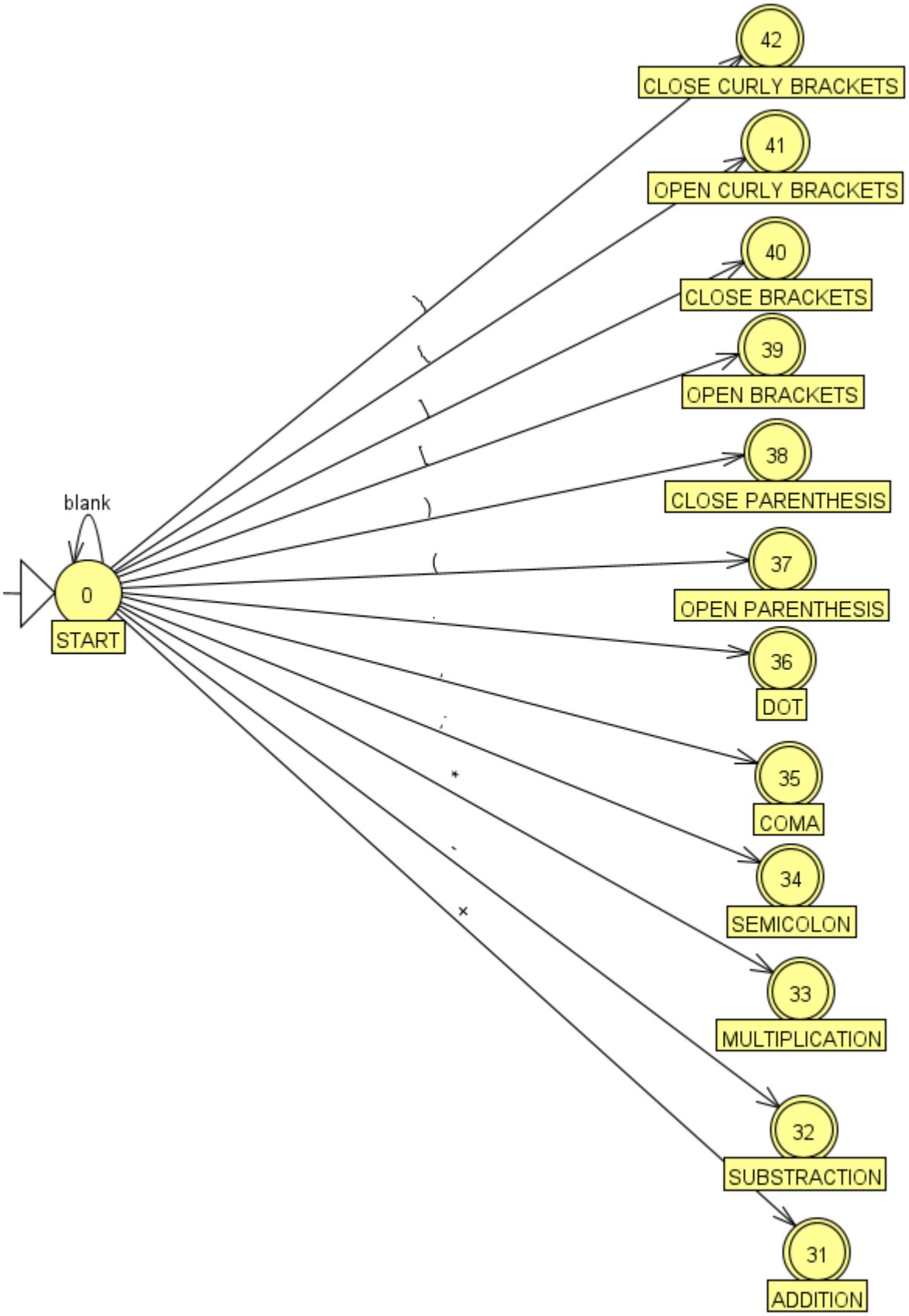

Figure 6. DFA section for special characters with one possible final state

Although many of the previous parts explore edge cases when the token leads to an error final state, it is important remember that if the next potential token to be proceed by the lexical analyzer starts with a character that is not recognized by the rules of the language, the scanner must be stopped and the error identified as it is one of the core functions of the scanner (Figure 7).

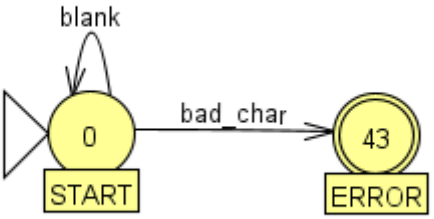

Figure 7. DFA section for errors

Tables of the scanner

Transition table

Completion of the DFA diagram means that the transition table can finally be done. Now what once was a state in the DFA is represented by each row in the column and every cell represents the transition or change of state. Not final states are colored in white while accepted final states are in green and error states in red (Table 1). This transition table (Table 2) went through a couple of iterations to optimize minimum space required and legibility. First, just as the DFA, the transition table saves a lot of space by categorizing identifiers and keywords in the same space since they share the same logic to form a token. The other breakthrough was achieved by pushing at the end all the final states which makes the transition table easier to read and saves space as well. Another step to minimize space that could have been implemented is to group all the special characters with one possible final state and identify each character in the code similar to what was done for the identifier and keyword tokens.

	Accepted States
	Error State
letter	[a-zA-Z]
digit	[0-9]
blank	(space|tab)
new_line	\n
EOF	End of File
STRING	"•*"
ID	letter (letter | digit)*
NUMBER	digit+ (. digit+)?

Table 1. Legend of the transition table

	letter	digit	+	-	*	/	<	>	=	!	;	,	"	.	(	)	[	]	{	}	blank	new_line	EOF	bad_char	
0	1	2	31	32	33	5	9	11	13	15	34	35	17	36	37	38	39	40	41	42	0	0	0	43	
1	1	1	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	43	
2	43	2	20	20	20	20	20	20	20	20	20	20	20	3	20	20	20	20	20	20	20	20	20	43	
3	43	4	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	
4	43	4	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	43	
5	21	21	21	21	6	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	
6	6	6	6	6	7	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	43	6	
7	6	6	6	6	6	8	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	43	6	
8	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	
9	23	23	23	23	23	23	23	23	10	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	
10	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	
11	25	25	25	25	25	25	25	25	12	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	
12	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	
13	27	27	27	27	27	27	27	27	14	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	
14	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	
15	43	43	43	43	43	43	43	43	16	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	
16	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	
17	17	17	17	17	17	17	17	17	17	17	17	17	18	17	17	17	17	17	17	17	17	43	43	17	
18	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	
19																									ID/KEYWORD
20																									NUMBER
21																									/
22																									COMMENT
23																									<
24																									<=
25																									>
26																									>=
27																									=
28																									==
29																									!=
30																									STRING
31																									+
32																									-
33																									*
34																									;
35																									,
36																									.
37																									(
38																									)
39																									[
40																									]
41																									{
42																									}
43																									ERROR

Table 2. Transition table

Token Table

After the scanner has gathered all the tokens it is time to output them in a way that maps an identifier with a token. Since the transition table has way more states than there are tokens a new table must be put into place to represent the tokens. The token table contains just that (Table 3). The arrangement of the tokens is in a particular way that would make sense for the later stages of compiler or interpreter. The first eleven tokens correspond to all keywords from which tokens one to four are data types, five to nine are for control statements and the next two are for input and output functions. Tokens twelve to fourteen are for regular expressions and although they are ignored by most compilers, comments must be recognized as tokens and thus have the fifteenth identifier in the table. Special characters have spots sixteen to thirty-six and thirty-seven is reserved for all the error tokens.

ID	Token
1	int
2	float
3	string
4	void
5	if
6	else
7	while
8	for
9	return

10	read
11	write
12	ID
13	STRING
14	NUMBER
15	COMMENT
16	+
17	-
18	*
19	/
20	<
21	<=
22	>
23	>=
24	==
25	!=
26	=
27	;
28	,
29	“
30	.
31	(
32	)
33	[
34	]
35	{
36	}
37	ERROR

Table 3. Token table

Types of errors

There are a couple of errors that can be extracted from the DFA and transition table. The scanner must be able to identify all of them and report them accordingly. Here is a list with types of errors expected to be encountered by the scanner.

- **Bad character inside the token**
  - A token contained a character outside of the English alphabet.
- **A float number was not properly stated**
  - A float number ended with a '.' instead of a digit.
- **A number contains a letter**
  - A token whose first character is a digit has also a letter in it.
- **A comment was not properly stated**
  - Comment (/* ... */) was not properly closed.
- **A string was not properly stated**
  - String constants (" ... ") were not properly closed.
- **Logic operator different is incomplete**
  - '!' was written but never followed by a '='.

Design

Once the DFA and transition tables have been created it is time to code the rest of the behavior of the scanner using a programming language. Usually scanners are coded in lightweight, low level programming languages such as C but for the scope of this activity it was decided to be done using python since it has a wide set of functions to manipulate strings easily and requires fewer lines of code in contrast to a language like C. This section of the essay aims to explain in detail how each of the parts thus far come together with the code to finalize the implementation of the scanner and to provide an example of the type of input it might receive and how the outputs of the application look like.

Implemented code

This section of the paper has been devised to help explain the implementation of each of the components and functions of the program and to offer enough context and traceability to the design of the model.

Global variables and imported libraries

The only requirement a user must meet to be able to run this application is to have the Pandas library installed in their virtual environment. Here are created the variables to store all the tokens and table information.

```
'''
Samuel Alejandro Diaz del Guante Ochoa - A01637592
Computer Science Advanced Applications Development - TC3002B
Module #3: Compilers
Lexical Analysis
'''
```

```
# Import libraries
import pandas as pd
import os

# Set global variables
DIR = (os.path.join(os.path.dirname(__file__)))

transitionTable = pd.read_csv(DIR+'/input/TransitionTable.txt',
sep='\t')
validTokens = pd.read_csv(DIR+'/input/ValidTokens.txt', sep='\t')

token_list = []
identifiers_table = []
numbers_table = []
strings_table = []
```

Main function

Contains the path to the source code to analyze by the scanner. It can be modified to find another file to analyze or just go to that directory and write your own C– code.

```
def main():

    pathToSource = './input/source.txt'
    with open(pathToSource, 'r') as f:
        text = f.read()

    scanner(text)

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

Scanner function

It contains all the logic of the program to be able to tokenize incoming characters. It starts by fetching the string of characters it is going to analyze, passing some hygiene data checks and finally going through each single character one by one until it reaches the end of the file. If an error state is reached then the scanner stops getting new characters and determines what went wrong and outputs whatever it had scanned up to that point. Otherwise, it completes the reading of the whole input and besides outputting the tokens and tables, it lets the user know the lexical analysis went smoothly.

```
# Main function of the program
# Tokenizes each keyword, special character using a transition table
# Outputs .txt file that each represent tokens, identifiers, numbers
and string tables
def scanner(text:str) -> None:
```

```
    # Scanner wont do anything if text is empty
    if not len(text): return
    # Add next line char at the end of the input string if file does
not already have it to avoid errors while reading file
    text += '\n' if text[-1] != "\n" else ''
    # Flag to stop process if a bad token has been reached
    err = False
    # Process every char in the input file or until an error is
detected
    while text and not err:
        # Pop and store the first char in queue
        char = text[0]
        text = text[1:]
        # Try again if the first char of the token is a blank
        if char in "\t\n ":
            continue
        # Store the first char of the token and get its first state
        token = char
        state = getState(0,char)
        # Continue to get next token char and state until it has
reached a final state
        while text and state < 19:
            state = getState(state,text[0])
            if state < 19:
                char = text[0]
                text = text[1:]
                token += char
        # Error state
        if state < 19 or state == 43:
            errorHandler(token,37)
            err = True
        # Valid state
        else:
            processToken(token,state)
    # Output scanner and symbol tables contents
    outputScanner()
    outputTables(identifiers_table,'IdentifiersTable.txt',['Lexeme'])
    outputTables(numbers_table,'NumbersTable.txt',['Lexeme', 'Type'])
    outputTables(strings_table,'StringTable.txt',['Lexeme'])

    print("❌ Something went wrong! Check the scanner output to verify
the bad token.") if err \
```

```
        else print("✅ Looks green to me! Check the scanner output and
lexeme tables.")
```

Get state and return transition function

This function is in charge of taking the current state and next character in the string and returning the next state in the sequence by looking into the transition table provided at the beginning of the program.

```
# Get the current state and map it to the next state from the
transition table
# Get current character and state
# Return the next state of the token
def getState(state:int, char:str) -> int:

    blank = "\t "
    new_line = "\n"
    specialChars = "+-*/<>=!;,\".()[]{}"
    alphabet = [chr(value) for value in range(97, 123)]

    if char in blank:
        col = transitionTable['blank']
    elif char in new_line:
        col = transitionTable['new_line']
    elif char.lower() in alphabet:
        col = transitionTable['letter']
    elif char.isdigit():
        col = transitionTable['digit']
    elif char in specialChars:
        col = transitionTable[char]
    else:
        col = transitionTable['bad_char']

    return col.loc[state]
```

Store token function

If the token has reached a final accepted state then that token is further analyzed so it can be paired with its token ID, stored and added to one of the tables if it applies to it. Token ID information is provided by the valid token table.

```
# Process token to store in transition table and (if it applies) to one
of the symbol tables
# Get token and the ending state to be translated to state found in the
ValidToken.txt table
def processToken(token:str, state:int) -> None:

    t = list(validTokens['Token'].values)
```

```
    t.insert(0,'')

    # IDs/Keywords
    if state == 19:
        # Lowercase input since IDs/Keywords are NOT case sensitive
        token = token.lower()
        # Keywords
        if token in t:
            s = t.index(token)
            token_list.append([s])
        # IDs
        else:
            # Determine if it is already in the table
            if not token in identifiers_table:
                identifiers_table.append(token)
            s = t.index('ID')
            index = identifiers_table.index(token)+1
            token_list.append([s,index])
    # Numbers
    elif state == 20:
        # Determine the type of the number
        num_type = 'float' if '.' in token else 'int'
        numbers_table.append([token,num_type])
        s = t.index('NUMBER')
        index = len(numbers_table)
        token_list.append([s,index])
    # Strings
    elif state == 30:
        strings_table.append(token)
        s = t.index('STRING')
        index = len(strings_table)
        token_list.append([s,index])
    # Comments
    elif state == 22:
        s = t.index('COMMENT')
        token_list.append([s])
    # Special characters
    elif token in t:
        s = t.index(token)
        token_list.append([s])
```

Error handler function

This function can only be called if an error token has been detected. Its main purpose is to determine the root cause of the bad token to then be displayed to the user.

```
# This function manages errors if a token finalizes in said state
# Get token, final state of token and analyzed text
# Return text string with modifications
def errorHandler(token:str, state:int) -> None:

    # Default message, unrecognizable character
    message = "Token has an unknown character"
    # '!=' was not fully stated
    if token == '!':
        message = "'!' has to be followed by a '='"
    # Number end with a dot
    elif token[-1] == '.':
        message = "Float numbers cannot end by with a '.'"
    # String was not closed
    elif token[0] == '"':
        message = "String was not properly closed"
    # Comment was not closed
    elif token[:2] == '/*':
        message = "Comment was not properly closed"
    # Number contains a letter
    elif token[0].isdigit() and (token.isupper() or token.islower()):
        message = "Numbers cannot contain letters"

    token_list.append([state,message])
```

Output results functions

A collection of functions that are in charge of taking all the stored tokens and tables from the scanner and output them in a user friendly format in a couple of .txt files the user can latter check if source code provided complies with all the rules of the C– language.

```
# Formats and outputs the different types of tables to a .txt
# Gets the data structure that contains the information of the table
# (usually contains Entry number and lexeme), file and column names
def outputTables(l:list, fileName:str, col_names:list) -> None:

    table = pd.DataFrame(l, columns=col_names)
    table.insert(0, 'Entry', range(1,len(l)+1))
    table.to_csv(DIR+'/output/'+fileName, sep='\t', index=False)

# Formats and outputs the scanner tokens to a .txt
def outputScanner() -> None:

    output = ''
```

```
    for item in token_list:
        if len(item) == 1:
            output += '<' + str(item[0]) + '>\n'
        else:
            output += '<' + str(item[0]) + ',' + str(item[1]) + '>\n'

    f = open(DIR+'/output/ScannerOutput.txt', 'w')
    f.write(output)
    f.close()
```

Code inputs

The scanner must be capable of processing all types of the characters even from other alphabets and including blank characters such as white spaces, newlines, tabs and empty strings and point correctly where the error is located and what type of error it is if one is found. Here is an example of an input where all the tokens are valid by the language's rules.

Source code:

```
/* Program that reads a 10 element array of
integers, and then multiply each element of
the array by a float, stores the result into an
array of floats. Subsequently, the array of
floats is sorted and display it into standard
output.*/

int x[10];
string s;
float f1;
float f2[100];

int miniloc(float a[], int low, int high){
   int i; float y; int k;

   k = low;
   y = a[low];
   i = low + 1.5;
   while (i < high){
      if (a[i] < x){
         y = a[i];
         k = i;
      }
      i = i + 1.0;
   }
   return k;
}/* END of miniloc() */
```

```
void sort(float a[], int low, int high){
   int i; int k;

   i = low;
   while (i < high - 3){
      float t;
      k = miniloc(a,i,high);
      t = a[k];
      a[k] = a[i];
      a[i] = t;
      i = i + 0.2;
   }
   return;
}/* END of sort() */

void readArray(void){
   int i;
   s = "Enter a float number: ";
   write(s);
   read(f1);
   while (i < 20){
      s = "Enter an integer number: ";
      write(s);
      read x[i];
      f2[i] = x[i]*f1;
      i = i + 1;
   }
   return;
}/* END of readArray() */


void writeArray(void){
   int i;
   i = 0;
   while (i < 5.5){
      write f2[i];
      i = i + 4;
   }
   return;
}/* END of writeArray() */


void main(void){
   s = "Reading Information...";
   write(s);
   readArray();
```

```
    s = "Sorting...";
    write(s);
    sort(f2,2,50);
    s = "Sorted Array:";
    write(s);
    writeArray();
    return;
}/* END of main() */
```

Code outputs

Following that code sample, here are the expected results from the scanner once it has finished the analysis.

Terminal: ✅ Looks green to me! Check the scanner output and lexeme tables.

Scanner output: Each token is represented with this structure '<token ID,optional>'. All token at least must have the first argument that represents their token ID. Second argument is only mandatory for those lexemes that are tied to a regular expression (IDs, NUMBERS, STRINGS) that references the position of the token in its respective table. The only other occasion this second argument might be used is with error tokens and its corresponding error message.

```
<15>
<1>
<12,1>
<33>
<14,1>
<34>
<27>
<3>
<12,2>
<27>
<2>
<12,3>
<27>
<2>
<12,4>
<33>
<14,2>
<34>
<27>
<1>
<12,5>
<31>
<2>
<12,6>
<33>
```

<34>
<28>
<1>
<12,7>
<28>
<1>
<12,8>
<32>
<35>
<1>
<12,9>
<27>
<2>
<12,10>
<27>
<1>
<12,11>
<27>
<12,11>
<26>
<12,7>
<27>
<12,10>
<26>
<12,6>
<33>
<12,7>
<34>
<27>
<12,9>
<26>
<12,7>
<16>
<14,3>
<27>
<7>
<31>
<12,9>
<20>
<12,8>
<32>
<35>
<5>
<31>
<12,6>
<33>
<12,9>
<34>

<20>
<12,1>
<32>
<35>
<12,10>
<26>
<12,6>
<33>
<12,9>
<34>
<27>
<12,11>
<26>
<12,9>
<27>
<36>
<12,9>
<26>
<12,9>
<16>
<14,4>
<27>
<36>
<9>
<12,11>
<27>
<36>
<15>
<4>
<12,12>
<31>
<2>
<12,6>
<33>
<34>
<28>
<1>
<12,7>
<28>
<1>
<12,8>
<32>
<35>
<1>
<12,9>
<27>
<1>
<12,11>

<27>
<12,9>
<26>
<12,7>
<27>
<7>
<31>
<12,9>
<20>
<12,8>
<17>
<14,5>
<32>
<35>
<2>
<12,13>
<27>
<12,11>
<26>
<12,5>
<31>
<12,6>
<28>
<12,9>
<28>
<12,8>
<32>
<27>
<12,13>
<26>
<12,6>
<33>
<12,11>
<34>
<27>
<12,6>
<33>
<12,11>
<34>
<26>
<12,6>
<33>
<12,9>
<34>
<27>
<12,6>
<33>
<12,9>

<34>
<26>
<12,13>
<27>
<12,9>
<26>
<12,9>
<16>
<14,6>
<27>
<36>
<9>
<27>
<36>
<15>
<4>
<12,14>
<31>
<4>
<32>
<35>
<1>
<12,9>
<27>
<12,2>
<26>
<13,1>
<27>
<11>
<31>
<12,2>
<32>
<27>
<10>
<31>
<12,3>
<32>
<27>
<7>
<31>
<12,9>
<20>
<14,7>
<32>
<35>
<12,2>
<26>
<13,2>

<27>
<11>
<31>
<12,2>
<32>
<27>
<10>
<12,1>
<33>
<12,9>
<34>
<27>
<12,4>
<33>
<12,9>
<34>
<26>
<12,1>
<33>
<12,9>
<34>
<18>
<12,3>
<27>
<12,9>
<26>
<12,9>
<16>
<14,8>
<27>
<36>
<9>
<27>
<36>
<15>
<4>
<12,15>
<31>
<4>
<32>
<35>
<1>
<12,9>
<27>
<12,9>
<26>
<14,9>
<27>

<7>
<31>
<12,9>
<20>
<14,10>
<32>
<35>
<11>
<12,4>
<33>
<12,9>
<34>
<27>
<12,9>
<26>
<12,9>
<16>
<14,11>
<27>
<36>
<9>
<27>
<36>
<15>
<4>
<12,16>
<31>
<4>
<32>
<35>
<12,2>
<26>
<13,3>
<27>
<11>
<31>
<12,2>
<32>
<27>
<12,14>
<31>
<32>
<27>
<12,2>
<26>
<13,4>
<27>
<11>

<31>
<12,2>
<32>
<27>
<12,12>
<31>
<12,4>
<28>
<14,12>
<28>
<14,13>
<32>
<27>
<12,2>
<26>
<13,5>
<27>
<11>
<31>
<12,2>
<32>
<27>
<12,15>
<31>
<32>
<27>
<9>
<27>
<36>
<15>

Identifiers table: Records the entry number and lexeme of an identifier token. Once it has been registered on the table, the following instances of the identifier found in the source code must reference this recording and not add another one to the table.

Entry	Lexeme
1	x
2	s
3	f1
4	f2
5	miniloc
6	a
7	low
8	high
9	i
10	y
11	k
12	sort

13	t
14	readarray
15	writearray
16	main

Numbers table: Records the entry number and lexeme of a number token including the type of number i.e. integer or float.

Entry	Lexeme	Type
1	10	int
2	100	int
3	1.5	float
4	1.0	float
5	3	int
6	0.2	float
7	20	int
8	1	int
9	0	int
10	5.5	float
11	4	int
12	2	int
13	50	int

String table: Records the entry number and lexeme of a string token.

Entry	Lexeme
1	"""Enter a float number: """
2	"""Enter an integer number: """
3	"""Reading Information...""""
4	"""Sorting...""""
5	"""Sorted Array:"""

Testing

To validate that the scanner works and properly tokenize characters, several test cases have been developed. From a regular instance where all the tokens are valid and passing through each of the types of lexical errors a user might commit when trying to write C– code. Hereunder are the case scenarios with their explanation, input and output.

Software Test Cases design

Case 0

Explanation: All tokens are accepted by the scanner.

Terminal: ✅ Looks green to me! Check the scanner output and lexeme tables.

Source code:

```
/* Program that reads a 10 element array of
integers, and then multiply each element of
the array by a float, stores the result into an
array of floats. Subsequently, the array of
floats is sorted and display it into standard
output.*/

int x[10];
string s;
float f1;
float f2;
```

Scanner output:

```
<15>
<1>
<12,1>
<33>
<14,1>
<34>
<27>
<3>
<12,2>
<27>
<2>
<12,3>
<27>
<2>
<12,4>
<27>
```

Identifiers table:

Entry	Lexeme
1	x
2	s
3	f1
4	f2

Numbers table:

Entry	Lexeme	Type
1	10	int

String table:

Entry	Lexeme

Case 1

Explanation: Float numbers cannot end with a dot (.) character,' it should have been a digit.

Terminal: ❌ Something went wrong! Check the scanner output to verify the bad token.

Source code:

```
int miniloc(float a[], int low, int high){
   int i; float y; int k;

   k = low;
   y = a[low];
   i = low + 1;
   while (i < high){
      if (a[i] < x){
         y = a[i];
         k = i;
      }
      i = i + 1.;
   }
   return k;
}/* END of miniloc() */
```

Scanner output:

```
<1>
<12,1>
<31>
<2>
<12,2>
<33>
<34>
<28>
<1>
<12,3>
<28>
<1>
<12,4>
<32>
<35>
<1>
<12,5>
<27>
<2>
<12,6>
<27>
<1>
<12,7>
<27>
```

<12,7>
<26>
<12,3>
<27>
<12,6>
<26>
<12,2>
<33>
<12,3>
<34>
<27>
<12,5>
<26>
<12,3>
<16>
<14,1>
<27>
<7>
<31>
<12,5>
<20>
<12,4>
<32>
<35>
<5>
<31>
<12,2>
<33>
<12,5>
<34>
<20>
<12,8>
<32>
<35>
<12,6>
<26>
<12,2>
<33>
<12,5>
<34>
<27>
<12,7>
<26>
<12,5>
<27>
<36>
<12,5>
<26>

<12,5>
<16>
<37,Float numbers cannot end by with a '.'>

Identifiers table:

Entry	Lexeme
1	miniloc
2	a
3	low
4	high
5	i
6	y
7	k
8	x

Numbers table:

Entry	Lexeme	Type
1	1	int

String table:

Entry	Lexeme

Case 2

Explanation: Comment (/* ... */) was not properly closed, it missed its second part (*/).

Terminal: ❌ Something went wrong! Check the scanner output to verify the bad token.

Source code:

```
void sort(float a[], int low, int high){
   int i; int k;

   i = low;
   while (i < high - 2){
      float t;
      k = miniloc(a,i,high);
      t = a[k];
      a[k] = a[i];
      a[i] = t;
      i = i +1;
   }
   return;
}/* END of sort()
```

Scanner output:

<4>
<12,1>

<31>
<2>
<12,2>
<33>
<34>
<28>
<1>
<12,3>
<28>
<1>
<12,4>
<32>
<35>
<1>
<12,5>
<27>
<1>
<12,6>
<27>
<12,5>
<26>
<12,3>
<27>
<7>
<31>
<12,5>
<20>
<12,4>
<17>
<14,1>
<32>
<35>
<2>
<12,7>
<27>
<12,6>
<26>
<12,8>
<31>
<12,2>
<28>
<12,5>
<28>
<12,4>
<32>
<27>
<12,7>
<26>

<12,2>
<33>
<12,6>
<34>
<27>
<12,2>
<33>
<12,6>
<34>
<26>
<12,2>
<33>
<12,5>
<34>
<27>
<12,2>
<33>
<12,5>
<34>
<26>
<12,7>
<27>
<12,5>
<26>
<12,5>
<16>
<14,2>
<27>
<36>
<9>
<27>
<36>
<37,Comment was not properly closed>

Identifiers table:

Entry	Lexeme
1	sort
2	a
3	low
4	high
5	i
6	k
7	t
8	miniloc

Numbers table:

Entry	Lexeme	Type
1	2	int

2 1 int

String table:
Entry Lexeme

Case 3

Explanation: String (“ ... “) was not properly closed, it missed its second semicolon (“).

Terminal: ❌ Something went wrong! Check the scanner output to verify the bad token.

Source code:

```
void main(void){
   s = "Reading Information...";
   write(s);
   readArray();
   s = "Sorting...;
   write(s);
   sort(f2,0,10);
   s = "Sorted Array:";
   write(s);
   writeArray();
   return;
}/* END of main() */
```

Scanner output:

```
<4>
<12,1>
<31>
<4>
<32>
<35>
<12,2>
<26>
<13,1>
<27>
<11>
<31>
<12,2>
<32>
<27>
<12,3>
<31>
<32>
<27>
<12,2>
<26>
```

<37,String was not properly closed>

Identifiers table:

Entry	Lexeme
1	main
2	s
3	readArray

Numbers table:

Entry	Lexeme	Type

String table:

Entry	Lexeme
1	"""Reading Information...""""

Case 4

Explanation: The logic operator different (!=) was not fully written.

Terminal: ❌ Something went wrong! Check the scanner output to verify the bad token.

Source code:

```
void writeArray(void){
   int i;
   i = 0;
   while (i ! 10){
      write f2[i];
      i = i + 1;
   }
   return;
}/* END of writeArray() */
```

Scanner output:

```
<4>
<12,1>
<31>
<4>
<32>
<35>
<1>
<12,2>
<27>
<12,2>
<26>
<14,1>
<27>
<7>
<31>
```

<12,2>
<37,'!' has to be followed by a '='>

Identifiers table:

Entry	Lexeme
1	writeArray
2	i

Numbers table:

Entry	Lexeme	Type
1	0	int

String table:

Entry	Lexeme

Case 5

Explanation: A token contained a character outside of the English alphabet.

Terminal: ❌ Something went wrong! Check the scanner output to verify the bad token.

Source code:

```
void readArray(void){
   int i;
   sórt = "Enter a float number: ";
   write(s);
   read(f1);
   while (i < 10){
      s = "Enter an integer number: ";
      write(s);
      read x[i];
      f2[i] = x[i]*f1;
      i = i + 1;
   }
   return;
}/* END of readArray() */
```

Scanner output:

<4>
<12,1>
<31>
<4>
<32>
<35>
<1>
<12,2>
<27>

<37,Token has an unknown character>

Identifiers table:

Entry	Lexeme
1	readArray
2	i

Numbers table:

Entry	Lexeme	Type

String table:

Entry	Lexeme

References

Conway, M. E. (1963). Design of a separable transition-diagram compiler. Communications of the ACM, 6(7), 396-408.

Conway, R. W., & Wilcox, T. R. (1973). Design and implementation of a diagnostic compiler for PL/I. Communications of the ACM, 16(3), 169-179.

Lucas, S. M., & Reynolds, T. J. (2005). Learning deterministic finite automata with a smart state labeling evolutionary algorithm. IEEE transactions on pattern analysis and machine intelligence, 27(7), 1063-1074.