Lexical Analyzer for the C Minus Language

Víctor Emmanuel Guerra Aguado - A01568075

Summary

The following document will report upon the development of a lexical analyzer for the proposed C Minus programming language in the Python programming language, including in the complete phases of **Analysis**, **Design**, **Implementation**, and **Testing.** Each phase will contain its appropriate documentation and implementation.

Context & Notation

Regular Expressions (RegEx, or REs)

Regular Expressions are a special notation that represent patterns of strings or characters, and facilitate the task of consistent and precise representation of the lexical rules of a language.[1]

A language is composed by a set of strings, and the strings are formed from a set of characters, these characters are the **legal symbols** or **alphabet** of the language and are represented by the Greek letter Σ (sigma).[1]

Finite State Machines(FSMs)

Also known as Finite Automata (FAs), they are a mathematical way of describing algorithms for behavioral systems. Finite State Machines can be used to describe the process of recognizing patterns in input strings, and can therefore be used for a lexical analyzer. They are strongly related to Regular Expressions; an FSM can always be constructed from a valid Regular Expression.[1]

In a Finite State Machine, the circles represent states which are locations in the process of recognition that record how much of the pattern has been recognized; the arrowed lines represent the transitions that record the change from one state (initial) to another (final), it is labeled with the character or characters that trigger said transition. Accepting states are indicated by a circle with double lined border, and they signify the recognition of a valid string.[1]

A Deterministic Finite Automaton (DFA) is one where any final state is **uniquely** given by the current initial state and the current input character of its transition.[1]

Transition Tables

A DFA can be expressed as a data structure together with a generic code that will use the information stored in order to take proper action. The transition table is one of such data structures; implemented as a two dimensional array, the transition table is indexed by a state number and an input character, which provide the values for the transition function.[1]

Programming Language

The chosen programming language for the compiler and lexical analyzer is Python. It was chosen for its simplicity and straightforwardness regarding the management of data structures, memory, and data types; its flexibility would make it possible to take some load from the implementation phase that could be used during other phases.

Analysis

Requirements

1. The automata of the language.
    a. The automaton of the language shall categorize each word read according to the specification of the C Minus Language.
    b. The system shall recognize comments, as defined by the C Minus language specification and ignore their contents.
2. Tokens and their identification.
    a. The system shall receive an input file to scan.
    b. The system shall recognize every given valid word according to the C Minus Language Specification.
    c. The system shall create a token for every valid word recognized in the input file.

   d. The system shall recognize identifiers and numerical constants as defined by the C Minus language specification and create a token for each found in the input file.
   e. The system shall recognize the keywords and special symbols of the C Minus language and create a special token for each found in the input file.
   f. The system shall create tokens from the input file completely, correctly, and in the order recognized, according to the C Minus language specification.
   g. The system shall create and output an ordered token list for the valid words of the C Minus language found in the input file.
3. Transition Table.
   a. The transition table shall represent completely and correctly the definition of the Deterministic Finite Automaton for the C Minus Language.
   b. The transition table shall be implemented in code, in order to be able to apply the language DFA in the compiler.
4. Symbol Tables.
   a. The system shall create and use different Symbol Tables for Identifiers and Numeric Constants.
   b. The system shall create an entry in a Symbol Table for every unique, valid identifier or numeric constant found in the input file.
   c. The Symbol Tables shall be able to handle the addition of columns when necessary.
   d. The Symbol Tables shall be able to handle entry Id and Value.
   e. The system shall output all Symbol Tables created and required.
5. Error Messages generated by the scanner.
   a. The automaton and scanner shall be able to recognize an invalid character and produce an error appropriate to the situation.
   b. The automaton and scanner shall be able to recognize an open comment symbol with no corresponding closing symbol, and produce an appropriate error.
   c. The automaton and scanner shall be able to recognize an invalid Identifier or Keyword and produce an appropriate error.
   d. The automaton and scanner shall be able to recognize an invalid Numerical Constant and produce an appropriate error.

e. The scanner shall be able to provide the number of the line in code where an error was found.

The Definition

In order to create the necessary transition table for recognizing valid words of the C Minus language, it is useful to first define the DFA that accepts all of the RegEx, keywords, and symbols shown in the definition of the language. The RegEx and valid words and symbols for the Language are defined as follows:

a) The Keywords of the language are:

else
if
int
return
void
while
input
output

All keywords are reserved words and they are NOT case sensitive […] .

b) Special symbols are the following:

+ arithmetic addition operation
- arithmetic subtraction operation
* arithmetic multiplication operation
/ arithmetic division operation
< logic operator less than
<= logic operator less or equal than
> logic operator greater than
>= logic operator greater or equal than
== logic operator equal
!= logic operator different
= assignation
; semicolon
, coma
(open parenthesis
) close parenthesis
[open square brackets
] close square brackets
{ open curly brackets
} close curly brackets
/* open comment
*/ close comment

c) Other tokens are ***ID*** and ***NUM***, their corresponding Regular Expressions definitions are as follows:

***ID* = *letter*+**
***NUM* = *digit*+**
***letter* = [a-zA-Z]**
***digit* = [0-9]**

Identifiers are letter sensitive, i.e., lower and uppercase letters are distinct.

d) White space consists of *blanks, newlines,* and *tabs.* Whitespace is ignored, but it MUST be recognized. White space together with ***ID’***s*,* ***NUM’s,*** and **keywords**, are considered as delimiters.

e) Comments are **C** language style, i.e., they are enclosed by /* … */. Comments can be placed anywhere white space can appear, i.e., comments cannot be placed within tokens. Comments may include more than one line.

Given these specifications the DFA in **Figure 1.1** was designed:

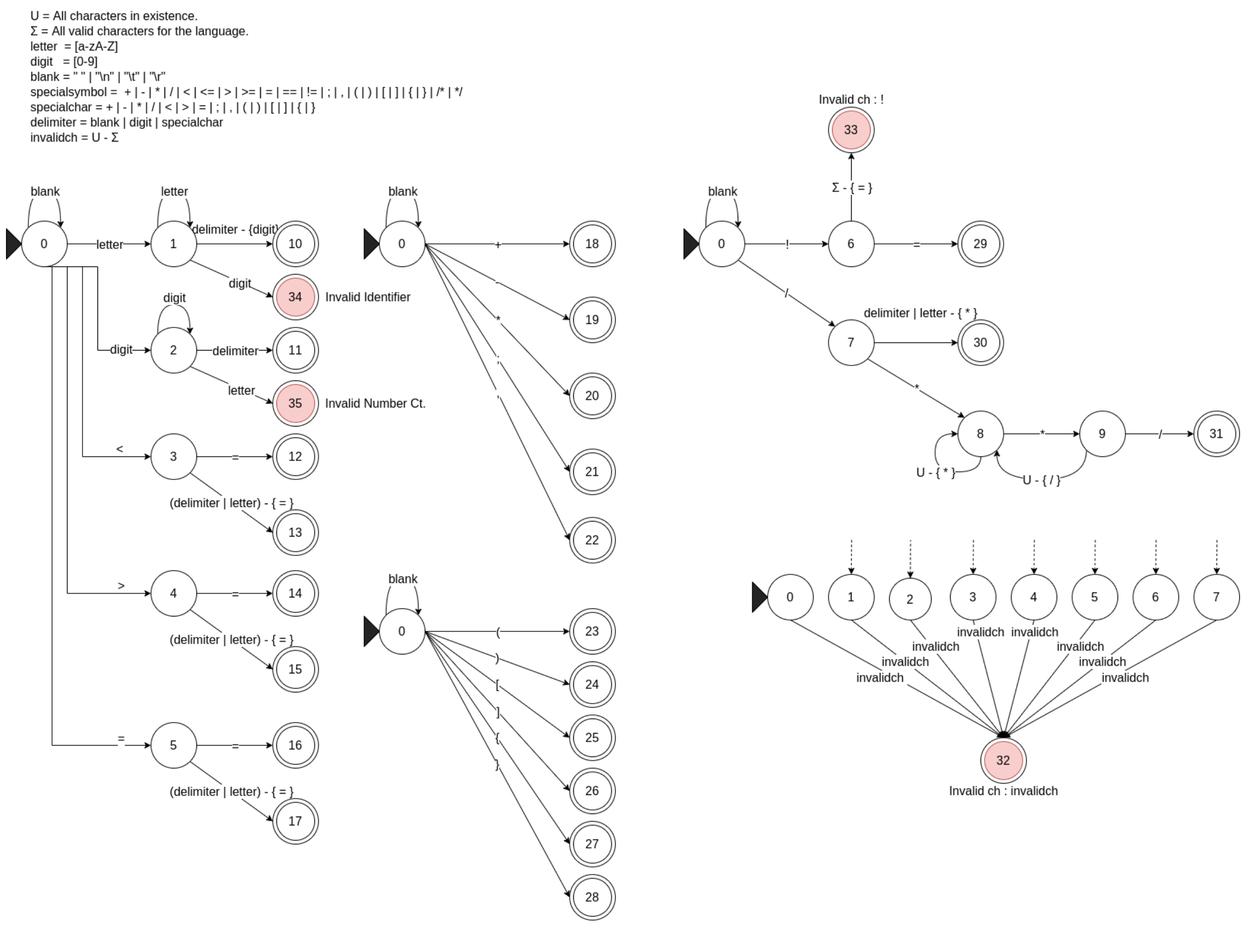

Figure 1.1 The Whole DFA for the C Minus Language

The definitions given on the top left of **Figure 1.1** are as follows:

U = All characters in existence.

Σ = All valid characters for the C Minus Language.

letter = [a-zA-Z]

digit = [0-9]

blank = “ ” | “\n” | “\t” | “\r”

specialchar = + | - | * | / | < | > | = | ; | , | (|) | [|] | { | }

specialsymbol = specialchar | <= | >= | == | != | /* | */

delimiter = blank | digit | specialchar

invalidch = U - Σ

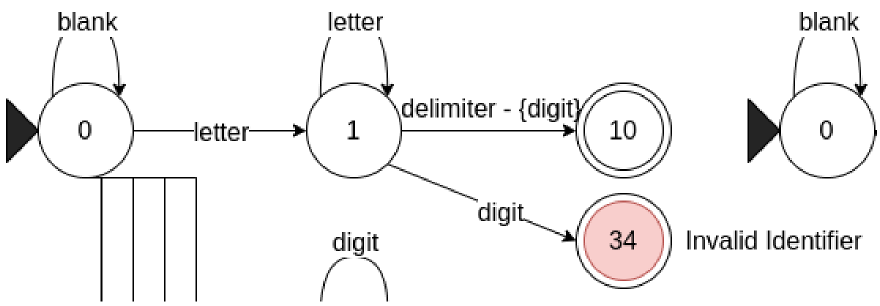

Figure 1.2 The keyword and identifier recognition of the DFA for the C Minus Language.

State 0 of the DFA, shown in **Figure 1.2**, is where the lexical recognition begins. It loops back to itself whenever it encounters a BLANK, and starts recognition of all other valid words in the language. It has transitions to states: 1, 2, 3, 4, 5, 6, 7, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 and 28.

States 1, 10, and 34 of the DFA, shown in **Figure 1.2**, are in charge of recognizing valid *keywords* and *Identifiers*; state 1 loops for every *letter* found, whenever a non-digit delimiter is found the automaton changes its state to 10, an accepting state, and finishes recognition; if, while in state 1, it receives a *digit* the DFA changes to accepting state 34, thus indicating that an invalid *identifier* was found.

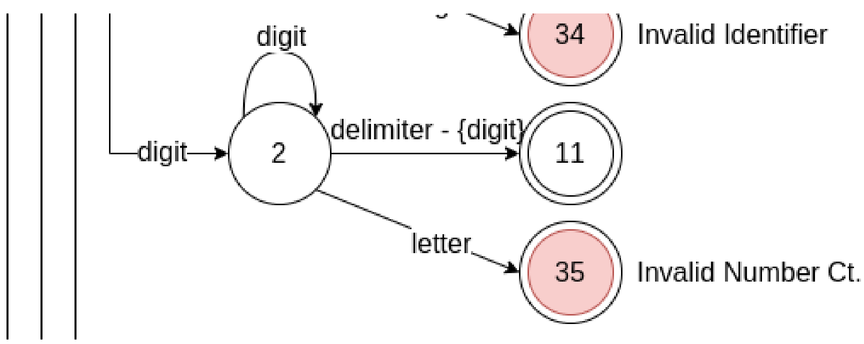

Figure 1.3 The numerical constant recognition of the DFA for the C Minus Language.

States 2, 11, and 35 of the DFA, shown in **Figure 1.3**, are concerned with recognizing *Numerical Constants*; state 2 loops for every *digit* found, whenever a non-digit delimiter is found the automaton changes its state to 11, an accepting state, and finishes recognition; if, while in state 2, it receives a *letter* the DFA changes to accepting error state 35, thus indicating that an invalid *identifier* was found.

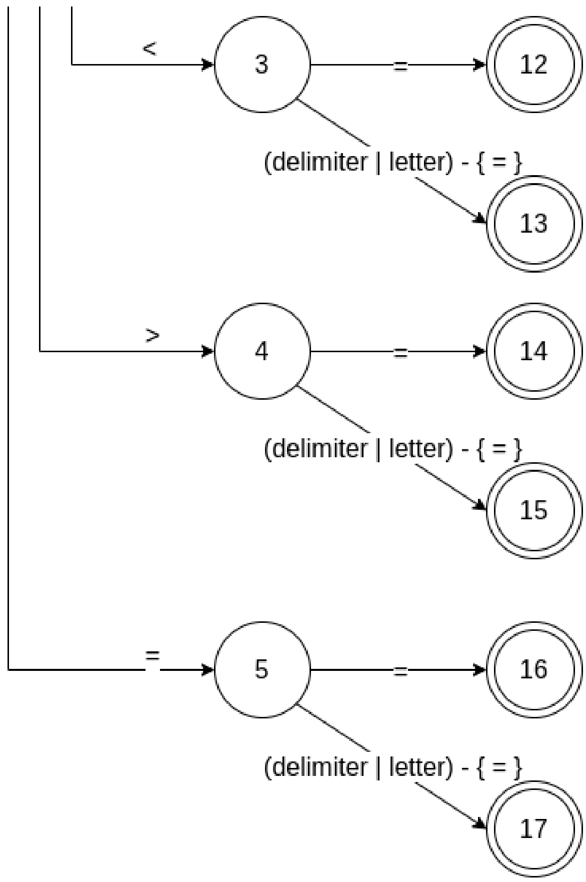

Figure 1.4 Recognition of Compound Symbols of the DFA for the C Minus Language.

States 3, 4, and 5, shown in **Figure 1.4**, are concerned with the recognition of Compound Symbols: <=, >=, and ==; Upon receiving one of the symbols: <, >, or = (all of which are valid symbols in the C Minus Language) the automaton looks at the following character, if it is anything but a "=", the automaton moves to accepting state 12, 14, or 16 respectively, and indicates recognition of one of the single-character-symbols shown previously; otherwise, the automaton moves to accepting state 13, 15, or 17, respectively, and indicates the recognition of one of the compound symbols shown previously.

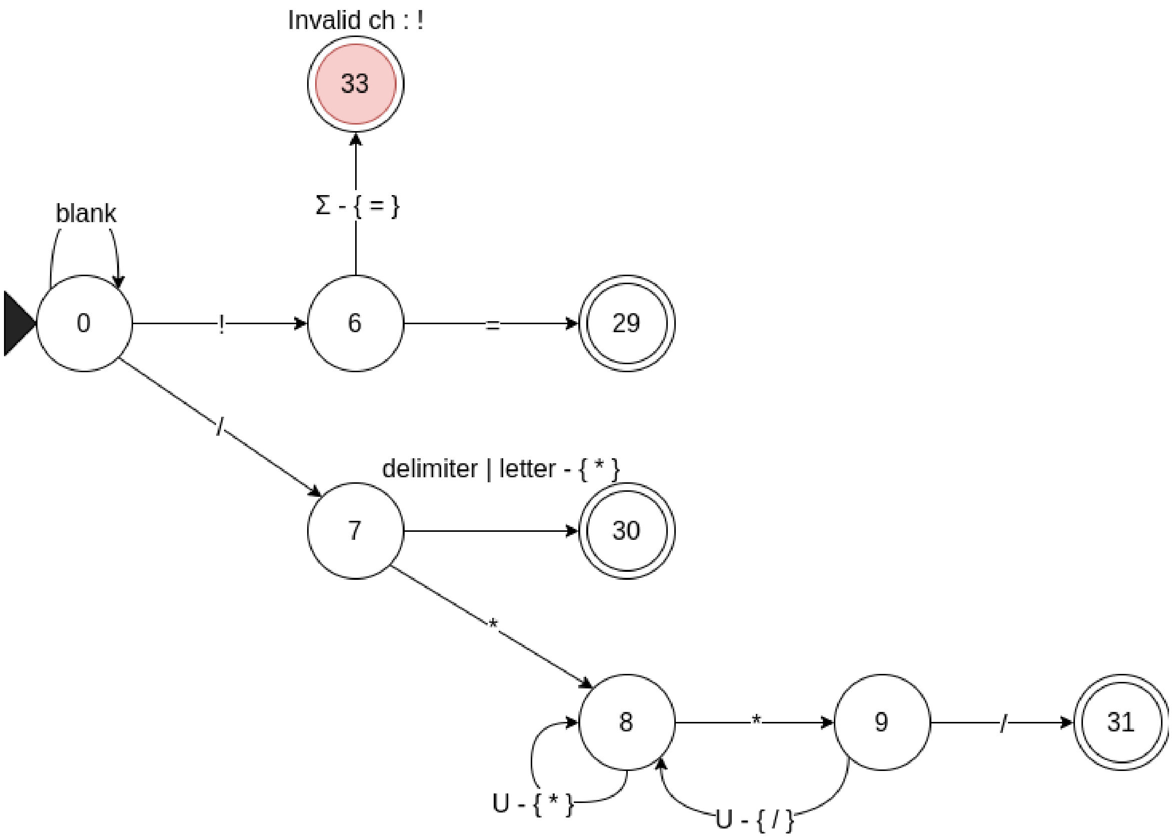

Figure 1.5 Other compound symbols and comment recognition.

State 6, shown in **Figure 1.5**, is in charge of recognizing a "!=" symbol, however, since the exclamation point by itself is not a valid word in the C Minus language if it is encountered, an error should be raised.

State 7, in **Figure 1.5**, recognizes the "/" symbol, but it also has a very important role with the recognition of comments; if the forward slash ("/") is followed by anything but an asterisk ("*") the forward slash is then recognized as one of the valid words of the language, but if it is followed by an asterisk then a comment has been opened, and states 8 and 9 have the responsibility of looking for the closing tag for said comment. If the closing tag is found, then all collected characters are ignored, and the accepting state of 31 is reached, ending the iteration, otherwise it is an error that shall be notified to the user.

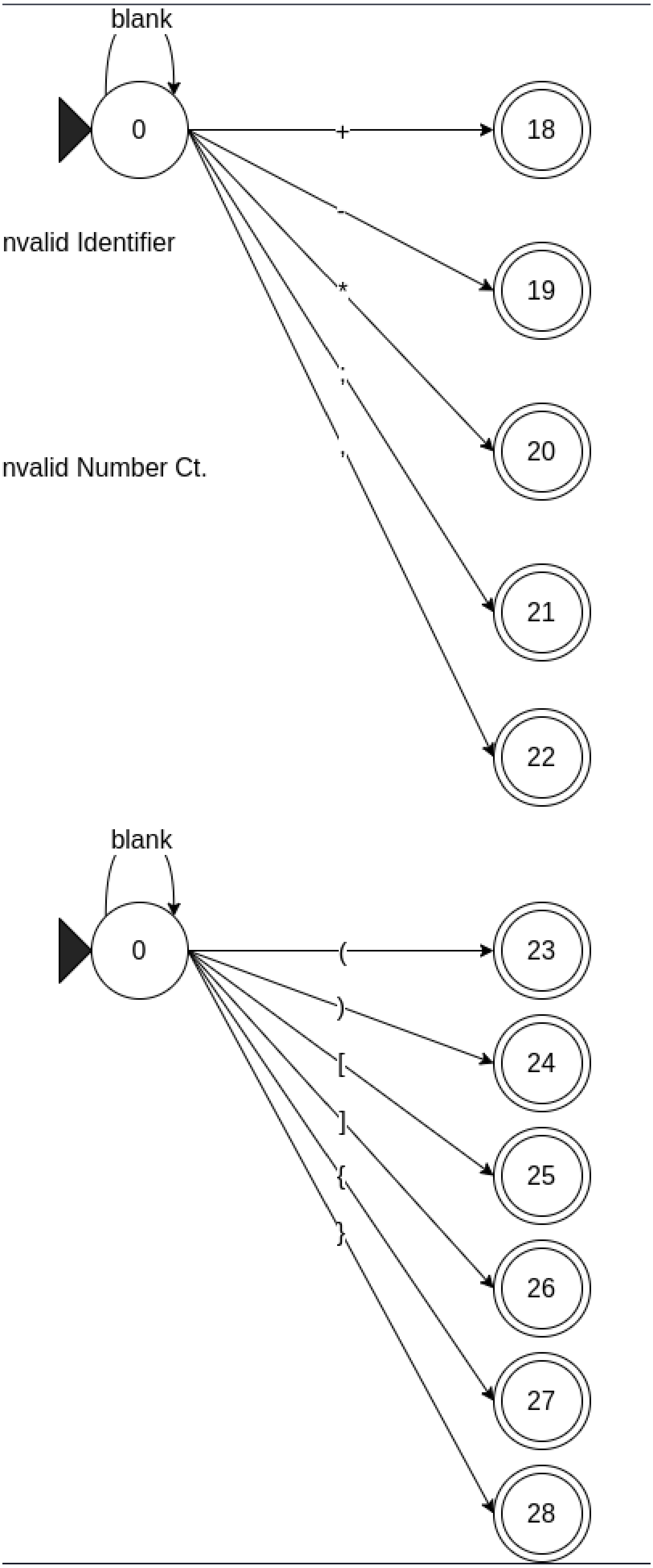
blank
0
+
18
-
19
*
20
;
21
,
22
nvalid Identifier
nvalid Number Ct.
blank
0
(
23
)
24
[
25
]
26
{
27
}
28

Figure 1.6 States 18 through to 28.

States 18 through 28, shown in **Figure 1.6**, provide the recognition of all remaining one-character symbols of the language. The recognition is simple since it only requires one transition, from initial state to accepting state; after reading one of the following characters: “+”, “-”, “*”, “;”, “,”, “(“, “)”, “[“, “]”, “{“, “}”, the state is accepting, thus, there is little to no opportunity for errors to arise during these recognitions.

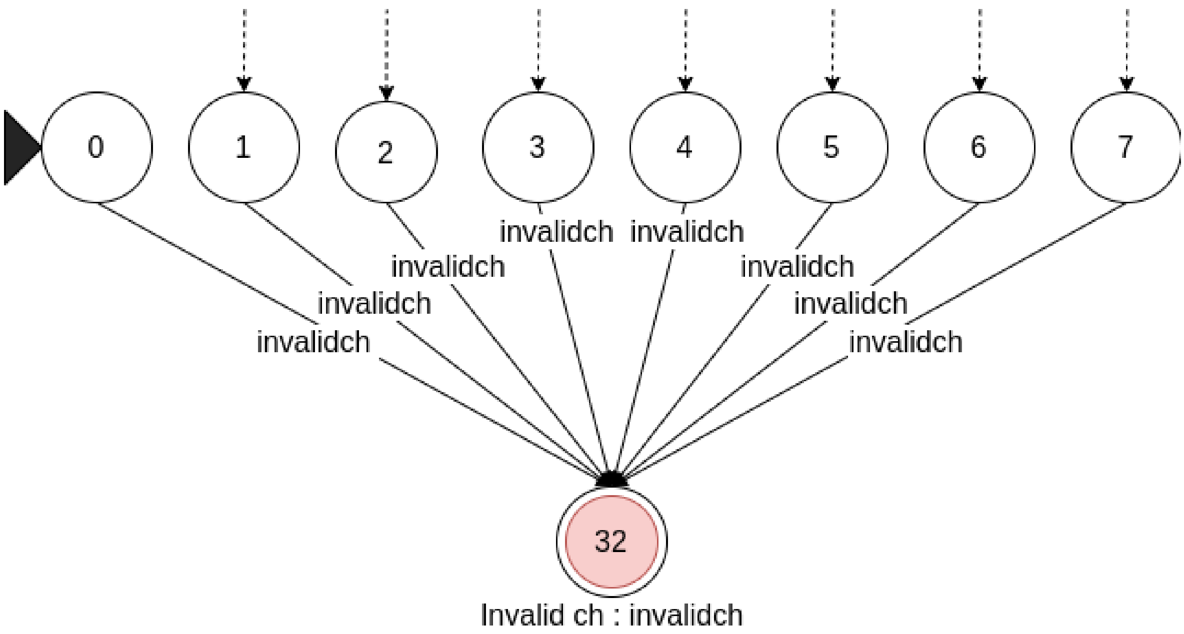

Figure 1.7 The Error State 32, for invalid characters.

Finally, states 32 33, 34, and 35, 32 pictured in **Figure 1.7** and the rest can be seen in **Figures 1.5, 1.3,** and **1.2** respectively, are the error states. State is a common error state to all non-accepting states of the DFA; it recognizes when a given character is not a valid character according to the C Minus language specification, that is any character outside of the 26 letters of the alphabet (uppercase and lowercase), the 10 digits, and any of the following: “+”, “-”, “*”, “/”, “<”, “>”, “=”, “;”, “,”, “(“, “)”, “[“, “]”, “{“, “}". Any other character is outside the definition of the language and, therefore, an error in lexical analysis.

Transition Table

The Following considerations were made, in order to best translate the DFA into a Transition Table:

1. The DFA shall be expressed as a transition table data structure.
2. The ID's and keywords are going to be differentiated within the code, not the DFA.
3. Non-accepting states were numbered first, followed by Accepting States, and Error Accepting States, in order to better organize the Transition Table.

Given the generated DFA and taking the previous considerations. the following Transition Table was generated:

STATE	ACCEPTING	LETTER	DIGIT	BLANK	<	>	=	!	/	+	-	*	;	,	(	)	[	]	{	}	INVALIDCH
0	FALSE	1	2	0	3	4	5	6	7	18	19	20	21	22	23	24	25	26	27	28	32
1	FALSE	1	[34]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	[10]	32
2	FALSE	[35]	2	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	[11]	32
3	FALSE	[13]	[13]	[13]	[13]	[13]	12	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	[13]	32
4	FALSE	[15]	[15]	[15]	[15]	[15]	14	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	[15]	32
5	FALSE	[17]	[17]	[17]	[17]	[17]	16	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	[17]	32
6	FALSE	33	33	33	33	33	29	33	33	33	33	33	33	33	33	33	33	33	33	33	32
7	FALSE	[30]	[30]	[30]	[30]	[30]	[30]	[30]	[30]	[30]	[30]	8	[30]	[30]	[30]	[30]	[30]	[30]	[30]	[30]	32
8	FALSE	8	8	8	8	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8
9	FALSE	8	8	8	8	8	8	8	31	8	8	8	8	8	8	8	8	8	8	8	8
10	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
19	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
21	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
22	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
23	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
24	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
25	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
26	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
27	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
28	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
29	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
31	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
32	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
33	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
34	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
35	TRUE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 1.8 The Transition Table from the C Minus DFA.

In **Figure 1.8**, the rows in yellow correspond to the non-accepting states, the rows in green to accepting states, and the rows in orange to accepting error states. The first column determines whether the state is an accepting state, however, since this column does not represent a transition, it shall not be reflected in the transition table itself, but as a different data point.

Accepting states have no specified transition since once the automaton is at one it shall stop execution and return its state and recognized string, therefore there is no need for an accepting state to have a transition.

In addition, the transitions that are displayed surrounded by square braces (“[“, “]”) are the **non-consuming** transitions, that is when triggered, these transitions will not “consume” the character that was read.

Tokens

The valid words of the language shall be organized into the following structure:

1. else
2. if
3. input
4. int
5. output
6. return
7. void
8. while
9. +
10. -
11. *
12. /
13. >
14. >=
15. <
16. <=
17. ==
18. !=
19. =
20. ;
21. ,
22. (
23.)
24. {
25. }

26. [
27.]
28. IDENTIFIER
29. NUM_CONSTANT
30. COMMENT

Where the number represents the id that said word will receive, and will identify it during the rest of the compilation process. The order was decided upon alphabetical order for the keywords; as presented in the specification for the special symbols (exceptuating the last four); and as seen in class[1] for Identifiers, Numerical Constants, and Comments.

Design

During the design phase, multiple diagrams were made, including: Class Diagram, Sequence Diagram, and a Flow Chart, along with a preliminary diagram of the architecture of the entire Compiler. With respect to the Requirements, the following considerations were made:

1. The entire scanner for the compiler can be abstracted into a class.
2. The Transition Table can be abstracted into a class, composed of transitions, which themselves are put into arrays with size adequate to index by state number; these arrays will be indexed, with a tuple of characters serving as key, within a dictionary.
3. The Symbol Tables can be abstracted into a class, composed of entries abstracted into classes, with the attributes for their value and any possible fields that might be required.
4. The Symbol Tables will have an attribute for the possible additional fields as an array of the values as required; said array will be expandable as needed.
5. The Tokens that are to be the output of the Scanner can be abstracted into a class, leaving them as a tuple was considered, but the necessary methods for adapting and manipulating these tokens demand a more complicated data structure.
6. The Compiler itself can be abstracted into a class, with its multiple phases abstracted into classes and added as attributes, which can then be called in the order necessary.

A diagram of a preliminary architecture for the entire compiler, seen in **Figure 2.1**, was made with the intention of establishing a structure in which the steps of compilation and their inputs and outputs can be visualized and the shared data that each class utilizes; therefore more easily placed within the *compile* method:

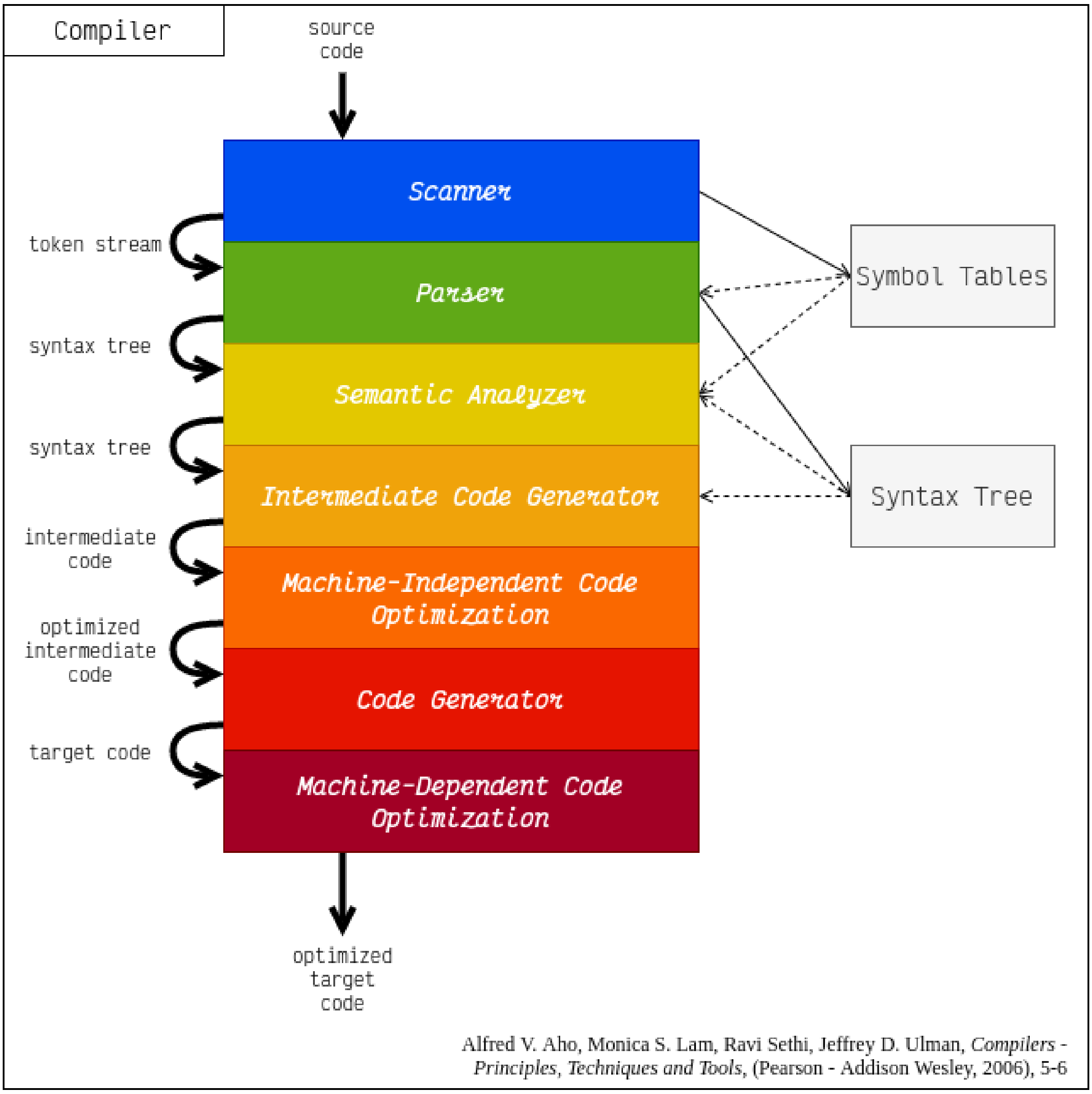

Figure 2.1 The preliminary architecture diagram of the compiler.

The diagram in **Figure 2.2** shall illustrate the interactions between the classes described previously; the classes for steps of compilation other than lexical analysis have been omitted as they are not relevant at this stage of compilation.

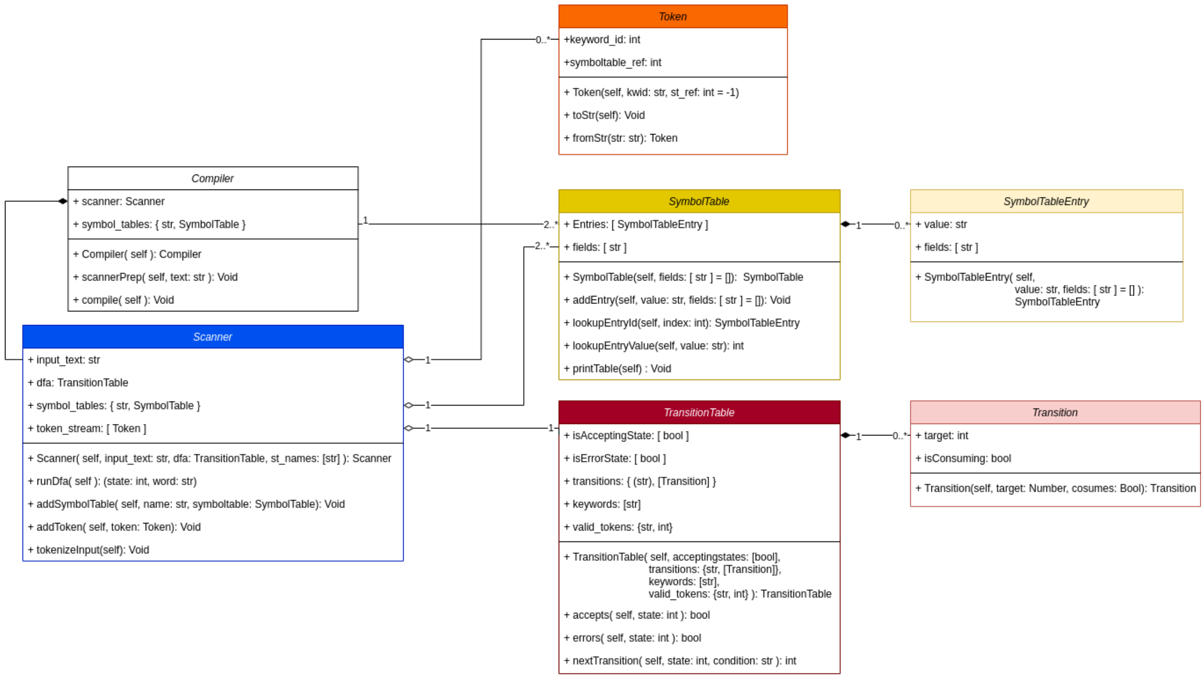

Figure 2.2 - The entire Class Diagram for the considerations described.

As considered, the *Compiler* class is composed of a *Scanner* object, as seen in **Figure 2.3**, and will be composed of a *Parser* object, a *SemanticAnalyzer* class, etc. It also has a 'symbol_tables' attribute which incorporates all the resulting *SymbolTable* objects from the Lexical Analysis phase. Its only methods so far are the *scannerPrep* method, which sets up the scanner with the necessary information; and the *compile* method, which starts the compilation process by calling the scanner to produce its output.

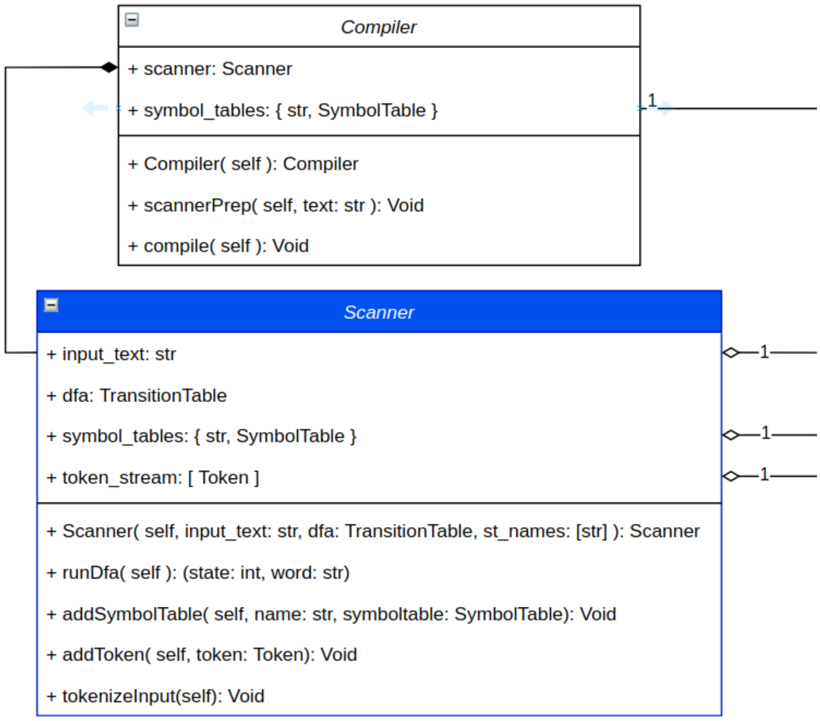

Figure 2.3 - The Compiler Class and the Scanner Class.

The *Scanner* Class itself, as seen in **Figure 2.3**, is aggregated from the *Token*, *SymbolTable*, and *TransitionTable* classes, and incorporates them as the attributes *token_stream*, *symbol_tables*, and *dfa* respectively, where *token_stream* represents the output of the Scanner as a list of *Token* objects; *symbol_tables* stores the scanners' *SymbolTable* objects by name via a dictionary structure; and *dfa* stores the C Minus Language specifications as a *TransitionTable* object. Its methods include: *runDfa,* which runs the "DFA" through its transition table exactly once; *addSymbolTable,* which receives a string and a *SymbolTable* object, and saves them within its *symbol_tables* attribute with the string as a key to access the object in the dictionary; *addToken*, which appends the given *Token* object to the *token_stream* attribute; and *tokenizeInput*, which executes the whole scanning phase and fills the *token_stream* and *symbol_tables* attributes with their respective entries.

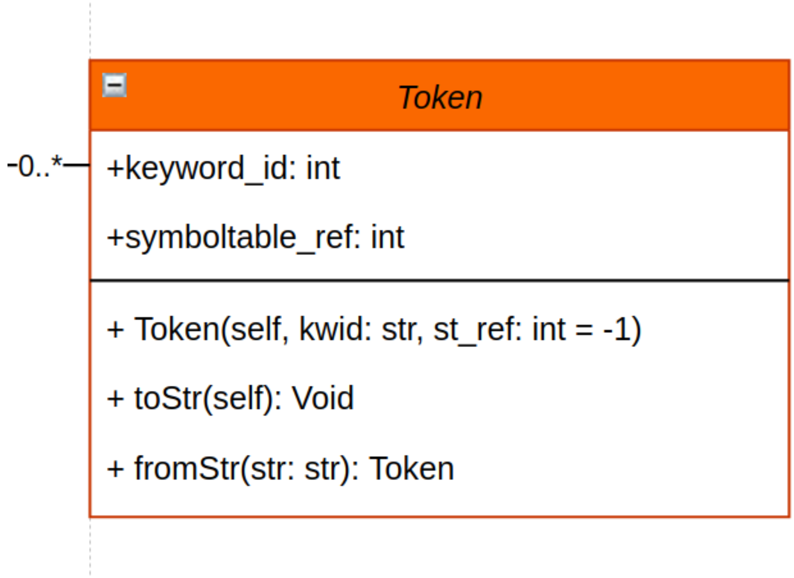

Figure 2.4 - The Token Class

The *Token* Class, seen in **Figure 2.4**, abstracts the components of the tokens generated by a compilers' scanner, and provides some relevant methods for input and output of said tokens.

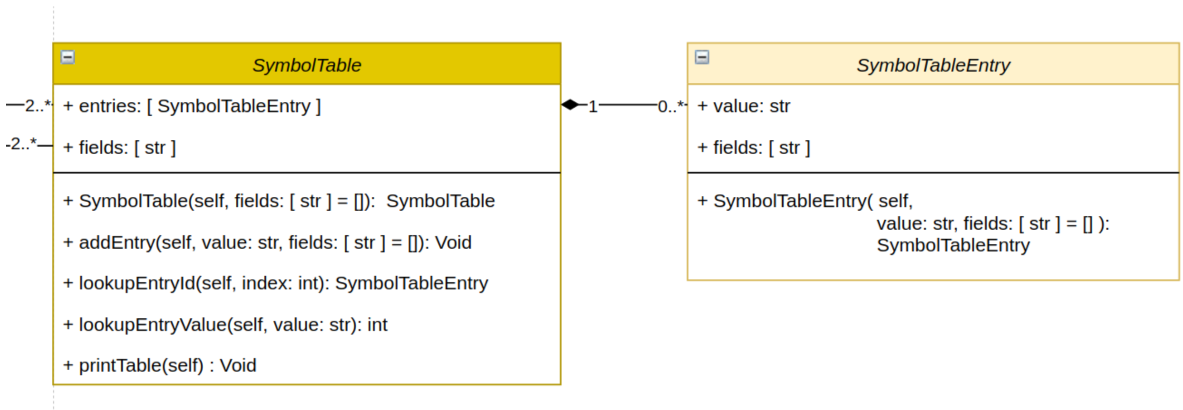

Figure 2.5 - The SymbolTable and SymbolTableEntry Classes.

The *SymbolTable* Class, seen in **Figure 2.5**, acts as an abstraction of the symbol table structure used during the compilation process. It is composed of a list of *SymbolTableEntry* Objects, which represent the unique identifiers or numerical constants found within the source code. Its methods include: *addEntry,* which appends an entry to the *entries*; *lookupEntryId*, which returns an entry given an index; *lookupEntryValue*, which returns an entry given a value (this works since every entry is supposed to have a unique *value*); and *printTable*, which provides visualization of the tables' contents.

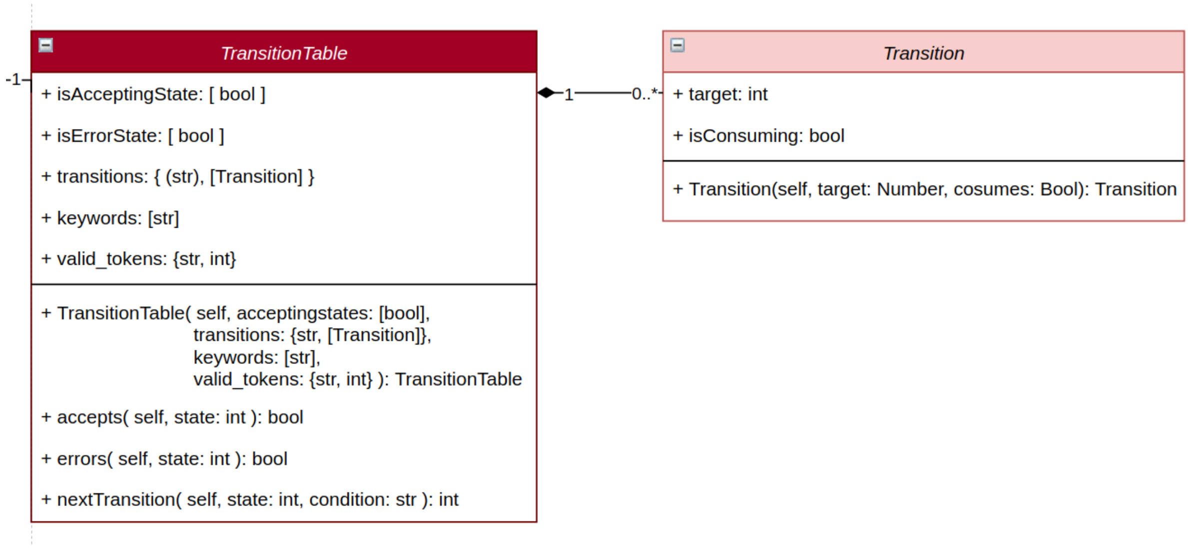

Figure 2.6 - The TransitionTable and Transition Classes.

The *TransitionTable* Class, as seen in **Figure 2.6**, works as a container for the definition of a language, it abstracts the components of a DFA and language specifications necessary for the lexical analysis. It contains the attributes: *isAcceptingState*, which is an array of boolean values intended to list all states of a DFA, and differentiate those that are accepting states; *isErrorstate* is similar to the *isAcceptingState* attribute, instead of differentiating accepting states, it differentiates the error states; the *keywords* attribute is a list of all the reserved words, or keywords, of a given language, the attribute is necessary due to the adjustments made to the DFA of the C Minus Language, namely, the differentiation between keywords and identifiers being made in code, as opposed to the DFA; *valid_tokens*, which lists the valid words of the language and assigns them an ID number by which the tokens created by the scanner will refer to; finally the *transitions* attribute is a dictionary with tuples of strings as keys, and *Transition* objects as values.

The *Transition* object acts as a value object for the transitions used in a transition table, and provides no additional functionality.

The sequence diagram in **Figure 2.7** was made to project the path of function calls along the instances of the previously illustrated object classes for the scanning process. It only takes into account the classes: *Compiler*, *Scanner*, *TransitionTable*, *SymbolTable*, and *Token* since the other classes act just as structures for the data in the shown classes:

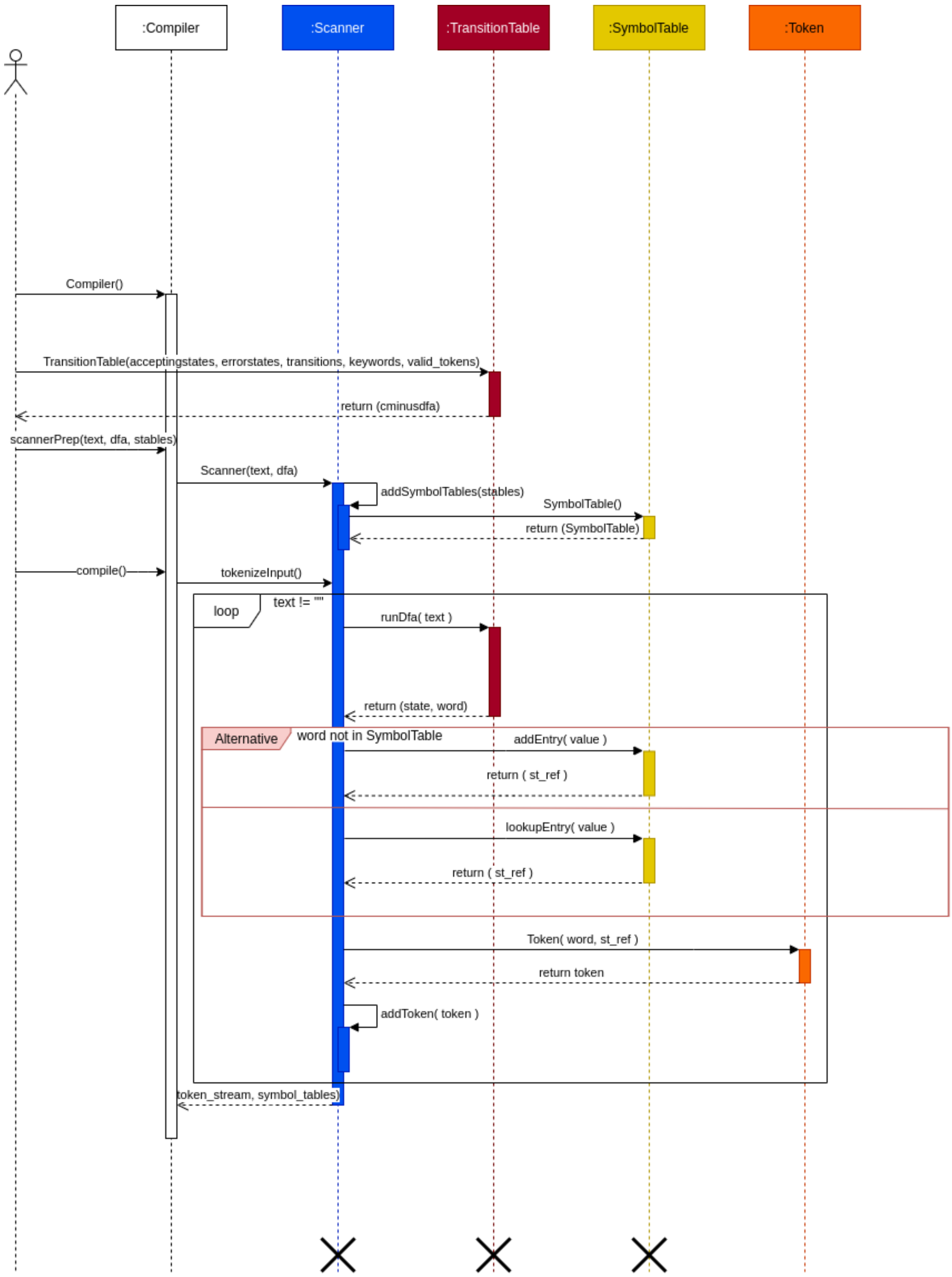

Figure 2.7 - The Sequence Diagram.

The diagram begins with the assumption that a user will call the compiler class from a main program. Within the main program, the user will also provide the necessary information about the definition of the language in the form of a *TransitionTable* object; this information is then passed to the *Scanner* object along with the source code and the names of the required *SymbolTable* objects for the scanner. After the *Scanner* object is set up, the *compile* method of the *Compiler* object can be called, which calls the *tokenizeInput* method of the scanner which returns the resulting *token_stream* and *symbol_tables*. Inside the *tokenizeInput* method, *TransitionTable* objects' method *runDfa* will be run until the *text* field is empty, creating tokens for every recognized word, and creating entries in the symbol tables if necessary. Since at this point the only compilation possible is the lexical analysis of the given source code, the diagram stops after said process.

Although the previous diagram provides a "big picture" of the scanning process, a further breakdown of the *runDfa* method is needed; for this purpose the *runDfa* flowchart, shown in **Figure 2.8**, was made:

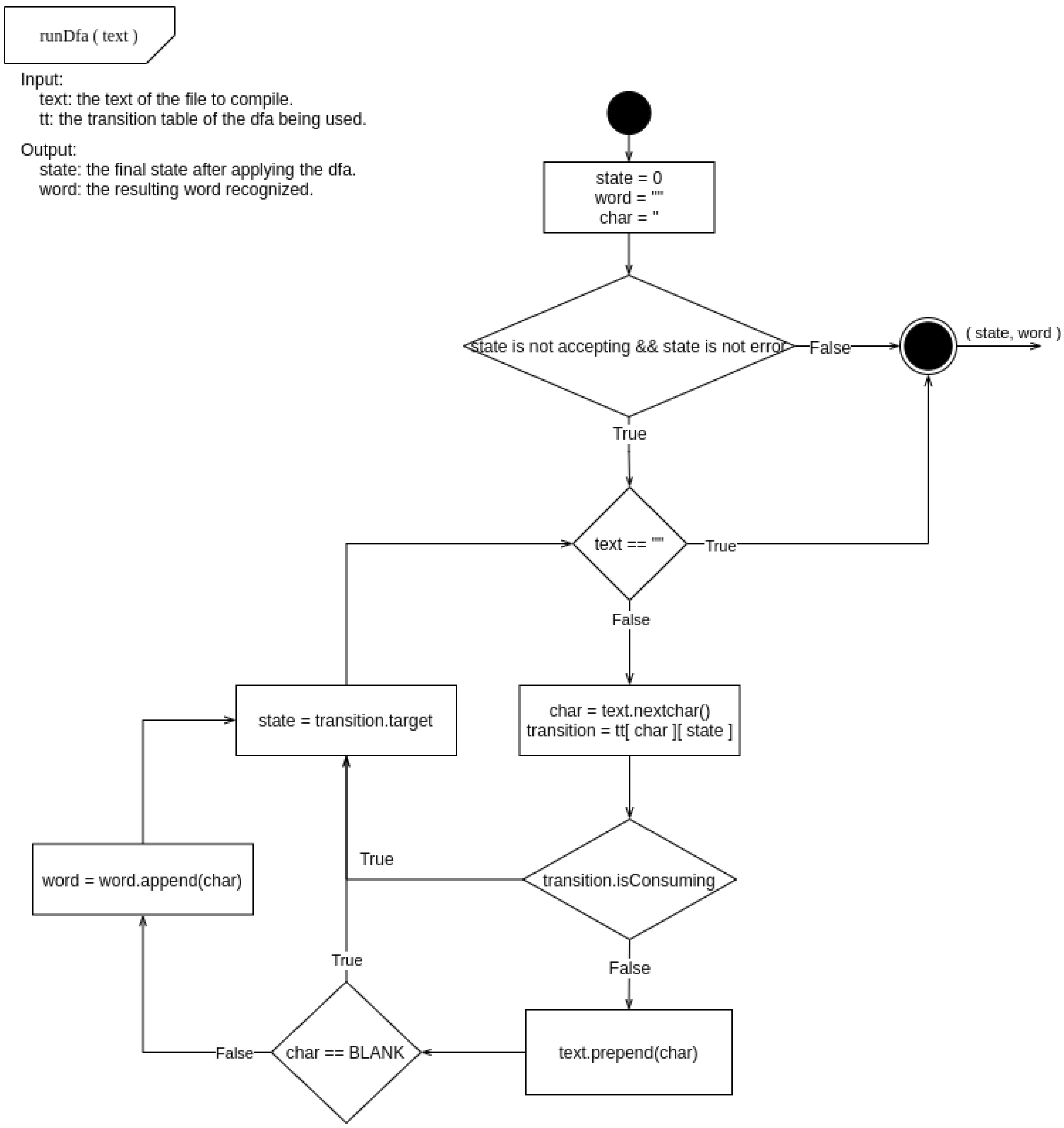

Figure 2.8 The Flowchart for the runDfa method.

The flowchart illustrates the specifics of what happens inside the *runDfa* method; it provides a pseudocode implementation of any given language specification DFA. The pseudocode equivalent would be the following:

```
method runDfa( text, tt ) {
    state = 0
    word = ""
    char = ''

    while state is not accepting AND state is not error {

        if text == "" {
            break
        }

        char = text.nextchar()
        transition = tt [ char ][ state ]

        if transition.isConsuming {
            state = transition.target
            continue
        }
        else {
            text.prepend( char )
            if char == BLANK {
                state = transition.target
                continue
            }
            else {
                word = word.append( char )
                state = transition.target
                continue
            }
        }
    }

    return ( state, word )
```

Figure 2.9 The pseudocode for the flowchart.

For the case of Error Recognition, these will be pointed out by the *TransitionTables*' method *runDfa*; after executing, the method will return the state it ended with and based in the states' status, the *Scanner* object will raise an error message that, with the help of a line counter, includes the line in code in which the error occurred. For the purpose of reporting an unclosed comment this counter will have to take into account the amount of lines that have passed after a comment is opened, in case the comment does not close, so that the reported line number is accurate to where the comment was first opened.

Implementation

```python
from scanner import Scanner
from symboltable import SymbolTable
from transitiontable import TransitionTable

"""
Compiler Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Compiler Class serves as an interface to the multiple steps of compilation.

Attributes:
    + scanner: Scanner; The compilers Scanner, in charge of the lexical analysis.
    + symbol_tables: {str, SymbolTable}; The symbol tables created during the
      lexical analysis.
"""
class Compiler:
    """
    Compiler Class Constructor

    __init__(self) -> Compiler
    """
    def __init__(self):
        self.scanner: Scanner = None
        self.symbol_tables: {str, SymbolTable} = {}

        # The Compiler Class expects further expansion, to account for the
        # following phases of compilation
        self.parser = None
        self.syntax_tree = None

```

```
    """
    Compiler scannerPrep method

    Description:
        Executes the necessary preparations for the compilers' scanner given a
        text to scan, and a dfa to set the languages' constraints.

    scannerPrep(self, text: str, dfa: TransitionTable, st_names: [str]) -> None
    """
    def scannerPrep(self, text: str, dfa: TransitionTable, st_names: [str]):
        self.scanner = Scanner(text, dfa)
        for name in st_names:
            self.scanner.addSymbolTable(name, SymbolTable())

    """
    Compiler compile method

    Description:
        Executes all of the main components' methods to produce an output as
        indicated by the architecture diagram. Said output is usually passed
        as an argument as input to the next method.

    compile(self) -> None
    """
    def compile(self):

        # Lexical Analysis Phase
        self.scanner.tokenizeInput()
        token_stream = self.scanner.token_stream
        self.symbol_tables = self.scanner.symbol_tables

        # Visualization of both outputs of the Lexical Analysis
        for token in token_stream:
            print(token.toStr())

        for table in self.symbol_tables:
            print(table)
            self.symbol_tables[table].printTable()
```

Figure 3.1 The Compiler Class implementation.

```python
import sys
from symboltable import SymbolTable
from transitiontable import TransitionTable
from ctoken import Token
"""
Scanner Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Scanner class provides management and operation of the Scanner phase of a
    Compiler.

Attributes:
    + input_text: str;
      The text to scan and tokenize; It is consumed along with the scanning and
      recognition.
    + dfa: TransitionTable;
      The DFA, or rather its Transition Table implemented as a TransitionTable
      Object.
    + symbol_tables: {str, SymbolTable};
      The symbol tables created during the lexical analysis.
    + token_stream: [Token];
      The list of tokens that have been recognized by the scanner.
    + line: int;
      The counter for the lines read during scanning.
    + comment_displacement: int;
      The counter for the lines that occurred inside of a comment, in case said
      comment is not closed.
"""
class Scanner:
    """
    Scanner Class Constructor

    __init__(self, input_txt: str, dfa: TransitionTable ) -> Scanner
    """
    def __init__(self, input_txt: str, dfa: TransitionTable):
        self.input_text: str = input_txt
        self.dfa: TransitionTable = dfa
        self.symbol_tables: {str, SymbolTable} = {}
        self.token_stream: [Token] = []
        self.line: int = 1
        self.comment_displacement = 0
```

```
    """
    Scanner addSymbolTable method

    Description:
        Allows for the addition of a new distinct Symbol Table for the tokens
        produced by the Scanner.

    addSymbolTable(self, name: str, symboltable: SymbolTable) -> None
    """
    def addSymbolTable(self, name: str, symboltable: SymbolTable):
        self.symbol_tables.setdefault(name, symboltable)

    """
    Scanner addToken method

    Description:
        Allows for the addition of a new token to the token stream of the scanner.

    addToken(self, token: Token) -> None
    """
    def addToken(self, token: Token):
        self.token_stream.append(token)

```

```
    """
    Scanner runDfa method

    Description:
        Runs the TransitionTable equivalent of the DFA provided exactly once,
        and returns the resulting word and end state.

    runDfa(self) -> (state: int, word: str)
    """
    def runDfa(self) -> (int, str):
        self.input_text = self.input_text[::-1]
        state = 0
        word = ""
        current_ch = ""
        while not self.dfa.accepts(state) and not self.dfa.errors(state):
            if not self.input_text:
                return (state, word)
            current_ch = self.input_text[-1]
            #print("current_ch: " + current_ch)
            trans = self.dfa.nextTransition(state, current_ch)

            if trans.isConsuming:
                self.input_text = self.input_text[:-1]
                # if current_ch is not BLANK
                if current_ch not in ["\n", "\t", " ", "\r"]:
                    word = word + current_ch

            state = trans.target
            #print("state: " + str(state))

            if current_ch == '\n':
                self.line = self.line + 1
                if state == 8 or state == 9:
                    self.comment_displacement = self.comment_displacement + 1

        self.input_text = self.input_text[::-1]
        #print("word: " + word)
        return (state, word)
```

```
    """
    Scanner tokenizeInput method

    Description:
        Runs the runDfa Class function until the input_text provided is empty,
        and adds a token to the token_stream for every valid word recognized.

    tokenizeInput(self) -> None
    """
    def tokenizeInput(self):
        while self.input_text != "":
            state, word = self.runDfa()

            # In state 10 the word is either a keyword or an identifier
            if state == 10:
                # If word is a keyword
                if word.lower() in self.dfa.valid_tokens.keys():
                    tok = Token(self.dfa.valid_tokens[word.lower()])
                    self.addToken(tok)

                # If word is an identifier
                else:
                ## Name of the SymbolTable should also be dynamic, not hardcoded

                    valid_token_id = self.dfa.valid_tokens["IDENTIFIER"]
                    stid = self.symbol_tables["IDENTIFIER"].lookupEntryValue(word)
                    # If there is already an entry in the ST with this value
                    if stid != None:
                        tok = Token(valid_token_id, stid)
                        self.addToken(tok)
                    # If there is NO entry with this value
                    else:
                        self.symbol_tables["IDENTIFIER"].addEntry(word)
                        st_ref = self.symbol_tables["IDENTIFIER"].lookupEntryValue(word)
                        tok = Token(valid_token_id, st_ref)
                        self.addToken(tok)

```

```python
# In state 11 the word is a  numerical constant
elif state == 11:
    valid_token_id = self.dfa.valid_tokens["NUM_CONSTANT"]
    stid = self.symbol_tables["NUM_CONSTANT"].lookupEntryValue(word)
    # If there is already an entry in the ST with this value
    if stid != None:
        tok = Token(valid_token_id, stid)
        self.addToken(tok)
    # If there is NO entry with this value
    else:
        self.symbol_tables["NUM_CONSTANT"].addEntry(word)
        st_ref = self.symbol_tables["NUM_CONSTANT"].lookupEntryValue(word)
        tok = Token(valid_token_id, st_ref)
        self.addToken(tok)

elif state == 32:
    sys.exit(f"Invalid Character Error in line: {self.line}")
elif state == 33:
    sys.exit(f"Invalid Character '!' Error in line: {self.line}")
elif state == 34:
    sys.exit(f"Invalid Identifier Error in line: {self.line}")
elif state == 35:
    sys.exit(f"Invalid Number Constant Error in line: {self.line}")
elif state == 8 or state == 9:
    sys.exit(f"Incomplete Comment Error in line: {self.line - self.comment_displacement}")
elif state == 31:
    self.comment_displacement = 0
    pass
elif state in range(12,31):
    tok = Token(self.dfa.valid_tokens[word.lower()])
    self.addToken(tok)
```

Figure 3.2 The Scanner Class implementation.

```python
"""
Token Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Token Class provides a value object for the tokens created in the
    Lexical Analysis phase of a compiler.

Attributes:
    + keyword_id: int;
      The number that tells what type of valid word for the language the token
      represents.
    + symboltable_ref: int;
      The ID number that indicates an entry to a Symbol Table, if not an
      IDENTIFIER or NUMERICAL_CONST, it defaults to -1.
"""
class Token:
    """
    Token Class Constructor

    __init__(self, kwid: int, st_ref: int) -> Token
    """
    def __init__(self, kwid: int, st_ref: int = None):
        self.keyword_id = kwid
        self.symboltable_ref = st_ref

    """
    Token toStr method

    Description:
        Provides a way to transform the Token Object into a string of characters
        that can be printed onto the screen for visualization or written into a
        file for storage.

    toStr(self) -> str
    """
    def toStr(self) -> str:
        if self.symboltable_ref is None:
            return f"({self.keyword_id})"
        return f"({self.keyword_id}, {self.symboltable_ref})"

```

```python
    """
    Token fromStr method

    Description:
        Provides a way to transform Strings of characters created by the
        Token.print method into an actual Token Object in order to work with
        previously printed Token Objects.

    fromStr(str: str) -> Token
    """
    def fromStr(tokenstr: str):
        temp = tokenstr[1: -1:].split(",")
        return Token(int(temp[0]), int(temp[0]))

```

Figure 3.3 The Token Class implementation.

```python
from transition import Transition
"""
TransitionTable Class
author: Victor Emmanuel Guerra Aguado

Description:
    The TransitionTable Class represents the implementation of a Deterministic
    Finite Automaton for a given language into code. It performs everything the
    DFA would.

Attributes:
    + isAcceptingState: [bool];
      The list that represents the Accepting States of the Automaton.
    + isErrorState: [bool];
      The list thar represents the Error States of the Automaton.
    + transitions: {str, [Transition]};
      The list of transitions of the DFA, the lists in the dictionary act as the
      columns of a transition table, The specific transition is then obtained by
      using the index as the row of the table.
    + keywords: [str];
      The Keywords of the language, exclusively keywords, no symbols.
    + valid_tokens: {str, int};
      The enumeration of the valid tokens of the language
"""
class TransitionTable:
    """
    TransitionTable Class Constructor

    __init__(self,
                acceptingstates: [bool],
                errorstates: [bool],
                transitions: {str, [Transition]},
                keywords: [String],
                valid_tokens: {String: Number}) -> TransitionTable
    """
    def __init__(self, acceptingstates: [bool], errorstates: [bool],
        transitions: dict, keywords: [str], valid_tokens: {str, int}):
        self.isAcceptingState = acceptingstates
        self.isErrorState = errorstates
        self.transitions = transitions
        self.keywords = keywords
        self.valid_tokens = valid_tokens

```

```
    """
    TransitionTable accepts method

    Description:
        Method for finding out if the state provided is an Accepting State

    accepts(self, state: int) -> bool
    """
    def accepts(self, state: int) -> bool:
        return self.isAcceptingState[state]

    """
    TransitionTable errors method

    Description:
        Method for finding out if the state provided is an Error State

    errors(self, state: int) -> bool
    """
    def errors(self, state: int) -> bool:
        return self.isErrorState[state]

    """
    TransitionTable nextTransition method

    Description:
        Method for finding the transition that results by the given state and
        condition.

    nextTransition(self, state: int, condition: str) -> Transition
    """
    def nextTransition(self, state: int, condition: str) -> Transition:
        transition = self.transitions["INVALIDCH"][state]
        for col in self.transitions:
            if condition in col:
                transition = self.transitions[col][state]
                break

        return transition
```

Figure 3.4 The TransitionTable Class implementation

```python
"""
Transition Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Transition Class provides a structure for the entries made into the
    Transition Table made as an implementation of the DFA of a language.

Attributes:
    + target: int;
      The state to which this transition points.
    + isConsuming: bool;
      Whether this specific transition causes the DFA to advance to the next
      character.
"""
class Transition:
    """
    Transition Class Constructor

    __init__(self, target: int, consumes: bool) -> Transition
    """
    def __init__(self, target: int, consumes: bool):
        self.target = target
        self.isConsuming = consumes
```

Figure 3.5 The Transition Class implementation.

```python
import symboltableentry as ste
"""
SymbolTable Class
author: Victor Emmanuel Guerra Aguado

Description:
    The SymbolTable Class represents one of the Symbol Tables created during the
    Lexical Analysis phase of a compiler.

Attributes:
    + fields: [str];
      Names or descriptions of the fields contained within an entry to the
      SymbolTable.
    + entries: [SymbolTableEntry];
      The entries of the Symbol Table.
"""
class SymbolTable:
    """
    SymbolTable Class Constructor

    __init__(self, fields: [str]) -> SymbolTable
    """
    def __init__(self, fields: [str] = []):
        self.fields = fields
        self.entries = []

    """
    SymbolTable addEntry method

    Description:
        Method for adding an entry to the Symbol Table, given its value and fields.

    addEntry(self, value: str, fields: [str]) -> None
    """
    def addEntry(self, value: str, fields: [str] = []):
        if len(fields) != len(self.fields):
            print("Entries' fields do not match the Tables' fields.")
            exit()
        else:
            temp = ste.SymbolTableEntry(value, fields)
            self.entries.append(temp)

```

```python
    """
    SymbolTable lookupEntryId method

    Description:
        Method for looking up an entry in the SymbolTable given an index or
        entry ID.

    lookupEntryId(self, index: int) -> SymbolTableEntry
    """
    def lookupEntryId(self, index: int) -> ste.SymbolTableEntry:
        return self.entries[index]

    """
    SymbolTable lookupEntryValue method

    Description:
        Method for looking up an entry in the SymbolTable given a String value.
        Returns the entry index.

    lookupEntryValue(self, value: str) -> int
    """
    def lookupEntryValue(self, value: str) -> int:
        for entry in self.entries:
            if entry.value == value:
                return self.entries.index(entry)

        return None

    """
    SymbolTable printTable method

    Description:
        Method for printing the contents of the SymbolTable, in case visualization
        of the table is needed.

    printTable(self) -> None
    """
    def printTable(self):
        print("| ID | VALUE | " + "|".join(self.fields))
        for entry in self.entries:
            print(str(self.lookupEntryValue(entry.value)) + " | " + entry.value
            + "|".join(entry.fields))
```

Figure 3.6 The SymbolTable Class Implementation.

```python
"""
SymbolTableEntry Class
author: Victor Emmanuel Guerra Aguado

Description:
    The SymbolTableEntry Class provides a structure for the entries made into a
    Symbol Table during the Lexical Analysis phase of a Compiler.

Attributes:
    + value: str;
      The value to be stored as an entry of the symbol table, represents a lexeme
      recognized by the scanner.
    + fields: [str];
      Fields where any necessary extra information about the lexeme will be stored.
"""
class SymbolTableEntry:
    """
    SymbolTableEntry Class Constructor

    __init__(self, value: str, fields: [str]) -> SymbolTableEntry
    """
    def __init__(self, value: str, fields: [str] = []):
        self.value = value
        self.fields = fields
```

Figure 3.7 The SymbolTableEntry Class implementation.

```python
import sys
from compiler import Compiler
from transitiontable import TransitionTable
from transition import Transition

"""
    Main Program File for the C Minus Compiler
    author: Victor Emmanuel Guerra Aguado

Description:
    The main file of the entire compiler program, where the language definition is
    provided, and the source code text is obtained. So far, the hardcoded language
    definition is compliant to both that of the C Minus language, and that of the
    class diagrams designed during the design phase.

Attributes:
    None
"""
if __name__ == "__main__":
    # Obtain a Compiler Object
    comp = Compiler()

    # Get the file to parse
    if len(sys.argv) < 2:
        sys.exit("Input file not provided.")

    filename = sys.argv[1]
    input_file = open(filename, 'r')

    input_txt = input_file.read()

    input_file.close()


```

```
    # To prepare the Scanner, define a transition table of a DFA
    acceptingstates = [
        0,0,0,0,0,
        0,0,0,0,0,
        1,1,1,1,1,
        1,1,1,1,1,
        1,1,1,1,1,
        1,1,1,1,1,
        1,1,1,1,1,
        1
    ]

    errorstates = [
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,1,1,1,
        1
    ]
```

```
    transitions = {
        ('a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z',
        'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'): [
            Transition(1,True),Transition(1,True),Transition(35,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('0','1','2','3','4','5','6','7','8','9'):  [
            Transition(2,True),Transition(34,False),Transition(2,True),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('\n', ' ', '\r', '\t'): [
            Transition(0,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('<'): [
            Transition(3,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
```

```
        ('>'): [
            Transition(4,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('='): [
            Transition(5,True),Transition(10,False),Transition(11,False),Transition(12,True),Transition(14,True),
            Transition(16,True),Transition(29,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('!'): [
            Transition(6,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('/'): [
            Transition(7,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(31,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
```

```
        ('+'): [
            Transition(18,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('-'): [
            Transition(19,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('*'): [
            Transition(20,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(8,True),Transition(9,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        (';'): [
            Transition(21,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
```

```
        (','): [
            Transition(22,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('('): [
            Transition(23,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        (')'): [
            Transition(24,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('['): [
            Transition(25,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
```

```
        (']'): [
            Transition(26,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('{'): [
            Transition(27,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('}'): [
            Transition(28,True),Transition(10,False),Transition(11,False),Transition(13,False),Transition(15,False),
            Transition(17,False),Transition(33,True),Transition(30,False),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
        ('INVALIDCH'): [
            Transition(32,True),Transition(32,True),Transition(32,True),Transition(32,True),Transition(32,True),
            Transition(32,True),Transition(32,True),Transition(32,True),Transition(8,True),Transition(8,True),
            Transition(10,False),Transition(11,False),Transition(12,False),Transition(13,False),Transition(14,False),
            Transition(15,False),Transition(16,False),Transition(17,False),Transition(18,False),Transition(19,False),
            Transition(20,False),Transition(21,False),Transition(22,False),Transition(23,False),Transition(24,False),
            Transition(25,False),Transition(26,False),Transition(27,False),Transition(28,False),Transition(29,False),
            Transition(30,False),Transition(31,False),Transition(32,False),Transition(33,False),Transition(34,False),Transition(35,False)
        ],
    }
```

```
keywords = ["else", "if", "input", "int", "output", "return", "void", "while"]

valid_tokens = {
    "else": 1,
    "if": 2,
    "input": 3,
    "int": 4,
    "output": 5,
    "return": 6,
    "void": 7,
    "while": 8,
    "+": 9,
    "-": 10,
    "/": 11,
    ">": 12,
    ">=": 13,
    "<": 14,
    "<=": 15,
    "==": 16,
    "!=": 17,
    "=": 18,
    ";": 19,
    ",": 20,
    "(": 21,
    ")": 22,
    "{": 23,
    "}": 24,
    "[": 25,
    "]": 26,
    "IDENTIFIER": 27,
    "NUM_CONSTANT": 28,
    "COMMENT": 29,
}

cminusdfa = TransitionTable(acceptingstates, errorstates, transitions, keywords, valid_tokens)

# We add the DFA and the Input text to the scanner
comp.scannerPrep(input_txt, cminusdfa, ["IDENTIFIER", "NUM_CONSTANT"])

# Start the compilation process
comp.compile()
```

Figure 3.8 The main file for the compiler program.

Verification & Validation

The Verification and Validation phase consists of making sure that the product is being well developed and that the developed product is the correct one. In this section a test model, consisting of several test cases will be presented, along with their purpose, input, expected result, result, and evaluation. The Test Case templates are based on the ones suggested by Thomas Hamilton.[2]

Test Cases

CMCTC-01

Description/Purpose:

To verify that the scanner object creates tokens completely and correctly from the given input text and according to the C Minus language definition.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-01 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test input:

```
“void main(){
    else if input int output return void while;
    i = 0;
    i + 1
    i - 1
    i * 1
    i / 1
    i > 1 >= 0
    i < 1 <= 0
    i == 1
    i != 1
    [, ]
    /**/
}“
```

Expected Result:

(7), (28, 0), (22), (23), (24), (1), (2), (3), (4), (5), (6), (7), (8), (20), (28, 1), (19), (29, 0), (20), (28, 1), (9), (29, 1), (28, 1), (10), (29,1), (28,1), (11), (29,1), (28,1), (12), (29,1), (28,1), (13), (29,1), (14), (29,0), (28,1), (15), (29,1), (16), (29,0), (28,1), (17), (29,1), (28,1), (18), (29,1), (26), (21), (27), (25)

IDENTIFIER

| ID | VALUE |

0 | main

1 | i

NUM_CONSTANT

| ID | VALUE |

0 | 0

1 | 1

Actual Result:

(7)

(28, 0)

(22)

(23)

(24)

(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(20)

(28, 1)

(19)

(29, 0)

(20)

(28, 1)

(9)

(29, 1)

(28, 1)

(10)

(29, 1)

(28, 1)

(11)

(29, 1)

(28, 1)

(12)

(29, 1)

(28, 1)

(13)

(29, 1)

(14)

(29, 0)

(28, 1)

(15)

(29, 1)

(16)

(29, 0)

(28, 1)

(17)

(29, 1)

(28, 1)

(18)

(29, 1)

(26)

(21)

(27)

(25)

IDENTIFIER

| ID | VALUE |

0 | main

1 | i

NUM_CONSTANT

| ID | VALUE |

0 | 0

1 | 1

Evaluation: PASS

```
    Test Case CMCTC-01

    Description/Purpose:
        To verify that the scanner object creates tokens completely and correctly
        from the given input text and according to the C Minus language definition.
    """
    def test_token_creation(self):
        input_txt = """void main(){
    else if input int output return void while;
    i = 0;
    i + 1
    i - 1
    i * 1
    i / 1
    i > 1 >= 0
    i < 1 <= 0
    i == 1
    i != 1
    [, ]
    /**/
}"""

        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        expected_token_stream = [Token(7),Token(28,0),Token(22),Token(23), Token(24),
        Token(1), Token(2), Token(3), Token(4), Token(5), Token(6), Token(7), Token(8),
        Token(20), Token(28,1), Token(19), Token(29,0), Token(20), Token(28,1), Token(9),
        Token(29,1), Token(28,1), Token(10), Token(29,1), Token(28,1), Token(11),
        Token(29,1), Token(28,1), Token(12), Token(29,1), Token(28,1), Token(13),
        Token(29,1), Token(14), Token(29,0), Token(28,1), Token(15), Token(29,1),
        Token(16), Token(29,0), Token(28,1), Token(17), Token(29,1), Token(28,1),
        Token(18), Token(29,1), Token(26), Token(21), Token(27), Token(25)]

        self.compiler.compile()

        expected_tokens = []
        for tok in expected_token_stream:
            expected_tokens.append(tok.toStr())

        actual_tokens = []
        for tok in self.compiler.scanner.token_stream:
            actual_tokens.append(tok.toStr())

        self.assertListEqual(expected_tokens, actual_tokens)
```

Figure 4.1 CMCTC-01 Test Case Implementation.

CMCTC-02

Description/Purpose:

To verify that the scanner completely and correctly creates Symbol Table entries of any valid given type.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-02 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test Input:

```
"void main() {
	one = 0;
	two = 1;
	three = 2;
}"
```

Expected Result:

(7), (28,0), (22), (23), (24), (28,1), (19), (29,0), (20), (28,2), (19), (29,1), (20), (28,3), (19), (29,2), (20), (25)

IDENTIFIER

ID | VALUE

0 | main

1 | one

2 | two

3 | three

NUM_CONSTANT

ID | VALUE

0 | 0

1 | 1

2 | 2

Actual Result:

(7)

(28, 0)

(22)

(23)

(24)

(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(20)

(28, 1)

(19)

(29, 0)

(20)

(28, 1)

(9)

(29, 1)

(28, 1)

(10)

(29, 1)

(28, 1)

(11)

(29, 1)

(28, 1)

(12)

(29, 1)

(28, 1)

(13)

(29, 1)

(14)

(29, 0)

(28, 1)

(15)

(29, 1)

(16)

(29, 0)

(28, 1)

(17)

(29, 1)

(28, 1)

(18)

(29, 1)

(26)

(21)

(27)

(25)

IDENTIFIER

| ID | VALUE |

0 | main

1 | i

NUM_CONSTANT

| ID | VALUE |

0 | 0

1 | 1

Evaluation: PASS

```
    """
    Test Case CMCTC-02

    Description/Purpose:
        To verify that the scanner completely and correctly creates SymbolTable
        entries of any valid given type.
    """
    def test_symbol_table_entries(self):
        input_txt = """void main() {
    one = 0;
    two = 1;
    three = 2;
}
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        expected_token_stream = [Token(7), Token(28,0), Token(22), Token(23), Token(24),
        Token(28,1), Token(19), Token(29,0), Token(20), Token(28,2), Token(19),
        Token(29,1), Token(20), Token(28,3), Token(19), Token(29,2), Token(20), Token(25)]

        self.compiler.compile()

        expected_tokens = []
        for tok in expected_token_stream:
            expected_tokens.append(tok.toStr())

        actual_tokens = []
        for tok in self.compiler.scanner.token_stream:
            actual_tokens.append(tok.toStr())

        self.assertListEqual(expected_tokens, actual_tokens)
```

Figure 4.2 The CMCTC-02 Test Case Implementation.

CMCTC-03

Description/Purpose:

To verify that the scanner raises an error when encountering an unclosed comment.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-03 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test Input:

“void main() {

/* Main Function */

if(true){

/* Comment

        }
}
"

Expected Result:

Incomplete Comment Error in line: 4

Actual Result:

Incomplete Comment Error in line: 4

Evaluation: PASS

```
    """
    Test Case CMCTC-03

    Description/Purpose:
        To verify that the scanner raises an error when encountering an unclosed
        comment.
    """
    def test_unclosed_comment(self):
        input_txt = """void main() {
            /* Main Function */
            if(true){
                /* Comment
            }
        }
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        with self.assertRaises(SystemExit) as res:
            self.compiler.compile()
        print(res.exception)
        self.assertEqual(res.exception.__str__(), "Incomplete Comment Error in line: 4")
```

Figure 4.3 The CMCTC-03 Test Case Implementation

CMCTC-04

Description/Purpose:

To verify that the scanner raises an error when encountering an invalid character for the language.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-04 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test Input:

“void main() {

int 編譯器;

編譯器 = 10;

}

”

Expected Result:

Invalid Character Error in line: 2

Actual Result:

Invalid Character Error in line: 2

Evaluation: PASS

```
    """
    Test Case CMCTC-04

    Description/Purpose:
        To verify that the scanner raises an error when encountering an invalid
        character for the language.
    """
    def test_invalid_character(self):
        input_txt = """void main() {
                int 編譯器;
                編譯器 = 10;
        }
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        with self.assertRaises(SystemExit) as res:
            self.compiler.compile()
        print(res.exception)
        self.assertEqual(res.exception.__str__(), "Invalid Character Error in line: 2")
```

Figure 4.4 The CMCTC-04 Test Case Implementation.

CMCTC-05

Description/Purpose:

To verify that the scanner accepts any character, even invalid characters for the language, if it is placed inside a comment.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-05 test input source code.
3. Run the compiler.

4. Compare the result with the expected result.

Test Input:

“void main() {

/* 冰淇凌 */

}

”

Expected Result:

[(7), (28,0), (22), (23), (24), (25)]

IDENTIFIER

ID | VALUE

0 | main

NUM_CONSTANT

ID | VALUE

Actual Result:

(7)

(28, 0)

(22)

(23)

(24)

(25)

IDENTIFIER

| ID | VALUE |

0 | main

NUM_CONSTANT

| ID | VALUE |

Evaluation: PASS

```
    """
    Test Case CMCTC-05

    Description/Purpose:
        To verify that the scanner accepts any character, even invalid characters
        for the language, if it is placed inside a comment.
    """
    def test_characters_in_comment(self):
        input_txt = """void main() {
                /* 冰淇凌 */
}
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        expected_token_stream = [Token(7), Token(28,0), Token(22), Token(23), Token(24),
        Token(25)]

        self.compiler.compile()

        expected_tokens = []
        for tok in expected_token_stream:
            expected_tokens.append(tok.toStr())

        actual_tokens = []
        for tok in self.compiler.scanner.token_stream:
            actual_tokens.append(tok.toStr())

        self.assertListEqual(expected_tokens, actual_tokens)
```

Figure 4.5 The CMCTC-05 Test Case Implementation.

CMCTC-06

Description:

To verify that the scanner can raise an error when recognizing an invalid identifier.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-06 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test Input:

“void main() {

int i2;

i2 = 0;

}

”

Expected Result:

Invalid Identifier Error in line: 2

Actual Result:

Invalid Identifier Error in line: 2

Evaluation: PASS

```
    """
    Test Case CMCTC-06

    Description/Purpose:
        To verify that the scanner can raise an error when recognizing an invalid
        identifier.
    """
    def test_invalid_identifier(self):
        input_txt = """void main() {
                int i2;
                i2 = 0;
}
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        with self.assertRaises(SystemExit) as res:
            self.compiler.compile()
        print(res.exception)
        self.assertEqual(res.exception.__str__(), "Invalid Identifier Error in line: 2")
```

Figure 4.6 The CMCTC-06 Test Case Implementation.

CMCTC-07

Description:

To verify that the scanner can raise an error when recognizing an invalid numerical constant.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-06 test input source code.
3. Run the compiler.
4. Compare the result with the expected result.

Test Input:

"void main() {

int i;

i = 2b;

}

"

Expected Result:

Invalid Number Constant Error in line: 3

Actual Result:

Invalid Number Constant Error in line: 3

Evaluation: PASS

```python
    """
    Test Case CMCTC-07

    Description/Purpose:
        To verify that the scanner can raise an error when recognizing an invalid
        numerical constant.
    """
    def test_invalid_num_constant(self):
        input_txt = """void main() {
        int i;
        i = 2b;
}
"""
        self.compiler.scannerPrep(input_txt, self.dfa, self.symbol_table_names)

        with self.assertRaises(SystemExit) as res:
            self.compiler.compile()
        print(res.exception)
        self.assertEqual(res.exception.__str__(), "Invalid Number Constant Error in line: 3")
```

Figure 4.7 The CMCTC-07 Test Case Implementation.

References

1. R. Castelló, Class Lecture, Topic: "Chapter 2 - Lexical Analysis." TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, April, 2022.
2. Thomas Hamilton, "How to Write Test Cases: Sample Template with Examples", *Guru99*, 2 April 2022 [journal on-line]; available from https://www.guru99.com/test-case.html; Internet; accessed 18 April 2022.