Finite Automata and Lexical Analysis

Vaishali P. Bhosale¹, Dr. S. R. Chaudhari²

¹Yashwantrao Chavan School of Rural Development, Shivaji University, Kolhapur, India
²Dept. of Mathematics, North Maharashtra University, Jalgaon, Maharashtra, India

Abstract—This paper introduces basics of lexical analysis. The finite automata concepts such as regular expression, non-deterministic finite automata, deterministic finite automata forms the basis for lexical analyser i.e. scanner are explained next. The lexical analyser for the C-Minus language is constructed.

Keywords—Lexical analysis, RE, NFA, DFA, C-Minus.

I. INTRODUCTION

In lexical analysis (scanning) phase of compiler the source program is analysed character by character and tokens are separated. Each token is a sequence of characters that represents a unit of information of the source program. Keywords are the tokens of fixed strings of letters. For example “int”, “if” etc. Identifiers are the tokens that are user defined strings usually begins with letter and consisting of letters and digits. For example temp1, num2 etc. Special symbols are also treated as tokens that are arithmetic symbols as well as multi-character symbols. For example +, *, <=, => etc. For more details on lexical analysis we referred to [1]. In the following Figure 1, we explain the lexical analysis phase with an example of variable declaration sentence.

```
int number1, number2, sum;
```

Secondly, it also stores the user defined symbols in a table called the symbol table. This shows that the functionalities of scanning and parsing are highly coupled.

Generally speaking lexical analysis is treated as a special case of pattern matching. Regular expressions and finite automata appear to be the proper tools for pattern specification and recognition required in lexical analysis. Regular expressions are algebraic notations for describing the sets of strings. The language generated by the regular expression “R” is represented as L(R). The set of all symbols that are used in the language is called the alphabet of the language and we shall denote it by the symbol Σ. For example, the set of English letters is the alphabet Σ = {A, B, C,…..,a,b,c…}. The set of digits is the alphabet Σ = {0, 1, ….,9}. A finite sequence of symbols from the alphabet is called a string or a word over that alphabet. The regular expression generally involved special characters called meta characters. For example *, + etc. Three important terms in lexical analysis (with similar meanings) are lexeme, token and patterns. Lexemes are possible input values for a given type of token, tokens are classes of similar lexemes and pattern gives formal or informal description for a token. For example in the statement below:

```
int number1, number2, sum;
```

```
number1, number2 and sum are lexemes of token type identifier with pattern specification as: an identifier is a string, with first character as an alphabet and the successive characters are either digits or alphabets. We now state few theorems that establish relation between regular expressions, DFA’s and NFA’s.
```

---

**Theorem 1.1[2]:** Let R is a regular expression. Then there exists an NFA with ε-transitions that accepts L(R).

**Theorem 1.2[2]:** If L is accepted by an NFA with ε-transitions, then L is accepted by an NFA without ε-transitions

**Theorem 1.3[2]:** Let L be a set accepted by NFA. Then there exists a DFA that accepts L.

**Theorem 1.4[2]:** If L is accepted by a DFA, then L is denoted by a regular expression.

---

The lexical analyzer (i.e. scanner) filters out white spaces, new lines and comments. The scanner is operated under the control of the syntax analyzer (i.e. parser). It returns a single token from the input, whenever demanded by the parser. Thus, the primary task of the scanner is to convert the entire source program into a sequence of tokens based on the request from the parser.

---

Figure 1 Lexical Analysis

The lexical analyzer (i.e. scanner) filters out white spaces, new lines and comments. The scanner is operated under the control of the syntax analyzer (i.e. parser). It returns a single token from the input, whenever demanded by the parser. Thus, the primary task of the scanner is to convert the entire source program into a sequence of tokens based on the request from the parser.
II. FRACT AUTOMATA TO DESIGN LEXICAL ANALYZER

In this section we discuss lexical analyzer and its implementation for the C-minus language [1]. We have begun with the construction of regular expressions for tokens of the C-minus language. The tokens of C-Minus language fall into three categories: reserved words, special symbols, and other tokens. There are six reserved words (with familiar meanings). There are 21 special symbols, including four basic arithmetic operations, six relational operators, assignment operator, parentheses, comments specific symbols, semicolon and comma. Comments are generally enclosed in /*…*/ and cannot be nested. The other tokens are numbers (sequences of one or more digits) and identifiers (sequences of one or more letters). Lexical structure of the C-Minus language is summarized in Table I:

<table>
<thead>
<tr>
<th>TABLE I: Tokens in C-Minus</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reserved words</strong></td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>while</td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>

The regular expressions for tokens in the C-Minus language are defined as follows:

Keywords = if | else | return | while | int | void
Special symbols = + - * / | <= | >= | = | != | [ ] | { } ; |
NUM=digit digit*  where, digit=0|1|…|9
ID= letter letter* where, letter=a|b|…|z|A|B|…|Z

According to Thompson’s rule [3] and Theorem 1.1 these regular expressions are converted into NFA with $\varepsilon$-moves. These NFAs with $\varepsilon$-moves are then converted into DFAs by successive applications of Theorem 1.2 and Theorem 1.3. For this we need the concept of $\varepsilon$-closure. We briefly discuss it below for convenience.

The $\varepsilon$-closure (T) of the state set T is the set of states that can be reached from T on $\varepsilon$-transitions. The procedure to find $\varepsilon$-closure (T) is given as follows:

1. Elements in state T are added to $\varepsilon$-closure (T).
2. If t is in $\varepsilon$-closure (T), and there is an edge labeled $\varepsilon$ from state t to state u, then state u is also added to $\varepsilon$-closure (T).
3. Rule 2 is repeated until no more states can be added to $\varepsilon$-closure (T).

Once the $\varepsilon$-closure of the set of states has been evaluated, we get a set of states of NFA that is equivalent to a single state of DFA. Now we extend this state on a particular input to get a set of new states of NFA. This is equivalent to the new states of DFA. Let T be the current state of DFA. Let the state T accept input c. Using the subset construction algorithm the DFA moves to the new set of states of NFA as given below:

$$DFAEdge(T,c) = \varepsilon - closure \bigcup_{q \in T} move(q,c)$$

We use the operations in Table II to keep the track of sets of NFA states (s represents an NFA state and T a set of NFA states).

<table>
<thead>
<tr>
<th>TABLE II: Sets of NFA states</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>$\varepsilon$-closure (s)</td>
</tr>
<tr>
<td>$\varepsilon$-closure (T)</td>
</tr>
<tr>
<td>Move(T,a)</td>
</tr>
</tbody>
</table>

The following steps are necessary to convert an NFA with $\varepsilon$-transitions to the equivalent DFA.

1. The initial state of DFA is the initial state of NFA together with the state that can be reached on $\varepsilon$-transitions.
2. The accepting states of DFA are those set of states that contain at least one accepting state of NFA.
3. The DFA is then minimized by finding equivalent states and using minimization algorithm as stated in [2].

The following algorithm is generally followed to minimize the DFA:

1. Partition the states into two groups of accepting and non-accepting states.
2. In each group find a sub-group of states having the same property. The states which have the transition on each input symbol in the alphabet to the same group are grouped together.
3. Step 2 is continued until no further partitioning of the set of states can be carried out.
4. Now each group in the partition becomes a state in the minimized DFA.

There are two main approaches for the construction of lexical analyzers.

1. Hand coded lexical analyzers: this actually involves writing a program to perform lexical analysis.
2. Using lexical analyzer generator: the lexical analyzer generator is a tool that automatically generates the lexical analysis program from a formal description of the tokens of the language and their attributes.

The lexical analyzer generated by the first method is possibly more efficient, whereas the lexical analyzer generated by second method is faster and is less prone to errors. The lexical analyzer is usually implemented as a subroutine or a co-routine of the parser.

III. IMPLEMENTATION FOR THE C-MINUS LANGUAGE

With this as a base we have coded the scanner for the C-Minus language, for more detailed description we refer to [35]. We have applied all the concepts of (crisp) automata theory to code the C-Minus scanner. Our task here is to completely specify the lexical structure of the C-Minus language. In designing the scanner for this language, we have begun with regular expressions and developed NFA and converted it into equivalent DFA. We have developed the minimized state DFA for the scanner directly. The final DFA for the C-Minus language is given in Figure 2.

We have not included reserved words in this DFA. Since it is easiest, in the view of the DFA to consider reserved words as identifiers, and then look for the identifiers given in the Table II for reserved words, after acceptance. Thus, reserved words are generally considered after an identifier has been recognized.

The easiest way to simulate this DFA is to write the code as follows:

{Begin in S state}
1. if the next character is a letter then
   Advance the input;
   {Now in state ID}
   While the next character is a letter do
      Advance the input;
      {Stay in state ID}
   End while
   {Go to state D without advancing the input}
   Accept token as an identifier;
   State = S;

2. else if the next character is a digit then
   Advance the input;
   {Now in state NUM}
   While the next character is a digit do
      Advance the input;
      {Stay in state NUM}
   End while
   {Go to state D without advancing the input}
   Accept token as a number;
   State = S;

3. else if the next character is special symbol
   \{+ - * ; ( ) [ ] \} \)
   then
      Accept as special symbol;
      Advance the input;
      State = S;


Figure 2 DFA for C-Minus Tokens

The easiest way to simulate this DFA is to write the code as follows:
4. else if the next character is NOT then
   Advance the input;
   \{Now in state NEQ\}
   If the next character is EQ then
   Accept as NOTEQ;
   State = S;
   Else
   Error
   End if
5. Else if the next character is LT then
   Advance the input;
   \{Now in state LT\}
   If the next character is EQ then
   State = D;
   Accept as LTEQ;
   Advance the input;
   State = S;
   Else
   \{Go to state D without advancing the input\}
   Accept as LT;
   State = S;
   End if
6. Else if the next character is GT then
   Advance the input;
   \{Now in state GT\}
   If the next character is EQ then
   State = D;
   Accept as GTEQ;
   Advance the input;
   State = S;
   Else
   \{Go to state D without advancing the input\}
   Accept as GT;
   State = S;
   End if
7. Else if the next character is EQ then
   Advance the input;
   \{Now in state ASS\}
   If the next character is EQ then
   State = D;
   Accept as REQ;
   Advance the input;
   State = S;
   Else
   \{Go to state D without advancing the input\}
   Accept as ASS;
   State = S;
   End if
8. Else if the next character is ‘/’ then
   Advance the input;
   \{Now in state DIV\}
   If the next character is ‘*’ then
   Advance the input;
   \{Now in state CMT\}
   Done=false;
   While not done do
   While the input character is not ‘*’ do
   Advance the input;
   \{Stay in state CMT\}
   End while
   State = INC;
   Advance the input;
   While the next character is not ‘/’ do
   Advance the input;
   \{Now in state CMT\}
   End while
   State = S;
   Done = true;
   End while;
   Else
   Accept the token as DIV;
   \{Go to state S without advancing the input\}
   End if
End if.

REFERENCES