COMPILER CONSTRUCTION (Lexical Analyser) **Fall 2019**

Lecture 5, 6: July 25-26, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

5.1 Introduction

The first phase of compiler is lexical analyser, whose main task is to read the input characters from the source program and assemble into lexemes, and produce output as sequence of tokens. These tokens are input to parser for syntax analysis. The lexical analyser interacts with the symbol table also. When it encounters a lexeme in the form of identifier, it is entered into the symbol table. Some times it reads information from the symbol table to determine the proper token.

A commonly used way for above is to implement the lexical analyser as a *routine* of the parser. The lexical analyser is called by a call *nextToken* (see Fig 5.1), in return the lexical analyser returns the next token to the parser. The parser checks the syntax by constructing syntax tree for each statement of the language, and sends this to semantic analyser.

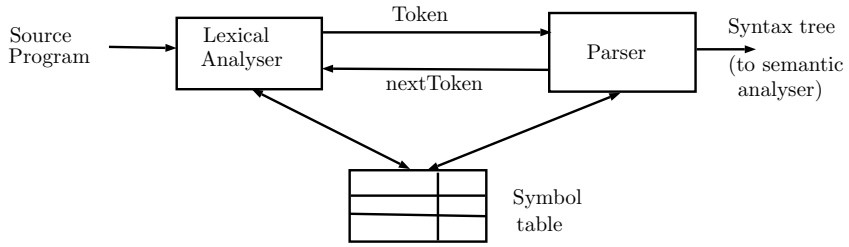

Figure 5.1: Lexical analyser as routine of parser

The lexical analyser not only generates the tokens but performs some other tasks also; these are stripping of *whitespace* (blanks, new line, tab, some other characters which may be used to separate the token in the input), and comments.

5.2 Lexical Analysis vs. Parsing

There are number of reasons why the analysis portion of a compiler is separated into lexical analysis and syntax analysis (parsing) phases. One reason is due to simplicity of design.

The separation of these two allows us to simplify at least one of these tasks. For example, the parser becomes simpler if *whitespaces* and comments are removed. If there is a new language, this separation simplifies the design.

The efficiency is improved for the compiler – a separate lexical analyser allows use of specialised technique that serve only the lexical task. Apart from this, a separate buffering is used for reading input into that before it goes to lexical analyser. This buffering speeds up the process for lexical analyser.

The compiler portability is enhanced, as input-device specific properties can be confined to the lexical analyser.

5.2.1 Token Attributes

In many cases, the lexical analyser returns to parser not only the token name, but *attribute value* that describe the lexeme represented by the token. The token name influence parsing decision, while attribute value influence translation of token after the parse.

We assume that every token has at most one attribute associated with it, however this attribute may have further structure in it. E.g., the token **id**, using which we can associate a great deal of information with token. This information may be, its *lexeme*, its *type*, and *location* where it is found in the program. The latter is needed, to flag the error for that location. All this information is kept in the symbol table. The appropriate value of the identifier is a pointer to the symbol-table entry for that identifier. In some languages it is challenging to recognize the lexemes, e.g., in Fortran, the statement "DO5I = 1,20". In this statement, until we come to comma sign, it is not clear whether DO5I is a variable or a DO statement.

Examples of tokens are all the keywords in any program, all identifiers, operators, punctuation marks, parentheses, curly braces, etc. While the non-tokens are comments, preprocessor directives, macros, tabs, blanks, and newlines.

Example 5.1 *Token names and associated attribute values for the statement* $e = m * c * *2$.

The token names and associated attribute names as follows:

$\langle \textbf{id}, 1 \rangle$
$\langle \textbf{assignop} \rangle$
$\langle \textbf{id}, 2 \rangle$
$\langle \textbf{multiop} \rangle$
$\langle \textbf{id}, 3 \rangle$
$\langle \textbf{expop} \rangle$
$\langle \textbf{number}, \textit{integer value } 2 \rangle$

In $\langle id, 1 \rangle$, 1 is pointer to symbol-table entry for e, 2 for m, and 3 for c. Note that for certain pairs, specially operators, punctuations, and keywords there is no attribute value required. The "number" has been given integer values attribute. In practice, for constants numbers, a digits sequence is stored and its attribute is a pointer to that string of digit sequence.

5.2.2 Lexical Errors

It is difficult for a lexical analyser to tell about any lexical errors, without the help of other components. For example, if the lexical analyser first time encounters the word `fi`, in the statement

$$\texttt{fi}(\texttt{a} == \texttt{b})....$$

the lexical analyser cannot say, whether `fi` is the keyword, like, `if` or an undeclared function identifier for function $fi(a == b)$. Since `fi` is a valid lexeme for any token **id**, the lexical analyser will return **id** to the parser. The reason for this is that lexical cannot say any thing about the structure of statement. This error will be handled by the parser phase of the compiler. Suppose the situation in the lexical analyser is such that none of the patterns for the tokens matches prefix of the remaining input. The simplest solution in this case is "panic mode" recovery. The successive characters from the remaining input (e.g., characters after fi in this case) are deleted until the lexical analyser can find a well formed token at the beginning of what input is left.

The other possible error recovery methods are: delete one character from the remaining input[1], insert a missing character into the remaining input, replace a character by other, transpose two adjacent characters.

5.3 Input buffering

We normally consider that source file is stored on disk, therefore, we are able to read one character at a time, and that too at slow speed. The input buffering stores the source file in (RAM) buffer, hence makes it possible to foresee a lexeme beyond its boundary to determine it correctly. This will help in the solution of problems discussed above (subsection 5.2.2). In the C language, the single characters, like, $-, +, >, <, =$ are operators themselves, as well as they are beginnings of two characters, $--, ++, >=, <=, <>, ==$. Hence, unless the second character is read we are not in a position to determine, whether it is '$-$' or '$--$' lexeme, and similar in the others.

The size of buffer is usually equal to a block (1, or 2, or 4 k bytes). By storing the source file fully or in part in the buffer makes it possible to read the input at speed, as well fore see beyond any lexeme.

To scan the lexeme present in the buffer, two pointers are needed: one points to the begin of a lexeme and other goes forward to detect the end of a lexeme, as shown in Fig. 5.2, where begin and forward pointers' position indicate that >= lexeme has been detected.

Once a lexeme is found the "forward pointer" is set to a character at its right end. Then, after the lexeme is recorded, an attribute value is passed to the parser, and the lexeme's begin pointer is set to the character immediately following the end character of the current lexeme. In the Fig. 5.2, we see that end of the lexeme ">=" is at position "forward pointer value-1". Once the lexeme is recorded, the forward pointers value is assigned to begin pointer, and forward pointer is advanced to record the next lexeme. Once, the forward

[1]That is to delete a character for consideration of token, and not permanently deleting it from the input text.

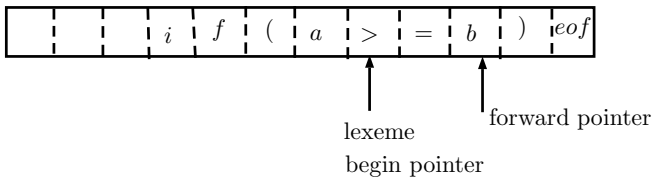

Figure 5.2: Input buffer

pointer moves to end of buffer (equal to buffer size N), the buffer is reloaded from the beginning, and pointers are realigned for the current lexeme. Note that, the "eof" marker is not the actual end of source file, but used as a *sentinal* character.

5.4 Regular Expressions

The identifiers of any program can be created by performing operations of union, concatenations, and kleene closure, on the sets of letters and digits. This may also include underscore characters(s). The identifiers so created are *regular expressions*. The regular expressions also represent all the languages that can be described by performing these operations on the symbols of these languages. If "*letter_*" stands for any letter or underscore, and *digit* stands for any digit, then the C language identifiers can be described by:

$$letter\_(letter\_ \mid digit)^*$$

Note that, "letter_" is a representation for letter or underscore, and not both. For construction of regular expressions, we will prefer to drop the parentheses, which enforce the rules of precedence. Following conventions are adopted for association and precedence of operators:

1. The operator $*$ is of highest precedence, and it is left associative[2].
2. The concatenation is second highest associative, and it is also left associative.
3. "$|$" is left associative, and it has lowest precedence.

Example 5.2 *Regular expressions for the set* $\Sigma = \{a, b\}$.

1. Regular expression $a|b$ is equal to $\{a, b\}$.
2. $(a|b)(c|d)$ denotes set $\{ac, ad, bc, bd\}$.
3. a^* denotes the set $\{\varepsilon, a, aa, aaa, ...\}$ □

Example 5.3 *Some definitions using regular expressions.*

[2]The operators $+, -, *$ and concatenation ($\circ$) are left associative for a expressions, say, $a + b + c$, where first $a + b$ is computed, result is substituted in place of that, then c is added into the result. However, in $a * *b * *c$, first b^c is computed, then its resulted, say t, is used to compute a^t, so the net expression result is a^{b^c}.

$$
\begin{aligned}
letter\_ &\rightarrow A \mid B \mid ... \mid Z \mid a \mid b \mid ... \mid z \mid \_ \\
digit &\rightarrow 0 \mid 1 \mid ... \mid 9, \text{or} [0-9] \\
id &\rightarrow letter\_(letter\_ \mid digit)^* \\
digits &\rightarrow digit\ digit^* \\
fraction &\rightarrow .\ digits \mid \varepsilon \\
exponent &\rightarrow (E \mid + \mid - \mid \varepsilon)\ digits) \mid \varepsilon \\
number &\rightarrow digits\ fraction\ exponent \\
relop &\rightarrow <|>|<=|>=|=|<>
\end{aligned}
$$

□

5.5 Recognition of Tokens

In this section, given patterns for all needed tokens, we will design a piece of code that examines the input string and finds a prefix, i.e., finds prefix that is a lexeme, matching one of the patterns. We will make use of following standard examples, as grammar of branching statements.

$$
\begin{aligned}
stmt &\rightarrow if\ expr\ then\ stmt \mid if\ expr\ then\ stmt\ else\ stmt \mid \varepsilon \\
expr &\rightarrow term\ relop\ term \mid term \\
term &\rightarrow id \mid number
\end{aligned}
$$

In the above, the *relop*, i.e., relational operators are: $>, <, >=, <=, <>, =$, and they have attribute values respectively as, GT, LT, GE, LE, NE, EQ.

In addition to these, the lexical analyser must strip out the white spaces, by recognizing the token ws as defined below.

$$ws \rightarrow (blank \mid tab \mid newline)^+$$

The above are abstract symbols, which stands for their respective ASCII symbols. The Table 5.1 shows the relations between lexemes, tokens, and attributes values.

5.5.1 Transition Diagrams

The first step in construction of a lexical analyzer is to convert the patterns into special flow-charts, called "Transition diagrams", having nodes or circles (called *states*), and edges representing transitions. Each state represents a condition that occur during the scanning of the input, and looking for a lexeme that matches one of the several patterns. Each edge is labeled by a symbol or a set of symbols.

We assume that all the transition diagrams are deterministic, i.e., at every node, there is never more than one edge for one symbol. Certain states in the transition diagrams are called *accepting* or *final* states. These indicate that lexeme is found, and indicated by a double circle. If it is necessary to retract the *forward* pointer one position, then we shall

Table 5.1: Relation between lexemes, tokens, and values

Lexemes	Tokens	Attribute values
Any *ws*	-	-
if	**if**	-
then	**then**	-
Any *id*	**if**	Pointer to symbol table
Any *number*	**number**	Pointer to symbol table
<	**relop**	*LT*
>	**relop**	*GT*
>=	**relop**	*GE*
<=	**relop**	*LE*
<>	**relop**	*NE*
=	**relop**	*EQ*

additionally place $*$ near the accepting state. One state in the transition diagram is called start state, and labeled by *start*, with entering edge, which is not leaving from any state.

Example 5.4 *Transition diagram for relational operators* $>, >=, <, <=$.

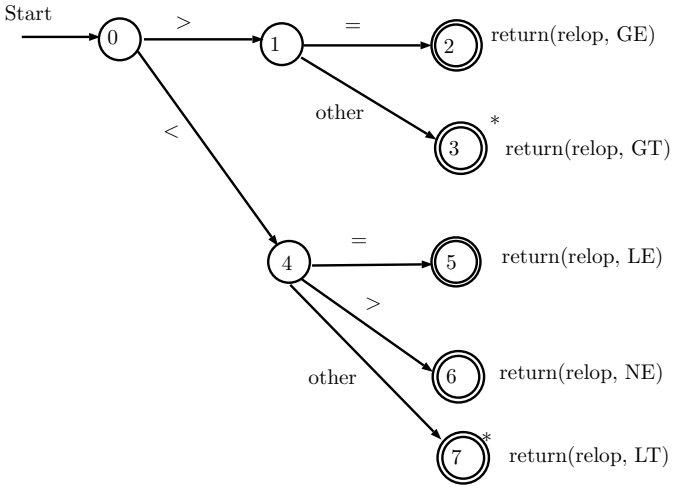

Figure 5.3: Transition diagram for *relop*

We consider the transition diagrams, for limited *relop* operators, as shown in Fig. 5.3. The others can be similarly represented. Note the difference between transition diagram and finite automata. In the FA there is "other" while in the transition diagram it is not. For details, see the Fig. 5.4.

5.5.2 Recognition of reserved words and identifiers

Usually the keywords *if, then* are reserved words, so they cannot be identifiers, though they may look like identifiers. However, it is difficult to reject them when used as identifiers,

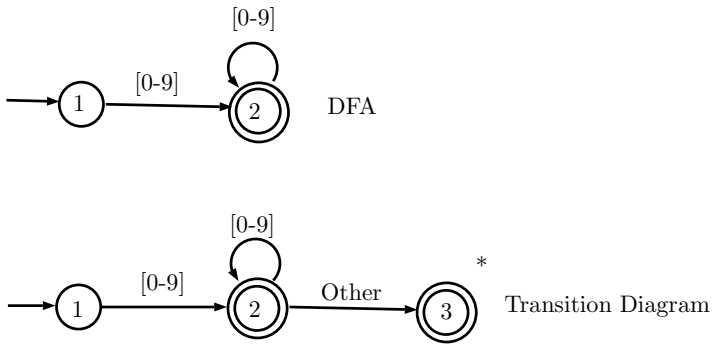

Figure 5.4: Transition diagram vs. FA

because identifiers are also the sequence of alphabets. However, the recognition of identifiers is strait forward. The Fig. 5.5 shows a transition diagram to recognize the identifiers.

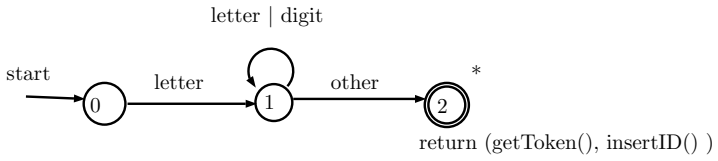

Figure 5.5: Transition diagram for ids

There are two approaches to handle reserved words:

1. One way is to install the reserved words in the symbol table initially. A tag for this in the symbol table will indicate that these are not ordinary tokens, and also tells that these are reserved words. This approach is applicable in Fig. 5.5. When an identifier is found using the transition diagram algorithm[3], a function call to $installID()$ places the token in the symbol table if it is already not there. A pointer is returned that points to the entry found. If it is not in the symbol table, its token **id** is returned by the $getToken()$, else the keyword is returned by it.

2. An alternative approach is to create separate transition diagram for each keyword, as shown in Fig. 5.6. The transition diagram shows transitions for each character in the lexeme. When "other" character (non letter and digit) is found, i.e. any character that cannot be continuation of an identifier, the **else** is returned. If "other" is any character, that makes the word as **else val**, then correct token is **id**, and same is returned. When this process is adopted, the reserved words need to be prioritized as keywords and not as **id**.

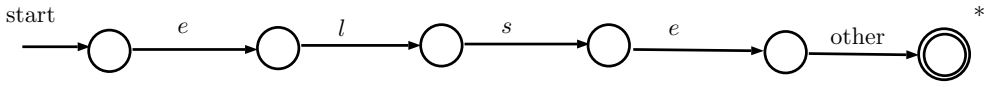

Figure 5.6: Transition diagram for the keyword "else"

The removal of whitespace (to skip it) is necessary for recognition of tokens. The transition diagram in Fig. 5.7 skips the whitespace characters, where *delim* is space or newline or tab

[3]The transition diagram in Fig. 5.5 shows scanning of a letter followed by zero or more number of letter/digit sequences. When a non-digit and non-letter symbol is encountered, the identifier is accepted. This is the description of an algorithm.

character.

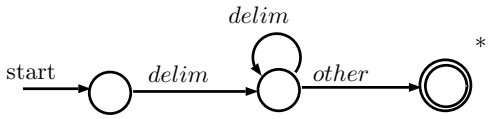

Figure 5.7: Transition diagram for skipping whitespace

There are several ways of using transition diagrams to build lexical analyser. Regardless of other strategy used, each state is represented by a piece of code. We may imagine a variable named as **state** holds the number of the current state for a transition diagram. Now, a switch statement may take us to the code based on value of that variable (i.e., state).

Disadvantages of Lexical Analyser Inspite of number of interesting fact and the benefits of lexical analyser, it has some disadvantgaes also. Here are the following:

1. Significant amount of time is consued in reading the source program and partitioning it in the form of tokens.
2. More difficult to develop and debug the lexer and its token dependencies.
3. Additional run time overhead is required to generate the lexer tables and construct tokens.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.