International Journal on Emerging Technologies (Special Issue NCETST-2017) **8**(1): 377-381(2017)
(Published by Research Trend, Website: www.researchtrend.net*)*

ISSN No. (Print) : 0975-8364
ISSN No. (Online) : 2249-3255

An Analysis of Compiler Design in Context of Lexical Analyzer

Praveen Saini and Renu Sharma
Department of Computer Science & Engineering,
Amrapali Institute of Technology& Sciences, Haldwani (U.K.)

ABSTRACT: In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands. A program for a computer must be built by combining these very simple commands into a program what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates a program written in a high-level programming language into the low-level machine language that is required by computers. On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well structured programs. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. A typical way of doing this is to split the compilation into several phases with well-defined interfaces. Conceptually, these phases operate in sequence. It is common to let each phase be handled by a separate module. We are presenting a review; of working of the very first and important phase of the compiler known as lexical analyzer. The lexical analysis programs written with Lex. Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. The final outcome of this paper is to present working of lexical analyzer in a simplest way to provide depth knowledge about lexical analyzer phase which is very crucial phase of the compiler design.

Index Terms- Lex, Yacc Parser, Parser-Lexer,

I. INTRODUCTION

A compiler is system software that converts a high-level programming language program into a target language equivalent to low-level (machine) language program. It validates the input program and shows the error message or warnings if there is any. Obviously it attempts to mark and detail the mistakes done by the programmer [1]. Many of the techniques used to construct a compiler are useful in a wide variety of applications involving symbolic data. In particular, every man-machine interface is a form of programming language and the handling of input involves these techniques.

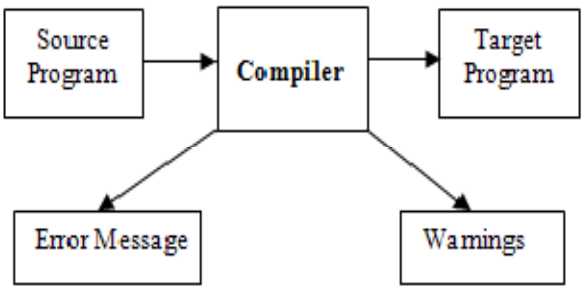

Fig. 1. Basic diagram of compiler.

The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. The very basic diagram for compiler is shown in figure 1.

II. PHASES OF GENERAL COMPILER

The compiler is made up of different modules or phases. Starting with token recognition, it runs through generation of context free grammar, parsing sequence, checking acceptability, machine independence intermediate code generation to finally target code generation state. These act as a basis for communication interface between user and processor [1, 3]. The first phase of the compiler is lexical analysis. The word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens. The main phases of a compiler include and undergo through Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Target Code Generation.

The various phases of the compiler is shown in figure 2.

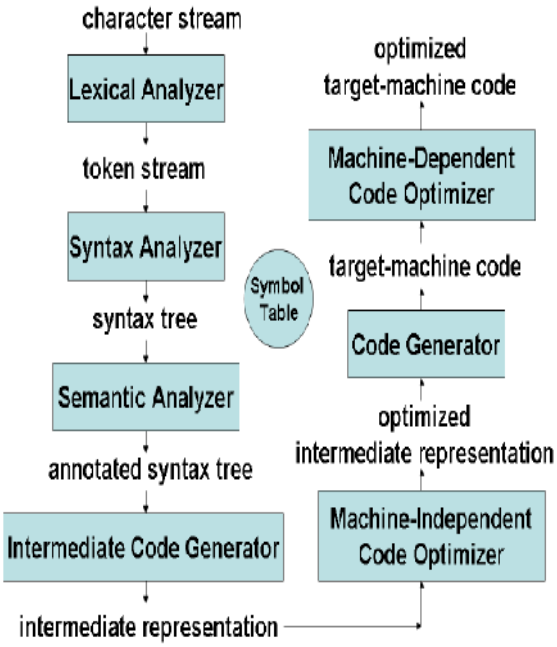

Fig. 2. Phases of compiler.

III. WORKING PRINCIPLE OF LEXICAL ANALYZER

A lexical analyzer or lexer for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines etc.) and comments. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. Lex and yacc were both developed at Bell.T. Laboratories in the 1970s. Yaccwas the first of the two, developed by Stephen C. Johnson. Lex was designed by Mike Lesk and Eric Schmidt to work with yacc. Both lex andyacc have been standard UNIX utilities since 7th Edition UNIX.

Lex takes raw input, which is a stream of characters and converts it into a stream of tokens, which are logical units, each representing one or more characters that belong together."

Typically,

1. Each keyword is a token, e.g, **then, begin, integer.**
2. Each identifier is a token, e.g., **a, zap.**
3. Each constant is a token, e.g., **123, 123.45, 1.2E3.**
4. Each sign is a token, e.g., **(, <, <=, +.**

A. Approaches to Building Lexical Analyzers

The lexical analyzer is the only phase that processes input character by character, so speed is critical. Either write it yourself; control your own input buffering, or use a tool that takes speculations of tokens, often in the regular expression notation, and produces for you a table-driven LA.

Lexical Analysis group the stream of refined input characters into tokens. Figure 3 describes this fact.

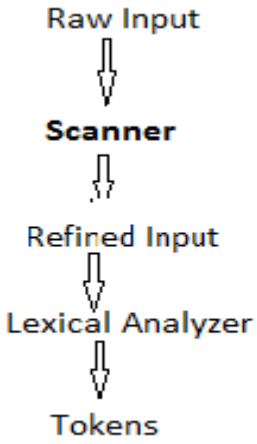

Fig. 3. Phases in Lexical Analysis.

A lexer (Lexical Analysis) has to distinguish between several different types of tokens, e.g.

numbers, variables and keywords. A lexer does not check if its entire input is included in the languages defined by the regular expressions. Instead, it has to cut the input into pieces (tokens), each of which is

included in one of the languages. If there are several ways to split the input into legal tokens, the lexer has to decide which of these it should use.

The simplest approach would be to generate a DFA for each token definition and apply the DFAs one at a time to the input. This can, however, be quite slow, so we will instead from the set of token definitions generate a single DFA that tests for

all the tokens simultaneously. This is not difficult to do: If the tokens are defined by regular expressions r1, r2. . . rn, then the regular expression r1| r2| . . .| rn describes the union of the languages r1; r2; : : : ; rn and the DFA constructed from this combined regular expression will scan for all token types at the same time. However, we also wish to distinguish between different token types, so we must be able to know which of the many tokens was recognized by the DFA [4].

B. Splitting the input stream

As mentioned, the lexer must cut the input into tokens. This may be done in several ways. For example, the string if17 can be split in many different ways:

- As one token, which is the variable name if17?
-As the variable name if1 followed by the number 7.
-As the keyword if followed by the number 17.
-As the keyword if followed by the numbers 1 & 7.
-As the variable name i followed by the variable name f17.
-And several more

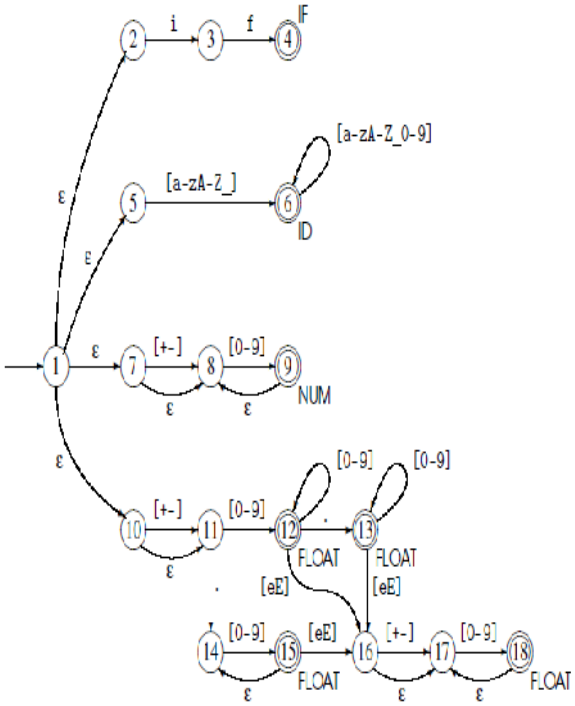

Fig. 4. Combined NFA for several tokens.

A common convention is that it is the longest prefix of the input that matches any token which will be chosen. Hence, the first of the above possible splitting of if17 will be chosen. Note that the principle of the longest match takes precedence over

the order of definition of tokens, so even though the string starts with the keyword if, which has higher priority than variable names, the variable name is chosen because it is longer.

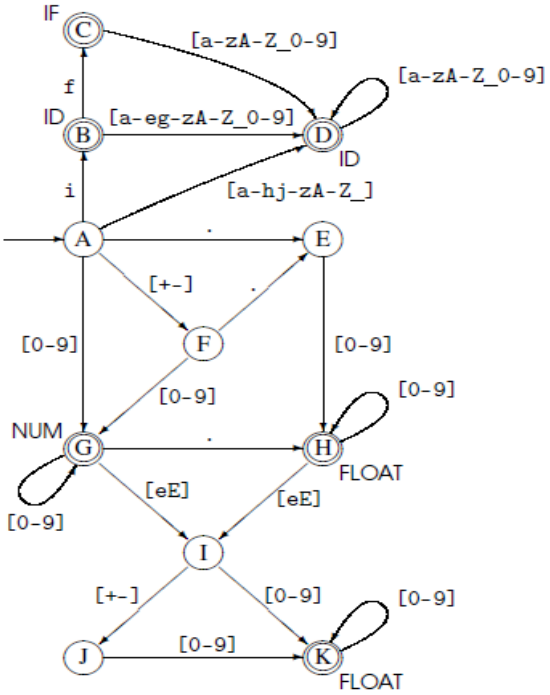

Fig. 5. Combine DFA for several tokens.

To illustrate the precedence rule, figure 4 shows an NFA made by combining NFAs for variable names, the keyword if, integers and floats. When a transition

is labeled by a set of characters, it is a shorthand for a set of transitions each labeled by a single character. The accepting states are labeled with token names

as described above. The corresponding minimized DFA is shown in figure 5. Note that state G is a combination of states 9 and 12 from the NFA, so it can accept both NUM and FLOAT, but since integers take priority over floats, we have marked G

with NUM only.

Let us discuss this with the help of an example, suppose the pseudo code:

```
if (x*y<10)
 {
Z = x;
}
```

Let's consider the first statement of the above code. The corresponding token stream of pairs <type, value>

is shown in Figure 6. Lex and input systems together constitute layers of Lexical Analyzer [5].

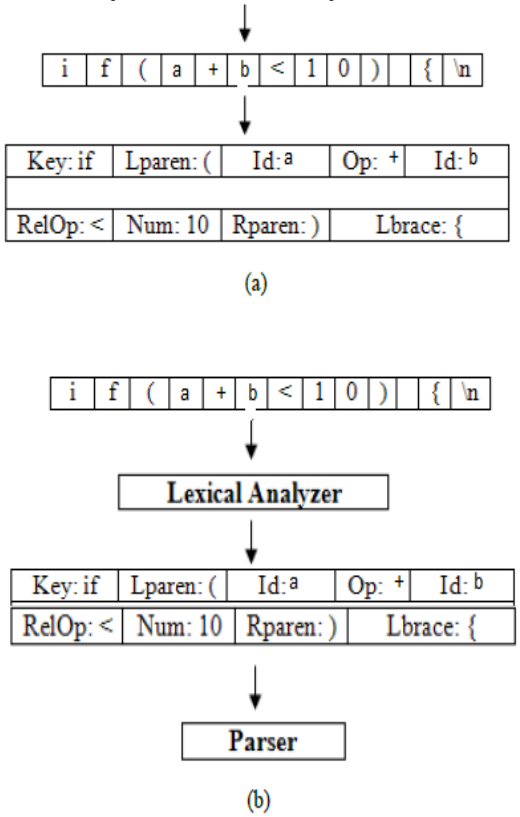

Fig. 6. Output Stage of Lexical Analyzer.

Sample program for lex

```
%%
      int k;
  -?[0-9]+   {
    k = atoi(yytext);
    printf("%d",
  k%7 == 0 ? k+3 : k);
        }
  ?[0-9.]+            ECHO;
  A-Za-z][A-Za-z0-9]+   ECHO;
```

The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7.

C. Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of **yyfex().** Not all tokens are of interest to the parser-in most programming languages the parser doesn't want to hear about comments and white space. The lexer and the parser have to agree what the token codes are.

D. Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line. Consider the following problem: copy the input to the output, changing the word magic to first on every line which began with the letter a, changing magic to second on every line which began with the letter b, and changing magic to third on every line which began with the letter c. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
int flag;
 %%
 ^a      {flag = 'a'; ECHO;}
 ^b      {flag = 'b'; ECHO;}
 ^c      {flag = 'c'; ECHO;}
 \n      {flag =  0 ; ECHO;}
  magic   {
   switch (flag)
    {
case 'a': printf("first"); break;
case 'b': printf("second"); break;
case 'c': printf("third"); break;
default: ECHO; break;
     }
          }
```

E. Error Handling

The error handling in the Lexer is basically concerned with the errors in the compiler or its environment, design errors in the program being compiled, an incomplete understanding of the source language, transcription errors, incorrect data, etc. The tasks of the error handling process are to detect each error, report it to the user, and possibly make some repair to allow

processing to continue. It cannot generally determine the cause of the error, but can only diagnose the visible symptoms. Similarly, any repair cannot be considered a correction (in the sense that it carries out the user's intent); it merely neutralizes the symptom so that processing may continue. The purpose of error handling is to aid the programmer by highlighting inconsistencies. It has a low frequency in comparison with other compiler tasks, and hence the time required to complete it is largely irrelevant, but it cannot be regarded as an 'add-on' feature of a compiler. We distinguish between the actual error and its symptoms. The diagnosis always involves some uncertainty, so we may choose simply to report the symptoms with no further attempt at diagnosis. Thus the word 'error' is often used when 'symptom' would be more appropriate. A simple example of the symptom/error distinction is the use of an undeclared identified LAX. The use is only a symptom, and could have arisen in several ways:

- The identifier was misspelled on this use.
- The declaration was misspelled or omitted.
- The syntactic structure has been corrupted, causing this use to fall outside of the scope of the declaration.

Most compilers simply report the symptom and let the user perform the diagnosis. An error is detectable if and only if it results in a symptom that violates the definition of the language. This means that the error handling procedure is dependent upon the language definition, but independent of the particular source program being analyzed. For example, the spelling errors in an identifier will be detectable in LAX (provided that they do not result in another declared identifier) but not in FORTRAN, which will simply treat the misspelling as a new implicit declaration.

We shall use the term anomaly to denote something that appears suspicious, but that we cannot be certain is an error. Anomalies cannot be derived mechanically from the language definition, but require some exercise of judgment on the part of the implementers. As experience is gained with users of a particular language, one can spot frequently-occurring errors and report them as anomalies before their symptoms arise.

IV. CONCLUSION

This paper outlines a novel approach to lexical phase in compiler construction. Furthermore, expressiveness is barely sacrificed; the compiler can be boot strapped provided there is enough run-time support. In spite of the scope of data storage is limited and symbols used are a few, the main aim has been just cleared conception and application of efficient look up table approach in finite states generation for lexical analysis. The next phase of compilation is just introduced to represent its utility, for the sake of completion and better understanding. Further study on extending this model with parser generation to generate language constructs as well as error recovery in lexical analysis is in progress.

The compiler has been used, for example, to study advanced topics such as the implementation of first-class continuations and register allocation.

REFERENCES

[1]. Alfred V.Aho, Ravi Sethi, Jeffery D. Ullman, Addison-Wesley, 2007. Compilers- Principles, Techniques, and Tools.

[2]. Torben Ægidius Mogensen, May 28, 2009. Basics of Compiler Design", lulu, Extended Edition.

[3]. David Galles, 2005. Modern Compiler Design, Addison-Wesley.

[4]. Torben Ægidius Mogensen, May 28, 2009. Basics of Compiler Design", lulu, Extended Edition

[5]. International Journal of Computer Applications (0975 – 8887) Volume 6– No.11, September 2010

[6]. William M. Waite Department of Electrical Engineering University of Colorado Boulder, Colorado 80309USAemail: William.Waite@colorado.edu.

[7] Aho, Alfred V. and Ullman, Jeffrey D. [1972]. The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Englewood Cliffs.

[8] Aho, Alfred V. and Ullman, Jeffrey D. [1977]. Principles of Compiler Design. Addision.

[9] Ross, D. T. [1967]. The AED free storage package. *Communications of the ACM*, **10**(8):481492.

[10] Rutishauser, H. [1952]. Automatische Rechenplanfertigung bei Programm-gesteuerten

[11] Rechenmaschinen. Mitteilungen aus dem Institut f•ur Angewandte Mathematik der ETHZurich, 3.

[12] Sale, Arthur H. J. [1971]. The classification of FORTRAN statements. *Computer Journal*,**14**:1012.

[13] Sale, Arthur H. J. [1977]. Comments on 'report on the programming language Euclid'. *ACM SIGPLAN Notices*, **12**(4):10.

[14] Sale, Arthur H. J. [1979]. A note on scope, one-pass compilers, and Pascal. Pascal News, **15**: 6263.