A New Approach of Complier Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages

Biswajit Bhowmik[1]
MIACSIT, MIAENG, MPASS, MIAOE

Abhishek Kumar[2]
Abhishek Kumar Jha[2]
Rajesh Kumar Agrawal[2]

ABSTRACT
A compiler translates and/or compiles a program written in a suitable source language into an equivalent target language through a number of stages. Starting with recognition of token through target code generation provide a basis for communication interface between a user and a processor in significant amount of time. A new approach GLAP model for design and time complexity analysis of lexical analyzer is proposed in this paper. In the model different steps of tokenizer (generation of tokens) through lexemes, and better input system implementation have been introduced. Disk access and state machine driven Lex are also reflected in the model towards its complete utility. The model also introduces generation of parser. Implementation of symbol table and its interface using stack is another innovation of the model in acceptance with both theoretically and in implementation widely.

General Terms
Compiler, Language Processor.

Keywords
Tokens, Lexeme, Lex, Tokenizer, PDA, Lookahead, Pushback.

1. INTRODUCTION
A compiler is system software that converts a high-level programming language program into an equivalent low-level (machine) language program. It validates the input program conforming the source language specification and violation of the same is stipulated as error message or warnings. Obviously it attempts to mark and detail the mistakes done by the programmer [1]. The idea is shown in Figure 1.

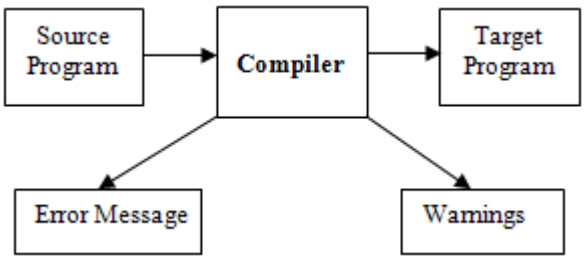

Figure 1: Working Methodology of a Compiler

Beginning 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 tool usually used to construct lexical analyzer is Lex. Syntax directed translation is achieved through parser. A data structure, symbol table interacts with different phases. For parser part the codes are directed towards implementing attributed grammars in PDA, top-down parsing [2].

2. PHASES OF GENERAL COMPILER
Writing a compiler is a nontrivial task. It will be a very nice practice to structure its principles. Conceptually a compiler works in phases. The key phases include and undergo through Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Target Code Generation [2]. These are shown in Figure 2.

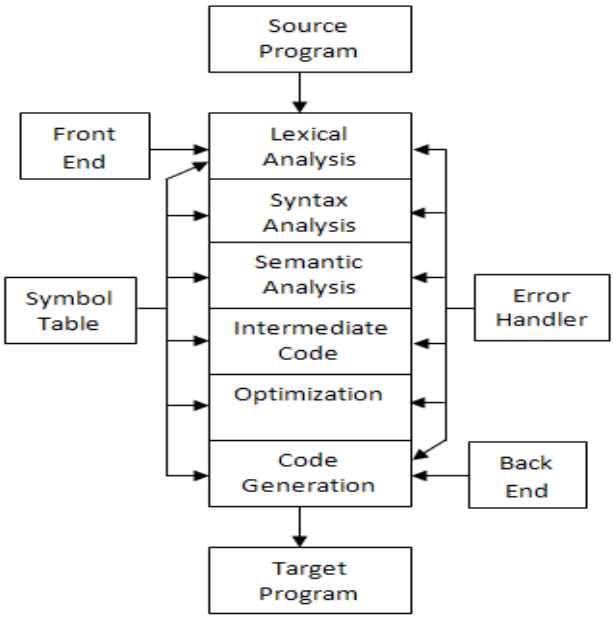

Figure 2: Phases of a Typical Compiler

3. GLAP Model
Lexical analysis, parsing, and symbol tables are those implementation techniques that support general design principles. We also suppose that a modern compiler construction till now is

[1] Sr. Lecturer, Department of Computer Science & Engineering, Bengal College of Engineering & Technology, Durgapur, 713212, India.

[2] UG Students, Department of Computer Science & Engineering, Bengal College of Engineering & Technology, Durgapur, 713212, India.

at an abstract level [8]. In a lexical analyzer, the scanning of a whole symbol table for a finite input string incurs very high computational cost. We present a lexical analyzer designed to focus on a very limited sub-set of the whole dictionary in least cost [5]. The model proposed describes the following:

i. Working Principle of Lexical analyzer.

ii. Input Systems.

iii. Optimization Issues

iv. Look-ahead and Pushback.

v. State Machine Implementation.

vi. Proposed Algorithms.

vii. Performance Analysis.

viii. Parser Generator.

3.1 Working Principle of Lexical Analyzer

A lexical analyzer (also known as lexer), a pattern recognition engine takes a string of individual letters as its input and divides it into tokens [1]. Additionally, it also filters out whatever (usually white-space, newlines, comments, etc) separates the tokens. The main purpose of this phase is to make the subsequent phase easier [2].

Some key definitions [4, 5] related to this phase include:

- Lex: A set of buffered input routines and constructs. It translates regular expression into lexical analyzer.
- Tokens: Basic indivisible lexical unit or language elements. These are terminal symbols in a grammar, Constants, Operators, Punctuation, Keywords, Classes of sequences of characters with collective meaning, arbitrary integer values, etc that represent the lexemes [1].
- Lexeme: These are original string (character sequence) comprised (matched) by an instance of the token. E.g. "sqrt" [6].

Lexical Analyzer is basically a part of compiler which:

i. Translates lexemes into tokens (arranged in symbol table for compilation references) with the help of Lex [13].

ii. Communicates with parser for serving token requests.

iii. Discards comments and skips over white spaces.

iv. Keeps track of current line number so that parser can detect errors [8].

Working methodology of lexical analyzer has been traced in some interesting phases as stated below:

First, it acts as an interface for parser and symbol table with input stream as reference as shown in Figure 3.

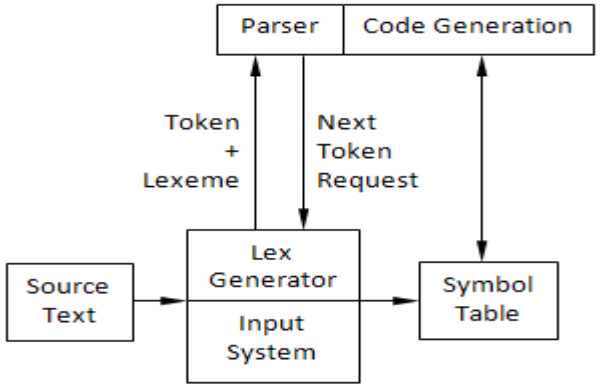

Figure 3: Lex as a Tokenizer

Second, internal working procedure of Lexical Analyzer that generates a stream of tokens. Suppose the pseudocode:

```
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 4. Lex and input systems together constitute layers of Lexical Analyzer.

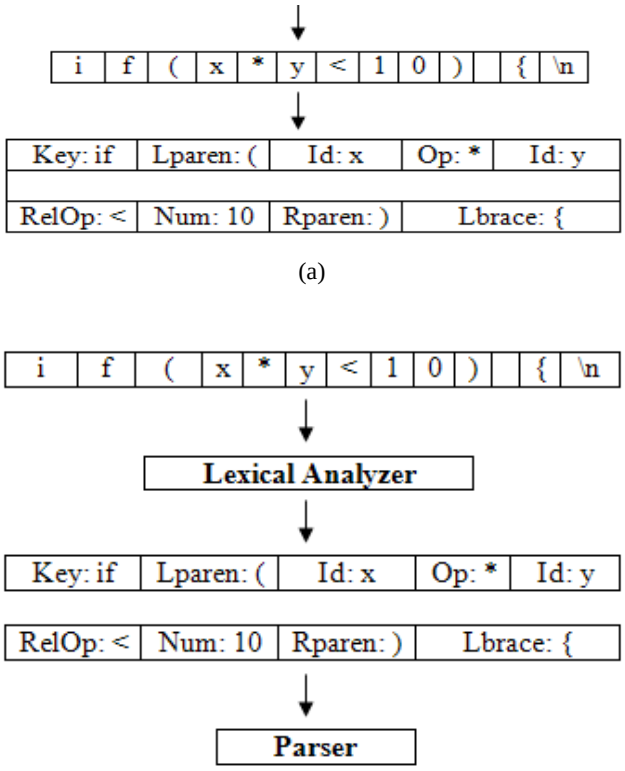

Figure 4: Output Stage of Lexical Analyzer

3.2 Input System

Input System is the lowest-level layer of the lexical analyzer which consists of group of functions that actually read data from the operating system. This is the reason that refers the lexical analyzer as a distinct and independent module [4]. This independency may derive several advantages such as:

- Change in the phase doesn't affect compiler as a whole.
- Enhanced portability in absence of inter and intra dependability.
- Efficient speed to read large data. Thus, optimized read time.
- Code recycling supported is same for every compilation utility.

An input system must possess the following design criteria [6].

- Efficient disk access.
- Supported reasonable lexemes of finite length.
- Availability of both the current and previous lexemes.
- Availability of pushback and lookahead of several characters.
- Faster routines as possible, with little or no copying of the input strings.

3.3 Optimization Issues

A Lexical analyzer consumes a good portion of compilers time since the number, size, and complexity of software systems are increasingly in nature. Moreover, programming languages, which are likely to make the programming task easier, are still frequent to error prawn. This is because of either the new languages' features do not exclude all the causes of errors or C, C++, etc like some old languages are in use [4, 7, 9].

E.g. standard C buffered input system is actually a poor choice as it copies the input characters thrice. From Disk to two buffers and then to the string that holds the lexeme [3, 6]. Consequently it is worthwhile to optimize the input systems.

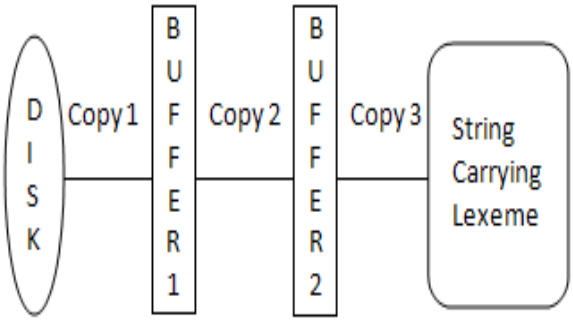

Figure 5: Multistage Copying Problem

3.4 Lookahead and Pushback

A Lexical Analyzer looks ahead several characters in input to distinguish a token from other and then it must push extra characters back to input. These functionalities can be described in accordance to input system reference through a problem statement and its solution flowchart [1, 9] as shown in Figures 6 and 7.

Available tokens:

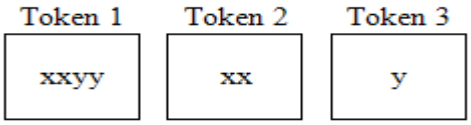

Figure 6: Some available tokens

Given Input: xxy

The problem statement: generate tokens for the above input string.

The solution flowchart for the generation of suitable tokens deals with only steps towards solution or marked with circle otherwise.

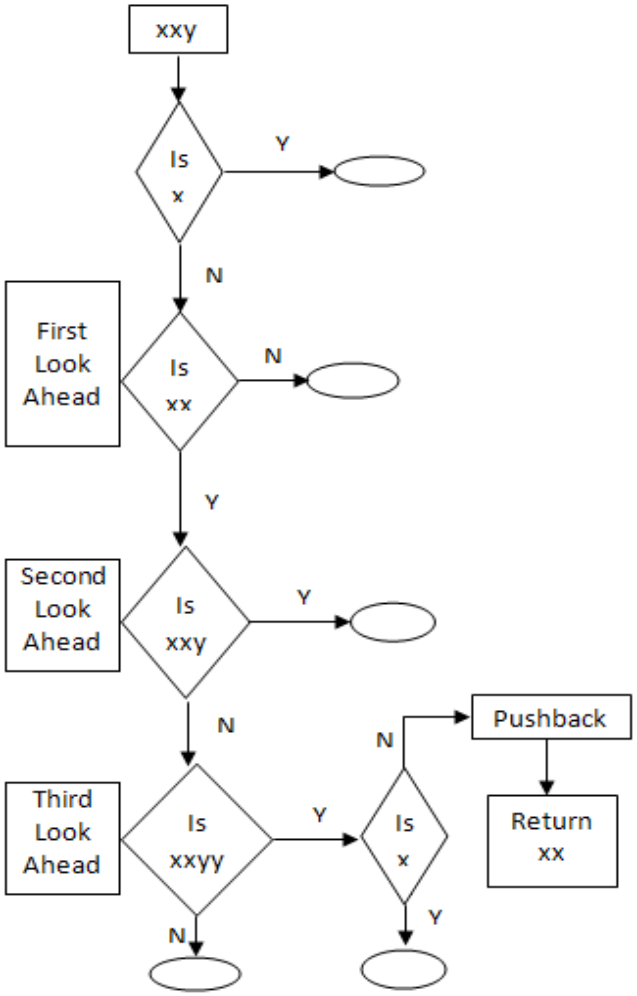

Figure 7: Flowchart Showing Multi-character Lookahead and Pushback

Thus the solution steps can be traced with carry ahead and pushbacks. As clear from the flowchart is that lookahead is not any big issue for input system [1]. But the major problem is pushback, more specifically multiple character pushback. It can be fixed fix by adding a layer around (e.g getc() if implemented in C language) that gives more pushbacks using a stack [5, 8].

3.5 Lexical analysis through FSM

The Efficiency of a Lexical Analyzer can be improved through:

a. A set of developed input routines which have been applied in lexical analysis applications in two primary approaches:
   - Hard code the analyzer that identifies lexemes with nested if/else statements, switches and so forth.
   - A series of look up tables to recognize tokens if lexemes are small enough.

b. Lex (uses approach of finite state machine) which takes a set of regular expressions to describe tokens, and create DFA or NFA [11] that recognizes the expressions and finds the longest possible sequence of input characters forming tokens. Thus, a Lexfile (a text file for token description) consists of regular expression / action pairs, where actions are represented by blocks of C code. The same functionality can also be provided by greedy algorithm [10].

3.6 Algorithm for Lexical Analyzer

Building a Lexical Analyzer needs a language that must describe the tokens, token codes, and token classification. It also needs to design a suitable algorithm to be implemented in program that can translate the language into a working lexical analyzer. We have used C language in particular, for implementation as powerful tool enough to describe the metasymbols used in regular expressions, as well as non-printable ASCII characters [9]. We have also described a shorthand notation for the range of ASCII characters, e.g. all lower-case letters [4]. The algorithm designed for the proposed GLAP model for the generation of tokens is named as Tokenizer and is written below.

Algorithm: Tokenizer(S)

Where S = Input string.

Output: A set of tokens

Step 1: Initialize S.

Step 2: Define symbol table.

Step 3: Repeat while scanning (left to right) S is not completed

i. If blank (empty space)
   a. Neglect and eliminate it.

ii. If operator op // arithmetic, relational, etc.
   a. Find its type.
   b. Write op.

iii. If keyword key // if, while, for, etc.
   a. Write key.

iv. If identifier id // a, b, c, etc
   a. Write id.

v. If special character sc // (,), etc.
   a. Write sc.

Step 4: Exit

Complexity Analysis: Initially, the input string is considered in an array. The already built symbol table is used here. Thus, the running time of the above algorithm will be the scanning of the string from left to right i.e. linear in nature.

3.7 Parser Generator

Parsing (also called syntax analysis) is another most important phase of a compiler. The parser (syntax analyzer) works hand-in-hand with pervious lexical analysis phase [1]. The lexical analyzer determines the tokens occurring next in the input stream and returns the same to the parser when asked for. The parser considers the sequence of tokens for possible valid constructs of the programming language [4, 9]. Role of a typical parser is two-fold:

a. Identify the language constructs from a given input. A parser outputs and represents valid input in the form of a parse tree [3].

b. For grammatically incorrect input string, the parser declares the detection of syntax error. No parse tree [7] in this case is generated.

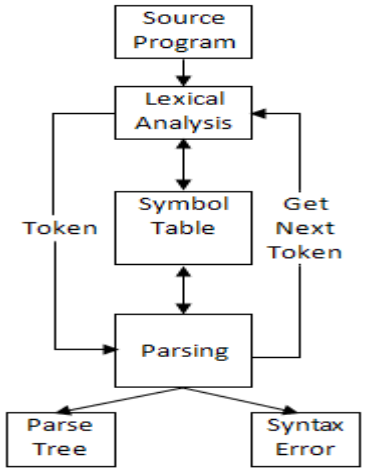

Figure 8: Working Principle of parser

As an alternative to parsing, based on the repetitive application of regular expressions using a shortest-match strategy, we may apply an iterative lexical technique. The approach in general recognizes syntactic elements using iterative sophistication, where constructs those are unambiguous are identified to provide related reminder for the identification of more ambiguous constructs [1, 6].

3.8 Performance Analysis

The efficiency measurement of a compiler is basically a tradeoff between complexity and time. Here, performance of the GLAP model is viewed w.r.t. to lexical analyzer.

Instead of using three levels of copying mutual data transfer a fragmented FILE buffer to serve this utility is used to some extent. The work becomes a bit easier as here the bulky symbol table has not been used. Thus, data chunks are lighter and access frequency is smaller and faster one.

With consideration of the design issues, the input system becomes efficient in speed factor and pushback as well as lookahead parts. The routines have been optimized to make them fast. However, the independent checking cannot be performed.

4. EXPERIMENTAL RESULTS

The above Tokenizer() algorithm has been implemented in Turbo C++ Version 3.0 and the input is simulated using both valid and invalid strings.

Observation 1:

Valid input: for(x1=0; x1<=10; x1++);

Output Analysis:

for : Keyword

(: Special character

x1 : Identifier

= : Assignment operator

0 : Constant

; : Special character

x1 : Identifier

<= : Relational operator

10 : Constant

; : Special character

x1 : Identifier

+ : Operator

+ : Operator

) : Special character

; : Special character

Tokens generated.

Observation 2:

Invalid input: for(x1=0; x1<=19x; x++);

Output Analysis:

for : Keyword

(: Special character

x1 : Identifier

= : Assignment operator

0 : Constant

; : Special character

x1 : Identifier

<= : Relational operator

Token cannot be generated.

5. CONCLUSION

The novelty of this model not only provides the variation of the existing lexical analyzer but also reduces the computational cost to a large extent. Furthermore, the nature of the string whether eligible for generating tokens could be analyzed by proposed algorithm with satisfactory results. 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.

6. ACKNOWLEDGMENTS

Our special thanks to Mr. Parag Kumar Guha Thakurta, Asst. Prof., NIT Durgapur, Dr. Subrata Trivedi, Asst. Prof. and Mr. Jishnu Mondal, Asst. Librarian, BCET Durgapur for their valuable suggestions and co operations towards development of the paper.

7. 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] William A. Barrett, 2005. Compiler Design, CmpE 152, FALL Version, San Jose State University.

[5] G.M6nier, G.Lorette, 1997. Lexical Analyzer based on a Self-Organizing Feature Map, IEEE Xplore, 0-8186-7898-4.

[6] Anthony Cox, Charles Clarke, 2003. Syntactic Approximation Using Iterative Lexical Analysis, IWPC'03, IEEE Xplore,1092-813A8

[7] Davide Pozza, Riccardo Sisto, Luca Durante, Adriano Valenzano, 2006. Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software, IEEE Xplore, 0-7803-9575-1.

[8] William M. Waite, Assad Jarrahian, Michele H. Jackson, Amer Diwan, 2006. Design and Implementation of a Modern Compiler Course, ACM 1595930558/06/0006.

[9] Allen I. Holub, 2009. Compiler Design in C, Phi Learning.

[10] Santanu Chattophadhyay, 2005. Compiler Design, PH I.

[11] K.L.P. Mishra, N.Chandrasekharan, 2007. Theory of Computer Science, PHI, Third Edition.

[12] Lex table look up: www.gradsoft.kiev.ua

[13] Symbol table management: www.faculty.washington.edu