Mayabini – A Practical Implementation of a Bangla Compiler

A Thesis submitted

By

Istiyak Amin Santo (ID:CSE02107131)
Md. Maharaj Hossen (ID:CSE02107122)

Under the supervision of

Manoara Begum
Assistant Professor

Computer Science Department
Faculty of Science
Port City International University - Bangladesh

September 2023

Design and Implementation of a Bangla Compiler

A Thesis submitted to the Computer Science Department of the Science Faculty, Port City International University - Bangladesh (PCIU) in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Istiyak Amin Santo (ID:CSE02107131)
Md. Maharaj Hossen (ID:CSE02107122)

Computer Science Department
Faculty of Science
Port City International University - Bangladesh

Spring Semester

September 2023

Declaration

This is to certify that this project is our original work. No part of this work has been submitted elsewhere partially or fully for the award of any other degree or diploma. Any material reproduced in this project has been properly acknowledged.

Students' names and Signatures.

1. _____
Istiyak Amin Santo

2. _____
Md. Maharaj Hossen

Approval

The Thesis titled “Design and Implementation a Bangla Compiler” has been submitted to the following respected members of the Board of Examiners of the Faculty of Science in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science on 16 September 2023 by the following students and has been accepted as satisfactory.

1. Istiyak Amin Santo (ID:CSE02107131)
2. Md. Maharaj Hossen (ID:CSE02107122)

_____
(Signature of the supervisor)

Manoara Begum
Assistant Professor,
Department of Computer Science and Engineering
Port City International University
Email: manoara.cse34@gmail.com
Telephone: 01845110925

Acknowledgements

"Design and Implementation of a Bangla Compiler" – is the job for which our effort was up to the boundary of our ability. But at last we could do it. At this precious moment we should not forget those who were behind us whole the way.

First of all, we would like to thank our honorable supervisor **Manoara Begum,** Assistant Professor, Department of Computer Science, Port City International University-Bangladesh (PCIU), for his knowledgeable suggestions and skillful instructions about our thesis. In fact, without his proper help and supervision we may not be able to reach our goal in time.

Moreover, our gratefulness to our respectable parents, our family and other teachers knows no bound, who not only patiently listened to us but also helped us to complete our project properly within due time.

Last but not least, our cordial thanks go to all martyr who have died at language movement. Because only for them we get Bangla and Bangla is the heart of our work. Hats off to all of them.

Contents

Abstract

It is evident that the familiar programming environment accelerates the efficiency of the programmer. If a programmer has a chance to write a program in his/her native language then undoubtedly it will enhance the overall performance of programming. Keeping in this mind, this project tries to design and implement a Bangla compiler. This compiler has been named as *Mayabini*. The whole project may be divided into two phases. The first phase is the designing of a high-level Bangla programming language. The implemented language follows the top down approach of popular structured programming languages. Many basic features of imperative programming languages such as: loop, block structure,conditional branching etc have been included in *Mayabini.*

In the second phase the compiler has been designed and implemented. This part actually includes lexical, syntax and semantics issues of the language. Finally, it deals with generating assembly code. The final job is to execute the file by creating executable file.

However, as this is the first step of our venture, many complexities of the language have been avoided. But the major features of any contemporary languages have been included in our implemented compiler *Mayabini.*

Chapter 1

Introduction

1.1 Introduction

Before we begin our exposition of the designing of a Bangla Compiler, we must consider a few preliminaries. In this chapter the reasons behind adopting this project and the underlying issues of designing a Bangla compiler has been discussed. Then a brief review on previous works on compiler in Bangla has been provided. Finally, the structure of our thesis document has briefly described. This work includes both design and implementation issues of a compiler. So, this thesis title is “Design and Implementation of a Bangla Compiler.”

1.2 Rationale

Survival of the fittest - the best example of the best survived species in this earth is human beings. In fact human are the only species that managed to survived from the beginning of creation to now, and doubtlessly they will survive till doomsday. One of the most important natures of human that help them to survive is that they could learn themselves, they could learn from their environment, they can store their learning in their mind; and they can process it to gain more information. Human is the species that could store their learning for next generation. For collect and process their information throughout generations human has been inventing many tools and techniques. Among the different tools invented by human computer is one of the best.

When people and computer have started working together, they have come up with a united force, which leads to change the human civilization. Computer is distinct from other tools in that they could be instructed by a man to perform task. The underlying CPU (Central Processing Unit) can recognize a set of instructions that form the machine language of computer. Generally instruction set for machine language consists of binary codes. The early days of computer programming, people used to do give machine

instruction to PC for process. Suppose instruction 1011001 might represent arithmetic addition operation.

But very soon they do realize that giving instruction in binary code is really critical. Necessity is the mother of invention – the human then come up with the logic of assembly language which is very near to machine instruction but can be written into natural English language. Such as: ADD, SUB etc. Assembly language can be said the first layer of in human computer interaction. Using assembly language is more understandable and thus more manageable. A tool was designed that translates the assembly language into machine language known as assembler.

Human never be satisfied with the same thing for long time – this is the one major key of human success over other species. So, it is very common that human get tired with assembly language very soon. Assembly language still requires knowledge about machine hardware. But users want to instruct machine in a more lucid way. They also want that the instructing language should be like their natural language. As a result of this demand High-Level language with instructions that closely resembles human language and mathematical notation appeared in the market. Like assembly language a High-Level language also needs to be translated to machine readable format. Two types of tool are used for this purpose Compiler and Interpreter or Hybrid Interpreter.

A compiler directly translates the source program to machine language. This method has the advantage of very fast program execution, once the translation process is complete. [11]. Such as: C, COBOL, Ada, Pascal etc.

An interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high level language program rather than machine language. Then, the high level language translates source code to machine language. Though it is easy to implement but 10 to 100 times slower than compiler [11]**.** Such as: APL, LISP, UNIX shell script and DOS batch files.

A hybrid interpreter translates source code to an intermediate language which is easier to interpret. It is quit faster than pure interpreter [11]**.** Such as: Perl.

High level language gets tremendous popularity very soon. But as human nature they began to yawn at high level language very soon. They want something more robust. As a result fourth-generation languages appeared. The object oriented programming approach has been introduced and getting popularity.

Whatever, the popularity of many programming languages nowadays, most of the languages are written in English. So people need to learn English to get in the computer. If we think in the context of Bangladesh it is very common scenario that most of our people are illiterate and the literate people fell less comfort in English. This makes the major issue on design a Compiler of our own language. So that people can get closer to computer. If it is possible to design a very good compiler it is also possible to write operating system in our mother language using our own programming language.

At present there are many organizations that are dealing with Bangla operating system. It has become easier after the arrival of open source operating system such as LINUX. Among some popular organization BIOS [30]**,** *Mayabini* [31] in Bangladesh and Ankur
[32] in west Bengal are to be mentioned. They have designed some operating system and open source Bangla software. But none of them have yet do any work on Compiler. This is the reason that we have decided to design a Bangla Compiler "*Mayabini*".

1.3 Objectives of this Work

We have chosen to design and implement a Bangla programming language as our thesis. The major reasons for our choice are follows:

i. As a student of computer science we have done many course related to programming languages. Moreover we have done a course named compiler design. In fact, at that moment we were so much amazed about how a compiler

works. So, we have designed this Bangla Compiler to learn the zest of programming language as well as a compiler.

ii. We have designed a Bangla compiler. But there may be question why a Bangla compiler. Our intention was to develop such a project that can reach to everybody of our country. Where the literacy rate is too low in our country and many of the literate people are not too much comfortable with English, so we have decided to do our project related to Bangla language, our mother tongue.

iii. Nowadays, we are so much worried about to reach the facilities of IT to every door of Bangladesh. We believe that, only mother tongue can be the weapon to reach every door of the country. So, we have switched to do something in Bangla.

iv. Designing a good and fully functional Bangla compiler may lead us to design a full-fledged Bangla operating system.

Still now there is no academic work on Bangla compiler. The scope of working such a topic is not too much beneficiary in our country as far as monitorial is concerned. We have to do it within a very small context and very few support from other people. But we believe someone have to start it otherwise it will never be developed. We expect after the first release people feel interest on it and understand the reality of a Bangla programming language. As far as we have decided it to be open source anybody can make any development on it.

1.4 Background Studies

Reportedly no academic work on the design and implementation of Bangle compiler has been done in Bangladesh so far. However one works on a Bangle Interpreter and two works on assemblers based on the 80x86 family have done. The first was done in Dhaka University. The first assembler was done in 1993 by S.M.S Zabir, M.O. Haider and A. Rahman all from Department of Computer Science and Engineering, BUET. Their assembler was not a full-fledged one but the approach they made was generalized and hence it could be easily extended to support the full instruction set. The second work came in 1995 following this generalized approach where MD. R. Karim of Department of

Computer Science, Dhaka University, improved on the generalized algorithm. His assembler supports more 8086 instructions. Due to the flexible structure of the work some modification to the symbol table could bring another assembler of different microprocessor.

Now we will give our attention on the design and implementation of Bangla Interpreter. This was done in Department of Computer Science, Dhaka University at master's level thesis work by Mr. Naushad Jamil under the guidance of Sheikh Khaled Gafoor in 1996. Mr. Jamil took the first initiative to design a High Level Bangla Programming Language. He had got success in many cases. He designed a new language in Bangla, some keywords, primitive data types, different logical, relational and arithmetical operators for computations. He had also implemented some built-in functions and an interactive editor. First he had performed lexical analysis on the source program and made a symbol tables. Then he had built parser. After this, the source program had been transferred to C compiler for further computations. As an initiative work, he had done a lot of job.

Mr. Jamil had not designed any grammar for his Language though it is a necessary thing for every language. He had no precise syntax and semantics check and didn't generate any assembly code. Since, it is an interpreter so it is 10 to 100 times slower than in compiled system [11]. It is slow because decoding to C language is more complex than machine language instruction. Moreover, an interpreter takes more memory space than a compiler. Since the symbol table must be present during interpretation. In spite of some weaknesses, his approach was generalized and hence it could be easily extended to a more efficient interpreter.

In advanced countries especially in USA hundreds of papers are published every month processing new or improved algorithms and concepts on high level programming, compilers etc. Regarding programming languages the topics of interest are syntax-semantics, grammars, data types, operators, data abstraction etc. Papers on compiles/interpreters concentrate on symbol tables, parsing techniques, error checking,

debugging methods, linkers, preprocessors, incremental compiling and the more recent incremental linking.

The following sections will review some interesting works on programming languages. Most of these works will focus on improving specific areas of compiles/interpreters.

McCrosky and Sailor [26] published a paper in 1993 on **"**A synthesis of Type-checking and parsing**".** The context-free grammars and parsers conventionally used to define and interpret concrete syntax lead to constrained, more intuitive notations. These algorithms can be viewed as context-sensitive parser, where the context that is taken into account is the type structure of the expression or they can be viewed as type checkers that determine the form of the abstract syntax. The approach taken integrates parsing and type-checking in the formalism of Markov Algorithms. The interpretation of concrete syntax by these algorithms is unambiguous.

One year later McCrosky and Sailor [27] again came with a practical approach to type-sensitive parsing. Type sensitive parsing of expression is context-sensitive parsing based on type. Previous research reported a general class of algorithms for type-sensitive parsing. Unfortunately, these algorithms are impractical for language that infers type. This work describes a related algorithm which is more efficient- its incremental cost is linear (with a small constant) in the length of the expression, even when types must be deduced. Their method can be applied to any statistically typed language solving a variety of problems associated with conventional parsing techniques including problems with operator precedence and the interaction between infix operators and higher order functions.

A parser on the analyzability of pointer data structures in the design of programming languages was introduced by Hendren and Gao [29]. In this paper they proposed a programming language mechanism and associated compiler techniques which significantly enhance the analyzability of pointer-based data structures frequently used in non-scientific programs. Their approach was based on exploiting two important

properties of pointer data structures: *structured inductivity and speculative traversability*. Structural inductivity facilitates the application of a static interference analysis method for such pointer data structures based on path matrices, and speculative traversability is utilized by a novel loop unrolling technique for while loops that exploit fine-grain parallelism by speculative traversing such data structures. The effectiveness of this approach was demonstrated by applying it to a collection of loops found in typical non-scientific C programs.

A work of worth by Shaoying Liu[30] proposed an abstract programming language with correctness proofs. The realization of an abstract programming language is a good approach for automating the software production process and facilitating the correctness proof of a software system. This paper introduced a formal language for programming at the abstract level by combining Pascal with VDM (Vienna Development Method). The notation provided by the language obliges programmers to consider the correctness of programs throughout the whole process of programming and the proof axiom and rules presented in this paper may be used to prove the correctness of programs. A complete example was given to illustrate how to program using APL and how to prove the correctness of programs using the given axiom and rules.

An improved recursive ascent – descent parsing method has been found in the work of R.N.Horspool [31]. Generalized left corner parsing was originally presented as a technique for generating a parse for the SLR(1) class of grammars but with a far fewer states than the SLR(1) parser. This paper modifies and extends the formulation of left corner parsers so that it is possible to apply the technique to LALR(1) and LR(1) classes of grammars. It has been further shown that left-corner parsers can be converted into directly executed code in a manner that subsumes the parsing methods known as recursive descent and recursive ascent – hence the name recursive ascent-descent. The directly executed form has the advantage that it allows a compiler writer to insert semantic code into the parser incrementally, without having to re-execute the parser generator.

In 1992 Carvalho and Kowaltoski [32] came with a paper suggesting some new techniques to handle open arrays and variable number of parameter passing. Two facilities for programming language were described there: open arrays (an extension of Pascal conformant arrays) and automatic parameter conversion. As a result of combining these two mechanisms, it became possible to give compile time verifiable specification of procedures with a variable number of parameters and varying types. This ability is indeed very useful in many applications, and in particular in specifying I/O procedures within a programming language itself.

An exceptional design of a data control model in programming languages was placed in 1991 by M. J. Outshoorn and C.D. Marlin [33]. Their paper described a layered model of the semantics of the data control aspect of programming language; this aspect of programming language semantics concerns access to the data objects of the program. The model was an information structure model in which the information structures were defined in a relatively precise manner using algebraic specification techniques for abstract data types. The use of ADT (abstract data types) was also the key to the layering of the description: the outer most layers of the information structures used within the model, and the innermost layer contained precise descriptions of the information structures.

Since the mid 1970s and with the development of each new programming paradigm there has been an increasing interest in exceptions and the benefits of exception handling. With the move towards programming for even more complex architectures, understanding basic facilities such as exception handling as an aid to program reliability, robustness and comprehensibility has become much more important. Interest has sparked the production of many papers both theoretical and practical, each giving a view of exceptions and exception handling from a different standpoint. In an effort to provide a means of classifying exception handling models which may be encountered, taxonomy was presented in an outstanding paper of S. J. Drew and K.J. Gough [34] in 1994. As the taxonomy was developed some of the concepts of exception handling were introduced and discussed. The taxonomy was applied to a number of exception handling models in

some contemporary programming languages and some observation and conclusions were also offered.

An interesting work by Albert Nymeyer[35] has been published in 1995 which concerns to a grammatical specification of Human-computer dialogue. The Seeheim model of human-computer interaction partitions an interactive application into a user-interface, a dialogue controller is based on context free grammars and automata. In this work, the author modifies an off - the – shelf compiler generator (YACC) to generate the dialogue controller. The dialogue controller is then integrated into the popular X-WINDOW system, to create an interactive application generator. The actions of the user drive the automaton, which in turn controls the application.[36]

1.5 An Overview of the *Mayabini*

Mayabini is a fully functional compiler. It includes many of the features of a standard programming language. Though it has very few alternatives but it supports any kind of algorithm construct.

The design of *Mayabini* follows standard forward method of designing any compiler. It haslexical, syntax, semantic and target machine code generation phase. We have dropped only the intermediate code because it seems unnecessary to us. Two main reasons for dropping intermediate code are: firstly, we do not design a platform independent compiler and secondly, we have lake of time. But anybody can add the intermediate code phase if he wants and thus make the compiler platform independent.
We have designed our grammar in such a way that anybody can extend feature to it. Moreover, we have designed our own syntax analyzer. We do not use any parser to generate syntax tree. Moreover the semantic analyzer is also a pure implementation of DFA that checks the semantic of the language. We have followed Intel 8086 instruction set for generating target machine code.

Finally we want to give a briefing about the Bangla character encoding. We have followed Bijoy layout because it is the most popular layout. But anybody can use his own

layout if he choice without any hazard. We have used ASCII code for encode and decode Bangla to English.

1.6 Structure of the Thesis

This thesis paper is a self dependent document. People do not need too much interaction with other books to understand any technical term. Because most of the technical terms are clearly defined in this thesis document. The only thing they have to know is JAVA and assembly language because the compiler is written with Java and the target machine code is generated using Intel 8086 instruction set. Anybody can further develop the project with only the help of this report.

This thesis contains six chapters. The first chapter is introductory about the thesis. Here we have described the objective of the thesis, background studies and a brief overview of the thesis.

The second chapter includes all the design issues of a compiler. It includes the lexical phase, syntax analyzer, semantic analyzer and code generation phase. Algorithms that we have used have given here. Moreover, new algorithms that we have developed are also included here. We have tried to explain some critical features and few DFA also.

The third chapter includes some implementation details. Some critical codes are described here along with sample programs.

Chapter four describes about the text editor where user should compile and run his source code. The process of linking and loading is properly given here. Then we describe briefly about the tools that we have used on developing the language.
Chapter five is a brief overview of the language features comparatively to other popular languages. We have described here the features that we adopted from other and features that we have discarded.

The last chapter is the conclusion where we give suggestion for future development and discuss about our work, its strength and weakness.

In the appendix we have included all DFA that we have used for both lexical analyzer and semantic analyzer. The complete grammar is also given here. Some sample programs are given for user convenient. We have also included the Bijoy layout for new Bangla user.

1.7 Summary

Programming in Bangla is no more a dream, it is real. We believe that everybody can participate in further development of this compiler. This thesis provides enough resources about compiler study and implementation of a compiler. People do not need to be a computer scientist to participate in development of this compiler. In the following chapters each and every aspect of designing and implementing a Bangla compiler has described. This thesis not only helps to know about the development of our compiler but also helps to learn to design a new compiler.

Chapter 2

Design Issues

2.1 Introduction

This chapter focuses on all design issues that are relevant to compiler design. By reading this chapter, the reader will gain a comprehensive knowledge on compiler, language, grammar, parser, syntax and semantics etc. This will in turn enable him/her to understand the construction of the compiler. It also includes all the algorithms that have been followed to design the compiler *Mayabini*. It also describes the grammar of *Mayabini*.

2.2 Language

When we communicate with others we use some meaningful sound or symbol this is called language. Language can be divided into two types, natural and formal. “A natural language is any language used for general communication in a community, and often exists in both spoken and written forms [12]”, for example, Bangla or English language. They come naturally from a group of people. They come through a slow, unpredictable, and unplanned process. It is context oriented and ambiguous. In contrast “a formal language is created for intensive use with a specific precisely delineated set of purpose in view. Usually, it is created in written form only, all at once, either by a single person or by a group of people; it may be modified, but only by a conscious act. The rules by which a formal language is understood are precise, and are created as parts of the language itself.” [12], for example, Pascal, C/C++, and JAVA2. Formal languages are simpler and easier to understand than most natural languages. Most of them are context free and unambiguous. All over this book, language means formal language. Before examining the formal definition of language let’s look at some related topics.

2.2.1 Alphabets

To write a language we use some symbols which are fixed and have predefined meaning, this symbols are called alphabet. More exactly, “an alphabet is a finite, nonempty set of

characters [12]". Conventionally, we use the Greek symbol Σ for an alphabet [3]. For example

1. Σ = {a, b, c, … , y, z, A, B, …, Y, Z }, the set of English alphabet.
2. Σ = {0,1 } , the set of Binary alphabet.

a) Alphabets used in *Mayabini* Language:

Mayabini language contains the following alphabet

Σ = {w … } U { ○ … ²}

b) Punctuations:

In *Mayabini* ";" is used for indicating end of a simple statement. Parenthesis ((,)) are used for function. They are also used after ץ and ⅈא. Curly brackets ({,}) are used for compound statement and brackets ([,]) are used for array.

2.2.2 Word/String

Word is the smallest unit of a sentence which has some meaning. Formally, "A string (or sometimes word) is a finite sequence of symbols chosen from some alphabet [3]". For example

1. Bangladesh is a string from the above English alphabet.
2. ম া য় া ি ব নী is a string from above *Mayabini* alphabet.

The set of all strings over an alphabet Σ is conventionally denoted Σ*. For instance, {0, 1}* = {ε, 0, 1, 00, 01, 10, 11, 000, …} [3].

a) Empty String

An empty string is a string where the number of symbol is zero, i.e. zero occurrences of symbols. We denote this string by the Greek symbol ε (epsilon). This is such a string that may be chosen from any alphabet. It may seem that empty string is unnecessary. But this is very useful. It is an identity of concatenation operation.

2.2.3 Grammar

Simply grammar means a set of rules which helps us to read, to write, and to speak a language properly. But the grammar of computer or formal language indicates what sort of statements, expressions and other features will be supported by the language. It also ensures that a particular language obey the grammar. So unlike natural language, for every computer language, first the language developers construct grammars then the actual languages are constructed. Like the natural language each computer language has its own alphabets, vocabulary and punctuation. We used Context-Free Grammar (CFG) to describe our grammar. "It was first described independently by Chomsky [1956] and Backus [1960]. Backus's notation was immediately adopted for describing ALGOL 60, and has come to be called Backus-Naur Form (BNF)." [10].

2.2.4 A Formal Definition of Language

Now we will give a formal definition of language. Formally, "A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a language. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ" [3]. We can define a language through set. Such as

{ w | something about w}

This expression is read as "the set of words (string) w such that (whatever is said about w to the right of the vertical bar)" [3]. For examples:

1. { w | w is a odd number }
2. { w | w is a even number }

2.2.5 A Formal Definition of grammar

A grammar for a language is a systematic presentation of its syntax. Thus the grammatical rules are sufficient to determine whether any given string is a member of a language specified by the grammar. Normally formal/programming languages are defined by an inherently recursive structure called Context-Free Grammar (CFG). We can represent a formal grammar or Context-Free Grammar G by its four components, that is, G = { V, T, P, S }. Where

1. V is a finite set of variables, also called sometimes non terminals or syntactic categories. Each variable represents a language; *i.e*., a set of strings.
2. T is a finite set of symbols (alphabet) that form the strings of the language being defined. We call this alphabet the terminals, or terminals symbols.
3. P is a finite set of recursive productions or rules that represent the recursive definition of a language.
4. S is one of the variables represents the language being defined; it is called the start symbol. Conventionally if nothing is stated then the left side of the first production is the start symbol.

For example, if we want to construct a grammar which will only generate all strings of balanced parentheses then our grammar may be G = { V, T, P, *A* }.

Here,

Set of non terminals, V = {*A*}

Set of terminals T = { (,) }.

Set of production rule is, P, *e.g.* $A \rightarrow$ **(** *A* **)** *A* | ε

Starting symbol is *A*.

a) Grammar for *Mayabini* Language

If we represent *Mayabini* grammar by G_{bangla} then

G_{bangla} = { V_{bangla}, T_{bangla}, P_{bangla}, *prog*}

V_{bangla} = { *prog, decl, A, type, main_fun, fun,fh, comp_stmt, stmt_list,stmt, select_stmt, stmt', iter_stmt , as_fun_stmt ,B,Q,C, io_stmt,D,* j*ump_stmt,* e*xpr_list , expr_list'*, *M, M',N, N',O, O', expr, expr', term, term' , factor, P, uniop, addop, mulop, relop, eq, E* }

T_{bangla} = { **int , char, float, main, id, [, num,], (,), {, }, if, else, while, break, continue, ; ,= ,|| ,&& ,== ,!= ,> ,>= ,< ,<= ,+ ,- ,* ,/ ,% ,! ,printf, scanf, letter, str** }

P_{bangla} set of productions is given in appendix-A.

prog is starting symbol.

2.2.6 Derivation

Derivation means to derive a string from a grammar. We use the production rules from head to body. We expand the start symbol using one of its productions. We further expand the resulting string by replacing one of the variables by the body of one of its production, and so on, until we derive a string consisting entirely of terminals [3]. There are two types of derivation leftmost and rightmost.

a) Leftmost Derivation

If we replace the leftmost variable by one of the productions in each step then the derivation is called leftmost derivation. For example, for the above grammar leftmost derivation will be

A=>(A) A=>((A) A)A=>((ε) A)A=> (()(A) A)A=>(()(ε) A)A=>(()( ) ε)A

=>(( ) ( )) ε=>(( ) ( )) .

b) Rightmost Derivation

If we replace the rightmost variable by one of the productions in each step then this is called rightmost derivation, e.g., for the above grammar rightmost derivation will be

A=>(A)A=>(A) ε=>((A)A)=> ((A)(A)A)=> ((A)(A) ε)=> ((A) (ε))

=>((ε) ( )) =>(( ) ( )) .

2.2.7 Parse Tree

Parse tree shows the derivations pictorially. Let G = {*V, T, P, S*} is a grammar." The parse trees for G are trees with the following conditions:

1. Each interior node is labeled by a variable in *V*.
2. Each leaf is labeled by either a variable/terminal, or ε. However, if the leaf is labeled ε, then it must be the only child of its parent.
3. If an interior node is labeled *A*, and its children are labeled

   X1, X2, ..., Xn

respectively, from the left, then $A \rightarrow X1, X2, ..., Xn$ is a production in P."[3].

The corresponding parse tree of the above derivations is 2.1.

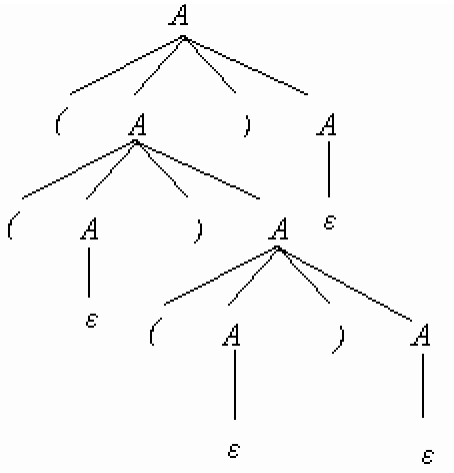


Figure 2.1: Parse Tree

If the leaves of a parse tree are concatenated from the left then we get a string which called the yield of the tree, *e.g.*, (( ) ( )) is yield of the above parse tree.

2.2.8 Parsing

Parsing means construct a parse tree. Using this parse tree we determine whether a string can be generated by a grammar. We can construct a parse tree in the following two ways:

a) Top-Down parsing

When we construct a parse tree expanding the root, then expand all the non terminals until we get the leaves. We use top-down parsing method for *Mayabini*.

b) Bottom-up parsing

When we construct a parse tree from bottom i.e. from leaf and get the root this parsing process is known as bottom-up parsing.

2.2.9 Ambiguity

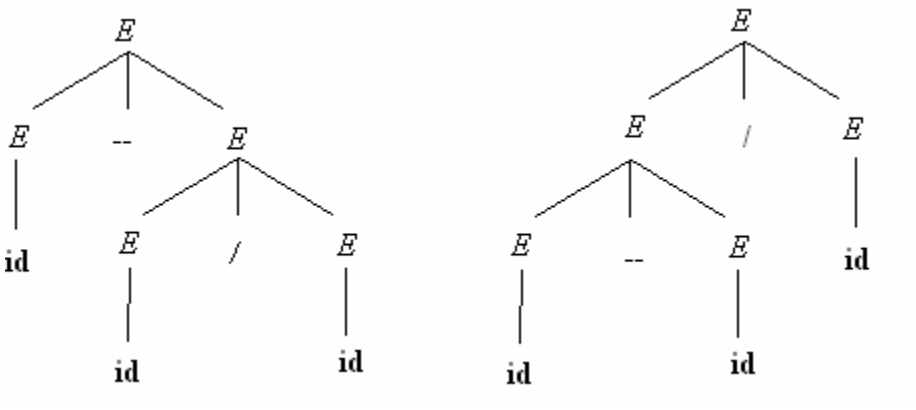
If a language has more than one meaning then it is called ambiguous language. Our natural languages are often ambiguous and context oriented, for example, "visiting relatives are interesting". One possible meaning may be relatives who are visiting us are interesting. Another meaning may be when we visit our relative this thing is interesting. If we command our computer by this type of sentence (ambiguous sentence) it may not respond properly. So we should construct such language which is free from ambiguity. Formally "A grammar that produces more than one parse tree for some sentence is said to be ambiguous."[1]. Consider the following grammar.

$E \rightarrow E\ E$

$E \rightarrow E/E$

$E \rightarrow$ **id**

It produces two different parse trees .2.2 (a) and 2.2 (b) for **id – id / id.**



(a) (b)

Figure 2.2: Two parse tree for **id – id / id.**

a) Causes of Ambiguity

1. If we don't respect precedence of operator then the grammar will be ambiguous. / (division) operator has precedence over – operator. We don't respect this precedence so our above grammar becomes ambiguous.
2. If identical operators are not group together then the grammar will be ambiguous. If the / operator is replaced by – then we get two different parse trees for the string *E - E - E* though the value will not be changed.

2.2.10 Removing Ambiguity

We must remove ambiguity from grammar otherwise we cannot produce unambiguous instructions for computer. We adopt the following techniques for removing ambiguity from our grammar.

a) Maintaining precedence

"The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of "binding strength." Specially:

1. A factor is an expression that cannot be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses.
2. A term is an expression that cannot be broken by the + operator.
3. An expression will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +."[3] for example

expr → *term* | *expr* + *term*
term → *factor* | *term* *
factorfactor→ **id** | (*expr*)

We have sent the highest precedence operator in the deepest of parse tree, *i.e.*, the highest the precedence the deepest in the parse tree. So our grammar is now in the following form:

expr_list → *M* **||** *M* |
MM→ *N* **&&** *N* | *N*
N → *O eq O*| *O*
O → *expr relop expr*| *expr*
expr → *uniop term addop term* |
termterm → *factor mulop factor*|
factor factor → **id** *P* | **num** | **(**
expr_lsit **)**
P→ [*expr_list*] |
ε*uniop* → + | **-** | **!**
| ε*addop* → + | -
mulop → * | **%** |
/
relop→ > | >= | < | <=
eq → = = | **!=**

b) Maintaining Associative rules

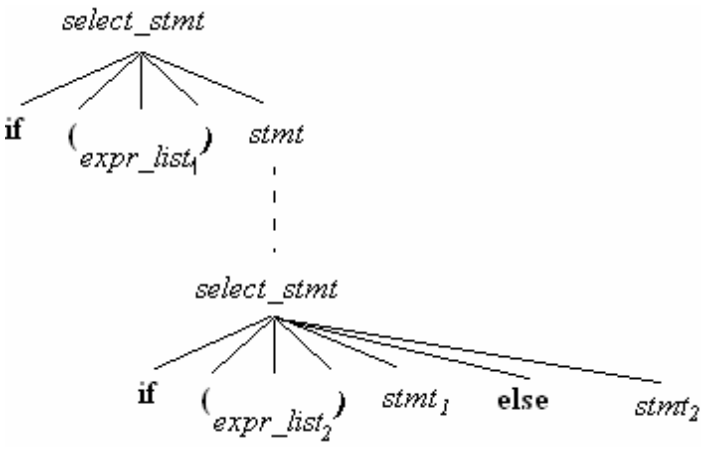
We have group the identical operator from left to right (+, -, /, %, *, and all relational and logical operator).

c) Dangling –Else Ambiguity

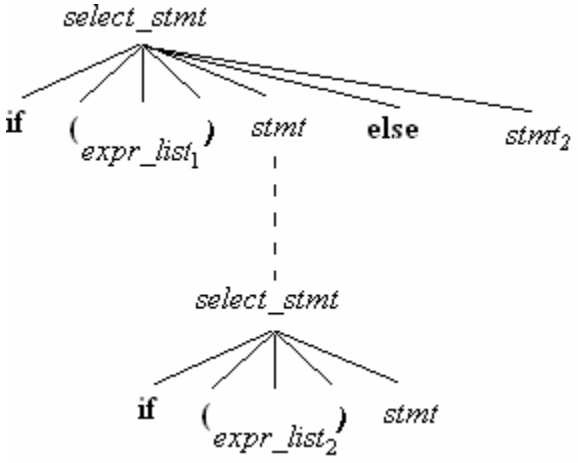
A well known example of syntactic ambiguity is the dangling-else ambiguity, which arises if a grammar has the following productions

select_stmt→ **if** (*expr_list) stmt*
select_stmt → **if** (*expr_list) stmt* **else**
stmt

The statement "**if (***expr_list*$_1$**) if (***expr_list*$_2$**)** *stmt* $_1$ **else** *stmt* $_2$" has two parse trees 2.3 (a) and 2.3 (b).



(a)



(b)

Figure 2.3: Two different parse trees

d) Solution of Dangling – Else Ambiguity

Since all programming languages prefer the first parse tree, so we have chosen first one for *Mayabini*. To solve this ambiguity the general rule is, "Match each **else** with the closest previous unmatched "**if (***expr_list***)"** "[1]. We will rewrite the grammar in such a way that "a statement appearing between a "**if (***expr_list***)**" and an **else** must be "matched;" i.e., it must not end with an unmatched "**if (***expr_list***)**" followed by any statement, for the **else** would then be forced to match this unmatched "**if (***expr_list***)**". A matched statement is either an "**if (***expr_list***)-else"** statement containing no unmatched statements or it is any other kind of unconditional statement." [1].

2.3 Lexical Analyzer

The lexical analyzer is the first phase of a compiler which reads the input characters and produces as output a sequence of token that a parser use for syntax analysis. [1]. A lexical analyzer can insulate a parser from the lexeme representation of tokens. The following sections describe some necessary terms of lexical analyzer.

2.3.1 Token

Token is treated as terminal symbols in the grammar for the source language. Boldface names are used to represent tokens. In most programming languages, the following constructs are treated as token: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parenthesis, commas, and semicolons. [3]

2.3.2 Lexeme

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

For example, the following statement

int number **;**

The substring ‘number’ is a lexeme for the token “identifier” or “ID”, ‘**int**’ is a lexeme for the token “keyword”, and ‘**;**’ is a lexeme for the token ” **;**”.

2.3.3 The Role of Lexical Analyzer

The main purpose of a lexical analyzer in a compiler application is to translate the input stream into a form that is more manageable by the parser. [13]. However the tasks of a lexical analyzer can be divided into two phases. They are: Scanning & Lexical analysis. [1]. Lexical analyzer can detect also some lexical errors. The figure 2.4 shows the interaction of Lexical Analyzer with Parser

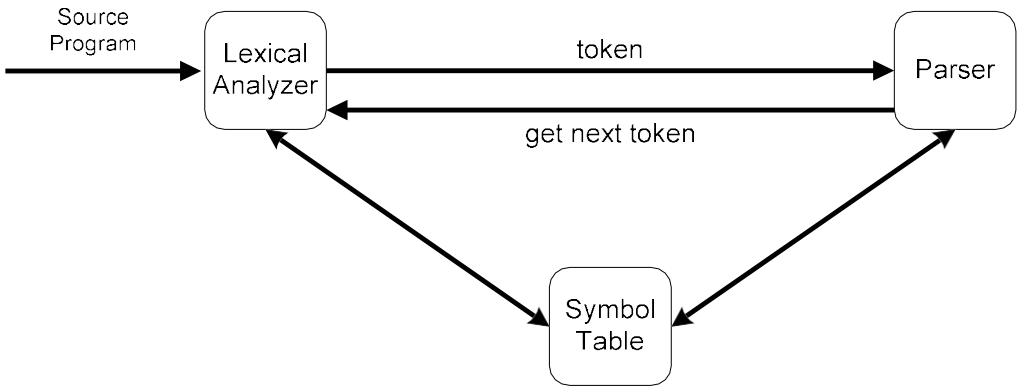


Figure 2.4: Interaction of Lexical Analyzer with Parser

a) Scanning:

In the scanning phase it scans the input file and eliminates comments and white spaces in the form of blank, tab and new-line characters. So the parser will have not to consider it. The alternative is to incorporate white space into the syntax which is not nearly as easy to implement. This is why most compilers do such tasks at scanning phase.

b) Lexical Analysis:

At the second phase it matches pattern for each lexeme for generate token. In some compilers, the lexical analyzer is in charge of making a copy of the source program with the error message marked in it. It may also implements preprocessor functions if necessary. The last two options are not supported by *Mayabini*. After identifying a lexeme itupdate its symbol table and lexical table. After identifying a lexeme it update its symbol table and lexical table. In the lexical analysis phase it can also check for some lexical errors such as: if any keyword is declared as identifier.

2.3.4 Working method of Lexical Analyzer

Most of the lexical analyzer identifies tokens by a rule called pattern associated with the token [1]. The pattern is said to match each string in the set. It generally uses DFA (Deterministic Finite Automata) to match a particular pattern.

a) DFA

A DFA is a set of finite states, having a set of finite input symbols and transition functions, and must have a start state and a set of final or accepting state [3]. This is also called finite state machine (FSM) [18]. To represent a DFA as a notation regular expression is used.

b) Expression

A regular expression is any well formed formula over union, concatenation and "Kleene" closure. In the case of compiler design it is an alternative way of describing a language's token set that takes a more grammatical approach [18]**.** Regular expression is used to represent any grammar using notation. Every language defined by a DFA can also be defined by a regular expression and vice versa [3].

c) Recognition of Tokens using DFA

This is also called transition diagram that depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

Let us define a regular expression to recognize any valid number in our compiler. A valid number is a set of digits that can followed by a decimal. No exponential is supported in *Mayabini*. Therefore, the expression is:

num → digit^{+} (.digit^{+})?

digit → ০ | ১ | ২ | ৩ | ৪ | ৫ | ৬ | ৭ | ৮ | ৯

alpha → W -

The figure 2.5 is transition diagram to recognize a digit.

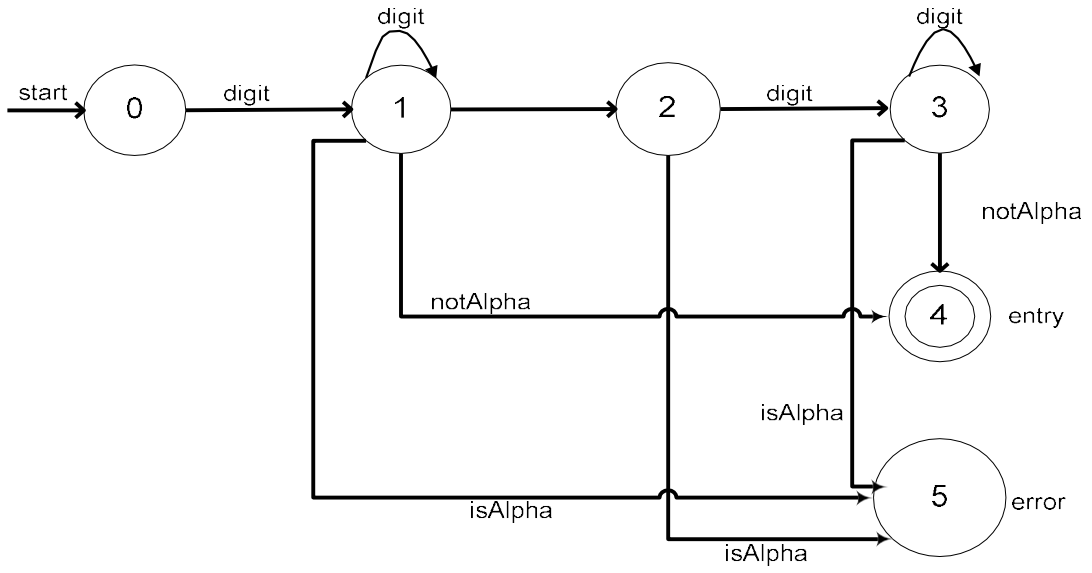


Figure 2.5: DFA for pattern matching of any digit.

The transition diagram starts from state 0. If it finds a digit then it moves to state 1. From here if another digit is found it will remain at the same state. If a decimal point is found it moves to next state 2. Here it searches for another digit. If a digit is found it moves to state 3. If another digit is found at state 3 it remains in this state. Otherwise it moves to state 4 if it founds a notAlpha. A notAlpha means that a char is found that is not in between w - . It is the general flow to search a digit. After it gets a valid digit it makes an entry at LexicalNode. (Details of LexicalNode is described at Implementation chapter.) Any valid digit may be , QQ, .Q, .QQ etc.

The lexical analyzer also has the capability of find lexical errors. In the previous diagram from state 1, 2, or 3 if it finds any alphabet (isAlpha) then it will generate an error. Suppose a digit is found like 3wt which is neither a valid digit nor a valid identifier or keyword. So, lexical analyzer will generate an error for identifying it as an invalid digit. Some examples of invalid digits are: w , QQw , .Qw , .QQo etc. Moreover, any undefined state for lexical analyzer is treated as an error.

Let us have another transition diagram in figure 2.6 that depicts the DFA for search an identifier or keyword.

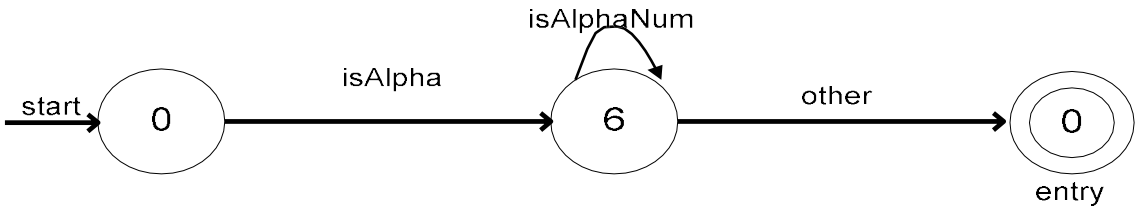


Figure 2.6: DFA for pattern matching of ID & Keyword

Like the previous one it also starts from state 0. If it finds an alphabet (isAlpha) it moves to state 6. From here if it finds an alphabet or any number (isAlphaNum) it remains at state 6. Otherwise it moves to state 0. Here it makes a dual entry. First of all, the identifier is searched whether it is a keyword or not. If it is a keyword it makes an entry at LexicalNode as a keyword and keeps the index of the keyword at Symbol Table. Here the lexeme is the keyword such as: Æ , ⅈא etc found and the token is "keyword". On the other hand, if it is an identifier then the entry is made to Symbol Table along with its name. The entry at LexicalNode is made named ID having the index at Symbol Table. In this case the lexeme is the identifier such as , " etc and the token is "id". The regular expression for this DFA is:

id → alpha(alpha | digit

)*alpha → w -

digit → ○▮ ▮ Q▮ ⊂▮ 8▮ •▮ ▮ ▮ ▮ ²

Both of the cases, the pointer for reading the input file moves one step ahead to determine about the lexeme. So, finally the pointer moves one character before.

2.3.5 Dealing with Symbol Table

Symbol Table is generally used to store information about various source language construct. The information is collected by the lexical analysis phase and used by the

synthesis phase to generate target machine code. For example, during lexical analysis, the character string, or lexeme, forming an identifier is saved in a symbol-table entry.

The symbol-table is concerned primarily with saving and retrieving lexemes. When a lexeme is saved, the token associated with the lexeme is also saved. The reserved words are saved at the beginning of the symbol table. When a lexeme is found it is entered at the symbol table only if it can be an entry of symbol table. A sample *Mayabini* program along with its symbol table entry is given below:

```
/* মায়াবিনী();
 * মজার মজার কোড লিখুন! রান করুন! সাবমিট করুন!
 */
```

```
/* মায়াবিনী();
 * মজার মজার কোড লিখুন! রান করুন! সাবমিট করুন!
 */

//*** প্রোগ্রাম: ৩ বড় না ৪ বড় ? ***//
যদি(৩ থেকে ৪ বড় হয়){
    দেখাও("বল্টুঃ আমি আগেই জানতাম ৪ বড়।");
}নাহলে{
    দেখাও("পল্টুঃ ঘোড়ার ডিম");
}


//*** প্রোগ্রাম: নন্দলাল ***//
ধরি জগত = "সুন্দর";
যদি (জগত দেখতে "সুন্দর" হয়){
    দেখাও("থাকবো নাকো বদ্ধ ঘরে দেখবো এবার জগতটাকে");
}নাহলে{
    দেখাও("আমি নন্দলাল হব, এ জগতকে ভয় পাব");
}
```

A conventional symbol table for a lexical analyzer is depicted at following table 2.1:

Æ

w ew
- Aw

Table 2.1: Conventional Symbol Table for a Lexical Analyzer

Symbol table also keeps other necessary records for each token like its value, type, and some other attributes as necessary by language designer.

2.3.6 Reading the Input File

Mayabini never reads directly from file. It copies the content of the input file to its editor. Then it reads source program from the editor. It has used input buffering to read from editor.

For the case of convenience a data structure named InputBuffer is used.

Algorithm for read from the source program is given here:

Algorithm 2.1.: InputBuffer

Input: Source file

Output: buffer []; contains all the input characters.

```
while(!EOF)
{
        ch = getchar();  //Read from source file.
        buffer [index++] = char;
}
buffer [index] = -1;     //-1 is the end marker or Sentinels.
```

This algorithm is written as generic algorithm. *Mayabini* will never read directly from the source file. It makes a copy of source file in its own editor and then does the rest. The source file is send to InputBuffer as a string. So we can redefine the algorithm as follows:

```
While (!EndOfInputString)
        Read each character from editor;
```

After reading from the editor *Mayabini* will copy the content to its Buffer.

At the end of the input array an end marker is kept that indicates the end of source which is called “Sentinels”. A “Sentinels” is a special character that can not be part of source program. [1].

2.3.7 Matching Pattern to Find Tokens and Update Symbol Table

The major task of the Lexical Analyzer is to find out all lexemes from a given source program. To do so it needs to do some pattern matching to check whether a lexeme is matched with a valid pattern.

We have used DFA for matching pattern. Algorithm for implementation of the DFAs is given below.

Algorithm 2.2: LexicalAnalyzer

Input: LexicalNode, SymbolTable, InputBuffer

Output: SymbolTable, LexicalNode

```
BEGIN
1. While (!EOF)
BEGIN
2. Read a Lexeme from InputBuffer
3. IF Lexeme is a digit
       Enter Lexeme at LexicalNode
   ELSE IF Lexeme is an ID
       IF ID is a Keyword
            Enter “keyword” at LexicalNode
            Enter index of SymbolTable at LexicalNode
        END IF
        IF Lexeme is not in SymbolTable        //i.e., neither a keyword nor already entered
            Enter name of ID at SymbolTable
            Enter “ID” at LexicalNode
            Enter index of SymbolTable at LexicalNode
         END IF
  ELSE IF Lexeme is an arithmetical or relational operator
       Enter Lexeme at LexicalNode
  ELSE IF Lexeme is Quoted String
       Enter Lexeme at LexicalNode
```

```
  ELSE IF Lexeme is a character
        Enter Lexeme at LexicalNode
  ELSE IF Lexeme is comments
        Skip
  ELSE
        Generate Error Message
4. GOTO step 1
END
END
```

Though this algorithm does not seem complex but its implementation is a bit tricky.

2.4 Syntax Analyzer

Each language has some predefined syntax in which the words are arranged together to form sentence. The word syntax comes from the Greek word syntax, meaning "setting out together or arrangement", and refers to the way words are arranged together [14]. For simple English sentence first comes subject (noun or pronoun) then verb followed by objects. For example," I love Bangladesh" is a syntactically correct. But "I Bangladesh love" is wrong. To ensure each sentence's form is correct, each compiler has a phase where the form/syntax is checked. This phase is called syntax analyzer. To check the syntax is called syntax analysis. Normally, syntax analysis is done by using different types of parser and parsing algorithm.

2.4.1 Transformation on Grammar

Syntax analysis can be done using a non-recursive predictive parser. A non-recursive predictive parser is a method or a program where a parse tree is constructed unambiguously depending on the input symbol and corresponding production rules [1]. If the input symbol and corresponding production rules don't match then errors are generated. For non-recursive predictive parser, the grammar should be unambiguous. There also needs some transformations on the grammar. In section 3.2.9 we have discussed about the ambiguity and in section 3.2.9 we have discussed how to remove

them from a grammar. Now we will examine some transformations on grammar which are necessary for non-recursive predictive parsing.

a) Left Recursion

Non-recursive predictive parser methods cannot handle left-recursive grammars, so a transformation is needed. "A grammar is left recursive if it has a non terminal *A* such that there is a derivation $A \overset{+}{\Rightarrow} A$ a for some string a" [1]. For example,

$$A \rightarrow A\ a \mid \beta$$

b) Elimination of Left Recursion

We can easily remove left recursion by the following process. First, we will group the production which starts with *A* in the following way:

$$A \rightarrow A\ a_1 \mid A\ a_2 \mid \ldots \mid A\ a_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Where no β_i starts with an *A.* then we replace *A*-production by

$$A \rightarrow \beta_1 A` \mid \beta_2 A` \mid \ldots \mid \beta_n A`$$

$$A` \rightarrow a_1 A` \mid a_2 A` \mid \ldots \mid a_m A` \mid$$

$$\varepsilon$$

The following algorithm will eliminate left recursion systematically from a grammar if it has no cycles (derivatives of the form $A \overset{+}{\Rightarrow} A$) and ε-production (like $A \rightarrow \varepsilon$ form).

Algorithm 2.3: Eliminating left recursion

Input. Grammar *G* with no cycles or ε-productions.

Output. An equivalent grammar with no left recursion.

1. Arrange the non terminals in some order $A_1, A_2, \ldots, A_n$.
2. **for** *i*:= 1 **to** *n* **do begin**

   for *j*:= 1 **to** *i*-1 **do begin**

   replace each production of the form $A_i \rightarrow A_j \gamma$

   by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$

   where $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j production;

   end

   eliminate the immediate left recursion among the A_i production.

end

c) Definition of Left Factoring

If a non terminal produces alternative productions then we will not be able to decide which production we will choose. So the grammar should be transformed into such a form where there is no more confusion. This transformation is known as left factoring, *e.g.*

$$A \rightarrow a\beta_1 | a\beta_2$$

d) Transformation to Left Factored grammar

Let, $A \rightarrow a\ \beta_1 |\ a\ \beta_2$ be two productions and the input begins with a nonempty string derived from a. Now we don't know which rule A to a β_1 or a β_2 to expand. However, we may defer the decision by expanding A to a A'. Then, after seeing the input derived from a, we expand A' to β_1 or to β_2. [1].That is, after transforming to left factoring original productions become

$$A \rightarrow a$$
$$A'A' \rightarrow$$
$$\beta_1 |\ \beta_2$$

The following algorithm transforms a grammar to equivalent left factored grammar.

Algorithm 2.4: Left factoring a grammar.

Input. Grammar G

Output. An equivalent left factored grammar.

Method. For each non terminal A find the longest prefix a common to two or more of its alternatives. If $a \neq \varepsilon$, i.e., there is a non trivial common prefix, replace all the A productions $A \rightarrow a\ \beta_1 | a\ \beta_2 | \ldots | a\ \beta_n | \gamma$ where γ represents all alternatives that do not begin with a by

$$A \rightarrow a\ A' | \gamma$$
$$A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$$

Here A' is a new non terminal. Repeatedly apply this transformation until no two alternatives for a non terminal have a common prefix [1].

2.4.2 FIRST and FOLLOW

First and follow are two functions which help us to fill the entries of parsing table.

a) FIRST

The **FIRST**(*A*) is the set of terminals that occur first in the strings of *L*(*A*) [2]. If a is any string of grammar symbols then **FIRST**(a) be the set of terminals that begin the strings derived from a and if a $\stackrel{*}{\Rightarrow}$ ε then ε is also in **FIRST**(a).

The following rules apply to determine the **FIRST**(*X*) for all grammar symbol *X*:

1) If *X* is terminal, then **FIRST**(*X*) is {*X*}.
2) If *X*→ ε is a production, then add ε to **FIRST**(*X*).
3) If *X* is non terminal and *X*→Y_1 Y_2**. . .** Y_k is a production, then place a in **FIRST**(*X*) if for some *i*, a is in **FIRST**(*Y*), and ε is in all of **FIRST**(Y_1), **. . .,** **FIRST**(Y_{i-1}); that is, Y_1 **. . .** Y_{k-1} $\stackrel{*}{\Rightarrow}$ ε. If ε is in **FIRST** (Y_j) for all *j* =1, 2, **. . .** , *k*, then add ε to **FIRST**(*X*). [1]

b) FOLLOW

The follow set of *A* is the set of symbols that can follow a string derivable from *A* **[2] FOLLOW**(*A*), for non terminal *A*, to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form, that is, the set of terminals *a* such that there exits a derivation of the form, *S* $\stackrel{*}{\Rightarrow}$ a *A B* β for some a and β. To compute **FOLLOW**(*A*) for all non terminals *A*, apply the following rules until nothing can be added to any **FOLLOW** set**.**

1) Place $ in **FOLLOW**(*S*)**,** where *S* is the start symbol and $ is the input right end marker.
2) If there is a production *A*→ a *B* β, then everything in **FIRST**(β) except for ε is placed in **FOLLOW**(*B*).
3) If there is a production *A*→ a *B,* or a production *A*→ a *B* β where **FIRST**(β) contains ε (i.e., β $\stackrel{*}{\Rightarrow}$ ε) , then everything in **FOLLOW**(*A*) is in **FOLLOW**(*B*). [1]

2.4.3 Predictive Parsing Table

It is a table whose rows are the set of non terminals and columns are set of terminals and each cell contains either a production rule or an error message. The production rules of a predictive parsing table are free from all kinds of ambiguities and left recursion. A predictive parsing table can be constructed by the following algorithm [1]

Algorithm 2.5: Construction of a predictive parsing table.

Input. Grammar *G*.

Output. Parsing table *M*.

Method.

1. for each production $A \rightarrow a$ of the grammar, do steps 2 and 3
2. For each terminal a in **FIRST**(a), add $A \rightarrow a$ to $M[A, a]$.
3. If ε is in **FIRST**(a), add $A \rightarrow a$ to $M[A, a]$ for each terminal *b* in **FOLLOW**(*A*). If ε is in **FIRST**(a) and $ is in **FOLLOW**(*A*), add $A \rightarrow a$ to $M[A, \$]$.
4. Make each undefined entry of *M* be **error**.

2.4.4 LL(1) Grammar

In general, if there is no multiple production rule in a cell of Predictive parsing table then this grammar is called **LL**(1) grammar. "A grammar whose parsing table has no multiple-defined entries is said to be **LL** (1) Grammar" [1]. The **LL**(1) stands for Left- to-right (scanning the input string), and Left (leftmost derivation of the input string) and "1" for using one input symbol of look ahead at each step to make parsing action decision [2]. To be a **LL** (1) grammar, all ambiguities or left recursion must be removed from a grammar. A grammar *G* is **LL**(1) if only if whenever $A \rightarrow a \mid \beta$ are two distinct production of G the following condition hold:

i. For no terminal a do both a and β derive string beginning with a.
ii. At most one of a and β can derive the empty string.
iii. If $\beta \overset{*}{\Rightarrow} \varepsilon$ then a does not derive any string beginning with a terminal in **FOLLOW**(*A*).[1]

Normally, each parse table contains an ambiguity after removing left recursion and left-factoring, in the cell *M*[*stmt´*][**else**] Which contains both *stmt´*→ **else** *stmt* and *stmt´*→ ε, since the **FOLLOW**(*stmt´*)={ **else ,$**}. For the following grammar

stmt → **if (** *expr* **)** *stmt stmt´*|

astmt´ → **else** *stmt* | ε

e*xpr*→ *b*

Language designers have resolved this problem by choosing either of the productions. After performing the above task any grammar is free from ambiguity and it can be used for a predictive parser.

2.4.5 Working Method of Non-Recursive Predictive Parser

A non-recursive predictive parser can be implemented by maintaining an input buffer, a stack, a parsing table (where all production rules are resided), and an output stream. The input buffer contains the string to be parsed, followed by a symbol $ used as end of the input string. Grammar symbols with a $ on the bottom, indicating the bottom of the stack are resided in the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array *M*[*A , a*], where *A* is a non terminal, and *a* is a terminal or the symbol $ [1]. Pictorially a non-recursive predictive parser looks like the figure 2.7.

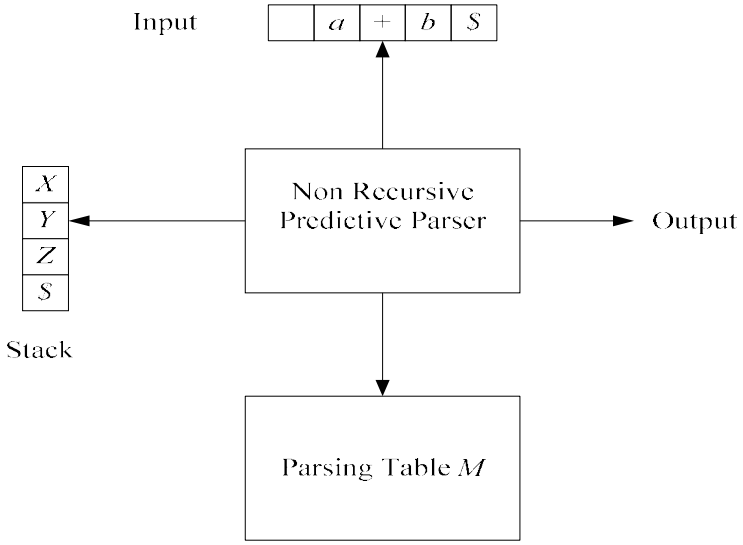


Figure 2.7: Model of a non-recursive predictive parser.

This parser is controlled by the following algorithm [1]

Algorithm 2.6: Non-recursive Predictive Parsing

Input. A string *w* and a parsing table *M* for grammar *G*.

Output. If w is in *L (G),* a leftmost derivation of w; otherwise an error indication.

Method. Initially, the parser is in a configuration in which it has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is given below:

Set *ip*(input pointer) to point to the first symbol of *w*$.
repeat
let *X* be the top stack symbol and a the symbol pointed to by *ip*
if *X* is a terminal or $ **then**
if *X* = *a* **then**
pop *X* from the stack and advanced *ip*
else *error()*
else /* *X* is a non terminal */
if *M*[*X, a*] = $X \rightarrow Y_1 Y_2 \ldots Y_K$ **then begin**
pop *X* from the stack;
push $Y_K, Y_{K-1} \ldots Y_1$ on to the stack, with Y_1 on top,
output the production $X \rightarrow Y_1 Y_2 \ldots Y_K$
end
else *error()*
until *X* = $ /* stack is empty */

This algorithm can be described in other way.

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops *X* off the stack and advances the input pointer to the next input symbol.

3. If *X* is a non terminal, the program consults entry *M*[*X, a*] of the parsing table *M*. This entry will be either an *X*-production of the grammar or an error entry [1]. Repeat all the steps until an error occurs or stack is empty.

2.5 Semantic Analyzer

The following sections will described various aspects of semantics of a language. It will also discuss about the semantic of *Mayabini*.

2.5.1 Semantics

Simply "semantics defines the meaning of the well-formed sentences" [2]. For example, "I drink rice" is not a valid sentence though it is syntactically (form) correct. Because, this sentence is not meaningful. In general semantic analyzer checks these types of error where form (syntax) of a sentence is correct but meaning (semantics) is not. So "Iraq has mass destructive weapons!" is a correct. In formal language "Semantics is usually defined informally in English, by attaching explanation and examples to syntax rules in a grammar for the language." [10]

2.5.2 Definition of Semantic Analyzer

Semantic Analyzer is the third phase of compiler where certain checks are performed to ensure that the components of a language fit together meaningfully. **"**The semantic analysis phase checks the source language for semantic errors and gather type information for the subsequent code generation phase." [8]

2.5.3 The Role of Semantic Analyzer

The main purpose of the semantic analyzer in a compiler is to check the source language is semantically correct or not before translate it in to machine language. In simple, it checks the type of identifier of the whole program. This check is done by type checker in single pass implementation [1]. The following figure 2.8 describes about the position of type checker in a compiler.

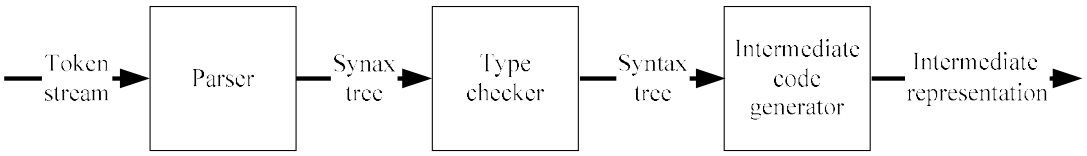


Figure 2.8: Position of type checker

a) Type Checker

A type checker verifies that the type of a construct matches that expected by its context. For example, the built-in operator **mod** in Pascal requires integer operands, so a type checker must verify that the operand of mod have type integer [1].

2.5.4 Working method of type checker

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notation of types, and the rules for assigning types to language constructs. [1]

a) Type expression

The type of a language construct will be denoted by a "type expression". Informally, a type expression is either a basic type or is formed by applying an operator called a *type constructor* to other type expressions. The set of basic types and constructors depend on the language to be checked.

We use the following definition of *type expression*:

i. A basic type is a type expression. Among the basic types are **int**(Æ), **char**(–), and **float**(א ¼).

ii. A type constructor applied to type expressions is a type expression. Constructors include *Arrays.* If *Type* is a type expression, then array (*I, Type)* is a type expression denoting the type of an array with element of type *T* index set *I*. *I* is often a range of int(Æ). For example, in Bangla C language [1]

Æ F ○ 1 ;

b) Type system

A *type system* is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type. The type systems are specified in a syntax-directed manner. [1]

type → **int**	{	*type.t* = **int**	}
type → **char**	{	*type.t* = **char**	}

c) Static and Dynamic type checking

After setting the type system it checks the type for assignment statement, relational or arithmetical expression etc. this check can be static and dynamic. Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of that element [1].

d) Error recovery

Since type checking has the potential for catching errors in programs, it is important for a type checker to do something reasonable when an error is discovered. At the very least, the compiler must report the nature and location of the error [1].This is called error recovery. It is desirable for the type checker to recover from errors, so it can check the rest of input. *Mayabini* does not support it due to simplicity.

2.5.5 Semantic rules of type checker

There are two notations for associating semantic rules with productions, syntax-directed definitions and translation schemes. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details but Translation schemes indicate the order in which semantic rules are to be evaluated. So they allow some implementation details to be shown. [1] *Mayabini* has been used the Translation schemes for semantic rule due to practical aspect.

a) Semantic rule for type setting of declaration

Mayabini has been used the following translation scheme to save the type of the declared variable:

prog → *decl main_fun*

fundecl → *type* **id** *A ;*

decl | ε

A → **[num]** | ε	{	*type.type* := array(0…num, *type.t*)	}
type → **int**	{	*type.type* := int	}
type → **char**	{	*type.type* := char	}
type → **float**	{	*type.type* := float	}

b) Semantic rule for type checking of expressions

Mayabini has been used the following translation scheme to check the mod(%) operation:

expr_list → *expr_list*1 **mod** *expr_list*2 { *expr_list.type* := **if** *expr_list*1*.type* = *int* **and** *expr_list*2*.type* = *int* **then** *int* **else** *type_error()* }

Mayabini has been used the following translation scheme to check the array index:

expr_list → **id** [**id**1 | **num**] { *expr_list.type* := **if** id1*.type* = *int* **or** num*.type* = *int* **then** *int* **else** *type_error*( ) }

c) Semantic rule for type checking of statements

Mayabini has been used the following translation scheme to check the type of assignment statement:

stmt → *as_fun_stmt* { *stmt.type* := **if** id*.type* = *expr_list.type* **then** void

as_fun_stmt → **id** *B* **else** type_error( ) }

B → = *C* | **[** *Q*

Q → *expr_list* **]** = *C* | **]** = **str ;**

C →*expr_list* **;** | **letter ;**

Mayabini has been used the following translation scheme to check the type of input output statement:

stmt → *io_stmt* { *stmt.type* := **if** id.*type* =*int* **or** char **or** float

io_stmt → **printf (** *D* | **scanf (id** *E* **) ;** **then** void

D → **str) ; | id** *E* **); | num) ;** **else** *type_error*( ) }

Mayabini has been used the following translation scheme to check the type of iterativestatement:

stmt → *iter_stmt* { *stmt.type* **: = if** *expr_list.type* =*boolean* **then**

iter_stmt → **while (** *expr_list* **)** *stmt1* *stmt1.type*

else *type_error*( ) }

Mayabini has been used the following translation scheme to check the type of selectionstatement:

stmt → *select_stmt* { *stmt.type* := **if** *expr_list.type* =*boolean*

select_stmt → **if (** *expr_list* **)** *stmt stmt′* **then** *stmt1.type*

else *type_error*( ) }

2.6 Code Generation:

Mayabini is an analysis-synthesis model compiler. In this type of compiler the front end translates a source program into an intermediate representation from which the back end generates the target code [1]. The figure 2.9 shows the position of intermediate code generator in a compiler.

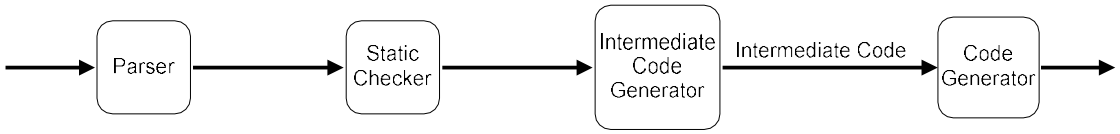


Figure 2.9: Position of Intermediate Code Generator in a Compiler

Many of the compilers omit the Intermediate phase. *Mayabini* also has dropped this phase and directly moves to Code Generator phase.

2.6.1 Reasons behind omission of Intermediate Code

Though Intermediate Code Generation phase is a good approach but it is not the only solution for generating target machine code. The main reasons are as follows:

i. *Mayabini* primarily is not a platform independent compiler because most of the common of Bangladesh uses INTEL base micro computer and Windows operating system.
ii. Dropping a phase saves time while compiling.
iii. It also saves space, though at present space is not a primary concern.

2.6.2 Issues in the Design of a Code Generator

There are several issues regarding the code generation. These issues are briefly discussed in the following sections.

a) Input (Source Program)

As *Mayabini* has dropped the Intermediate Code phase so the input to code generator is the source program. The Lexical Node (Described at Lexical Analyzer phase) keeps records of all lexemes that has found at the source program. Beside, it also keeps the Symbol Table. Before generation of target code compiler assumes that the source program is syntactically and semantically correct.

b) Output (Target Program)

Output of Code generator is obviously the target machine code. This output may take on a variety of forms: absolute machine language, re-locatable machine language, or assembly language. [1]

Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. The problem is generating absolute machine language is really a clumsy job.

Producing a re-locatable machine language program (object module) as output allows sub programs to be compiled separately. This can be done linking and loading [1]**.** The disadvantage of this is if the target machine does not handle re-location automatically, the compiler has to deal with it.

Producing an assembly language program as output makes the process of code generation somewhat easier. *Mayabini* uses the third approach to generate target code. It uses assemblyinstruction (INTEL 8086) to generate assembly code.

c) Memory Management

Mapping names in the source program to addresses of data object in run-time memory is done cooperatively by the front end and the code generator. The front end helps in memory management with the help of the Symbol Table. The type in a declaration which can be found from Symbol Table, determines the width of a variable i.e., the amount of storage, needed for the declared name.

d) Instruction Selection

The nature of the instruction set of the target machine determines the difficulty of instruction selection. If the target machine does not support each data type in a uniform manner, then each exception to general rule requires special handling. The quality of the generated code is determined by its speed and size [1]. A target machine with rich instruction set can generate more efficient code.

e) Register Allocation

The target machine that has more registers can make more use of its registers. But if it has few register, then register handling must be easier. As *Mayabini* uses 8086 instruction set it has only four data register and has to deal carefully.

f) Choice of Evaluation Order

The most important criterion for a code generator is that it produces correct code. To do this the compiler must have pre determined evaluation order of both operators and operands. This is probably the most important issue on designing a Code Generator.

2.6.3 Generating Target Machine Code in *Mayabini*

This section describes details about code generation. *Mayabini* is based on INTEL microprocessor and Windows operating system. 8086 instruction sets have used for generating assembly code. The code generation phase of *Mayabini* can be divided into threemajor modules:

i. The Expression Evaluation Module.
ii. The Label Generator Module.
iii. The Declaration Module.

a) The Expression Evaluation Module

The heart of run-time system is the expression parser. Whenever *Mayabini* finds an expression then it sends the expression to the Expression Evaluation module to evaluate it i.e., generate assembly code for that expression. There are numerous approaches to build an Expression Evaluation parser. Some tools are also available called 'Parser Generators' which take the grammar of a language and produces a parser for language as output. These tools are hard to learn, inflexible and very few are geared to interactive work [13]. It is reported that the parser they produce often contain potential bugs. Llama, Occs, Bison are well known parser generators. [13]

For the problems of existing parser generators *Mayabini* has used its own parser that evaluate all types of expressions. This module can be subdivided into two parts:

i. Convert the Infix Expression to Postfix Expression.
ii. Generate target code for Postfix Expression.

Both the parts are explained below in details.

i) Converting an Infix Expression to a Postfix Expression

Before going into detail some important definition are given below.

ii) Infix Expression

Any mathematical expressions are infix expression. e.g.:1+2*3. This expression can be found by traverse an ordered rooted tree in-orderly.

iii) Postfix Expression

An expression is a postfix expression if it the expression has found by traversing its ordered rooted tree post-orderly.

iv) Ordered Rooted Tree

An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered. They are drawn so that the children of each internal vertex are shown in ordered from left to right. [15]

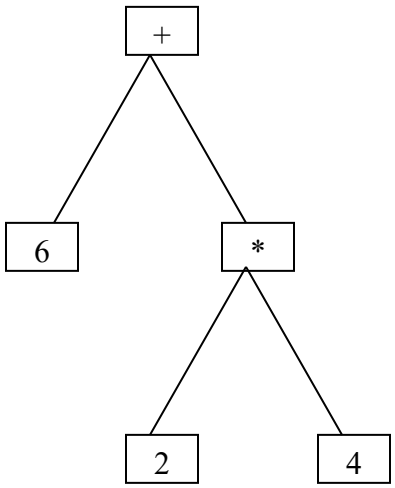


Figure 2.10: Ordered Rooted Tree of Mathematical Expression 6 + 2 * 4

The above figure is an ordered rooted tree of mathematical expression 6+2*4 where it follows the mathematical evaluation order *i.e*. the multiplication is calculated first then the addition.

v) Inorder Traversal

If T is an ordered rooted tree with root r and T consists only of r then r is the inorder traversal of T. Otherwise if T1, T2, T3 … are the subtree of r from left tot right then inorder traversal begins by traversing T1 in inorder, then visiting r. It continues by traversing T2, T3 and so on in inorder [15]. As for example the previous figure if traversed in-orderly will result the expression 6+2*4.

vi) Postorder Traversal

If T is an ordered rooted tree with root r and T consists only of r then r is the postorder traversal of T. Otherwise if T1, T2, T3 … are the subtree of r from left tot right then postorder traversal begins by traversing T1 in postorder, then visiting r. It continues by traversing T2, T3 and so on in postorder [15]. As for example the previous figure if traversed in-orderly will result the expression 624 * +.

vii) Traversing a tree post-orderly (An algorithmic Approach)

The following algorithm converts an infix expression to a postfix expression.

Algorithm 2.7: Post Order Traversal

POLISH (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix notation P.

1. Push “(” onto STACK and add “)” to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of stack) which has the same precedence as or higher precedence than the current operator.

b) Add current operator to STACK.

.END IF

6. If a right parenthesis is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
    b) Remove the left parenthesis.

   END IF

   [End of Step 2 Loop]

7. END.

[8]

viii) Generate target code for Postfix Expression

After getting the postfix notation of an expression we do evaluate the expression. Postfix expressions are easier to evaluate because they omit all kind of braces. The following algorithm has used the evaluation.

Algorithm 2.8: Evaluating Postfix Expression

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This act as sentinel]
2. Scan P from left to right and repeat steps 3 & 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator is encountered, then:
    a) Remove the two elements of STACK, where A is the top element and B is the next to top element.
    b) Evaluate B (as operand No. two) and A (as operand No. 1) with the associate operator.
    c) Place the result of b) on STACK.

   END IF

5. Set VALUE equal to the top element of STACK.
6. END.

[8]

In the case of *Mayabini* this algorithm does not do the evaluation part rather than it generates the actual assembly code. (Line 4b) This code is written to an assembly file that holds all assembly codes for an *Mayabini* source program.

b) The Label Generator Module

Whenever *Mayabini* finds a loop structure or a branching structure it has to generate labelsat its corresponding assembly file. Three types of labels can be found:

i. Loop structure label (ץ)
ii. Conditional structure label (אּ - w"–)
iii. Function calling label.

i) Loop structure label

Mayabini supports only while loop structure. Because it is enough for do all kind of iterative statements. Moreover, it makes program more readable by removing too much feature multiplicity. The general format for an iterative statement is:

```
[initial_statement];
While (expression_list)
{
        statement 1;
        statement 2;
        ……
        statement n;
        [loop_counter;]
}
```

expression_list can be a logical or Boolean expression but it can't call any function to evaluate the result of any expression inside the expression_list..

Assembly language can not have more than one label with same name. This is the one and only design issue on designing the algorithm. The while counter keeps track of generating these labels uniquely.

The algorithm for generating while loop in assembly language is given follows.

Algorithm 2.9:

```
If (input == "while")
{
        while_counter++;
        write ("while"+while+counter+":")   ;
        evaluate expression for looping condition;
        compare value found from expression_evaluation;
        write ("JNE end_while"+while_counter++);
        state = 0;
}
else if (input == "}")  //it should ensure "{" is for while loop only.
{
        pop from stack;
        if ( "}"  AND for_while ("}"))
        {
                write ("JMP while"+label_count);
                write ("end_while"+label_count+":");
                state = 0;
        }
}
```

ii) Conditional structure label

Mayabini supports three types of conditional expression. They are:

1. Single way selector.
   e.g. if (Boolean_Expression) statement_list;
2. Two way selector:
   e.g. if (Boolean_Expression) statement_list;
   else statement_list;
3. n-way selector or multiple way selector:
   e.g. if (Boolean_Expression) statement_list;
   else if (Boolean_Expression) statement_list;
   else if (Boolean_Expression) statement_list;
   ……..
   [else statement_list] ;

Assembly language can not have more than one label with same name. This is the one and only design issue on designing the algorithm. The counter keeps track of each if, else, else if expression.

The algorithm for generating selection statement in assembly language is given follows:

Algorithm 2.10:

Case 0: If (input == "if")

```
{
        else_count++;
        push (else_count,if);
        push ("if_bracket", "{");  //for pop"}"
        evaluate  expression for if condition;
        compare value found from expression_evaluation;
        write ("JNE else"+else_count);
        state = 0;
}
else if (input = "else")
```

```
{
        input = get value from Lexical_Node;
        If (input == “if”)
        {
                else_count = pop_from_stack();
                write (“JMP end_if”+end_if_count;
                write (“else”+else_c+”:”);
                else_count++;
                push (else_count,”if”);
                push (else_if_break, “{”);
                evaluate  expression for if condition;
                compare value found from expression_evaluation;
                write (JNE else”+else_count);
        }
        else
        {
                else_counter = pop_from _stack;
                write (“jmp end_if”+end_if_count);
                write (“else”+else_counter+”:”);
                push (end_if_count,”else”);
                state = 0;
        }
else if (input == “}” )
{
        //if  (input==”else”)
        end_if = pop_from_stack;
        write (“end_if”+end_if_count+”:”);
        end_if_count++;
        stare = 0;
}
```

iii) Function calling label

Unfortunately *Mayabini* can't support parameter passing to functions. Its functions also do not have any return type. However except this two its functions behave like other programming language's functions. Only recursive algorithm can't be supported in *Mayabini* for absence of parameter passing and return types.

A function calling of *Mayabini* will be:

```
- Aw ();
```

A function declaration will be:

```
- Aw ()
{
        //statement list;
}
```

Another modifiable thing is inside a function variable declaration is not allowed. In fact as all variable of *Mayabini* is global so this is a unnecessary thing.

The algorithm for designing function calling and generating label for function at assembly language is given below:

Algorithm 2.11:

```
If (input == id)
{
        if ( check next input ("(") == true )
                if (check next input (")") == true )
                        if  (check next input ("=") == false )
                                write ("call"+id);
}
```

c) The Declaration Module

Whenever *Mayabini* finds a variable declaration, it defines that variable at the data segment at Assembly file. Even any quoted string also needs to be declared at the data segment of Assembly file.

We have three primitive data types at *Mayabini*. They are:

i. Æ ¸ ¨
ii. א ¼4
iii. – ¨

For the case of declaration simplicity we have kept all data types as word i.e. 16 bit. Beside 8086 instructions are 16 bit instruction.

Algorithm 2.12: Declaration of Variable

Input: Source File

Output: Assembly file containing variable declaration.

```
While(! EOF)
{
    if (token == "Æ   " OR "א  ¼4  " OR "–   ")
    {
        variable = getNextToken();
        write("variable DW ?")  //Typical Assembly specification
    }
    else if (token == Quoted String)
    {
        variable = getNextToken();
        write("msg DB variable")  //Declaring message that will be displayed as output.
    }
```

Complete DATA segment declaration of assembly file can be found at Implementation chapter.

2.7 Summary

Now we hope the reader has got an overview of different issues like grammar designing, lexical analysis, semantics analysis, and code generation of a compiler. Having sufficient background of language, grammar and compiler design, the reader after reading this chapter will have a vivid picture of the design aspects of *Mayabini* compiler.

Chapter 3

Implementation

3.1 Introduction

This chapter may be termed as the extended part of Chapter 2. The algorithms and design issues that have been described in previous chapter, how they have been implemented into code is discussed here in details. It has also explained some critical code segments that have been written in JAVA2.

3.2 Lexical Analyzer

Mayabini has used finite automaton to design its lexical analyzer. This technique is used to identify a valid digit, identifier including keywords, all arithmetical and relational operators (*Mayabini* does not have any logical operators instead it has used keyword to perform logical operation.), punctuations, string literal, character, and comments. This section describes all the data structures, and algorithms that *Mayabini* has used to identify token and generate symbol table.

3.2.1 Data Structures

Two major data structures have been used in *Mayabini*. They are: SymbolTable and LexicalNode. Both of this data structures has used an internal data structures named SymbolNode and Node respectively.

a) Symbol -Table

The main field of Symbol-Table is the field SymbolTable which can hold an object of SymbolNode. A SymbolNode consists of

i. A name field that is the actual name given by the programmer for any valid identifier.
ii. A type field that is the data type of the identifier may be any primitive data type or a Function type.

iii. A Boolean value array that indicate whether the identifier is an array or not.

The Lexical Table and Syntax table for a sample program of our compiler has already been described at chapter table 2.2.

b) Lexical-Node

This data structure has kept a copy of each and every token from input file. If the token is entered at the Symbol-Table then it keeps the index of its entry at Symbol-Table. Actually this data structure is about similar to the Symbol Table.

Like the SymbolTable its main field is lexNode which can contain another data structure called Node. A Node consists of:

i. A data field that holds the name of the lexeme.
ii. An index that holds the index of the token lexeme at symbol table if exists.
iii. A type that is the data type of a number. (Not of identifier.)
iv. A strValue is the value of the number.

A lexical table for *Mayabini* can be found at Table 2.4.

Here, we discuss a typical JAVA2 code according to this algorithm described at Design part for searching an identifier.

c) Code dissection for design a Lexical Analyzer

```
1.char ch;
2.char token[]=new char[128];
3.int state=0,index=0;
4.Node node;
5.while(!EOF)
6.{
7.      switch(state)
8.      {
```

```
9.          case 0:// Initial state
10.                 index=0;
11.                 ch=buffer.getChar();  //Read from Input Buffer.
12.                 if(ch==EOF)
13.                         return;
14.                 if(isAlpha(ch))
15.                 {
16.                         token[index++]=ch;
17.                         state=6;
18.                 }
19.         case 6:
20.                 ch=buffer.getChar();
21.                 if(ch==EOF)
22.                         return;
23.                 while( isAlphaNum(ch) )
24.                 {
25.                         token[index++]=ch;
26.                         ch=buffer.getChar();
27.                         if(ch== -1)
28.                                 return;
29.                 }
30.                 buffer.unGetChar();
31.                 String s=String.copyValueOf(token,0,index);    //Take the Lexeme.
32.                 int n;
33.                 //Search for Keyword.
34.                 if( ( n=symbolTable.keywordSearch(s) ) >=0)
35.                 {
36.                         lexNode.add(SymbolTable.valueAt(n),n);
37.                  }
38.
39.                 //Search whether entried or not if not a keyword.
```

```
40.                     else if((n=symbolTable.idSearch(s))>=0)
41.                             lexNode.enq("id",n);    //If already entered
42.                     else  //else Enter at both Symbol & Lexical table.
43.                     {
44.                             symbolTable.add(s);
45.                             lexNode.enq("id",symbolTable.currentIndex());
46.                     }
47.                     index=0;
48.                     state=0;
49.                     break;
50.             }
51.}
```

N.B. This code does not reflect the actual coding. It is a kind of pseudo code.

In the code after read an alphabet from the source file (line 14) the Lexical Analyzer moves to a new state (line 17) where it searches for next characters or digit until it gets a non alphanumeric value. This is obviously the pattern for any identifier. After ensure an identifier it looks at the symbol table to find whether it is a keyword or not (line 34) . If so, it enters the keyword and its index at symbol table at LexicalNode (line 36). If it is not a keyword then the **id** must be a user defined identifier. If such an **id** is already entered at Symbol Table the new entry is made only at LexicalNode (line 41). Otherwise, both LexicalNode and SymbolTable get the new entry (line 44-45).

3.3 Syntax Analyzer

For *Mayabini*, non-recursive predictive parser has been used to analyze the syntax. We have designed a grammar for *Mayabini* and removed ambiguity from it according to section 2.2. Now we will transform our grammar which will be equivalent to left-recursion and left- factoring grammar. So it will perform syntax analysis on the source language.

3.3.1 Elimination of Left Recursion

In *Mayabini* before eliminating the left recursion the grammar was like this

expr → *uniop term addop term* |

termterm → *factor mulop factor*|

factor

factor → **id** | **id**[*expr_list*] | **num** | **(** *expr_lsit* **)**

uniop → + | **-** | **!** |

ε*addop* → + | -

mulop → * | **%** |

/

relop→ > | >= | < | <=

eq → = = | **!=**

Adopting the algorithm 2.3, we have eliminated left recursion. So *Mayabini* grammar become

expr → *uniop term expr´*

expr´→ *addop term expr´* |

ε*term* → *factor term´*

term´→ *mulop factor term´* | ε

factor → **id** *P* | **num** | **(** *expr_lsit* **)**

P→ [*expr_list*] |

ε*uniop* → + | **-** | **!**

| ε*addop* → + | -

mulop → * | **%** |

/

relop→ > | >= | < | <=

eq → = = | **!=**

We have eliminated left recursion from all grammar using this algorithm.

3.3.2 Transformation to Left Factored grammar

We took the following grammar from *Mayabini*. We have to transform it to left-factored grammar.

decl → *type* **id** *; decl* | *type* **id[num]** *; decl* |ε

type → **int | char | float**

After transformed to left factoring according to algorithm-2.4, our above grammar becomes

decl → *type* **id** *A ; decl* | ε

A → **[num]** | ε

type → **int | char | float**

We have transformed all other productions using this algorithm.

3.3.3 Determining of FIRST and FOLLOW Set

Before design the predictive parse table we have to determine the **FIRST** and **FOLLOW** set from *Mayabini* grammar.

A portion of *Mayabini* grammar is given below

prog → *decl main_fun*

fun decl → *type* **id** *A ;*

decl | ε*A* → **[num]** | ε

type → **int | char | float**

main_fun → **main ( )**

comp_stmtfun → *fh*

comp_stmt fun | ε

.

.

.

Using rules 2.4.2 the first and follow set of *prog* are respectively

FIRST (*prog*) = {**int, char, float, main**}

FOLLOW (*prog*) = {$}.

The first and follow sets of other grammar symbols have been added to Appendix-B.

3.3.4 Predictive Parsing Table

For parse table we designed a class called "ParseTable". We include three arrays of String class in it. They are given below:

String parseTable[TOTAL_NONTERM][TOTAL_TERM]

String nonterminal[TOTAL_NONTERM]

String terminal[TOTAL_TERM]

The “terminal” and “nonterminal” arrays contained all set terminals and non terminals respectively of *Mayabini* grammar, e.g.,

```
terminal[0]="int";
terminal[1]="char";  and
nonterminal[0]="Prog";
nonterminal[1]="Decl";
```

We have filled each cell of “parseTable” array by either a production rule or an error using algorithm-2.5, *e.g.*,

```
parseTable[0][0]="Decl Main_Fun Fun";
parseTable[0][4]="Error";
```

If we had to look up a production in a particular cell of parseTable then first we searched for non terminal and terminal in the “nonterminal” array using “indexOfNonterminal” method and “terminal” array using “indexOfTerminal” method respectively. These two methods returned either index of terminal or nonterminal or -1(if not found). Using these values we have directly get the value of that cell using the ”grammarAt” method.

3.3.5 LL(1) grammar

Still, there is an ambiguity in parsetable[*stmt´*][**else**] cell. Which contains both *stmt´*→ **else** *stmt* and *stmt´*→ ε, since the **FOLLOW**(*stmt´*)={ **if, while, id, printf, scanf, break, continue, {, }, else** }. For the following grammar

stmt → *select_stmt* | *iter_stmt* | *as_fun_stmt* | *io_stmt* | *jump_stmt* | *comp_stmt*

select_stmt → **if (** *expr_list* **)** *stmt stmt´*

stmt´ → **else** *stmt* | ε

e*xpr_list* → *M expr_list´*

The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an **else** is seen. We have resolved the ambiguity by choosing *stmt´*→ **else** *stmt.*

3.3.6 Non-Recursive Predictive Parser

At this stage our grammar is free from all kinds of ambiguity. So we can construct a non-recursive predictive parser. To construct the parser, we have designed a class "SyntaxBangla". It has a method "banglaParse", which successfully check syntactical error of the source language using the algorithm-2.6. It uses objects of "Stack", "ParseTable" and "LexicalNode" class. Input strings are kept in "input" object which is an instance of "LexicalNode" class. Output productions are kept in "stack" object of "Stack" class. All production rules are resided in "parseTable" object of "ParseTable". A very simple example and moves made by *Mayabini* predictive parser is given below:

```
 Ç y
 †   ()
{
          (  )
}
```

Corresponding input strings:

float id ; main ( ) { printf(id);}

Stack	Input	Output
$ prog	**float id; main({printf(id);}$**	
$ fun main_fun decl	**float id; main({printf(id);}$**	prog→ decl main_fun fun
$ fun main_fun decl **;** A **id** type	**float id; main({printf(id);}$**	decl→ type **id** A ; decl
$ fun main_fun decl **;** A **id** float	**float id** ; **main({printf(id);}$**	type→**float**
$ fun main_fun decl**;**A**id**	**Id ; main ( ){printf(id);}$**	
$ fun main_fun decl **;** A	**; main ( ){printf(id);}$**	
$ fun main_fun decl **;**	**; main ( ){printf(id);}$**	A→ ε

$ fun main_fun decl	**main ( ){printf(id);}$**	
$ fun main_fun	**main ( ){printf(id);}$**	decl→ ε
$ fun comp_stmt**) (main**	**main ( ){printf(id);}$**	main_fun→**main**()comp_stmt
$ fun comp_stmt **) (**	**){printf(id);}$**	
$ fun comp_stmt **)**	**){printf(id);}$**	
$ fun comp_stmt	**{printf(id);}$**	
$ fun **}**stmt_list **{**	**{printf(id);}$**	Comp_stmt→{stmt_list }
$ fun **}**stmt_list	**printf(id);}$**	
$ fun **}**stmt_list stmt	**printf(id);}$**	stmt_list → stmt stmt_list
$ fun **}**stmt_list io_stmt	**printf(id);}$**	stmt→io_stmt
$ fun **}**stmt_list D**(printf**	**printf(id);}$**	Io_stmt → **printf (** D
$ fun **}**stmt_list D**(**	**(id);}$**	
$ fun **}**stmt_list D	**id);}$**	
$ fun **}**stmt_list **;)** E **id**	**id);}$**	D → **id** E **);**
$ fun **}**stmt_list **;)** E	**);}$**	
$ fun **}**stmt_list **;)**	**);}$**	E→ ε
$ fun **}**stmt_list **;**	**;}$**	
$ fun **}**stmt_list	**}$**	
$ fun **}**	**}$**	stmt_list→ ε
$ fun	**$**	Fun→ ε
$	**$**	

Table 3.3: Moves by *Mayabini* predictive parser on the above input string.

3.4 Semantic analyzer

Mayabini checks the all kinds of type during the compilation of the source program. This implementation has followed the semantic rules which are defined in the design issues part. Now here described about how implementation is done by using transition diagrams and its corresponding algorithms.

3.4.1 Data structure

SymbolTable and LexicalNode two major data structures have been used at semantic analyzer in *Mayabini*. Both of this data structures has used an internal data structures namedSymbolNode and Node respectively.

3.4.2 Type setting of declared variables

At first, *Mayabini* sets the type of variable which is declared before the w ew function. Itis necessary to set the type of variables before doing any type checking operation on those variables. The valid declared variable will be look like this:

```
Æ   ;
Æ   F ○1;
w ew( )
{
}
```

It checks the variable weather it is already declared or not before setting the type of that variable and also checks the index of array variable. If the variable is already declared or index is not integer number, it will generate an error. The following two error cases will be look like this:

Case 1:

```
Æ   ;
Æ   F ○1;
Æ   ;
. . . . . .
```

Case 2:

```
Æ   ;
Æ   F ○.○1;
. . . . . .
```

Mayabini has been used the following transition diagram 3.1 to set the type of variables in the symbol table:

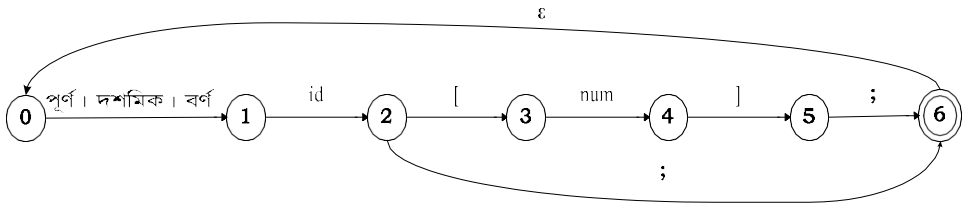


Figure 3.1: transition diagram for declared statement

Pseudo code for type setting of declared variable:

input is the variable which contain the token value from the lexical node.

```
Switch ( state )
begin
        case 0:
                if ( input == '$' then ) print("w" 56 7 <- e8 ▮")
                else if (input = = 'Æ *'|| '– '|| 'א 4 ') then
                begin
                        varType = input
                        state = 1
                end
                break;
        case 1:
                forward input pointer
                if (input(id) type = = null) then state = 2
                else error("m: Y <4 m * 3w w <5: – Y o =<o<>▮")
                break;
        case 2:
                forward input pointer
                if (input = = ';' ) then state = 6
                else state = 3
                break;
```

case 3**:**

forward *input* pointer

if (*input* = = ‘Æ ’|| **num**) **then** state = 4

else error (" m7 <o e3 w– 7: Æ eA 7 =<6 =<– ▮")

break;

case 4**:**

forward *input* pointer

array = **true** // to set the variable is array type in the symbol table

state = 5

break;

case 5**:**

forward *input* pointer

state = 6

break;

case 6**:**

set varType of the declared variable with array value in the symbol table

array = **false** // re-initialize array value

state = 0

end

3.4.3 Type checking of expression

Mayabini supports logical expressions and arithmetical expressions. Since both types of expressions are syntactically corrected by syntax analyzer, only mod(%) and index checking needs for semantic analyzer.

a) Type checking of mod (%) operation

The type checking of mod operation is that the left and right hand side sub-expressions of mod operator must be integer type. *Mayabini* checks all kinds of nested mod powerfully. If it finds any floating type within the boundary of left or right hand side sub-expressions, it have generated an error. Let us consider the following example:

```
Æ  ;
Æ  @ F ○1;
w ew()
{
      = c;
→ @F 1 = (  %•+  )%Q;
}
```

The above statement is semantically correct. But if the statement is instead of above one, it will generate error for invalid mod operation:

```
Æ  ;
Æ  @ F ○1;
w ew()
{
      = c;
→ @F 1 = (  %•.○+  )%Q;
}
```

Mayabini have used several transition diagrams to check the invalid mod operation in an expression which have been added in Appendix D.

Pseudo code for type checking of mod operation:

FUNCTION modAnalyzer(state : int) : boolean

switch(state)

begin

case 13**:**

backward *input* pointer

if (*input* == **id** || **num**)**then**

begin

if (**id or num** type = = ’Æ ‘) **then** state = 13

else error(" % wY7 Æ eA 7 – Æ @o<Yo 3w =<6 =<–∎")

end

else if (*input* == '/' || **'*') then** state = 13

else if (*input* == ')') **then** state = 14 with increment closeBrack

else state = 15

break

case 14**:**

backward *input* pointer

if (*input* = = **id** || **num**)**then**

begin

if (**id or num** type = = 'Æ ˜ ') **then** state = 13

else error("% wY7 Æ eA 7 – Æ @o<Yo 3w =<6 =<–∎")

end

else if (*input* == ')') **then** state = 14 with increment closeBracket

else if (*input* == '(') **then**

begin

state = 14 with increment openBracket

if (openBracket = =closeBracket) **then** state=13

else state=14

end

else if (*input* == '%' **) then** state = 14 with increment mod op

else state = 14

break

case 15**:**

forward *input* pointer up to mod(%) op

initialize openBracket && closeBracket

state = 16

break

case 16**:**

forward *input* pointer

if (*input* == **id** || **num**)**then**

begin

if (**id or num** type = = ' Æ¸ ˜ ‘) **then** state = 13

else error(”% wY7 Æ eA 7 – Æ @o<Yo 3w =<6 =<– ∎”)

end

else if (*input* == ‘(‘) **then** state = 17 with increment openBracket

else return true

break

case 17**:**

same check as **case** 13

break

end

b) Type checking of array index

Index checking is the most important part of the array type variable. Array index is called subscript of array. In the declaration section subscript value is the range of array and rest of the portion subscript value is element or position of array which must have integer type. *Mayabini* checks the subscript value of array variables for an expression weather element is integer type or not. In case of previous example it will generate an error:

```
Æ    ;
Æ  @ F ○1;
w ew()
{
      = c;
   @F .○1 = (  %●+  )%Q;
}
```

Mayabini has been used transition diagram to check the mod(%) operation which is in Appendix D.

Pseudo code for type checking of array index:

```
FUNCTION indexCheck(state : int) : boolean
begin
switch(state)
begin
    case 8:
        forward input pointer
        if ( input == num ) then
        begin
            if ( num type = = 'Æ  ') then ok
            else error(" m7 <o e3  w- 7: Æ  eA 7 =<6
            =<-▮")
        end
        else if ( input = = id ) then
        begin
            if ( id type = = 'Æ  ') then ok
            else error(" m7 <o e3  w- 7: Æ  eA 7 =<6
            =<-▮")
        end
        state = 9
        break
    case 9:
        forward input pointer
        if ( input = =']' ) then
            return true
        else if ( input = = '[' ) then error(" m7 <oo <76o m7 <o <Ao
          <- Y ▮")
```

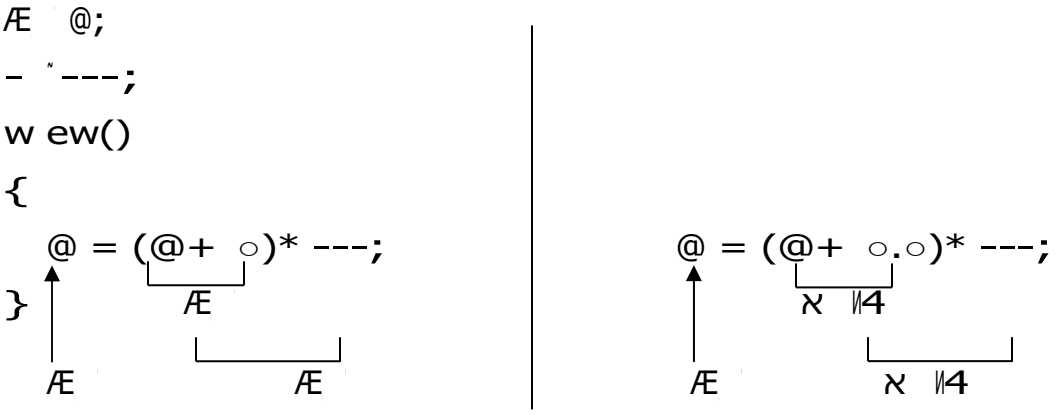
```
        else error("m7 <o e3   w- 7: Æ  eA 7 - 3w   =<6
        =<-▮")
end
end
```

3.4.4 Type checking of statement

Mayabini consider the four type of statement are assignment, input-output, iterative and selection statement. Each of these type statements are described below:

a) Type checking of assignment statement

The main purpose of type checking of the assignment statement is that to check the equality of left hand and right hand side type of the assignment operator. *Mayabini* save the type of left side variable at first. Then it checks the right side variable type one by one until get semicolon (;) from the lexical node. Then it matches the left and right side for assignment statement. The following example will be clear about that:



type match type mismatch

Mayabini has been used transition diagram to check the type of assignment statement whichis in Appendix D:

Pseudo code for type checking of assignment statements:

```
set left_type variable by id type
switch(state)
begin
        case 7:
                forward input pointer
                if ( input = ='[' ) then call indexCheck( )
                else state = 11
                break
        case 11:
                forward input pointer
                if ( input = = id || num ) then
                begin
                        if( id or num type = = 'א  ‖4   ' )then right_type = א  ‖4
                        else if(id or num type = ='Æ  ' && left_type = ='–  ') then
                        right_type= Æ
                end
                else if ( input = = '[' ) then call indexCheck( )
                else if ( input = = '%' ) then call modAnalyzer( )
                else if ( input = = ';' ) then state = 12
                else state = 11
                break
        case 12:
                if  (left_type = = right_type)  then state =0
                else error(" m  : C–‖ <G7o – <  7 Yo▮")
                break
end
```

b) Type checking of input output statement

In case of Input and output statement *Mayabini* checks the variable type is set or not to identify the undeclared variable and also checks the index of array variable is integer. If we consider the following two program:

Case 1:

```
Æ  7 ;
- ˜ --- ;
w ew()
{
  Æ?(---);
  Ww (@F 1);
}
```

@F 1 is undeclared variable here

Case 2:

```
Æ  @F 1 ;
- ˜ --- ;
w ew()
{
  Æ?(---);
  Ww (@F .○1);
}
```

@F .○1 is contained the floating point index although the variable id declared. So, in both cases *Mayabini* will generate an error.

Mayabini trace the above error by using the following transition diagram in figure 3.2:

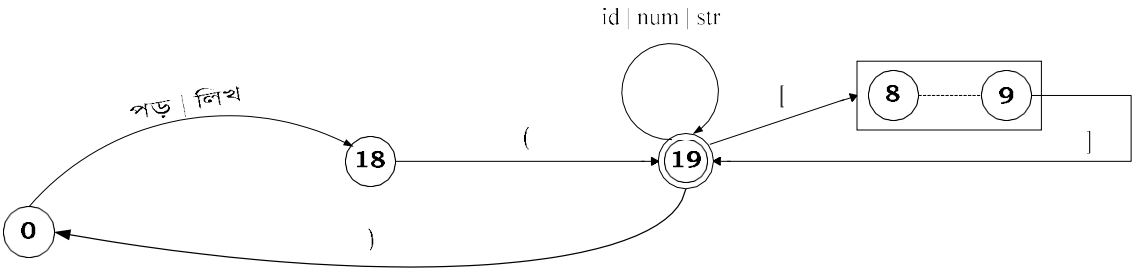


Figure 3.2: Transition diagram for input output statement

Pseudo code for type checking of input output statements:

```
switch(state)
begin
        case 18:
                forward input pointer
                state = 19
                break
        case 19:
                if (( input = = num || str ) || id = = not null ) then state = 19;
                else if ( input = ='[' ) then
                begin
                        call indexCheck( )
                        state = 19
                end
                else if ( input = =')' ) then state = 0
                break
end
```

c) Type checking of iterative statement

In the iterative statement *Mayabini* checks the mod operation of the expression and also checks the array index of the array variable. When *Mayabini* get any mod operator and an array variable in an expression of iterative statement it call the modAnalyzer(int state,Message msg) and indexCheck(int state,Message msg) function to check. Those two checks are discussed before in the type checking of mod operation and type checking of array index portion.

Mayabini has used transition diagram to check the iterative statement which has been addedin Appendix D:

Pseudo code for type checking of iterative statements:

```
switch(state)
begin
        case 21:
                forward input pointer
                state = 22
                breakf

        case 22:
                if ( input = = num || op || id ) then skip
                else if ( input = = '[' ) then
                begin
                        call indexCheck( )
                        state = 22
                end
                else if ( input = ='%' ) then
                begin
                        call modAnalyzer( )
                        state = 22
                end
                else if ( input = =')' ) then state = 0
        break
end
```

d) Type checking of selection statement

Type checking of selection statement is same as iterative statement. *Mayabini* has been used the transition diagram to check the selection statement which is in Appendix D. Pseudo code for type checking of selection statements is same as iterative statement

3.4.5 Type checking of function

Since *Mayabini* does not support the parameter passing to function and return from function, there is no type checking for passing parameter or returning value. *Mayabini* checks the function calling weather is valid or not. Let us consider with examples to verify how *Mayabini* checks error.

The correct program is:

```
Æ     F ○1;
w ew()
{
      - Aw ();
}
  - Aw ()
{
      ÇÇe – e – Aw <א   !!! ÇÇ
}
```

There are two possibilities to error in case of above function calling:

1) Any function can call which is not present in the program.

```
Æ     F ○1;
w ew()
{
      - Aw ();
}
```

Here - Aw (); is called in w ew() program but definition of - Aw () is not present here. So *Mayabini* will generate error.

2) Function name identifier can be variable identifier

```
Æ     F ○1;
w ew()
{
```

```
        ();
}
   - Aw ()
    {
      ÇÇe - e - Aw <א   !!! ÇÇ
    }
```

is an array variable identifier. So it is also an error for *Mayabini* compiler.

3.5 Code Generation

The most important part of a compiler is generating target machine code. It includes all types of memory allocation and run time evaluation. *Mayabini* has used Assembly code for generating target machine code i.e. *Mayabini* generates assembly code from a source *Mayabini* program. It uses INTEL 8086 instruction sets. *Mayabini* generates machine code in three different modules:

i. Expression Evaluation.
ii. Label Generator.
iii. Declaration.

Following sections describe about implementation details of the code generation phase.

3.5.1 Expression Evaluation

The heart of run-time system is the expression parser. Whenever *Mayabini* finds an expression then it sends the expression to the Expression Evaluation module generate assembly code for that expression. It does such kind of evaluation in two phases:

i. Firstly it converts the Infix Expression to Postfix Expression.
ii. Finally, it generates target code for Postfix Expression.

a) Getting the Postfix Expression

It uses the algorithm described at design part. Let us have a look on how the algorithm works. Consider the following pseudo code:

Code Listing:

```
1.void convertPostfix()
2.{
3.      Character c = new Character('(');
4.      stack.add(top,c);           //Add an opening brace at the top of stack.
5.      top++;
6.      for(int i=0;i<infix.size();i++)
7.      {
8.              Object sb1 = (Object)infix.get(i);  //Getting first element from infix.
9.              String s1 = sb1.toString();
10.             StringBuffer sb = new StringBuffer(s1);
11.             char ch = sb.charAt(0);
12.             switch(ch)
13.             {
14.                     case'+':checkPriorityPlus();
15.                     push(ch);
16.                     break;                          //Shows only for PLUS operator.
17.                     case'(':push(ch);
18.                             break;
19.                     case')':popAll();
20.                             break;
21.                     default:copy(sb);               //For Operand.
22.                             break;
23.}    }       }

24. void checkPriorityPlus()
25. {
26.      top--;
27.     Object ob1 = (Object)stack.get(top);
28.     String s1 = ob1.toString();
29.     StringBuffer sb = new StringBuffer(s1);
```

```
30.       char c1 = sb.charAt(0);
31.       if( c1=='-' || c1=='%' || c1=='/' || c1=='*')
32.       {
33.          for(k=top;c1!='&& c1!='e'&& c1!='G' && c1!='>' && c1!='<' && c1!='!'
34.          && c1!='+' && c1!='('; )
35.          {
36.                String sg = stack.get(k).toString();
37.                postfix.add(j,sg);
38.                j++;
39.                k--;
40.               Object ob2 = (Object)stack.get(k);
41.               String s2 = ob2.toString();
43.               StringBuffer sb2 = new StringBuffer(s2);
44.               c1 = sb2.charAt(0);
45.          }
46.          top=k;
47.          top++;
48.       }
49.      else
50.          top++;
51.  }
```

N.B. This code depicts the scenario only for '+' operator.

Let we have an expression 6 + 2 * 4 + 2. According to the code this expression will be written as 6 + 2 * 4 + 2). Now the following steps will be performed:

1. 6 has inserted at the Postfix Queue Q.
2. + has pushed at STACK.
3. 2 has inserted at the Postfix Queue Q.
4. * has pushed at STACK. Before it has pushed a checking should ensure that no operator having higher priority than * is on the STACK.
5. 4 has inserted at the Postfix Queue Q.

6. + has pushed at STACK. But before it * is popped from STACK at insert at Q.
7. 2 has inserted at the Postfix Queue Q.
8. + has pushed at Q.
9.) has found so al operator from STACK will be popped and insert at Q.
10. The resultant postfix expression was 6 2 4 * 2 + +.

b) Generating Target Assembly Code

16-bit assembly code (8086) has been used to write assembly code for *Mayabini*. Accordingto the algorithm *Mayabini* deals only with two operands and one operator at a time. It applies the operator on the operands and generates target code. While generating target code the operands may be a memory location (A declared variable.) or may be in the assembler stack. (8086 maintains own stack itself). Depending on this situation four things may happen that is reflected on the following code.

Code Listing :

```
1. void generateAssembly(String oprtr, String op1, String op2)
2.  if(oprtr.compareTo("+")==0)
3. {
4.       if( (op1.compareTo("pops")==0) && (op2.compareTo("pops")==0) )
5.       {
6.          pw.println("xor bx,bx");
7.          pw.println("pop  bx");      //Retrive value from Assembler's Stack --> op2.
8.          pw.println("mov temp,bx");
9.          pw.println("pop bx");       //Retrive value from Assembler's Stack --> op1.
10.         pw.println("add bx,temp");
11.         pw.println("push bx");
12.         push("pops");
13.      }
14.     else
15.     if(  (op1.compareTo("pops")!=0) && (op2.compareTo("pops")==0 ) )
16.    {
```

```
17.             pw.println("xor bx,bx");
18.             pw.println("pop  bx");       //Retrive value from Assembler's Stack --> op2.
19.             pw.println("mov temp, bx");
20.             pw.println("mov bx,"+op1);
21.             pw.println("add bx,temp");
22.             pw.println("push bx");
23.             push("pops");
24.     }
25,     else
26.     if(  (op1.compareTo("pops")==0) && (op2.compareTo("pops")!=0 ) )
27.     {
28.           pw.println("xor bx,bx");
29.           pw.println("pop  bx");       //Retrive value from Assembler's Stack --> op1.
30.           pw.println("add bx,"+op2);
31.           pw.println("push bx");
32.           push("pops");
33.      }
34.     else    //None of them have "pops"
35.     {
36.          pw.println("xor bx,bx");
37.          pw.println("mov bx,"+op1);
38.          pw.println("add bx,"+op2);
39.          pw.println("push bx");
40.          push("pops");
41.           }
42.       }
43. }
```

N.B. pw is a PrintWriter object (JAVA2) that is used to write on a file. This code depicts the scenario only for '+' operator.

Now considering the above code and algorithm (2.7, 2.8, 2.9, 2.10, 2.11, and 2.12) the assembly code will be:

1. XOR AX,AX
2. XOR BX,BX
3. MOV AX,2
4. MOV BX,4
5. MUL BX
6. PUSH AX
7. XOR BX,BX
8. POP BX
9. ADD BX,2
10. PUSH BX
11. XOR BX,BX
12. POP BX
13. MOV TEMP, BX
14. MOV BX,6
15. ADD BX,TEMP
16. PUSH BX

N.B. All instructions are 8086 instruction. [9] [16] [17]

When *Mayabini* found an operator it popped two consecutive operands from stack and depending on the operator wrote the code. As far as mathematical rule was concerned the multiplication will be done first (line 5). Then the result was added with next operand (line 9). Finally the summation was added with the last operand (line 16.)

3.5.2 Label Generator

In an *Mayabini* source program there can be three types of label. They are:

i. Loop structure label
ii. Conditional structure label
iii. Function calling label

The special attention has to be maintained on writing assembly file is that 8086 does not support multiple labels with same name.

Consider the following *Mayabini* source program:

***Mayabini* Source File:**

```
 . Æ  @;
Q. w ew()
c. {
8.    @ = 8;
•.    ϒ (@>= )
 .    {
 .          ⅈא (@==Q)
 .         {
².              ⅈw  (“ⅈÆo – Aw <א  ”);
 ○.         }
  .  }
 Q.}
```

This program will print “ⅈÆo – Aw <א ” only once. Now according to the label generation algorithm assembler will generate the following code:

```
1.  .model small
2.  .stack 100h
3.  .data
4.  temp dw 0
5.  msg0 db '  wcªq evsjv‡`k ',0dh,0ah,'$'
6.  a dw 0
7.  .code
8 .  include eks_out.asm
9.   main proc
10.  mov ax, @data
11.  mov ds, ax
```

```
12. mov a,4
13. while1:
14. mov bx,a
15. cmp bx,1
16. jge label0
17. mov bx,0
18. push bx
19. jmp label1
20. label0:
21. mov bx,1
22. push bx
22. label1:
23. pop bx
24. cmp bx,1
25. jne end_while1
26. mov bx,a
27. cmp bx,2
28. je label2
29. mov bx,0
30. push bx
31. jmp label3
32. label2:
33. mov bx,1
34. push bx
35. label3:
36. cmp bx,1
37. jne else1
38. lea dx,msg0
39. mov ah,9
40. int 21h
41. jmp endif0
```

42. else1:
43. endif0:
44. jmp while1
45. end_while1:
46. mov ah,4ch
47. int 21h
48. main endp
49. end main

N.B. ”@“ has interpreted as “a” and ”wÆo – Aw <א “ has interpreted as “wc^{a}q evsjv‡`k” at assembly according to BIJOY Bangla specification. The algorithms for generating all labels have been described 2.5.3

3.5.3 Declaration

The declaration parts include all types of variable declaration even quoted string. Consider the *Mayabini* Source code in section 3.5.2 the corresponding code segment of assembly file will be following:

1. .DATA
2. a DW ?
3. msg0 DB 'eee',0dh,0ah,'$'

N.B. @ is interpreted as ‘a’ and --- is interpreted as ‘eee’ at assembly according to BIJOY Bangla specification.

Line No. 2 declares the variable @ and line no. 3 declares the quoted string ---.

3.6 Summary

So far, we have discussed in this chapter, how we have implemented the *Mayabini* compiler. We hope the attentive reader has understood this complex task fully. In next chapter we will discuss about the different types of tools we have used to implement *Mayabini*.

Chapter 4

Tools and Editor Used for *Mayabini*

4.1 Introduction

This chapter includes the tools that have been used to implement *Mayabini*. By reading this chapter, the reader will get a full knowledge about the type of tools have been used to implement *Mayabini*. He/she will also understand why a particular tool has been preferred. He/she will also know about the editor which has been especially built for compiling and running source *Mayabini* program and different features about the editor.

4.2 Tools

The following languages and tools have been used to develop *Mayabini*. Most of the tools are free (except Bijoy). So, anybody can go for any further extension without any cost.

4.2.1 JAVA2 (J2SE)

JAVA2 have been used for coding the *Mayabini* compiler. The reasons behind using JAVA2 are as follows:

i. JAVA2 is the first language which successfully supports Unicode. Unicode is a 16-bit character set which includes all the alphabets currently available in different natural language. Though we didn't use Unicode, but it can be extended to Unicode. Then our language will not be Bijoy dependent. Moreover both Bangla and English alphabet can be added to *Mayabini* alphabet. So the programmer will able to use both the Bangla and the English characters in the same program.

ii. *Mayabini* is an open source language. It is possible to add byte code in *Mayabini* like JAVA2. Then it will be a platform independent language. If both the language and compiler is platform independent then it will be very easy to improve *Mayabini* for the programmers. Thus one day it may be a popular language like C.

iii. Object oriented approach was used for developing this project. Since object oriented approach increases reusability of code, so it will be helpful for those who want to improve it. Regarding these issues we used JAVA2 which is a popular object oriented language.

4.2.2 Turbo Assembler

Turbo Assembler has been used for running assembly file of the corresponding source language. We used 8086 micro-processor instructions set.

4.2.3 Microsoft Visual C++

Microsoft Visual C++ has been used because we have implemented JNI (JAVA2 Native Interface).

4.2.4 Bijoy and SutonnyEMJ font

Bijoy 2000 pro was used for writing *Mayabini* source language. Bijoy is widely accepted software for writing in Bangla. We also used SutonnyEMJ font for writing source language.

4.2.5 IntellIJ

IntellIJ has been used for debugging purpose only.

4.2.6 TextPad

Most of the source code has been written using TextPad.

4.3 *Mayabini* Editor

An interactive graphical editor has been built using JAVA2 for writing, compiling and running *Mayabini* source language. It has some menus, command buttons, an input text area for writing source language and another output text area for displaying error messages. These are given below:

i. V :w (File Menu): It has four sub menus. Y6Y to open a new file to write a new program. < w to open an existing file, -d o to close an already opened file and ⅈ–א o to exit from the *Mayabini* editor.

ii. *w (Tool Menu): It has two sub menus. First MÆ :w for compiling a program and another o Y for running the compiled program.

iii. e = 7 (Help Menu): There are two sub menus here. w 4 <אo m < is used for supplying information about *Mayabini* editor and =* ⅈ gives a list of hot keys.

iv. Y6Y (New Button): Y6Y button to open a new file to write a new program.

v. < w (Open Button): to open an existing file. Hot key is f3.

vi. eAoø (Save Button): to save a file. Hot key is f2.

vii. MÆ :w (Compile Button): to compile a program. Hot key is f5.

viii. o Y (Run Button): to run the compiled program. Hot key is f9.

ix. e = 7 (Help Button): for information. Hot key is f1.

x. ⅈ–א o (Exit Button):to exit from the *Mayabini* editor. Hot key is alt+f4.

xi. :YÆ* (Input Button): The user will give the necessary input for the program by clicking this button.

Some key features of the editor are given below:

i. All labels of menus, buttons are written in Bangla. Both input and output text area use Bangle. These components are user friendly.

ii. A user can open a new file as well as an existing file. He/she can edit, save, compile and run it.

iii. A user can use a mouse as well as hotkeys through keyboard to give a command.

iv. There is a help menu. A user can taking help and know more about *Mayabini* features.

v. All the buttons have hot keys for easy use.

4.4 Linking and Loading

After writing a source program the programmer want to execute the source program. The execution of a source *Mayabini* program can be divided into two major parts.

i. Compiling.
ii. Running.

4.4.1 Compiling

In this phase *Mayabini* have checked whether there are any logical errors in the sourcecode. Three things happen in this phase:

1. Make the Symbol Table and Lexical Table.
2. Check whether there is any syntax error or not.
3. Check whether there is ant semantics error or not.

If any phase fails *i.e.* generates errors then its successor phases do not execute.

4.4.2 Running

In this phase *Mayabini* create the assembly file which is a true assembly version of sourcefile. The following things happen in this phase:

1. Create the Assembly file.
2. Create a batch file to execute the assembly file.
3. Execute the batch file.

The last step is done by using JNI (JAVA2 native interface). The source code for executing the batch file has been written using Visual C++.

4.5 Summary

In this chapter, we have discussed the various tools that have been used for implementing *Mayabini.* Besides, we have also briefly discussed about the editor that we have developed for giving supports in a user friendly way for writing codes.

Chapter 5

Features of *Mayabini* Language

5.1 Introduction

Every language has some features which make it more acceptable over other languages. The programming languages are evolving continuously. So a new language inherits some popular features of old languages and it also builds its own. Thus, it enriches its features. Most of the feature of *Mayabini* is inherited from popular language C. This chapter will help the reader to understand the full features of *Mayabini*.

5.2 Level of Language

In the late 1940 and early 1950s, there was no programming language or even Assembly language. Programmers used to program using machine code, which was really difficult, tedious and error-prone.[11].To reduce these problems, Assembly, FORTRAN, ALGOL 58 and other languages came. These languages can be divided into the following two broad categories:

i. Low Level Language: These languages were designed to give better machine efficiency, i.e. faster program execution. They are machine oriented, e.g. Assembly and Machine language.

ii. High level Language: They are more like natural languages (English, Bangle, etc). These languages were designed to give better programming efficiency, i.e. faster program development. [4]

According to these categories, *Mayabini* falls into High level Language. Because it is more like Bangle language and a programmer can program more efficiently using it.

5.3 Genealogy of *Mayabini* Language

The following figure shows genealogy of *Mayabini* Language [4][11] Here we see that *Mayabini* has been directly inherited from C. Some other languages like C++, JAVA2 and C# also inherited from C.

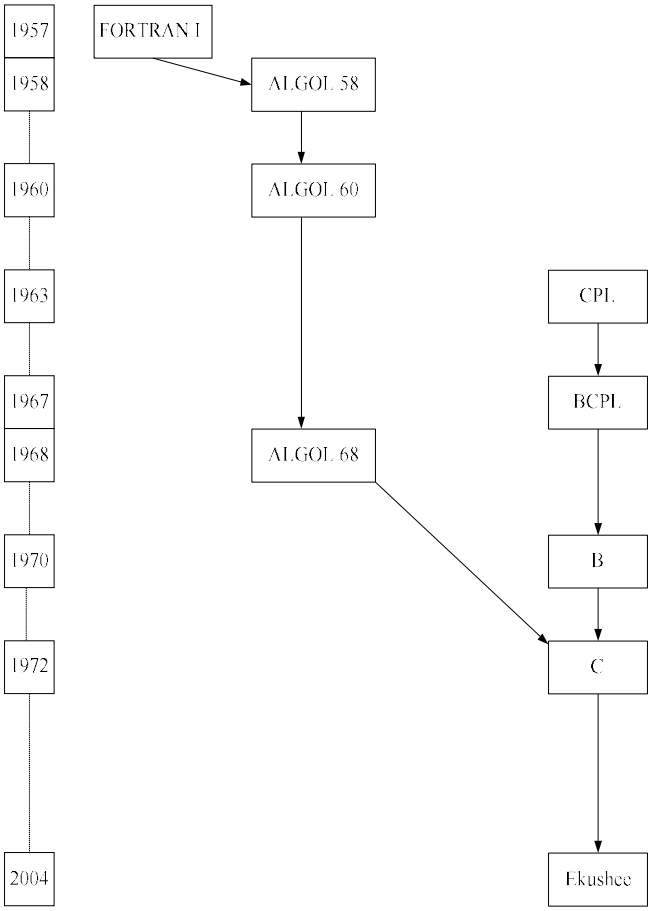


Figure: - Genealogy of *f* Language.

5.4 Names or Identifiers

"A name is a string of characters used to identify some entity in a program."[11]**.** For *Mayabini* it is related with variables, procedure, e.g. m is a name. Some key featuresof name are given below:

i. It must start with a character followed by any number of character or digit.
ii. Connector character (_)can be used in name.
iii. The length of the name is virtually unlimited.

5.5 Reserved Words

“Reserved word is a special word of a programming language that can not be used as name.”[11] It is very useful for programming languages. It increases the readability. *Mayabini* has some reserved words. These are given below:

```
w"-      w ew      m-A      3w
" 4      א  4      Æ?       Æ
-  "     -           Y      4א
4w
```

5.6 Variables

Variables play an important role for any programming language. This variable concept comes from mathematics. Normally, variable means an entity which can hold different value at different time, but in programming language it is more than that. ”A program variable is an abstraction of a computer memory cell or collections of cell.”[11]

5.6.1 Variables of *Mayabini*

All of the variables of *Mayabini* are static, i.e. it is bound to memory location at compile time and remains unchanged throughout the program execution. Variables in *Mayabini* must be declared by an explicit declaration statement in the top of the † function,

e.g.

```
א  4   w w;
Æ  @F  ○○1;
Æ  @ Y;
-  "- - ;
w ew()
{
      . . .
}
```

All variables default value is 0, and it implicitly initialized by the compiler. The length of *Mayabini* variable can be virtually infinite.

5.6.2 Scope

Scope means the range of statement where the variable is accessible [11]. Since all variables of *Mayabini* are static and declared globally so they are accessible from anywhere in the program.

5.6.3 Life Time

Life time of a variable means period of time a variable resides in memory. Since all variables of *Mayabini* are static and declared globally so their life time begins when the execution of the program starts and ends when the program terminates.

5.7 Data Types

Each computer programs compute results by manipulating data. Therefore, it is very much crucial that a language should support an appropriate variety of data types and structures [11] *Mayabini* has the following data types

5.7.1 Primitive Data Types

Primitive data types are those which can not be defined in terms of other data types. [11]. *Mayabini* has the following three primitive data types:

i. Integer (Æ): It is the most common data type for all programming language. In *Mayabini* 2 bytes are used for integer (Æ). Its range is within +32767 to -32768. A programmer can perform addition, subtraction, division, multiplication and mod operation between two integer ( ) types.

ii. Float (א 4): It is used to kept fraction value. Many hardwares don't support it. In *Mayabini* 4 bytes are used for float(א 4). Its range is within 2 byte. A programmer can perform addition, subtraction, division, and multiplication between two float(א 4) types.

iii. Character (–): It is used to keep character type data. *Mayabini* has 1 byte for character (–). Its range is within +127 to -128. A programmer can perform addition, subtraction, division, multiplication character (–) types.

iv. Boolean (e67/4"7): *Mayabini* has no Boolean type data like C. It uses zero (0) for a false(4"7) and non-zero for true (e67).

5.7.2 Array

Array is a collection of homogeneous data which are stored contiguously in the memory. Specific element of an array is accessed by means of a two-level syntactic mechanism, where the first part is aggregate name and second part is index [11]. Array of *Mayabini* may be all possible primitive data types. Array of *Mayabini* has the following characteristics

i. It is static. So subscript ranges and storage are allocated at compile time. Its size can not be changed at run time.
ii. Only dimension array is supported.
iii. Range of an array is not checked.
iv. Array cannot be initialized at declaration period.
v. Array index must be integer (Æ) type.

5.7.3 String Types

Many languages like JAVA2 use string as a primitive type. But *Mayabini* uses array of character to store string. The string length is fixed so the length of the string can not be changed at run time. A whole string can be initialized to a string array at a time.

5.8 Expressions

In General, every expression returns some value. So it is very crucial for a programmer to understand both the syntax and semantics of expressions. It is also fundamental means of specifying computation in a programming language. *Mayabini* has the following expressions

5.8.1 Arithmetic Expressions

Most of the features of *Mayabini* arithmetic expressions are inherited from mathematics. These features are given below:

i. The operator precedence rules (hierarchy of operators) of *Mayabini* have been taken from mathematics. It is similar to C. The complete rules have been given in Appendix E.

ii. Operator associativity rule is from left to right. When two operator of same precedence comes adjacently in an expression, they are evaluated from left to right.

iii. Operand evaluation order is from left to right. So, first the leftmost operand will be fetched from memory, then next one and so on.

5.8.2 Relational Expressions

A relational operator compares the values of its two operands. For relational expression *Mayabini* has either a non-zero value (true) or a zero value (false). The following relational operators are used in *Mayabini*

```
>      >=     <
<=     ==     !=
```

5.8.3 Logical Expressions

Mayabini logical operators return either a non-zero value (e67) or a zero value (M4"7) depending on the expression. *Mayabini* has the following logical operators:

```
m-A        -
```

5.8.4 Evaluation of Expressions

Mayabini evaluates a full expression like Pascal. For example

```
Mx((      >      R) m-A(44 > 44R))
{
    ...
}
```

If “ ” is less than or equal to “ R” .i.e. the value first sub-expression is false then *Mayabini* also evaluate the next part (4 4 > 4 4R) to get the final value which is ultimately false.

5.9 Type Conversions

Mayabini allows only widening type conversion. So an integer (Æ) type can be converted to float (4) type, but no vice versa. This type conversion is implicit or coercion, i.e. compiler automatically converts a small type to a wide type.

5.10 Statements

Statement is the basic part of a program. *Mayabini* has the following types of statements.

5.10.1 Assignments Statement

Mayabini supports single assignment as well as mixed mode assignments, i.e. it is possible to include integer, float and character type data to a single statement.

```
ϰ 4  w;
Æ  @F ○○1;
Æ  7;
- ˟---;
w ew()
{
      7= ○; --- = ′ ′; @F 1= - •○;
      w= w* ○○+ 7* @F 1/ ---;
}
```

So the above program is a valid *Mayabini* program.

5.10.2 Function Calling Statement

A function can be called very easily in *Mayabini* just using the name of the function, e.g.

```
w ew()
{
      w w();
}
w w()
{
      . . .
}
```

5.10.3 Selection Statements

Mayabini if(ⅈא) – else(w"–) statement for to allow branching in a program. For example

```
.  .  .
 ⅈא(@>  ○)
{
      @= ○;
}
w"–
{
      @=Q○;
}
.  .  .
```

The above program allowed branching in program. If @> ○ true then program will flow in a path else it will go another direction.

5.10.4 Iterative Statement

An iterative statement is one that causes a statement or collection of statements to be executed 0 or more times. So it is a very important and powerful tool for writing code. It increases efficiency for array handling. *Mayabini* uses only a single iterative statement for loop. It use while (ϒ) reserved word for looping. For example

```
. . .
   Y(@> ○)
{
      . . .
}
. . .
```

This loop will execute until @ is less then ○.

5.10.5 Input Output Statements

Mayabini uses Ww and Æ? function respectively for write and read something on or from console. A programmer can read as well as write any type of primitive data types, i.e. integer ( ), float(א 4) and character (-) with this two statements.

5.10.6 Compound Statements

A compound statement is a collection of more than one statement. These statements are joined together using curly brackets ({ }). For example

```
.  .  .
 Wא(@>  ○)
{
      @= ○;
      w=ø+ ;
}
.  .  .
```

5.10.7 Variable declaration Statements

All variables of *Mayabini* are declared explicitly at the top w ew function. For example:

```
א 4  w;
Æ 7;
- ˜---;
```

5.11 Functions of *Mayabini*

Mayabini has function. A programmer can call a function in side other function. The function definition must be come after the w ew function. Some features of *Mayabini* function

i. Its name can not be a reserved word. Its length can be virtually infinite.
ii. Each function has only single entry point.
iii. Only one function can be executed at a time.
iv. Control returns to caller when the function terminates.
v. Nothing is accepted as parameter and nothing is returned.

5.12 Summary

Though *Mayabini* has few alternatives but it has covered most of the features of contemporary programming language. As a first Bangla compiler, we have tried to add as much features as we can. These features can be easily extended and enhanced by interested developers.

Chapter 6

Conclusions

6.1 Introduction

The overall design and implementation of the present work have been tried to present in the previous chapters. This last chapter is intended to have some discussion on the quality of the work, how *Mayabini* performs in comparison to other products of similar nature. The interested readers will also get a comprehensive knowledge about the scope for future works.

6.2 Discussions

Mayabini can be used for write any types of algorithms except recursive. As this is the first Bangla compiler it has some weakness. The major weaknesses are:

i. It has designed for text-mode application.
ii. It can not be designed as platform independently.
iii. It has very few built-in functions.
iv. Numbers of primitive data types are very few.
v. Very common arithmetical and logical operators are supported though all arithmetical and logical calculation can be done.

6.3 Performance of *Mayabini* in comparison to others

In fact there is no Bangla Compiler at present. So we can not compare the performance of *Mayabini* with any of this arena. A Bangla interpreter has been designed at University of Dhaka but as an interpreter that have used C as the intermediate language and obviously very slow. If *Mayabini* is compared with other compilers it is clearly a far behind. On the other hand *Mayabini* is ahead because very few compilers have been completely developed in mother tongue of any nation.

6.4 Strength of *Mayabini*

Some major strengths of our compiler are given follows:

i. The main strength of *Mayabini* is that the Bangladeshi programmer can write program in their mother tongue Bangla. As the most of the people in our country are relatively weak in English, they will feel comfort to do coding in our Bangla compiler.

ii. Any kind of complex algorithm can be coded using this compiler except recursive algorithm.

iii. A very good graphical user interface (GUI) has been provided for user to write, compile and run source programs.

iv. The error messages, input, output, messages are also provided in Bangla.

6.5 Suggestions for Future Work

This thesis has been completed within a semester (4 months). But we got actually about 107 working days. This time is too short to do a good research work especially like a new compiler design. Yet we have tried our best to accomplish this project. *Mayabini* compiler can be extended in the following directions:

i. Though we have supported floating point operation, (continue), t (break) statement and **!** (not) operator in our grammar but could not implement due to time constraint. But these can be easily implemented within a short period.

ii. The I/O features can be improved and different built-in I/O functions can be supported by a reasonable effort.

iii. Different library functions can be developed for performing string operations, array operations, for graphics, mathematics and for other purposes.

iv. We could not support parameter passing to and returning value from a function. It can be easily implemented by modifying the grammar.

v. All of the *Mayabini* variables are static and there is no local variable for functionor statement blocks. Some improvement can be done in this field. Thus, the variables can be extended to stack dynamic (most of the C variable), explicit heap dynamic (variable declared using "new" in C++ and JAVA2) and implicit heap dynamic. To implement this feature we have to explicitly handle the memory in run time.

vi. It is possible to support user defined data types like structure, union, enumeration and multi-dimensional array by changing the grammar. More over pointers and reference type can be also implemented. This feature can be added by slightly changing the grammar.

vii. Unicode can be used to represent Bangla. Thus both Bangla and English can be written in the same program.

viii. User will get more flexibility if different increment (++) and decrement (--) operators are supported.

ix. Low level operations like bit-wise **and or**, **shifting, rotating**, **xor** and **nor** can be easily implemented by changing the grammar.

x. It is possible to implement an intermediate byte code like JAVA2 then both the compiler and *Mayabini* will be platform independent. For this feature, we have to design a unique intermediate representation for each statement.

xi. Different code optimization methods and efficient searching algorithms for symbol Table and lexical Table ```can be adopted which will improve performance of *Mayabini* compiler.

xii. Keeping the main structure unchanged, *Mayabini* can be extended to support Object Oriented Approach, multithreading, and web development.

xiii. Using the same parsing technique, a Bangle translator can be built for natural language processing, which will convert an English text to corresponding Bangla text.

6.6 Summary

Programming in Bangla is no more a dream, it comes to reality. *Mayabini* is a very easy language. Anybody who has a bit experience in any programming language can do programming in *Mayabini*. This may be the first step to design a more sophisticated Bangla programming language. Moreover, this may be a revolutionary change in the field of development of a Bangla operating system as well as Bangla software because the present Bangla operating system are just a Bangla mask in front of English operating system.

Dream is unlimited. But there should be the beginning of a dream. We believe *Mayabini* is the first stair of the dream of Bangla computing. We expect that optimistic researchers will participate in the further enhancement of *Mayabini*.

Appendices

Appendix A: Grammar of *Mayabini*

1. *prog* → *decl main_fun fun*
2. *decl* → *type* **id** *A ; decl* | ε
3. *A* → **[num]** | ε
4. *type* → **int** | **char** | **float**
5. *main_fun* → **main ( )** *comp_stmt*
6. *fun* → *fh comp_stmt fun* | ε
7. *fh* → **id ( )**
8. *comp_stmt* → **{** *stmt_list* **}**
9. *stmt_list* → *stmt stmt_list* | ε
10. *stmt* → *select_stmt* | *iter_stmt* | *as_fun_stmt* | *io_stmt* |*jump_stmt* | *comp_stmt*
11. *select_stmt* → **if (** ***e****xpr_list* **)** *stmt stmt′*
12. *stmt′* → **else** *stmt* | ε
13. *iter_stmt* → **while (** *expr_list* **)** *stmt*
14. *as_fun_stmt* → **id** *B*
15. *B* → **( ) ;** | = *C* | **[** *Q*
16. *Q* → *expr_list* **]** = *C* | **]** = **str ;**
17. *C* →*expr_list* **;** | **letter ;**
18. *io_stmt* → **printf (** *D* | **scanf (id E) ;**
19. *D* → **str) ;** | **id** *E* **);** | **num) ;**
20. j*ump_stmt* → **break ;** | **continue ;**
21. e*xpr_list* → *M expr_list′*
22. *expr_list′* → **||** *M expr_list′* | ε
23. *M* → *N M′*
24. *M′* → **&&** *N M′* | ε
25. *N* → *O N′*
26. *N′* → *eq O N′* | ε

27. O → expr O′

28. *O′ → relop expr O′* | ε

29. expr → uniop term expr′

30. *expr′→ addop term expr′* | ε

31. term → factor term′

32. *term′→ mulop factor term′* | ε

33. *factor* → **id** *P* | **num** | **(** *expr_lsit* **)**

34. *P*→ [*expr_list*] | ε

35. *uniop* → + | - | **!** | ε

36. *addop* → + | -

37. *mulop* → * | **%** | /

38. *relop*→ > | >= | < | <=

39. *eq* → = = | **!=**

40. *E* → [*expr_list*]

| ε**letter** □ I w -T

digit □ **[** ○ - ² **]**

id □ **letter (letter | digit)***

digits □ **digit digit***

num □ **(** + | - | s**) digits ((. ?) digits) ?**

str □ **" (letter | digit)* "**

The English keywords and corresponding Bangla keywords mapping are given below

English terminals	Bangla keywords
int	Æ
char	–
float	২ ৷4
main	w ew
if	৷২
else	w"–

while	Y
printf	⁄w
scanf	Æ?
break	" 4
continue	3w
&&	m–A
‖	–

Appendix B: FIRST and FOLLOW

FIRST Set:

1. FIRST(*prog*) = { **int, char, float, main** }
2. FIRST(*decl*) = { **int, char, float,** ε }
3. FIRST(*A*) = { **[,** ε }
4. FIRST(*type*) = { **int, char, float** }
5. FIRST(*Main_Fun*) = { **main** }
6. FIRST(*fun*) = { **id,** ε }
7. FIRST(*fh*) = { **id** }
8. FIRST(*comp_stmt*) = { **{** }
9. FIRST(*stmt_list*) = {**if, while, id, printf, scanf, break, continue, {,** ε }
10. FIRST(*stmt*) = { **if, while, id, printf, scanf, break, continue, {**}
11. FIRST(*select_stmt*) = { **if** }
12. FIRST(*stmt´*) = {**else,** ε }
13. FIRST(*iter_stmt*) = { **while** }
14. FIRST(*as_fun_stmt*) = { **id** }
15. FIRST(*B*) = { **(, =, [** }
16. FIRST(*Q*) = { **+, -, !, id, num, (,]** }
17. FIRST(*C*) = { **+, -, !, id, num, (,], letter** }
18. FIRST(*io_stmt*) = { **printf, scanf** }
19. FIRST(*D*) = {**str, id, num**}
20. FIRST(*jump_stmt*) = { **break, continue** }
21. FIRST(*expr_list*) = { **+, -, !, id, num, (,** }
22. FIRST(*expr_list´*) = { **+, -, !, id, num, (,** ε }
23. FIRST(*M*) = { **+, -, !, id, num, (,** }
24. FIRST(*M´*) = {**&&,** ε }
25. FIRST(*N*) = { **+, -, !, id, num, (,** }
26. FIRST(*N´*) = {**==, !=,** ε }
27. FIRST(*O*) = { **+, -, !, id, num, (,** }
28. FIRST(*O´*) = { **>, >=, <, <=,** ε }

29. FIRST(*expr*) = {**+, -, !, id, num, (**,}
30. FIRST(*expr´*) = {**+, -,** ε }
31. FIRST(*term*) = { **id, num, (** }
32. FIRST(*term´*) = { ***, /, %,** ε }
33. FIRST(*factor*) = { **id, num, (** }
34. FIRST(*P*) = { **[,** ε }
35. FIRST(*uniop*) = {**+, -, !,** ε }
36. FIRST(*addop*) = {**+, -**}
37. FIRST(*mulop*) = {***, /, %**}
38. FIRST(*relop*) = { **>, >=, <, <=** }
39. FIRST(*eq*) = {**==, !=**}
40. FIRST(*E*) = { **[,** ε }

FOLLOW Set:

1. FOLLOW(*prog*) = { **$** }
2. FOLLOW(*decl*) = { **main** }
3. FOLLOW(*A*) = { **;** }
4. FOLLOW(*type*) = { **id** }
5. FOLLOW(*main_fun*) = { **id**, **$** }
6. FOLLOW(*fun*) = { **$** }
7. FOLLOW(*fh*) = { **}**}
8. FOLLOW(*comp_stmt*) = { **id**, **$** }
9. FOLLOW(*stmt_list*) = {**}** }
10. FOLLOW(*stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
11. FOLLOW(*select_stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
12. FOLLOW(*stmt´*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
13. FOLLOW(*iter_stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
14. FOLLOW(*as_fun_stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
15. FOLLOW(*B*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
16. FOLLOW(*Q*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
17. FOLLOW(*C*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}

18. FOLLOW(*io_stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
19. FOLLOW(*D*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
20. FOLLOW(*jump_stmt*) = { **if, while, id, printf, scanf, break, continue, {, }, else**}
21. FOLLOW(*expr_list*) = { **], ;,)** }
22. FOLLOW(*expr_list´*) = { **], ;,)** }
23. FOLLOW(*M*) = { **||,], ;,)** }
24. FOLLOW(*M´*) = {**||,], ;,)**}
25. FOLLOW(*N*) = { **&&, ||,], ;,)** }
26. FOLLOW(*N´*) = {**&&, ||,], ;,)** }
27. FOLLOW(*O*) = { **==, !=, &&,||,], ;,)** }
28. FOLLOW(*O´*) = { **==, !=, &&,||,], ;,)** }
29. FOLLOW(*expr*) = { **>, >=, <, <=,==, !=, &&,||,], ;,)** }
30. FOLLOW(*expr´*) = { **>, >=, <, <=,==, !=, &&,||,], ;,)** }
31. FOLLOW(*term*) = {**+ ,-, >, >=, <, <=,==, !=, &&,||,], ;,)** }
32. FOLLOW(*term´*) = {**+ ,-, >, >=, <, <=,==, !=, &&,||,], ;,)** }
33. FOLLOW(*factor*) = {***, /, %, + ,-, >, >=, <, <=,==, !=, &&,||,], ;,)** }
34. FOLLOW(*P*) = {***, /, %, + ,-, >, >=, <, <=,==, !=, &&,||,], ;,)** }
35. FOLLOW(*uniop*) = { **id, num, (** }
36. FOLLOW(*addop*) = { **id, num, (** }
37. FOLLOW(*mulop*) = { **id, num, (** }
38. FOLLOW(*relop*) = { **+, -, !, id, num, (** }
39. FOLLOW(*eq*) = { **+, -, !, id, num, (** }
40. FOLLOW(*E)* = { **),** **if, while, id, printf, scanf, break, continue, {, }, else** }

Appendix C: Transition Diagrams for Lexical Analyzer

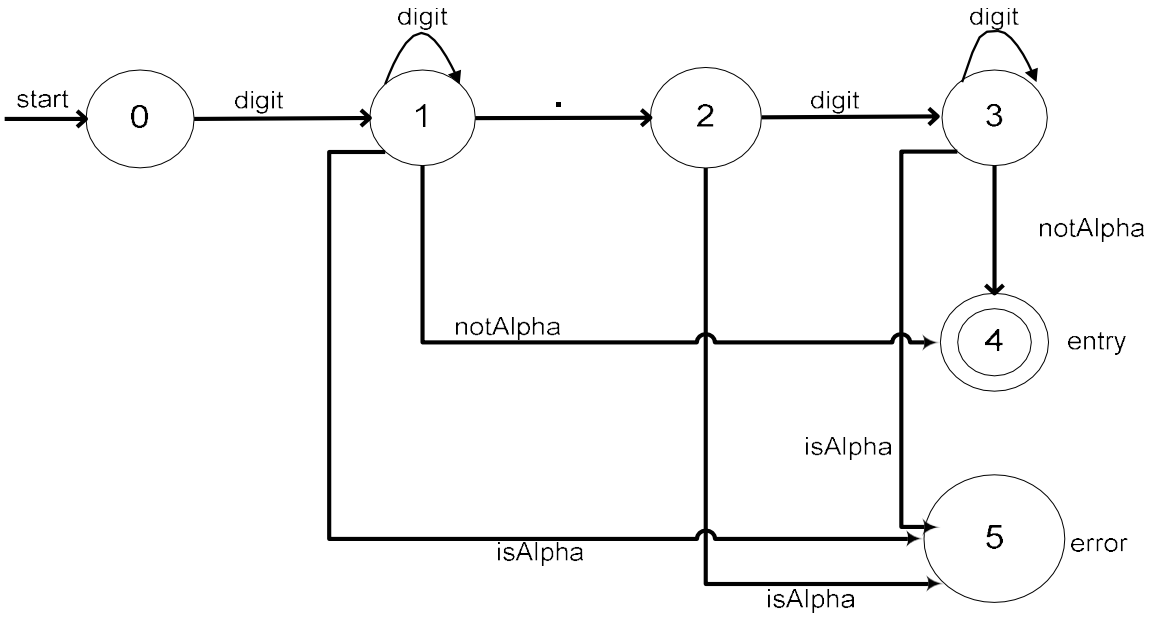


Figure: DFA for pattern matching of any digit.

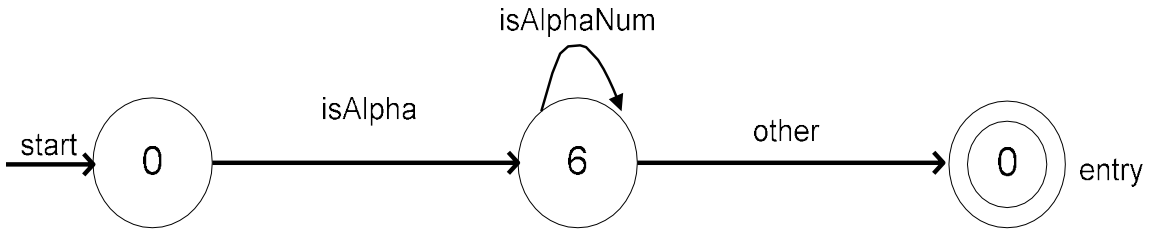


Figure: DFA for pattern matching of ID & Keyword

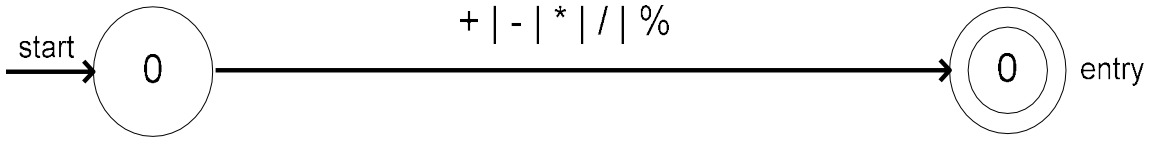


Figure: DFA for pattern matching of Operators. (+,-,*,/,%)

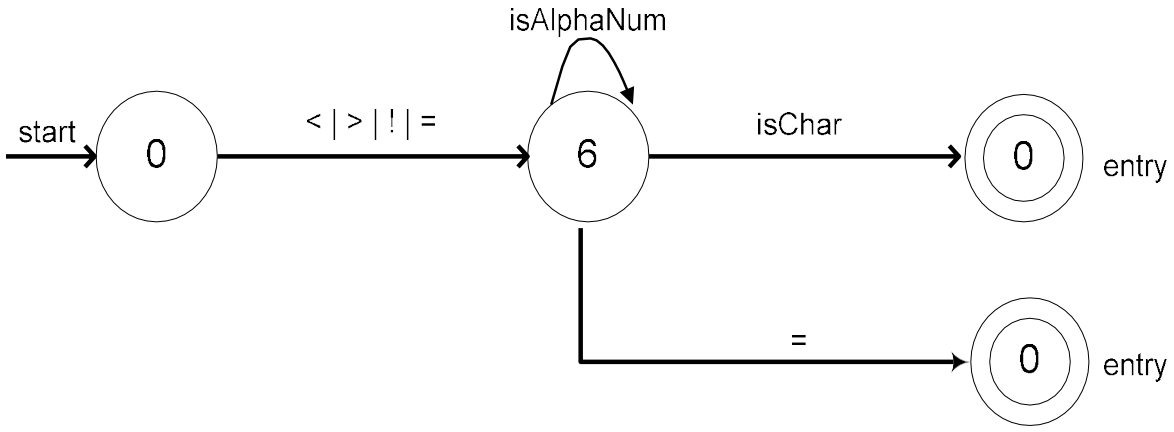


Figure: DFA for pattern matching of Relational Operators. (<.>,!,=,<=,>=,!=,==)

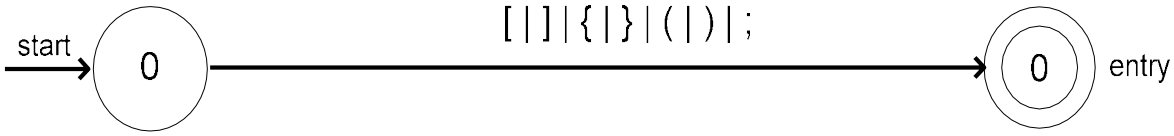


Figure: DFA for pattern matching of punctuation. ([,] , , , (,), ;)

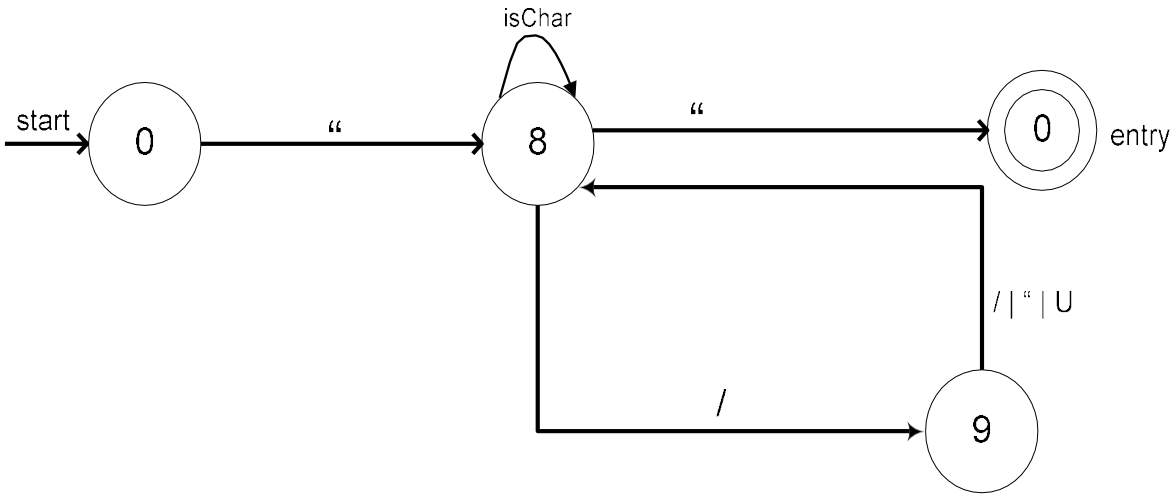


Figure: DFA for pattern matching of any String Literal. (“….”)

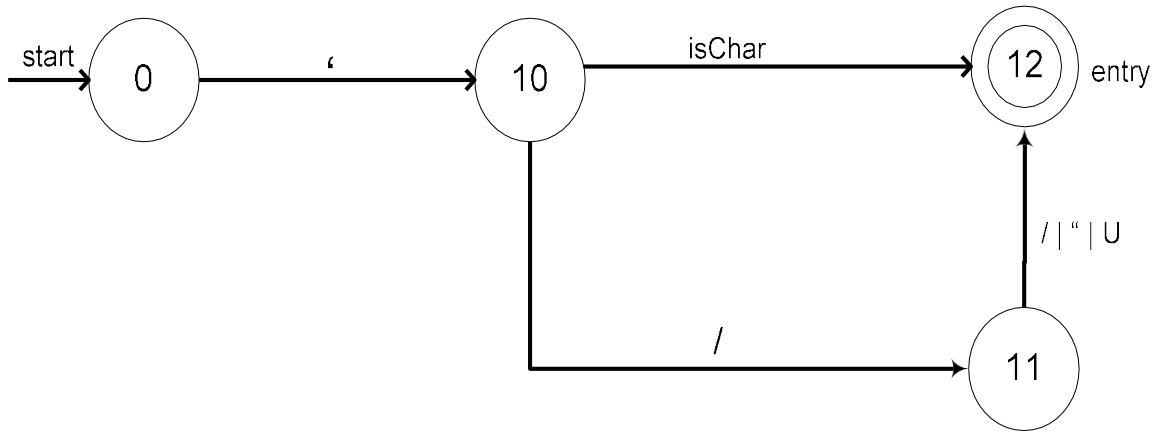


Figure: DFA for pattern matching of any Character. (‘.’)

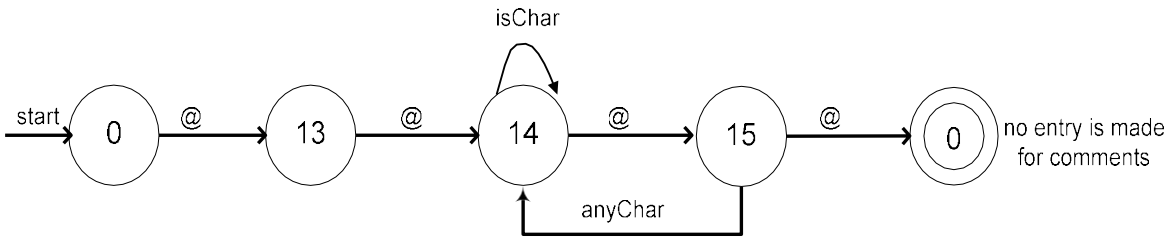


Figure: DFA for pattern matching of Comments. (@@...@@)

Appendix D: Transition Diagrams for Semantic Analyzer

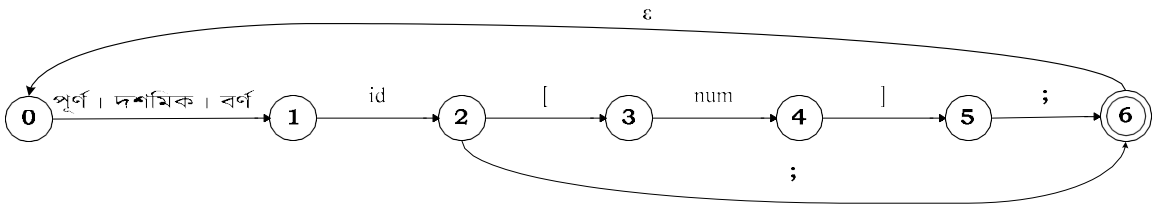


Figure: Transition diagram for declared statement

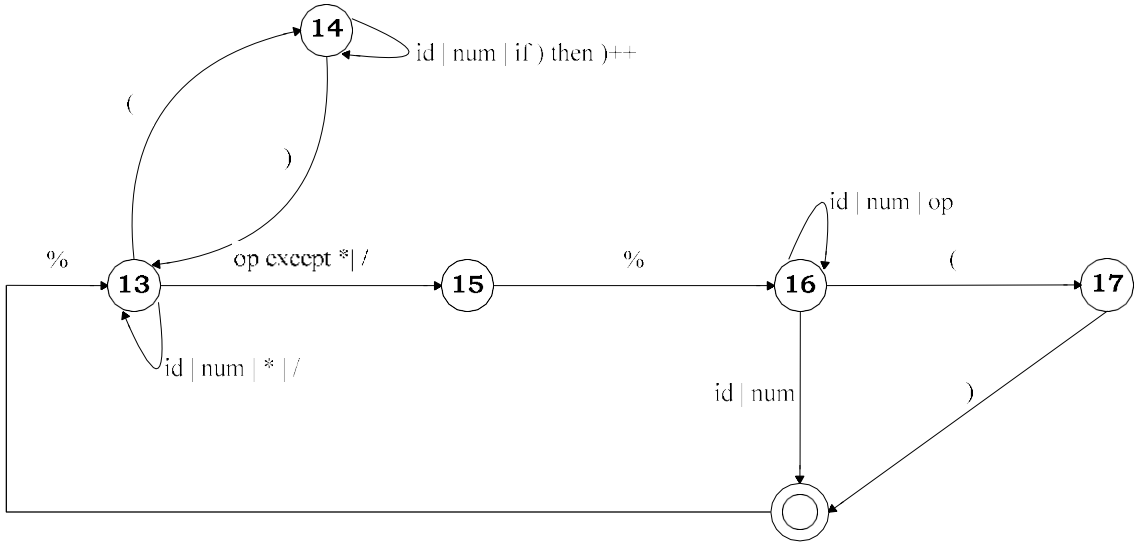


Figure: Transition diagram for mod operation

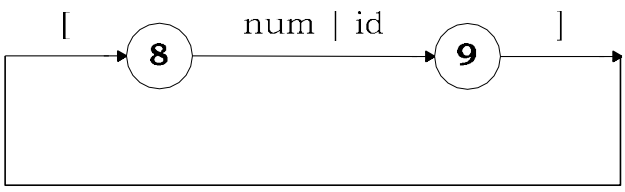


Figure: Transition diagram for array index checking

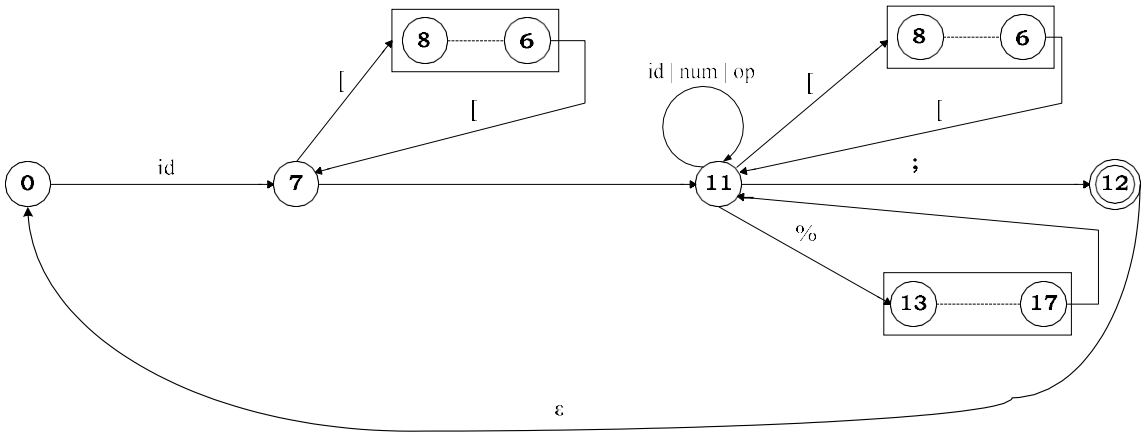


Figure: Transition diagram for assignment statement

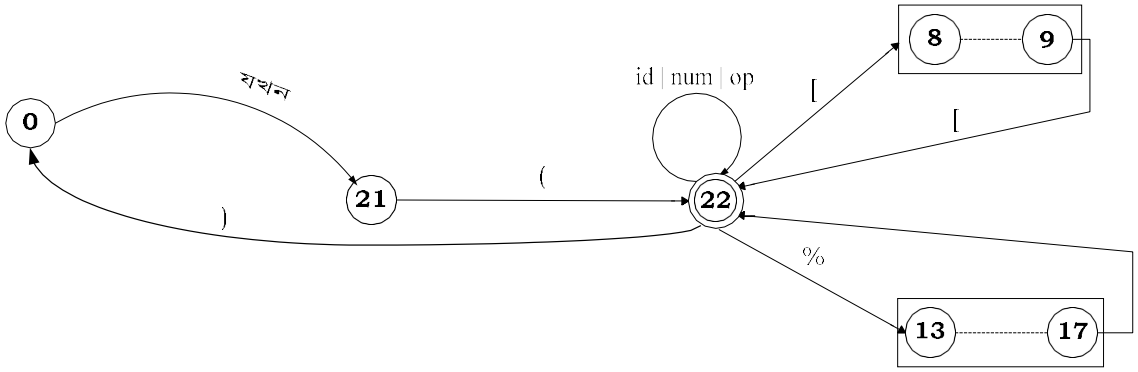


Figure: Transition diagram for iterative statement

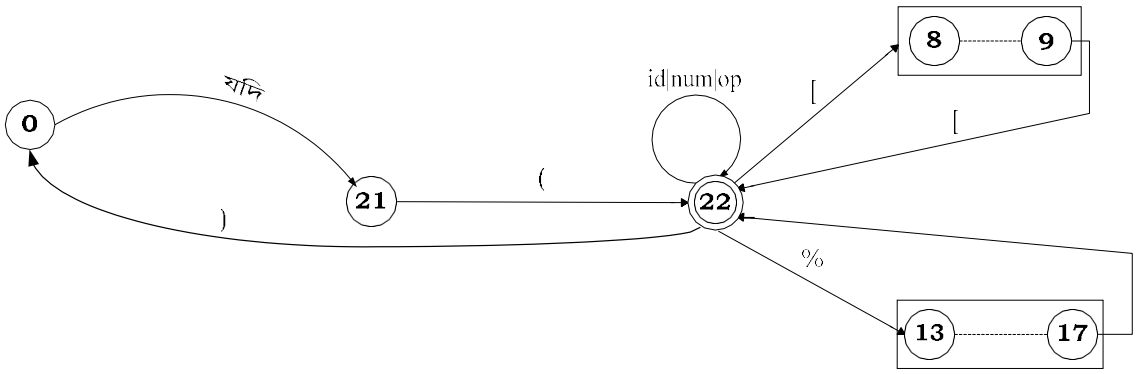


Figure: Transition diagram for selection statement

Appendix E: Operator Precedence Table

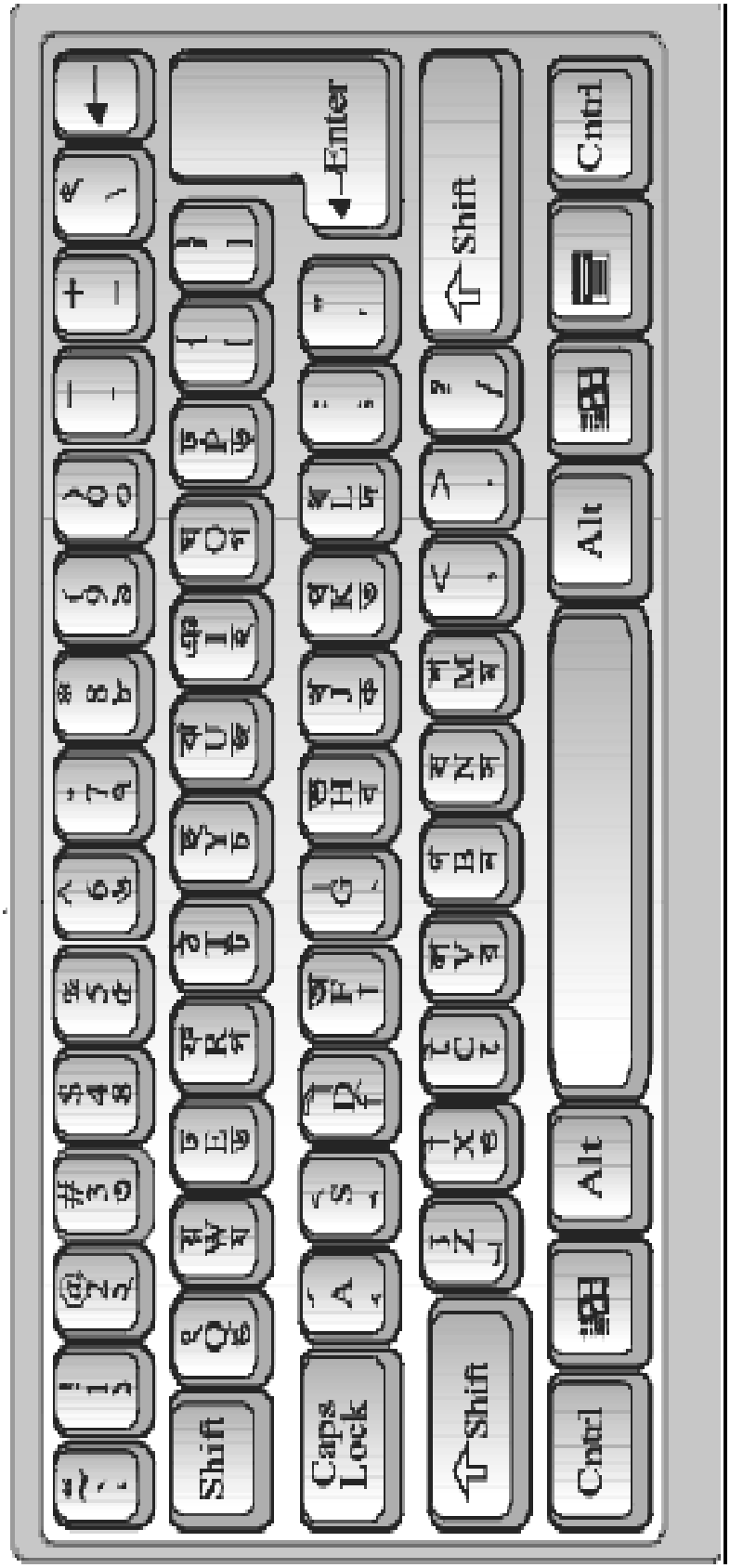
Description	Operator	Associativity
Function expression	**( )**	Left to right
Array expression	**[ ]**	Left to right
Unary minus	**-**	Right to left
Unary plus	+	Right to left
Negation	**!**	Right to left
Multiplication	*	Left to right
Division	/	Left to right
Modulus	**%**	Left to right
Less than	< =	Left to right
Less than or equal to	<	Left to right
Greater than	>	Left to right
Greater than or equal to	> =	Left to right
Equal to	= =	Left to right
Not equal to	**!**=	Left to right
Logical AND	m A	Left to right
Logical OR		Left to right
Assignment	=	Right to left

Appendix F: Notational Conventions

To avoid always having to state that "these are the terminals," "these are the non terminals," and so on, we have followed the following notational conventions [1].

1. these symbols are terminals:
   i. Lower-case letters early in the alphabet such as *a, b, c*.
   ii. Operator symbols such as +, -, etc.
   iii. Punctuation symbols such as parentheses, comma, etc.
   iv. The digits 0, 1, . . ., 9.
   v. Boldface strings such as **id** or **if**
2. these symbols are non terminals:
   i. Upper-case letters early in the alphabet such as *A, B, C*.
   ii. The letter *S*, which, when it appears, is usually the start symbol.
   iii. Lower case italic names such as *expr* or *stmt*.
3. Upper-case letters late in the alphabet such as *X, Y, Z* represent *grammar symbols*, that is, either non terminals or terminals.
4. Lower-case letters late in the alphabet such as *u, v, w. . . z* represent strings of terminals.
5. Lower-case Greek letters, a, β, γ, for example, represent strings of grammar symbols.
6. If $A \rightarrow a_1$, $A \rightarrow a_2$, . . ., $A \rightarrow a_k$ are all productions with A on the left, wemay write $A \rightarrow a_1 | a_2 | \ldots | a_k$.
7. Unless otherwise stated, the left side of the first production is the start symbol.

Appendix G: Keyboard Layout



122

Appendix H: Sample Source Codes

Sample 1:

Example of nested Selection statements. (Finding the smallest number among three numbers.)

```
Æ     ;
Æ     ;
Æ    5;

w ew()  ÇÇm*  m   *  <Æ 5 <4o *oWÇÇ
{
      Æ?(  );
      Æ?(  );
      Æ?(5);
       Wא(   <   )
      {
            Wא(   <5)
           {
                Ww  (Y    e–<3<o <> *Y);
           }
           w"–
           {
                Ww  (Y5 e–<3<o <> *Y);
           }
      }
      w"–   Wא(   <5)
      {
            Wא(   <   )
           {
                Ww  (Y    e–<3<o <> *Y);
           }
           w"–
           {
                Ww  (Y    e–<3<o <> *Y);
           }
      }
       w"–
      {
           Ww  (Y5 e–<3<o <> *Y);
      }
}
```

Sample 2:

Example of iterative statement. (Calculating Factorial.)

```
Æ  @;
Æ  w;
w ew()
{
      w= ;
      Æ?(@);
      Ⅱw  (@);
      Ⅱw  (Y! = Y);
         Y(@>○)
      {

            ÇÇⅡw  (@);   → m*  m   *  4~–7
            ÇÇ w=w * @;
            @ = @ - ;
      }
      Ⅱw  (w);
}
```

Sample 3:

Example of function calling.

```
w ew()
{
      Ⅱw  (Yw Ⅱ4 w ew   Y);
      e\();
      Ⅱw  (Yw Ⅱ4 w ew QY);
}

e\()
{
      Ⅱw  (Yw Ⅱ4 e\ !!Y);
      ÇÇⅡw  (Ym*  m   *  4~–7Y);ÇÇ
}
```

Sample 4:

Another example of function calling.

```
Æ  @;
Æ  Y;
Æ  Æ;

w ew()
{
      wA  ();
      <  5();
      Ⅱw  (Y<  5Vw=Y);
      Ⅱw  (@);
}

wA  ()
{
      Æ?(@);
      Æ?(Y);
      Æ?(Æ);
}

<  5()
{
      @ = @+Y+Æ;
}
```

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, "Compilers – Principles, Techniques, and Tools", Addison Wesley Longman (Singapore) Pte. Ltd., Delhi-110092, 2000, pp. 84-86, p. 91, pp. 175-178, 186-192, p. 279, pp. 344-348, p. 463, p. 514, p. 516.

[2] H. Alblas and A. Nymeyer, "Practice and Principles of Compiler Building with C", 1998, p. 4, p. 344, p. 366.

[3] John E. Hopcroft, Rajeev Motwani and Jeffery D. Ullman, "Introduction to Automata Theory, Language, and Computation", Addison Wesley Longman (Singapore) Pte. Ltd., Delhi-110092, 2001, p. 46, p. 90.

[4] Yashavanat Kanetkar, "Let Us C", Manish Jain for BPB Publications, New Delhi-110001, Third Revised Edition, 1999, p. 4, pp. 682-683, 714-715.

[5] Byron S. Gottfried, Ph.D., "Theory and Problems of Programming with C", Schaum's Outline Series-McGraw-Hill, Second Edition, 1998, pp. 46-61.

[6] Yashavanat Kanetkar, "Working with C", Manish Jain for BPB Publications, New Delhi-110001, First Edition, 1994, pp. 113-122, 141-144.

[7] Byron S. Gottfried, Ph.D., "Theory and Problems of Programming with PASCAL", Schaum's Outline Series-McGraw-Hill, Second Edition, 1994, p. 10, pp. 110-112, 120-127, p. 152.

[8] Seymour Lipschutz, Ph.D., "Theory and Problems of Data Structure", Schaum's Outline Series-McGraw-Hill, International Edition, 1986, p. 170, p. 171.

[9] Ytha Yu and Charles Marut, "Assembly Language Programming and Organization of the IBM PC", Mitchell McGraw-Hill, Revised Edition, 2002, pp. 167-169, 170-174, 372-374.

[10] Ravi Sethi, "Programming Languages Concepts and Construct", 2nd Edition, p. 52.

[11] Robert W. Sebesta, "Concepts of Programming Languages", Addison Wesley Longman (Singapore) Pte. Ltd., Delhi-110092, Fourth Indian Reprint, 2001, pp. 29-31, 39, p. 141, pp. 157-159, p. 175, pp. 196-197, p. 210.

[12] Robert McNaughton PRENTICE-HALL, "Elementary Computability, Formal Languages, and Automata", INC., Englewood Cliffs, New Jersey,2003, pp. 129-138.

[13] Allen I. Holub, “Compiler Design in C”, Prientince Hall of India, First Edition, p. 32.

[14] Daniel Jurafsky and James H. Martin, “Speech and Language processing, An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition”, Pearson Education (Singapore) Pte Ltd, Indian Branch, p. 324.

[15] Kenneth H. Rosen, “Discrete Mathematics and Its Applications”, McGraw-Hill, Fourth Edition, 2000, p. 532, p. 549.

[16] Raffiquzzaman, “Microprocessor Theory and Application – Intel and Motorolla”, Prentice-Hall of India Private Ltd., Fourth Edition, pp. 433-456.

[17] Douglas V. Hall, “Microprocessors and Interfacing Programming Hardware”, Second Edition, Tata McGraw-Hill Publishing Co. Ltd., pp. 204-207.

[18] Allen I. Holub , “Compiler Design In C”, Prientince, Second Edition, p. 56

[19] John Socha and Peter Norton, “Assembly Language for the PC”, Prientince Hall, Third Edition, pp. 599-607.

[20] Yu-Cheng Lice and Glenn A. Gibson, “Microcomputer System: The 8086/8088 Family Architecture, Programming and Design”, Prientince Hall, pp. 606-612.

[21] Andraw W. Appel, “Morden Compiler Implementation in JAVA”, Cambridge University Press, First Edition, p. 3.

[22] Steven S. Muchnick, “Advanced Compiler Design Implementation”, Harcourt Publication Pte Ltd., First Edition, pp. 43-64.

[23] John J. Denoven, “System Programming”, Tata McGraw-Hill Publishing Co. Ltd., Fourteenth Edition, pp. 452-469.

[24] Terrence W. Pratt and Marvin V. Zelkowitz, “Programming Language Deign and Implementation”, Printice Hall of India, Third Edition, p. 46

[25] Michael L. Scott, “Programming Language Pragmatics”, Harcourt Asia Publication Pte Ltd., First Edition, pp. 166-168.

[26] McCrosky, C. and K. Sailor, “A synthesis of Type-checking and parsing”, Computer Languages, 1993, Vol. 18, No. 4, pp. 241-250.

[27] K. Sailor and McCrosky, C., “A Practical Approach to Type-sensitive Parsing,” Computer Languages, 1994, Vol. 20, No. 2, pp. 101-106.

[29] Hendren and Gao, “Designing Programming Languages for the Analyzability of Pointer Data Structures”, Computer Languages, 1993, Vol. 19, No. 2, pp. 119-134.

[30] Shaoying, L., “An Abstract Programming Language and Correctness Proofs,” Computer Languages, 1993, Vol. 18, No. 4, pp. 273-282.

[31] Horspool, R.N., “Recursive Ascent-Descent Parsing”, Computer Languages, 1993, Vol.18

[32] Carvalho and Kowaltoski, “On Open Arrays and Variable Number of Parameters,”Computer Languages, 1993, Vol. 19.

[33] M. J. Outshoorn and C.D. Marlin, “A layered, Operational Model of Data Control in Programming Languages,” Computer Languages, 1991, Vol. 16, No. 2, pp.147-165.

[34] S. J. Drew and K.J. Gough, “Exception Handling: Exception Handling: Exception the Unexpected,” Computer Languages, 1994, Vol. 32, No.8, pp. 69-87

[35] Albert Nymeyer, “A Grammatical Specification of Human-computer dialogue”, Computer Languages, 1995, Vol. 21

[36] N. Jamil, “Design and Implementation of A High-Level Bangla Programming Language”, Department of Computer Science, University of Dhaka, 1996, pp. 1-107.

[37] http://www.bios.org.bd

[38] http://www.bengalilinux.org