Essentials of

COMPILER CONSTRUCTION

By

ODUWOLE O.A
B.Tech. (Hons), MSc., M.Tech., Ph.D. (Computer Science)
Lecturer, Computer Science Department,
Adeleke University, Ede.
Osun State. Nigeria.

LALA O. G.
BSc., MSc., Ph.D. (Computer Science)
Lecturer, Computer Science Department,
Adeleke University, Ede.
Osun State. Nigeria.

OYEBODE E.O
B Sc, MSc (Computer Science)
Lecturer, Ajayi Crowder University Oyo,
Oyo State. Nigeria.

Essentials of Compiler Construction

Oduwole O. A.; Lala O. G.; Oyebode E. O.

ISBN 978 – 46792 – 2 – 2

Printed @:

OKIKI Graphics & Printing Services
Shop #16, Ayoka Lad Plaza,
No. 6, General Hospital Road,
Atan-Oba, Offa, Kwara State.
07053302511, 07083210799
services.okiki@gmail.com

DEDICATION

This book is dedicated to God Almighty,
and
To all our families in general

ACKNOWLEDGEMENT

We first of all acknowledge God for giving us the grace and strength to write this text. May his name be glorified (Amen).

We also acknowledge the opportunity offered by the management of Adeleke University, Ede, Osun State to academic staff of the institution to publish books to be used by students.

The contributions of our colleagues and the interest of computer science students in this publication are well appreciated.

PREFACE

Compiler construction is an important aspect of Computer Science, which every potential computer analyst and software developers need to understand and be able to apply.

This book thus suits National Universities Commission (NUC) requirements for a detailed course in Compiler Construction for Computer Science, Computer Engineering, Pure and Applied Mathematics students, and for all institutions of higher learning. Readers are therefore advised to see the book as a companion and a tool for understanding compiling techniques.

TABLE OF CONTENT

CHAPTER ONE

1.0 PROGRAMMING LANGUAGE TRANSLASTION

The only language a computer can readily understand is machine language, and computers are made to carry out commands given in this language. Other programming languages like assembly and high level languages were developed as a result of the strain required in processing machine language mixed with the issue of machine dependency.

It is necessary to convert such codes into machine language codes that have the same meaning as the source code in order to enable the execution of programs written in non-machine languages. Thus, a program that converts instructions written in non-machine language into their equivalents in machine language is referred to as a translator. Assemblers, interpreters, and compilers are three prominent types of translators. While assembler turns assembly language programs into machine language, interpreters and compilers translate high level languages.

1.1 COMPILER, INTERPRETER AND ASSEMBLER

A compiler is a program that translates a high-level language program into a functionally equivalent low-level language program. So, a compiler is basically a translator whose source language (i.e. language to be translated) is the high-level language, and the target language is a low-level language; that is, a compiler is used to implement a high-level language on a computer. A compiler translates High Level Language (HLL)

to machine language equivalent for later execution while interpreter translates and executes the program statement by statement.

Compiler processes the program statements in their physical input sequence, while interpreter follows the logical flow of control through the program. Interpreter processes some statement repeatedly (if they were part of a loop) and might ignore others completely (if control never reaches them).

Object code is produced for subsequent executions with compilation: but with interpretation, no code is produced and re-translation is required.

A pure interpreter normally analyses a source program statement, each time it is to be executed in order to know how to carry out the execution. In the analysis, syntactic and semantic checks are performed on the source statement and the statement is then translated into an intermediate form, usually a polish notation is often used. This intermediate form is then executed (i.e. interpreted or simulated). This stage of analysis is similar to the first part of a multi-pass compiler.

Again, interpreters are very useful for debugging purposes since part output can be obtained as the program runs, but with a compiler, the whole program must be error free before it can be executed to obtain any result.

Moreover, compilation requires that both source program and object program be stored; while for interpretation there is no object program stored, only the source program has to be stored. For this reason, interpreters are preferable to compilers for very small computers.

On the other hand, a translated program may execute 20 to 100 times faster than an interpreted one. This is because certain portions of a program may need to be executed repeatedly, perhaps hundreds of times. An interpreter would have to analyze each statement every time it is to be executed. However, a compiler translates each statement only once; regardless of how many times the translated statement will eventually be executed.

Another significant difference between a compiler and interpreter is that only one error can be detected at a time by an interpreter, this error will cause program execution to halt until the error is corrected. However, as many syntax errors as possible are normally detected by a compiler.

Moreover, there is a guarantee for detection of all syntax errors in compilation but there is no such guarantee for interpretation, because control might never reach some statements that have syntax errors.

Assembly language is the oldest programming language, it bears the closest resemblance to native machine language. It provides direct access to computer hardware, requiring much understanding of computer architecture and operating system. An assembler is a utility program that converts source code programs from assembly language into machine language which is a numeric language specifically understood by a computer processor (the CPU). Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB and CALL. Assembly language has a one-to-one relationship with machine language. Each assembly language instruction corresponds to a single machine-

language instruction. High-level languages such as C++ and Java have a one-to-many relationship with assembly language and machine language. An important distinction between high level languages and assembly is that most high-level language are portable (they compile, and run on any machine)

However, assembly language is not portable because it is designed for a specific processor family, known processors include Motorola, 68x00, Intel IA-32, SUN Sparc. Summarily with respect to application, the table below compares High level language and assembly language.

Types of Application	**High Level Lang.**	**Assembly Lang.**
Business Application Software	Formal structures make it easy to organize and maintain large sections of code	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience.
Hardware Device Driver	Language may not provide for direct hardware access. Even, if it does, awkward coding techniques may be required, resulting in maintenance difficulties	Hardware access is straight forward and simple. Easy to maintain, programs are short and well documented

Business Application Platform	Usually portable, the source code can be recompiled and each target Operating System with minimal changes	Must be recorded separately for each platform, using an assembler with different syntax
Embedded Systems & Computer Games requiring direct Hardware access	Produces for much executable code and may not run efficiently	Ideal because the executable code is small and runs quickly.

1.2 CRITERIA FOR DETERMINING LANGUAGE IMPLEMENTATION APPROACH.

The following criteria are important factors to consider when deciding whether to implement a language as a compiler or an interpreter.

i. **Execution Efficiency**: Compiler is preferable otherwise interpreter

ii. **Environment**: Compiler is more suitable for production environment, while interpreter is for training environment.

iii. **Available Main Memory**: Compiler for unlimited memory, interpreter for limited memory.

iv. **Debugging and Diagnostic Support**: Compiler if its production is critical; otherwise interpreter.

v. **Detection of All Translation Errors**: Compiler, if its guarantee is of prime concern, otherwise, interpreter.

1.3 THE COMPILATION PROCESS

Compilation refers to the compiler's process of translating a high-level language into a low-level language program. This process is very complex; hence, from the logical as well as an implementation point of view, it is customary to partition the compilation process into several phases, which are nothing more than logically cohesive operations that input one representation of a source program and output another representation. The phases of compilation can be broadly divided into four. They are

(a) Lexical analysis

(b) Syntax analysis

(c) Semantic analysis; and

(d) code generation.

The compilation process is as represented in Figure 1.1

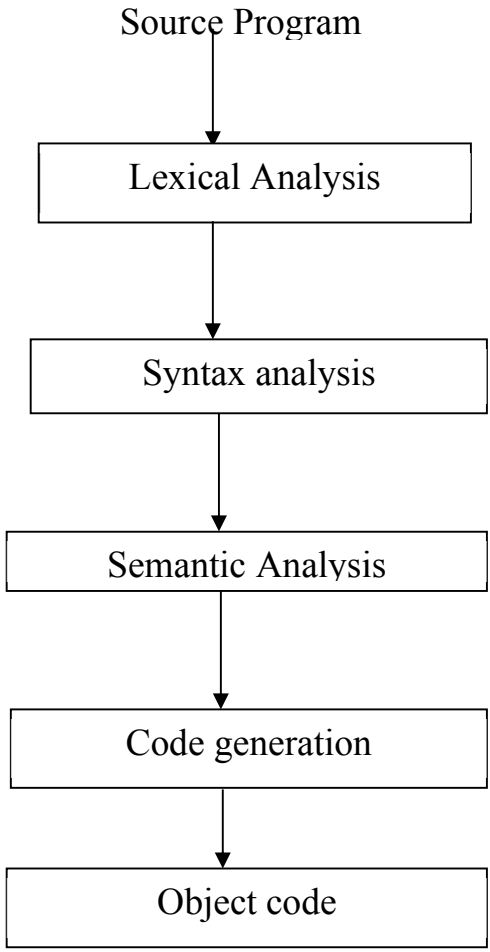


Figure 1.1: The Compilation Process

Exercise 1

1. Explain why codes in assembly language that runs on machine A will not run on equivalent machine B given the same condition.
2. In terms of speed and execution efficiency what are the necessary factors for consideration when thinking of implementing a language using compiler, interpreter and assembler
3. State clearly the relationship that exists in terms of code translation between compiler to machine language, interpreter to machine language and assembler to machine language.

.

CHAPTER 2

2.0 LEXICAL ANALYSIS

This is the first step in the compilation process. At the lexical analysis stage, a program called a lexical analyzer or scanner scans the characters of a source program from left to right and builds up the tokens of a program. The actual procedure of doing this is as follows: when a source statement is read, the first character scanned enables a token category to be identified. For example, scanning a digit first will indicate token category integer: similarly, an alphabet first will indicate token category identifier or reserved word. On identifying this token category, the lexical analyzer keeps scanning and concatenating, character by character repeatedly until a separator is found. The token of any language is the smallest basic unit of that language. The tokens of most languages fall into the following categories:

- **Identifiers** or **Variable Names** used by programmer
- **Keywords** e.g. DO, SELECT, BEGIN
- **Special Operators** e.g. <, >, =
- **Delimiters** e.g. comma, parenthesis curly bracket, semi-colon etc.
- **Literals** e.g. numeric constants, string constants
- **Separators**

Lexical analyzers are familiar with keywords, identifiers, operators, delimiters and punctuation symbols of the language to be implemented. It will scan the source program to see the occurrence of sequence of character patterns that resembles identifiers, operators, keywords, delimiters, punctuation symbols etc.

Token generation is usually by specifying the rules that govern the way that the language's alphabet symbols can be combined, so that the result will be a token of that language's identifiers, operators, and keywords. Redundancies such as comments, spaces etc are eliminated. In scanning the source program, errors can be detected e.g unknown character.

Lexical analyzer or scanner is a program that scans the characters of the source program from left to right and generate the tokens of a program. The first character scanned is used to determine the token category. For instance, scanning the first character to be numeric may mean integer. On identifying this token category, the lexical analyzer keeps scanning and concatenating, character by character, consecutive characters of that token until a separator is encountered.

A separator can be any of the following:

- blank character
- conventional arithmetic operator
- delimiters such as comma, semi-colon
- a character that does not actually belong to the set of the expected character that forms a particular token class. E.g. in the token class integer, a scan of any character other than digit 0-9 will serve as a separator.

The tokens of a source program are usually passed to the next phase of compilation (syntax analysis); in an internal form. For example, ID, LT, OP, KW and DL may be the internal representation of the token type identifiers, literals, operator, keywords and other delimiters respectively.

These internal representations and the actual values of token (i.e. the actual identifier name and actual literal) are passed in representation forms to the next phase of compilation. Alternatively, different integers may be used to denote each of these token types. For example, integers 01, 02 and 03 may be used to represent token type identifier, integer, and other literals respectively, while other unique integers may be used to represent each unique reserved words, operators and delimiters in the language.

This method allows the token to be treated in a uniform way. The internal forms in which tokens are represented have the same length regardless of the length of the token character. This allows the rest steps in compilation to be done in an efficient manner with fixed length symbols rather than variable length string of characters.

For example, let us consider the statement below

```
10      LET B = 5.3 * 3.2 + A
```

The scanner would pass the following internal representation of tokens to the syntax analyzer

	Token Passed	
Step	**Internal Representation**	**Value**
1	LT	10
2	KW	LET
3	TD	B
4	OP	=
5	LT	5.3
6	OP	*
7	LT	3.2
8	OP	+
9	TD	A

In addition to recognizing tokens, the scanner must recognize and delete blanks and redundant statements like comments. It is also the responsibility of the scanner to note and report lexical errors such as invalid characters or improperly formed identifiers.

2.1 LEXICAL ANALYZER DESIGN

The lexical analyzer's job is to scan the source program and generate a stream of tokens as output; hence the following problems must be addressed in its design:

(a) Identifying the tokens of the language for which the lexical analyzer is to be built, and to specify these tokens by using suitable notation, and

(b) Constructing a suitable recognizer for these tokens.

Therefore, the first thing that is required is to identify what the keywords are, what the operators are, and what the delimiters are. These are the tokens of the language. After identifying the tokens of the language, we must use suitable notation to specify these tokens. This notation should be compact, precise, and easy to understand. Regular expressions can be used to specify a set of strings, and a set of strings that can be specified by using regular-expression notation is called a "regular set". The tokens of this programming language constitute a regular set. Hence, this regular set can be specified by using regular-expression notation. Therefore, we write regular expressions for things like operators, keywords, and identifiers. For example, the regular expressions specifying the subset of tokens of typical programming language are as follows:

Operators = + | - | * | / | mod | div

Keywords = if | while | do | then

Letter = a | b | c | d |...| z | A | B | C | ...|Z

Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Identifier = letter (letter | digit)*

The advantage of using regular-expression notation for specifying tokens is that when regular expressions are used, the recognizer for the tokens ends up being a DFA (Deterministic Finite Automata). Therefore, the next step is the construction of a DFA from the regular expression that specifies the tokens of the language. But the DFA is a flowchart of the lexical analyzer. Therefore, after constructing

the DFA, the next step is to write a program in suitable programming language that will simulate the DFA. This program acts as a token recognizer or lexical analyzer. Therefore, we find that by using regular expressions for specifying tokens, designing a lexical analyzer becomes a simple mechanical process that involves transforming regular expressions into finite automata and generating the program for simulating the finite automata. Computing tools like LEX can be used to automate the procedure of obtaining the lexical analyzer from the regular expressions and specifying the tokens.

2.2 SYNTAX ANALYSIS

In the syntax-analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. If the tokens in a string are grouped according to the language's rules of syntax, then the string of tokens generated by the lexical analyzer is accepted as a valid construct of the language; otherwise, an error handler is called. Hence, two issues are involved when designing the syntax-analysis phase of a compilation process:

1 All valid constructs of a programming language must be specified; and by using these specifications, a valid program is formed. That is, we form a specification of what tokens the lexical analyzer will return, and we specify in what manner these tokens are to be grouped so that the result of the grouping will be a valid construct of the language.

2 A suitable recognizer will be designed to recognize whether a string of tokens generated by the lexical analyzer is a valid construct or not. Therefore, suitable notation must be used to specify the constructs of a language. The notation for the construct specifications should be compact, precise, and easy to understand. The syntax-structure specification for the programming language (i.e the valid constructs of the language) uses Context Free Grammar (CFG), because for certain classes of grammar, it is easy to automatically construct an efficient parser that determines if a source is syntactically correct. Later in this text we shall study Context Free Grammar (CFG).

Syntax Analysis performs syntactic checks on the source program statement by ensuring that rules governing formation of valid statements are obeyed.

The syntax of a programming language relates to the grammatical rules governing the formation of legal statements in a language meaning that the rules of combining tokens together in appropriate sequence to form a valid sentence or syntactic construct. Specifically, it ensures that appropriate tokens of a syntactic unit follow each other in appropriate sequence. For instance, a language that requires line number at the beginning of each statement and probably followed by keywords. The syntax analyzer then builds up tables of information like symbol table, table of constants and so on. Errors such as multiple declarations of identifiers can still be reported.

Examples of syntactic construct include:

- expression (arithmetic, relational, logical, etc).
- Assignment statements
- Declarative statements
- loops.

The syntax analyzer or parser ensures that the rules of the programming language in question are strictly followed. For example, parsers ensure that the number of parenthesis in the left hand and right hand are the same, it rejects the same or repeated operators from following each other.

Formation of syntactic entities can be done through the use of BNF (Backus Naur Form). BNF is a notation for describing the formation of syntactic entities. It is a popular notation for describing the syntax of context-free grammars. Items enclosed in (< >) are called syntactic entities or non-terminal symbols, the non-terminal symbols can be further broken down to terminal symbols as shown by the grammar.

In BNF, the symbol ::= is used to denote "may be replaced by" or "may be defined by" ,

⟶ can also be used instead of ::=

In BNF, the symbol | means "or"

"'double-quotation marks are for delimiting literal strings

[ ] square-brackets are for delimiting optional terms

( ) parenthesis is for grouping terms

Let us examine some examples on BNF:

BNF for a simple calculator language

```
<calculation>   ::=   <expression>
<expression>    ::= <value><operator><expression>
<value>         ::=   <number>
<value>         ::=   <sign><number>
<number>        ::=   <unsigned>
<number>        ::=   <unsigned>.<unsigned>
<unsigned>      ::=   <digit>
<unsigned>      ::=   <digit>.<unsigned>
<digit>         ::=   0
<digit>         ::=   1
<digit>         ::=   2
<digit>         ::=   3
<digit>         ::=   4
<digit>         ::=   5
<digit>         ::=   6
<digit>         ::=   7
<digit>         ::=   8
<digit>         ::=   9
<sign>          ::=   +
<sign>          ::=   -
<operator>      ::=   +
<operator>      ::=   -
<operator>      ::=   *
<operator>      ::=   /
```

Although there are many variants and extensions of BNF, such as Extended Backus-Naur Form (EBNF) and Augmented Backus-Naur Form (ABNF), these extensions can themselves be expressed in BNF, they offer additional notation for describing structures like repeated and optional terms.

The extended BNF for the grammar described above is as follows:

<calculation> ::= <expression>
<expression> ::= <value><operator><expression>
<value> ::= [<sign><unsigned>]<unsigned>
<unsigned> ::= <digit><unsigned>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sign> ::= + | -
<operator> ::= + | - | * | /

Through a number of techniques, the syntax analysis is able to detect whether or not entities are formed by appropriate combination of appropriate small syntactic entities and or tokens.

The example below is the production rule for If statement in JAVA programming language

If statement ::= "if" "(" expression ")" statement
["else" statement] ;

Let us consider another example that defines the PRINT

statement in a typical programming language

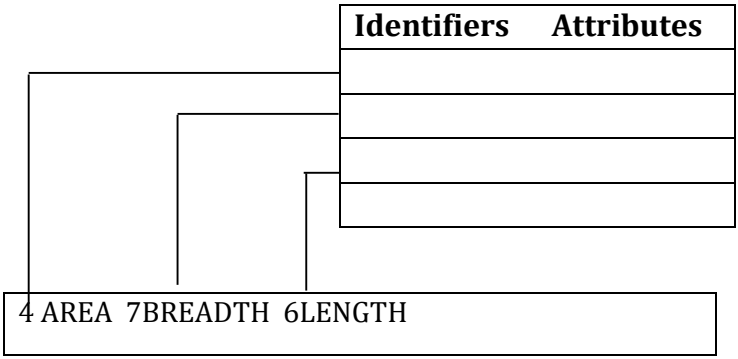
<PRINT> := PRINT|PRINT<EXPR>
<EXPR> := <ID>|<ID><OP><EXPR>
<ID> := <LETTER>|<LETTER><VAR>
<VAR> := <LETTER>|<LETTER><DIGIT>
<OP> := + | -
<LETTER> := A|B|...|Y|Z
<DIGIT> := 0|1|...|8|9

2.3 SYMBOL TABLE

At the syntax analysis stage, information about identifiers, literals for loops and the likes are kept in symbol or identifier table. It is a table containing the identifiers used in the source program along with their attributes. The attributes of an identifier include its type (integer, real, string), form (constant, simple variable, subscripted variable), block number for structured language like ALGOL, location in memory and other attributes relevant to the language.

	Identifiers	**Attributes**
Entry1	AREA	
Entry2	BREADTH	
Entry3	LENGTH	

Since the number of characters in identifier names varies, pointers to the identifiers are often stored in the column for identifier, while the identifiers themselves and their length are stored in a special string list. This approach keeps the size of the identifier fixed and thus facilitates searching for information about a particular identifier



Each time an identifier is encountered, the symbol table is searched to see whether the name has been previously entered. The information collected in symbol table is used in semantic analysis, which is checking that variable names are used in a manner that is consistent with their implicit and explicit declaration.

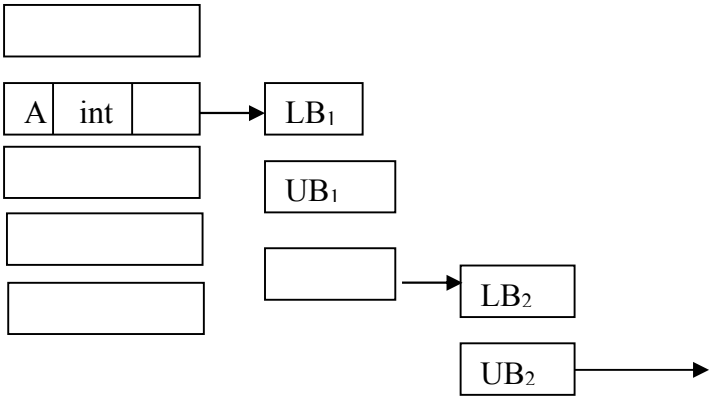
The translation process revolves around the symbol table. Every occurrence (use of declaration) of an identifier will call for need to search the symbol table. As a result, the symbol table must be structured in a manner which allows efficient extraction and insertion of information in the table.

A symbol table is a data structure used by a compiler to keep track of scope/binding information about names. This information is used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements. The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.

2.3.1 IMPLEMENTATION OF SYMBOL TABLE

Each entry in a symbol table can be implemented as a record that consists of several fields. These fields are dependent on the information to be saved about the name. But since the information about a name depends on the usage of the name (i.e. on the program element identified by the name), the

entries in the symbol table records will not be uniform. Hence, to keep the symbol table records uniform, some of the information about the name is kept outside of the symbol table record, as shown below. Here, the information about the lower and upper bounds of the dimension of the array named is kept outside of the symbol table record, and the pointer to this information is stored within the symbol table record.



2.3.2 ENTERING INFORMATION INTO THE SYMBOL TABLE

Information is entered into the symbol table in various ways. In some cases, the symbol table record is created by the lexical analyzer as soon as the name is encountered in the input, and the attributes of the name are entered when the declarations are processed. But very often, the same name is used to denote different objects, perhaps even in the same block. For example, in C language, the same name can be used as a variable name and as a member name of a structure, both in the same block. In such cases, the lexical analyzer only

returns the name to the parser, rather than a pointer to the symbol table record. That is, a symbol table record is not created by the lexical analyzer; the string itself is returned to the parser, and the symbol table record is created when the name's syntactic role is discovered.

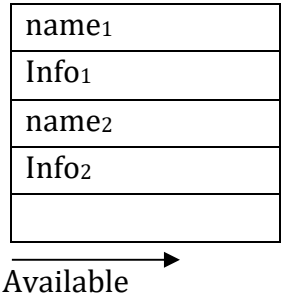
2.3.3 VARIOUS APPROACHES TO SYMBOL TABLE ORGANIZATION

Symbol table can be organized using any of these methods

(a) Linear List
(b) Search Trees
(c) Hash Tables

LINEAR LIST

A linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether the name is present or not, then the record for the new name is created and added to the list at a position specified by the available pointer as shown below:



A new record is added to the linear list of records

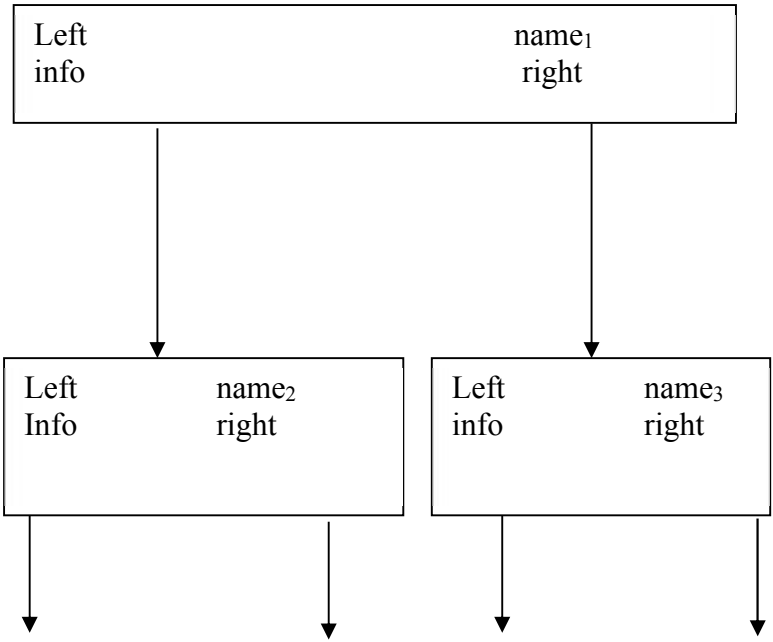
In order to retrieve information about a name, the table is searched sequentially, starting from the first record in the table. The average number of comparisons, p, required for search are p = (n+1)/2 for successful search and p = n for an unsuccessful search, where n is the number of records in symbol table.

The advantage of this organization is that it takes less space, and additions to the table are simple. This method's disadvantage is that it has a higher accessing time.

SEARCH TREES

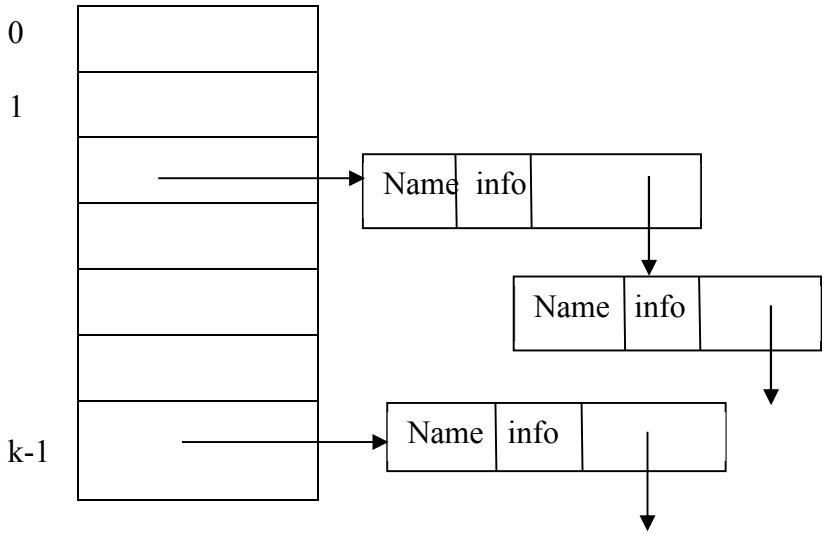
A search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a name is to be added, first the name is searched in the tree. If it does not exist, then a record for the new name is created and added at the proper position in the search tree.

This organization has the property of alphabetical accessibility; that is, all the names that precede name 1in alphabetical order will be accessible from name1by following a left link. Similarly, all the names that follow name 1 in alphabetical order will be accessible from name 1 by following the right link. It can be shown that the expected time needed to enter n names and to make inquiries is proportional to (m + n) $\log_2 n$; so for greater numbers of records (higher n) this method has advantages over linear list organization.



HASH TABLES

A hash table is a table of k pointers numbered from zero to k – 1 that point to the symbol table and a record within the symbol table. To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function. The hash function maps the name into an integer between zero and k-1, and using this value as an index in the hash table, we search the list of the symbol table records that is built on that hash index. If the name is not present in that list, a record for the name is created and inserted at the head of the list.



Hash table method of symbol table organization

When retrieving the information associated with the name, the hash value of the name is first obtained, and then the list that was built on this hash value is searched for information about the name

2.4 BASIC PARSING TECHNIQUES

Parsing is a stage in program compilation that maps from the linear strings of the program text into tree structures more easily dealt with by later stages of compilation. A parser obtains a string of tokens from the lexical analyzer and verifies whether or not the string is a valid construct of the source language that is, whether or not it can be generated by the grammar for the source language. And for this, the parser either attempts to derive the string of tokens "w" from the start symbol S, or it attempts to reduce w to the start symbol

of the grammar by tracing the derivations of w in reverse. An attempt to derive w from the grammar's start symbol S is equivalent to an attempt to construct the top-down parse tree; that is, it starts from the root node and proceeds toward the leaves. Similarly, an attempt to reduce w to the grammar's start symbol S is equivalent to an attempt to construct a bottom-up parse tree; that is, it starts with w and traces the derivations in reverse, obtaining the root S.

2.4.1 TOP DOWN PARSING

Top-down parsing attempts to find the left-most derivations for an input string w, which is equivalent to constructing a parse tree for the input string w that starts from the root and creates the nodes of the parse tree in a predefined order. Input string w is scanned by the parser from left to right, one symbol/token at a time, and the left-most derivations generate the leaves of the parse tree in left-to-right order, which matches the input scan order. Top-down parsing may require back-tracking (i.e. repeated scanning of the input); because in an attempt to obtain the left-most derivation of the input string w, a parser may encounter a situation in which a non-terminal A is required to be derived next, and there are multiple A-productions, such as $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_{n|}$. In such a situation, deciding which A-production to use for the derivation of A is a problem. Therefore, the parser will select one of the A-productions to derive A, and if this derivation finally leads to the derivation of w, then the parser announces the successful completion of parsing. Otherwise, the parser resets the input pointer to where it was when the non-terminal A was derived, and it tries another A-production.

The parser will continue this until it either announces successful completion of the parsing or reports failure after trying all of the alternatives.

Example

Consider the top-down parsing for the following grammar:

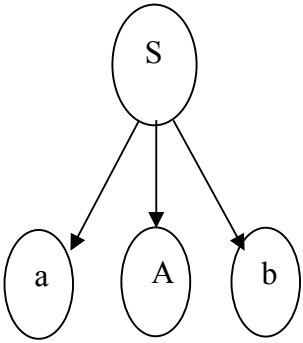
S -------> aAb

A -------> cd|c

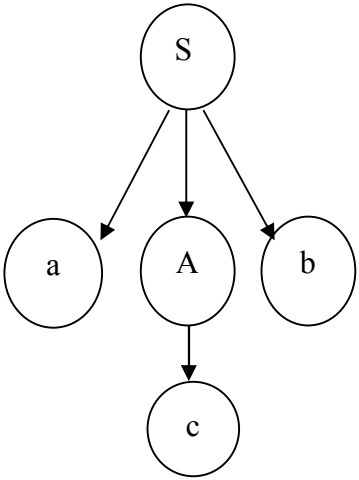
If the input string is w = acb, describe the Parsing tree using Top-Down Parsing.

Solution

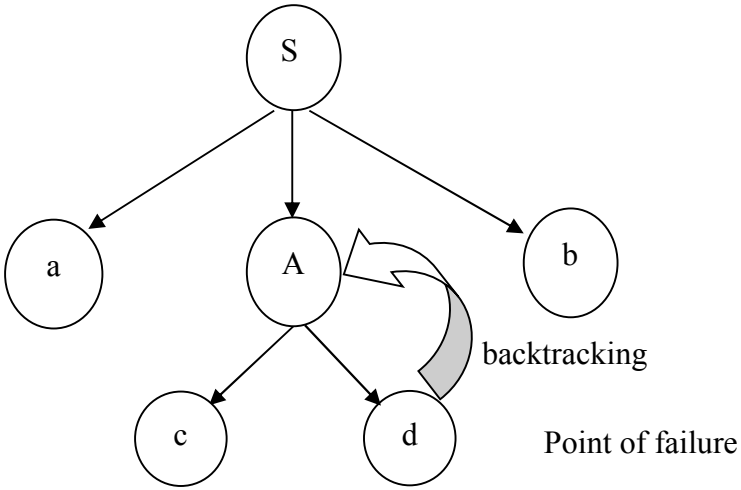
The parser initially creates a tree consisting of a single node, labeled S, and the input pointer points to a, the first symbol of input string w. the parser then uses the S-production S ---> aAb to expand the tree.



The left-most leaf, labeled a, matches the first input symbol of w. hence, the parser will now advance the input pointer to c, the second symbol of string w, and consider the next leaf labeled A. It will then expand A, using the first alternative for A in order to obtain the tree shown;



The parser now has the match for the second input symbol, and advances the pointer to b, the third symbol of w, and compares it to the label of the next leaf. If the leaf does not match d, it reports failure and goes back (backtracks) to A, (as shown below). The parser will also reset the input pointer to the second input symbol, the position it had when the parser encountered A and will try a second alternative to A in order to obtain the tree. If the leaf c matches the second symbol, and if the next leaf b matches the third symbol of w, then the parser will halt and announce the successful completion of parsing.



Note: If the parser fails to match a leaf, the point of failure, d, reroutes (back-tracks) the pointer to alternative paths from A.

Top-down parser can be implemented by writing a set of recursive procedures to process the input. One procedure will take care of left-most derivations for each of the non-terminal terms while processing the input. Each procedure should also provide for the storing of the input pointer in some local variables to enable proper resetting when the parser backtracks. This implementation is called recursive descent parser.

2.4.2 BOTTOM-UP PARSING

Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the right-most derivations of w in reverse. This is equivalent to constructing a parse tree for the input string w by starting

with leaves and proceeding toward the root, which is, attempting to construct the parse tree from the bottom-up. This involves searching for the substring that matches the right side of any of the productions of the grammar. This substring is replaced by the left-hand side non-terminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the right-most derivation. This process of replacing the right side of the production by the left side non-terminal is called "reduction". Hence reduction is nothing more than performing derivations in reverse. The parser scans the input string w from left to right, one symbol/token at a time. And to trace out right-most derivations of an input string w in reverse, the tokens of w must be made available in a left-to-right order.

Other common techniques of syntax analysis (parsing) are operator precedence and recursive descent. Operator precedence is especially suitable for parsing expressions, since it can use information about the precedence and associativity of operators to guide the parser.

Recursive descent uses a collection of mutually recursive routines to perform the syntax analysis. Particularly the earliest compiler of the several ages used operator precedence for expression and recursive descent for the rest of the language.

Exercise 2

1. Discuss the following methods of data organization in symbol table and state the relative advantage of one over the other

   (i) Linear List
   (ii) Search Trees
   (iii) Hash Tables

2. (a) Write a BNF that can define even numbers
   (b) Write a BNF that can define odd numbers

3. Distinguish between top down parsing and bottom up-parsing.

4. Using the grammar of the PRINT statement given in this chapter, state how many rules and how the rules are combined for the following strings:

   (i) PRINT
   (ii) PRINT K
   (iii) PRINT Y
   (iv) PRINT Q
   (v) PRINT T8
   (vi) PRINT M4

5. $S \longrightarrow A$
   $A \longrightarrow aAb$
   $A \longrightarrow ab$

   What language is described by the grammar above.

CHAPTER 3

3.0 SEMANTIC ANALYSIS

The semantic analysis phase deals with the interpretation of the meanings associated with a syntactic unit of the source program. When a parser has recognized a source language construct, the semantic analyzer will check for its semantic correctness, and store necessary information about it in the symbol table or create an intermediate form of the source program for it.

For instance, when an assignment statement of the form: <variable name> = <expression> is recognized, the semantic analyzer will check to ensure that there is type compatibility between the <variable name> and the <expression> and will create an intermediate form of source program for the assignment statement. For a simple declarative statement, the semantic analyzer will check to ensure that no same variable is declared more than once and necessary additional information about the declared variable are entered into the symbol table.

Other semantic checks include ensuring that every “BEGIN” statement has matching “CONTINUE” and every “ELSE” statement has matching “IF” and “ENDIF”. In addition, complex or compound statement, such as a DO LOOP, are broken down into simpler easy to follow statement at semantic analysis stage.

For example, consider the FORTRAN statement

```
DO 10  I = 1, 50, 2
       ---------
       ---------
       ---------
10 CONTINUE
```

This group statement may be broken down as

```
       I = 1
       J = 50
       K = 2
11     ---------
       ---------
       ---------
10 CONTINUE
        I=I+K
(I.LE.J) GO TO11
       ---------
       ---------
       ---------
```

The semantics analyzer also reports semantic errors discovered at this stage. Examples of such errors include type incompatibility, transfer of control to non-existing statement, duplicate labels on statements and so on.

3.1 THE INTERMIDIATE FORM OF SOURCE PROGRAM

Once the syntactic and semantic analysis has been performed on a syntactic unit, it is usual to put the source program in an intermediate form. The intermediate form of a source

program depends largely on how it is to be manipulated later. These include the parse tree, polish notation or a list of quadruple in the order in which they are to be executed.

(a) **PARSE / SYNTAX TREE**

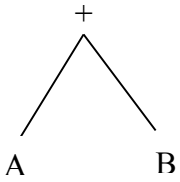
One intermediate form of an arithmetic statement is the parse tree (also called syntax tree or (binary tree). The rules for converting an arithmetic statement into a parse tree are:

i. any variable is a terminal node of the tree

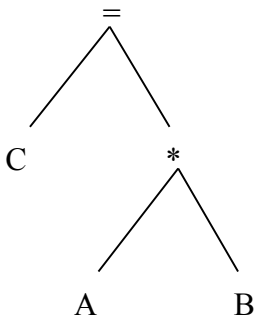
ii. for every operator, construct (in the order dictated by the rules of algebra) a binary (two branches) tree whose left branch is the operand and whose right branch is the tree for the second operand

Examples are shown below

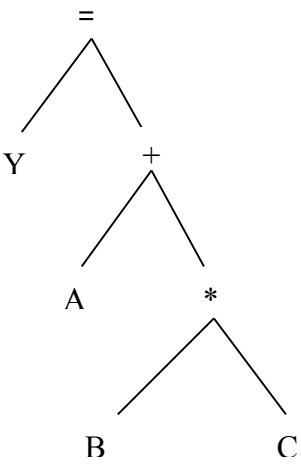
(i) A + B



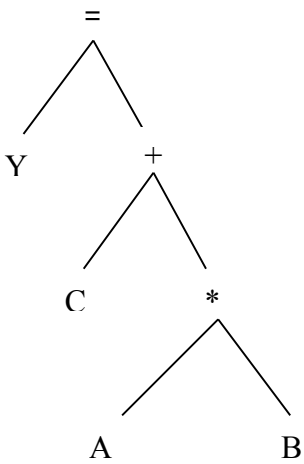
ii) C = A * B



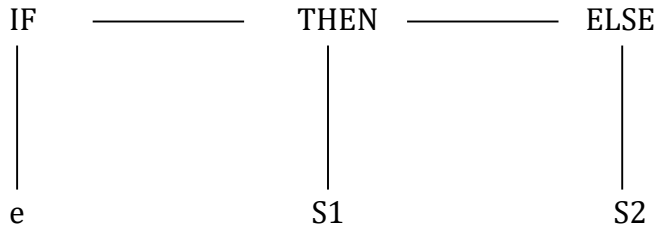
Y = A + B * C



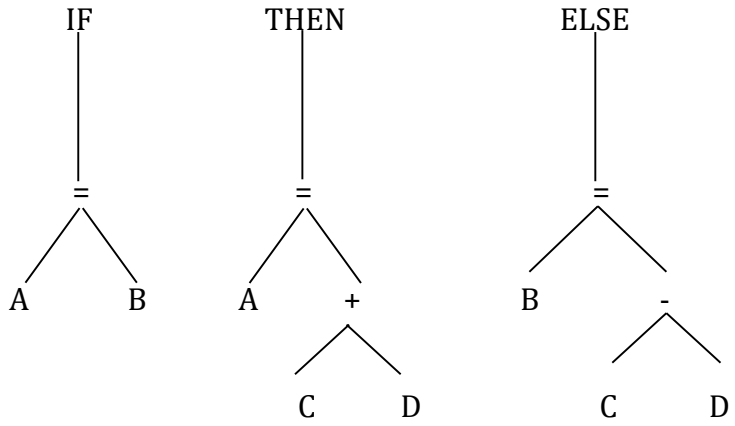
Y = A * B + C



The syntax tree for IF e THEN S1 ELSE S2 is:



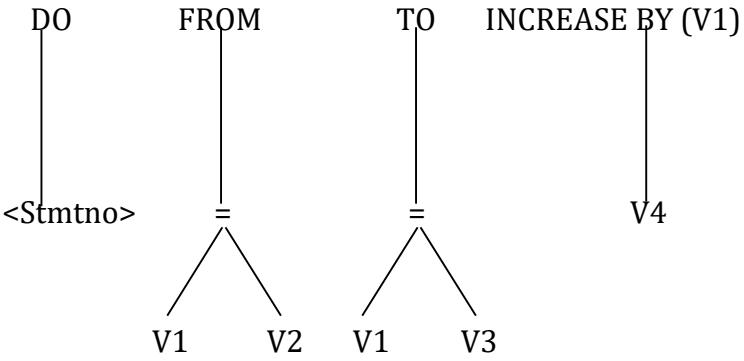
IF A = B THEN A = C + D ELSE B = C – D



The syntax tree for

DO <Stmtno> V1 = V2, V3 (Fortran Do Loop)

Could be



(b) **POLISH NOTATION**

Polish notation is used to represent arithmetic or logical expression in a manner which specifies simply and exactly the order in which operators are to be evaluated. In this notation, the operators come immediately after their operands. It is therefore, sometimes called suffix or postfix notation. For instance, we can have the following expression in infix notation with their corresponding polish notation.

	Infix notation	Polish notation
i	X + Y	XY+
ii	X * Y + Z	XY*Z+

Polish notation is the usual intermediate form of source program used by languages that are commonly interpreted e.g COBOL, BASIC.

It is easy to extend polish occasion to other operators, as long as operand are immediately followed by the operators. E.g assignment statement with the usual syntax

<variable name> = <Expression>

Now has the syntax

<variable name> <expression> =

A typical example is the statement

W = X * Y + Z

Which would now become

WXY * Z + =

Unary Operator

One simple way of handling unary operator like unary minus is to write them as binary operators, for example, O-B for –B. Thus:

Z + (-W + Y * X), in polish notation now becomes

Z0W – YX * + +

An alternative approach would be to introduce a special symbol for the operator. For example, @ for unary minus thus Z(-W + Y * X), in polish notation becomes Z W @ Y X * +

Unconditional Jump

An unconditional jump of the form

GO TO L

Would now have the syntax

L BR

Where L represent a number corresponding to the symbol table entry address of the label, and the operator BR means branch. BR is a unary operator

Conditional Jump

A conditional jump would be of the form

e i BZ

where e has a numeric value and i is the number or location of a symbol in the polish string. This polish notation means if e is zero then a branch should be made to i, otherwise the

usual order of evaluation should be followed. Similarly, operations such as branch if negative (BN), branch if positive (BP) and so on would be allowable. It should be noted that BZ, BN, BP etc are binary operators unlike BR which is a unary operator.

Conditional Statement

A conditional statement of the form:

IF <expr> THEN <Stmt 1> ELSE <Stmt 2>

would have the syntax:

<expr> L1 BZ <Stmt 1> L2 BR <Stmt 2>

Where <expr> has a Boolean value (1 = true, 0 = false), L1 is the label of the statement beginning <stmt 2>, L2 the label of the statement or symbol following <stmt 2>, (i.e a statement that follows the last statement in <stmt 2>. The operators BZ and BR are as defined previously for example, the statement

IF A = B THEN C = B + A ELSE D = A – B

would have its polish notation as

A B – L1 BNZ C B A + = L2 BR D A B - =

Where BNZ means branch if not zero, L1 represents the label for statement B = A – C and L2 represents the number of the statement or symbol following the complete IF statement given above. Alternatively, the polish notation could be written as;

(1) AB-(13) BNZ
(6) C B A + = (18) BR
(13) D A B - =
(18) _____

Where the number in parenthesis represents the position, location or number of symbol in the polish string. For example, the first A, BNZ. C and last B occupies the 1^{st} , 5^{th} , 6^{th} and 15^{th} position in the polish string respectively.

(c) **LIST OF QUADRUPLES**

A convenient form for a single binary operation is a list of quadruple it uses the format:

(Operators, Operand 1, Operand 2, Result)

Where operand 1 and operand 2 specify the argument and "Result" specifies the result of the operation. Thus X * Y might be represented by *, X, Y, T where T is an arbitrary temporary variable to which the result of X * Y is assigned.

Unary operators

In quadruple notation, common convention for unary operators (-) is that they do not make use of the second operand. For example, -X is represented in quadruple as:

-, X, , T

Which stores the value of –X in T.?

Consequently, the quadruple for –X * (Y+Z) is

+, Y, Z, T1

-, X, , T2

*, T1, T2, T3

Assignment Operator

This is handled like unary operators, the quadruple

W = X + Y * Z is represented as:

*, Y, Z, T1
+, X, T1, T2
=, T2, , W

Unconditional Jump

The quadruple for unconditional branch instruction is BR, i. This means branch to quadruple i.

Conditional jump

A quadruple for conditional branch statement can be

BZ . i, OP

This means that if the operand OP is zero then branch to quadruple i, BN (if OP is negative) and BP (if OP is positive are similarly defined.

Another quadruple for conditional jump (with a relational operator) can be

BE, i, OP1, OP2

Where BE means branch to quadruple I if OP1 = OP2

BG (for OP1 > OP2), BL (for OP1 < OP2) are similarly defined.

The quadruples defined for the jump instruction, can be applied to conditional statement as shown in the following examples

Example1

The quadruples for the statement:

IF I > J THEN K = K * 6

Could be

(1) -, I, J, T1
(2) BP (4), T1
(3) BR, (6)
(4) *, K, 6, T2
(5) =, T2, , K
(6) _____

Alternatively, it could be written as

(1) BG, (3), I, J,
(2) BR, (5)
(3) *, K, 6, T1
(4) =, T1, , K
(5) ……..

The number in parenthesis indicates quadruple sequence number

Example 2

The quadruple for the conditional statement:

IF A = B THEN A = A + D ELSE B = C- D

Could be the following:

(1) -, A, B, T1
(2) BZ, (6), T1
(3) -, C, D, T2

(4) =, T2, , B
(5) BR, (8)
(6) +, A, D, T3
(7) =, T3, , A
(8) _____

OR

(1) BE, (5) A, B
(2) -, C, D, T1
(3) =, T1, , B
(4) BR, (7)
(5) +, A, D, T2
(6) =, T2, , A
(7) _____

Where BE means branch if the 2 specified operands are equal.

3.2 CODE GENERATION

The last step in compilation is the code generation. This is the actual translation of the intermediate form of source program into machine language.

Consider the assignment statement:

, W = X * Y + Z

This could be represented by the sequence:

*, X, Y, T1

+, T1, Z, T2

=, T2, , W

Where T1 and T2 are temporary variables created by the compiler. Note that the quadruple will appear in the order in which they are to be executed. Furthermore, the operands in the above example would not be the symbolic names but pointers (or indexes) to the symbol table element which describes the operands.

The codes for the quadruples of the assignment statement would be generated as:

```
LDA X
MUL Y
STA T1
LDA TI
ADD Z
STA T2
LDA T2
STA W
```

Compiler-directed instructions such as PL/1's DECLARE statement or FORTRAN'S DIMENSION statement do not have intermediate form, they are not machine instructions but instructions that provides information that will help the compiler in doing its work of compilation. All information within these statements is normally entered into the symbol table and no machine code is required to be generated for them.

Types of Code

There are three types of code in which a source program can be translated to. These are:

(i) CODE IN ABSOLUTE MACHINE LANGUAGE

These are ML codes directly placed in fixed locations in memory which the compiler then executes immediately (load and go compiler)

Advantages

It is the most efficient from time point of view. Examples are WATFOR, ALGOL.

Disadvantages

(a) Subprogram cannot be pre-compiled separately; all Subprograms must be compiled at the same time.

(b) Storage is wasted as compiler occupies a large space.

(c) Compilation will be done every time the program is to be executed.

(ii) CODE IN ASSEMBLY LANGUAGE

Advantages

(a) It is the easiest, since AL is easier to deal with than binary digits

(b) Calls on macros whose definition has been previously written can be easily generated.

Disadvantage

It adds an extra step (assembly) to the process of translating a program and this extra step takes as long as the compilation itself.

(iii) CODE IN MACHINE LANGUAGE (placed in secondary storage)

It is flexible as in the AL format, but it does not require the extra assembly time. It is the standard in many systems. It however, requires linking and loading and this can be time consuming.

The two types of this code are:

(a) Absolute object code: This can only be loaded in a specific location in memory

(b) Relocatable object code: This can be loaded in any part of the memory.

3.3 STORAGE ALLOCATION

Prior to actual generation of code in either assembly language or machine language, storage would have to be assigned to all identifiers (both user-defined and compiler-generated, such as needed for temporarily storing an intermediate result) and literals. For instance, consider the FORTRAN declarative statement below:

INTEGER AREA, BREADTH, LENGTH

For this statement, a portion of the symbol table similar to the one below may be built. It is the responsibility of the storage

allocation routine to assign required location to each identifier.

As shown below, four bytes are reserved for each integer variable. The first variable will be assigned to a relative location zero, the second to 4, and so on but the absolute address cannot be determined until load time. These relative assigned addresses are used by the compiler in later phase for proper accessing.

Symbol Name	Other Attributes	Storage
AREA		0
BREADTH		4
LENGTH		8

In addition to storage allocation, some forms of intermediate symbolic codes are usually generated. The primary difference between this intermediate symbolic codes and assembly code is that the intermediate code need not specify the registers to be used for each operation.

3.4 CODE OPTIMIZATION

The translation of a source program to an object program is basically one of many mappings; that is, there are many object programs for the same source program, which implement the same computations. Some of these object-translated source programs may be better than other object programs when it comes to storage requirements and execution speeds. Code

optimization refers to techniques a compiler can employ in order to produce an improved object code for a given source program. Code optimization refers to the techniques used by the compiler to improve the execution efficiency of the generated object code. Any valid optimization should also protect the program's semantics. The following criteria are relevant for optimization:

(i) The optimization should capture most of the potential improvements without an unreasonable amount of effort.

(ii) The optimization should be such that the meaning of the source program is preserved.

(iii) The optimization should, on average, reduce the time and space expended.

During the intermediate code generation, it is possible to generate some redundant codes. Redundant in the sense that their elimination would not have any adverse effect on the expected result. The removal of this redundant codes is called code optimization. It is a step meant to improve the efficiency of the object code in terms of both storage utilization and execution time.

To illustrate this, it will be observed that eight codes were generated for the statement W = X * Y + Z. However, it is possible to generate a more efficient code for this statement. The 3rd and 4th line can easily be eliminated because the product of the contents of memory location X and Y are already in the accumulator; all that is needed is to just add the

content of memory location Z to it. The same principle applies to the sixth and seventh line of codes such that we have

```
LDA X
MUL Y
ADD Z
STA W
```

This is a more efficient code than the first. This type of optimization is called machine-independent optimization. Other machine independent optimization method include:

Performing operation whose operands are known at compile time. For example, in the Fortran statement

```
A = 7*22/7*R**2
```

The compiler can perform the multiplication and division involving constant and substitute 22*R**2 for the original expression. This is especially useful if such an expression occurs within a loop.

Removing from a DO-loop, the operations whose operands do not change. Consider the program segment below as an example:

```
ISUM = 0
DO 10 K = 1, 100
ISUM = ISUM + K
JX = 25
10 CONTINUE
```

It is easy to see that JX = 25 within the Do-loop will produce the same result. This could be removed out of the loop. There is also machine dependent optimization. It is done while actually generating code. This is done by exploiting the features of the machine such as better utilization of available registers, and instruction set of the machine.

3.5 ERROR HANDLING DURING COMPILATION

One of the important tasks that a compiler must perform is the detection and recovery from errors. Recovering from errors is important, because the compiler will be scanning and compiling the entire program, perhaps in the presence of errors; so as many errors as possible need to be detected. Every phase of compilation expects the input to be in a particular format, and whenever that input is not in the required format, an error is returned. When detecting an error, a compiler scans some of the tokens that are ahead of the error's point of occurrence. The fewer the number of tokens that must be scanned ahead of the point of error occurrence, the better the compiler's error-detection capability.

After detecting an error, the first thing that a compiler is supposed to do is to report the error by producing a suitable diagnostic. A good error diagnostic should possess the following properties:

(i) The message should be produced in terms of the original source program rather than in terms of some internal representation of the source program. For example, the message should be produced along with the line numbers of the source program.

(ii) The error message should be easy to understand by the user.

(iii) The error message should be specific and should localize the problem. For example, an error message should read, "x is not declared in function fun," and not just, "missing declaration."

(iv) The message should not be redundant; that is, the same message should not be produced again and again.

Errors can be detected at any phase of compilation. When errors are discovered, compilation cannot be completed. The usual practice is to continue to the end of the semantic analysis phase, in case any more errors would be encountered. Code generation in such instance cannot take place.

In some cases, it may be difficult to continue either the syntactic or semantic analysis as some assumption would have to be made about the correctness of code near the error. Analysis generally re-commences from the start of the next source program statement.

3.6 ERROR RECOVERY FROM LEXICAL PHASE ERRORS

The lexical analyzer detects an error when it discovers that an input's prefix does not fit the specification of any token class. After detecting an error, the lexical analyzer can invoke an error recovery routine. This can entail a variety of remedial actions.

The simplest possible error recovery is to skip the erroneous characters until the lexical analyzer finds another token. But this is likely to cause the parser to read a deletion error, which can cause severe difficulties in the syntax-analysis and remaining phases. One way the parser can help the lexical analyzer can improve its ability to recover from errors is to make its list of legitimate tokens (in the current context) available to the error recovery routine. The error recovery routine can then decide whether a remaining input's prefix matches one of these tokens closely enough to be treated as that token.]

3.7 ERROR RECOVERY FROM SYNTATIC PHASE ERRORS

A parser detects an error when it has no legal move from its current configuration. Some parsers use valid prefix property; they are capable of announcing an error as soon as they read an input that is not a valid continuation of the previous input's prefix. The advantages of using a parser with a valid–prefix-property capability is that it reports an error as soon as possible, and it minimizes the amount of erroneous output passed to subsequent phases of the compiler.

3.8 PANIC MODE RECOVERY

Panic mode recovery is an error recovery method that can be used in any kind of parsing, because error recovery depends somewhat on the type of parsing technique used. In panic mode recovery, a parser discards input symbols until a statement delimiter, such as a semicolon or an end, is encountered. The parser then deletes stack entries until it

finds an entry that will allow it to continue parsing, given the synchronizing token on the input. This method is simple to implement, and it never gets into an infinite loop.

3.9 ERROR RECOVERY FROM SEMANTIC PHASE

The primary sources of semantic errors are undeclared names and type incompatibilities. Recovery from an undeclared name is rather straightforward. The first time the undeclared name is encountered, an entry can be made in the symbol table for that name with an attribute that is appropriate to the current context. For example, if missing declaration error of x is encountered, then the error-recovery routine enters the appropriate attribute for x in x's symbol table, depending on the current context of x. A flag is then set in the x symbol table record to indicate that an attribute has been added, and to recover from an error or not in response to the declaration of x.

Exercise 3

1. Discuss the details of the semantic process
2. Write a quadruple for IF I < J THEN K = K * 5
3. Write explanatory note on Error handling
4. Write explanatory note on code optimization.

CHAPTER 4

4.0 INTRODUCTION TO FINITE AUTOMATA/REGULAR EXPRESSION

Let us remind ourselves of the basic set operations as this is part of the background for regular expressions

Definitions of Terms

Set: A set is a collection of objects. It is denoted by the following methods:

(i) The elements of a set are placed in curly brackets separated in comma ({ }), e.g C = {5, 6, 7}

(ii) We can use a predetermined notation in which the set is denoted e.g C = {x | x is an integer and x mod 3 = 0}.

The second example uses set builder notation. the notation is

{ template | condition}

This denotes the set of all items matching the template, which also meet the condition. This, combined with logic gives a natural way to concisely describe sets:

Examine the examples below:

$\{x \mid x < 1\} = \{0\}$
$\{x \mid x > 1\} = \{2, 3, 4, 5, \ldots\}$
$\{x \mid x \in R \Lambda x \in S\} = R \cap S$
$\{x \mid \exists y.x = 2y\} = \{0, 2, 4, 6, 8, \ldots\}$

The template can also be more complex expression.

For the above expressions, it is important to know that Λ is used for conjunction

∃y.x is used as existential quantification

Set Operations

(a) **Union**: If A and B are the two sets, then the union of A and B is denoted as: A U B = {x | x in A or x is in B}.

(b) **Intersection**: If A and B are the two sets, then the intersection of A and B is denoted as: A n B = { x | x is in A and x is in B}.

(c) **Set difference**: If A and B are the two sets, then the difference of A and B is denoted as A – B = {x | x is in A but not in B}

(d) **Cartesian product**: If A and B are the two sets, then the Cartesian product of A and B is denoted as: A x B = {(a, b) | a is in A and b is in B}

(e) **Power set**: If A is the set, then the power set of A is denoted as 2^A = P | P is a subset of A} (i.e, the set contains all possible subsets of A.) e.g A = {0, 1}

2^A = {Φ, {0}, {1}, {0,1}}

(f) **Concatenation**: If A and B are the two sets, then the concatenation of A and B is denoted as: AB = {ab | a is in A and b is in B}. For example, if A = {0, 1} and B = {1, 2}, then AB = {01, 02, 11, 12}.

(g) **Closure**: If A is a set, then closure of A is denoted as: $A^* = A^0 \cup A^1 \cup A^2 \cup ...A^\infty$, where A^i is the i^{th} power of set A, defined as A^i is the i^{th} power of set A, defined as A^i = A.A.A…i times.

$A^0 = \{\in\}$

(i.e, the set of all possible combination of members of A of length 0)

$A^1 = A$

(the set of all possible combination of members of A of length 1)

$A^2 = A.A$

(the set of all possible combination of members of A of length 2)

Therefore A* is the set of all possible combinations of the members of A.

Closure is a powerful idea. Suppose S is a set, for any $x \epsilon S$, $f(x) \epsilon S$, then S is said to be closed under f. for example, if we say that N is closed under squaring:

$\forall n\ n \epsilon N \Rightarrow n^2 \epsilon N$

What the above means in simple language is that n is a number, such that when we square it whatever we get can be found in N (where N itself is a set of natural number). This is a symbolic or mathematical way of writing volumes in small notations and it is a widely acceptable way of writing.

(a) Alphabet: It is a finite set of symbols, usually defined at the start of a problem statement.

Commonly, $\sum$ is used to denote alphabet.
$\sum = \{0, 1\}$
$\sum = \{a,b,c,d\}$

(b) String: a string over an alphabet $\sum$ is a finite sequence of symbols from alphabet, $\sum$. E.g if $\sum = \{0,1\}$, then 000 and 0100001 are strings over $\sum$. The strings provided in most programming languages are over the alphabet provided in the ASCII characters (and more extensive alphabets, such as Unicode, are becoming common).

Here are common mistakes students make when first confronted with sets and strings

Sets { a, b} = {b, a}
Strings ab ≠ ba
Sets {a, a, b} = {a, b}
Strings aab} ≠ ab

(c) Language is a set of string over an alphabet. It is not all strings that are valid in a language. Vocabulary of a language simply implies valid/meaningful strings of that language. $\sum^*$ is the set of all strings over alphabet $\sum$. The set $\sum^*$ contains all strings that can be generated by iteratively concatenating symbols from $\sum$, any number of times. if $\sum = \{a, b, c\}$

$\sum^* = \{ \epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, \ldots\}$

4.1 REGULAR EXPRESSION NOTATION

The regular expressions are another formal model of regular languages. Unlike automata, these are essentially given by bestowing syntax on the regular languages and the operations they are closed under. This can be used for specification of tokens as tokens constitute a regular set. It is compact, precise, and contains a Deterministic Finite Automata. The DFA is used to recognize the language specified by the regular expression and this can be built up so that it can recognize possible tokens of a language. Let us take a brief look into some regular expression notation.

- $a \in \mathbf{R}$, $a \in \sum$
- $\epsilon \in \mathbf{R}$
- $\mathbf{\emptyset} \in \mathbf{R}$
- $\mathbf{r_1 + r_2} \in \mathbf{R}$, **if** $\mathbf{r_1} \in \mathbf{R}$ ^ $\mathbf{r_2} \in \mathbf{R}$
- $\mathbf{r_1 . r_2} \in \mathbf{R}$, **if** $\mathbf{r_1} \in \mathbf{R}$ ^ $\mathbf{r_2} \in \mathbf{R}$
- $r^* \in \mathbf{R}$, **if** $\mathbf{r} \in \mathbf{R}$
- Nothing else is in **R**

Remark: this is an inductive definition of a set **R, the set is 'initialized' to have** ϵ and $\mathbf{\emptyset}$ **and all elements of** $\sum$. Then we use the closure operations to build the rest of the (infinite, usually) set **R,** the final clause disallows other random things being in the set.

Note: Regular expressions are syntax trees used to denote languages. The semantics, or meaning, of a regular expression is thus a set of strings i.e. a language.

A finite automaton consists of a finite number of states and a finite number of transitions, and these transitions are defined on certain, specific symbols called input symbols. One of the states of the finite automata is identified as the initial state the state in which the automata always starts. Similarly, certain states are identified as final states. Therefore, a finite automaton is specified as using five things:

(i) the states of the finite automata
(ii) the input symbols on which transitions are made;
(iii) the transitions specifying from which state on which input symbol where the transition goes;
(iv) the initial state; and

(i) The set of final states

$M = (Q, \sum, \delta, q_0, F)$ where

Q is a set of states of the finite automata

$\sum$ is a set of input symbols, and

δ specifies the transitions in the automaton.

If from a state p there exists a transition going to state q on an input symbol a, then we write $\delta(p, a) = q$. Hence, δ is a function whose domain is a set of ordered pairs, (p, a), where **p** is state and **a** an input symbol, and the range is a set of states.

Therefore, it is easy to see that δ defines a mapping whose domain will be a set of ordered pairs of the form (p, a) and whose range will be a set of states. that is, δ defines a mapping from:

Q X ∑ to Q,

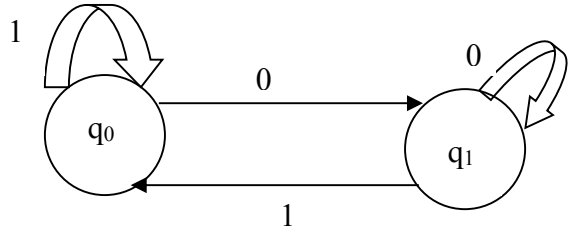
q_0 is the initial state, and

F is a set of Final states of the automata

$\delta(q_0, 0) = q_1, \delta(q_0, 1) = q_0$

$\delta(q_1, 0) = q_1, \delta(q_1, 1) = q_0$

A directed graph exists that can be associated with finite automata this is called transition diagram.



A tabular representation can also be used to specify the finite automata. A table whose number of rows is equal to the number of states, and whose number of columns equals the number of input symbols, is used to specify the transitions in the automata. The first row specifies the transitions from the initial state; the rows specifying the transitions from the final states are marked as *.

For example, for the above it can be specified as

	0	1
q_0	q_1	q_0
*q_1	q_1	q_0

A finite automaton can be used to accept some particular set of strings. If x is a string made of symbols belonging to $\sum$ of the finite automata, then x is accepted by the finite automata if a path corresponding to x in a finite automata starts in an initial state and ends in one of the final states of the automata, ie there must exist a sequence of moves for x in the finite automata that takes the transitions from the initial state to one of the final states of the automata. Since x is a member of $\sum^*$, we define a new transition function δ_1, which defines a mapping from Q X $\sum^*$ to Q. And if $\delta_1 (q_0, x)$ = a member of F, then x is accepted by the finite automata. If x is written as wa, where a is the last symbol of x, and w is a string of the remaining symbols of x, then:

$\delta_1 (q_0, x) = \delta\{\delta_1 (q_0, w), a\}$

For example:

Check if the string 010 is acceptable by the automata

Let x be 010, to find out if x is acceptable,

$\delta_1 (q_0, 0) = \delta (q_0, 0) = q_1$

Therefore, $\delta_1 (q_0, 01) = \delta\{\delta_1 (q_0, 0), 1\} = q_0$

$\delta_1 (q_0, 010) = \delta\{\delta_1 (q_0, 01), 0\} = q_1$

Since q_1 is a member of F, x = 010 is accepted by the automa.

Class work

Suppose that x = 0101 is it acceptable?

In the finite automaton discussed above, since δ defines mapping from Q X $\sum$ to Q, there exists exactly one transition from a state on an input symbol; and therefore, this finite automata is considered a deterministic finite automata (DFA)

4.2 DETERMINISTIC FINITE STATE AUTOMATA

It is a simple machine that reads an input string one symbol at a time-and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. Thus, a DFA conceptually consists of 3 parts:

- a tape to hold the input string. The tape is divided into a finite number of cells Each cell holds a symbol from Σ
- a tape head for reading symbols from the tape
- a control, which itself consists of 3 things:
- a finite number of states that the machine is allowed to be in
- a current state, initially set to a start state.
- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

The tape head reads the current tape cell and sends the symbols found there to the control. Then the tape head moves to the next cell. The tape head can only move forward. The control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been traversed, the final state is examined. If it is an accept; otherwise, the string is rejected. All the above can be summarized in the following formal definition.

A Deterministic Finite State Automaton, DFA is a 5-tuple:

$M = (Q, \sum, \sigma, q_0, F)$

Where

Q is a finite set of states
$\sum$ is a finite alphabet
$\Sigma: Q \times \sum \rightarrow Q$ is the transition function (which is b total)
$q_0 \in Q$ is the start state.
F Q is the set of accept states

Examples

DFA $M = (Q, \sum, \sigma, q_0, F)$ where
$Q = \{q_0, q_1, q_2, q_3\}$
$\sum = \{0,1\}$

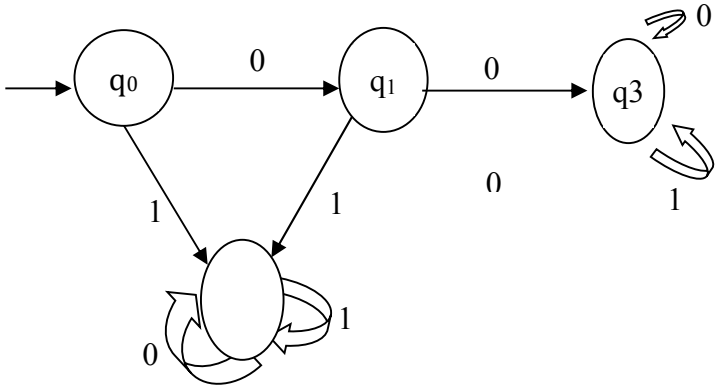
The start state is q_0 (this will be our convention)

$F = \{ q_1, q_2 \}$

Σ is defined by the following table:

	0	1
q_0,	q_1	q_3
q_1,	q_2	q_3
q_2	q_2	q_2
q_3	q_3	q_3

The state transition diagram is shown below:



Notice that the start state is designated by an arrow with no source. Final states are marked by double circles. The strings accepted by M are:

{0, 00, 000, 001, 000, 0010, 0011, 0001,…}

The transition function is total, so every possible combination of states and input symbols must be dealt with. Also, for every (q, a) ϵ Q x ∑, there is exactly one next state (which is why these are deterministic automata) thus given any string x over ∑, there is only one path starting from q_0, the labels of which form x.

If q is an accept state, then we call q a success state. If not, it is called a failure state. In the example above q_3 is a failure state and q_2 is a success state.

The language recognized by a DFA M is the set of all strings accepted by M, and is denoted by L(M).

What is the language accepted by M, i.e. what is L(M)?

L(M) consists of 0 and all binary strings starting with 00. Formally, we could write

$$L(M) = \{0\} \cup \{00x \mid x \in \sum^*\}$$

Exercise 4

1 (a) Define a DFA
(b) Define a Finite Automata

2 Construct the finite automata for accepting strings of zeros and ones that contain equal number of zeros and ones.

3 Explain the meaning of the following
(i) $\{x \mid x > 1\}$
(ii) $\{x \mid x \in R \land x \in S\}$
(iii) $\{x \mid \exists y.x = 2y\}$

CHAPTER FIVE

5.0 INTRODUCTION TO FORMAL GRAMMAR AND LANGUAGES

Grammar: The rules in a language for changing the form of words and joining them into sentences.

Lexicon: The list of verbal words in a particular language is called its lexicon.

Syntax: The rules that state how words and phrases must be used in a computer language.

Syntax Tree: A tree-like diagram that describes the structure or syntax of a sentence by breaking it down into its constituent.

Meta Language: A language used to describe the structure of other languages. E.g BNF

5.1 FORMAL GRAMMAR DEFINITION

Formally, a grammar is defined to be a quadruple (four-tuple)

G = (V, T, S, P)

Where V = set of non-terminal symbols

T = Set of terminal symbols

S = the starting symbols; a distinguished nonterminal symbol from which all strings of a language are derived.

P = Production rules

Those symbols appearing as a left part of rule are called non-terminals. The symbols on the right part of the rule which are not enclosed in corner bracket are called Terminal symbols.

		1 2
<ODD NUMBER>::=		<NO><LDIGIT>|<LDIGIT>
		3 4
<NO>::=		<NO><DIGIT>|<DIGIT>
		10
<DIGIT>::=		0 | 2 | 4 | 6 | 8 |<LDIGIT>
<LDIGIT>::=		1 | 3 | 5 | 7 | 9

In the grammar above, there are 15 production rules.

One purpose of a grammar is to describe all sentences of a language with a reasonable number of production rules. A grammar can contain more than one rule describing how a particular syntactic entity may be formed e.g.

<LDIGIT>::= 1
<LDIGIT>::= 3

<ODD NUMBER> regarded as the starting symbol, from it, all other strings are derived.

<ODD NUMBER>, <NO>, <DIGIT>, <LDIGIT> are non-terminal symbols: they can be further broken down by one or more rules to terminal symbols. 0, 2, 4, 6, 8, 1, 3, 5, 7, 9 are all terminal symbols. The terminal symbols can be likened to token of a language.

5.2 DERIVATION OF STRINGS FROM FORMAL GRAMMAR.

Derivation is done by reducing or re-writing the non-terminals into another form by applying one or more of the rules, beginning from the starting symbol by applying one or more of the production rules. For example, let us see how string 79 will be derived

<ODD NUMBER> ===> <NO><LDIGIT> from rule 1
===> <DIGIT><LDIGIT> from rule 4
===> <LDIGIT><LDIGIT> from rule 10
===> 7<LDIGIT> from rule 14
===> 79 from rule 15

Let G be a grammar, we say that string V directly produces the string w written v ===>w, if we can write V = xUy, w = xuy for some strings x and y, where U ===> u is a rule of G. We also say that w is a direct derivation of V, or that w directly reduces to V. the strings x and y, of course, may be empty. Therefore, for any rule U ===> u of the grammar G, we have U ===> u. In short, we say w is a direct derivation of V if we can derive w from V by the application of only one rule.

Note:
α===>β implies that α generates β by a direct application of exactly one production in the available set of production e.g

<NO><DIGIT> == => <DIGIT><DIGIT>

α===> +β means that the string β can be derived from the string α by one or more application of production of the grammar e.g.

<NO>===> +<DIGIT> <DIGIT><DIGIT>

α===> *β means that the string β can be derived from the string α by zero or more application of the production of the grammar. It simply means that we can have the rules repeatedly used.

E.g. <ODDNO>===> *<DIGIT> <DIGIT><LDIGIT>
(application of 3 rules)
<NO> ===> *<NO> (No rule is applied)
It then means α ===> *β, if α ===> +β or $\alpha = \beta$

5.3 CONSTRUCTION OF GRAMMAR FOR LANGUAGES

Example: Construct a grammar whose language is the set of even integer.

Solution: Sample sentences of the language are

2, 4, 6, 8, 10, 12……..

From these sentences, if length of the string |s| = 1 then it can be 2, 4, 6, and 8

If length of string |s| > 1, then the last symbol can only be any of the digit 0, 2, 4, 6, and 8 while other symbol can be any of thr digit 0 – 9.

The grammar below gives the answer

<EVEN NUMBER> ::= <NO><LDIGIT>|<LDIGIT>
<NO> ::= <NO><DIGIT>|<DIGIT>
<DIGIT> ::= <LDIGIT>1 | 3 | 5 | 7 | 9
<LDIGIT> ::= 0 | 2 | 4 | 6 | 8

5.4 CLASSIFICATION OF GRAMMAR

The definition of production allows for a wide variety of string transformation. Certain restrictions on the form of production give rise to different categories of grammar. Based on these restrictions formal grammars can be grouped into 4 classes or types.

a. **Type 0**

This is of the form α ===> β: (i.e. α, β belongs V union T)

In this type, the left part α can also be a sequence of symbols and the right part can be empty. There is no restriction imposed. It is also called phrase structure grammar. Example are

(i) aAbbAab ----> abAB

(ii) S -----> QNQ

QN -----> QR

RN -----> NNR

Note that some of the production on the left part contain more than one non-terminal symbol. Production that eliminate or erase symbols are permitted. This allows the intermediate string to expand or contract

An example of erasing production is

Aab -------> Ab

In the above example, A is erased from the context Aab

b. **Type 1**

If a grammar has the property that for all productions of the form α ------> β, $|\alpha| <= |\beta|$, where $|\alpha|$ denotes the length of α. It is also called context sensitive or context dependent grammar. In addition, the left hand side of production may not contain more than 1 non-terminal symbol.

Example are

i Bb ------>Bb
ii aAa-------->aaBa

c **Type 2**

If a grammar has the property that all left parts of production consist of a single non terminal symbol (and non blank right-hand string) it is called type two or context-free grammar. That is, all the left part of production consist of only one non-terminal. Most programming languages are defined by context-free grammar e.g the Odd and even grammars given as example belong to this category.

d **Type 3**

In this type, when at most, one non terminal symbol is used in both the right and left-hand side of s of production, the production is said to be linear. If the non-terminal occurs to the right of all other symbols on the right-hand side of production, the production is called, a right linear production. Left lineted bar is similarly defined. A language is called Regular if it can be generated by a right or left linear grammar.

That is, if each production of the grammar is one of the form AaA -------> aB where a is a terminal symbol and A and B are non-terminal, then the grammar is of type 3, right linear or regular. In general, a regular grammar may be either of the form: A -----> a, or A ------> Ba or A ------> aB.

Clearly, it can be shown that all type 3 grammars are type 2 grammar and all type 1are type 0.

CONTEXT-FREE GRAMMAR

Context Free Grammar (CFG) notation specifies a context-free language that consists of terminals, nonterminals, a start symbol, and productions.

The terminals are nothing more than tokens of the language constructs. Nonterminals are the variables that denote a set of strings. For example, S and E are nonterminals that denote statement strings and expression strings, respectively, in a typical programming language. The non-terminals define the sets of strings that are used to define the language generated by the grammar. Like natural languages, a programming language can be described by a grammar. A context-free grammar is a language generator just like regular expressions but it is more powerful. Formally, a context-free grammar is a finite set of rules. Each rule consists of a symbol (called the left-hand side), an arrow and a sequence of symbols (called the right-hand side). The purpose of grammar is to describe languages, that is set of possible sequences of symbols. Context-free grammars describe languages in a generative way: pick a nonterminal (in this case A) and apply the rules until there are no nonterminals left, and you have a sequence

of (terminal) symbols; do this in every possible way, and you have a set of such sequences. This set can be infinitely large.

Context Free Grammar (CFG) is a four-tuple denoted as:

G = (V, T, P, S)

Where:

V is a finite set of symbols called nonterminals or variables

T is a set of symbols called terminals

P is a set of productions, and

S is a member of V, called as start symbol.

CONTEXT-SENSITIVE GRAMMAR

A context-sensitive grammar (CSG) is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols. Context-sensitive grammars are more general than context-free grammars but still orderly enough to be parsed by a linear bounded automaton.

The concept of context-sensitive grammar was introduced by No am Chomsky in the 1950s as a way to describe the syntax of natural language where it is indeed often the case that a word may or may not be appropriate in a certain place depending upon the context. A formal language that can be described by a context-sensitive grammar is called a context-sensitive language.

In formal language theory, a context-sensitive grammar is defined by a finite set of rewriting rules of the form

[X]Y[Z] → W

Where X, Z, and W are strings of symbols, and Y is a single symbol.

Such a rule means: within a string of symbols, if a substring is equal to XYZ, the Y may be replaced with W.

This can be regarded as a restricted form of general grammar, namely, one in which all rules are of the form XYZ → XWZ. Unlike in general grammars, only one symbol can be written at a time. If all rewriting contexts are empty, we have a context-free grammar.

A context-sensitive grammar has productions of the form xAz → xyz,

Where A is a nonterminal and x, y, z are strings of grammar symbols with y → A. The production S → A is also allowed.

If S is the start symbol and it does not appear on the right side of any production. A context-sensitive language has a context-sensitive grammar.

Example

S→ aTb | ab

aT → aaTb | ac

Exercise 5

1. Explain Context free grammar and Context sensitive grammar.
2. Explain different classification of grammar.

REFERENCES

Kakde O. E.; Comprehensive Compiler Design, Lanmi Publication Ltd; 22 Golden House Daryagary, New Delhi – 110002.

Steven S. Muchnick; Compiler Design and Implementation, Morgan Kaufmann Publishers. Inc 340 Pine Street. Sinth Floor, San – Francisco CA 94104 – 3205 U.S.A, 1997.

Kip R. Iruine; Assembly Language for Intel-Based Computers, 5th Edition. Prentice Hall Pearson Education, Inc. Upper Saddle River, NJ 07458.

Ytha Yu and Charles Marnt; Assembly language programming and organization of the IBM PC. Mctrane. Hill, Inc; 1992.

Alfred V. A. John E. H. and Jeffrey D. U. D.; Data Structures and Algorithm, 9th Edition. Pearson Education (Singapore) Pte Ltd. Indi, 2003.

B. Ram; Fundamentals of Microprocessor and Microcomputers, 6th Edition. Phampat Rai Publication (P) Ltd 22, Ansari Rd Daryany, Mew Delhi – 110002.

Wilson J.R. Four Colors Suffice: How the Map Problem Was Solved. Princeton University Press, 2013.

Aho A., Lam M.S., Sethi, R. & Ullman, R. D.; Compilers: Principles, Techniques, and Tools", 2nd edition, Pearson, 2013.

NOTES