INF-3036

GAUHATI UNIVERSITY

Centre for Distance and Online Education

Third Semester

(Under CBCS)

M.Sc.-IT

Paper: INF 3036

COMPILER DESIGN

Contents: **Page No.**

Contributors:

Dr. Rita Chakraborty Asstt. Prof., GUCDOE	(Block 1 : Units- 1,2,3,4,5,6,7 & 8) (Block 2 : Units- 1,2,3,5 & 6)
Dr. Nabamita Deb Asstt. Prof., Dept. of IT, G.U.	(Block 1 : Unit- 9)
Mr. Chandan Kalita Asstt. Prof., Dept. of IT, G.U.	(Block 1 : Unit- 10), (Block 2 : Unit- 4)
Dr. Pranab Das Associate Prof., Dept. of Computer Applications Assam Don Bosco University	(Block 2 : Units- 7 & 8)

ISBN: 978-81-982028-9-5
October, 2024

BLOCK- I

UNIT: 1
INTRODUCTION TO COMPILER

Unit Structure

1.0 INTRODUCTION

In this unit, you will learn the basic concepts of compilers. A compiler is the system software that translates a program written in a source language into an equivalent target language. The source languages may be any high level language like C, C++ or Java. The target language may be any programming language or the machine language. Therefore, compiler is basically a translator that accepts programs written in high level language and translates them into machine language equivalent programs so that they can be executed. The output code produced by the compiler is termed as the object code. Apart from translation, compiler also produces error messages if the program does not abide by the language specification. You must have learnt to write programs in high level languages. Do you know what actually happens inside the computer, when you execute these programs? Here, the role of a compiler comes into play. It passes through different phases and finally it generates codes equivalent to what you have written. This is what the machine equivalent object code is. This code is machine dependent and carries out the intended instructions in order to produce results of computation. Further, you will get to learn how different phases work together, what their functionalities are and how they combine form the desired output.

1.1 UNIT OBJECTIVE

After going through this unit, you will be able to:

- Define complier
- Understand the different types of compilers
- Know some basic concepts of compilers
- Know how each phases of a compiler work

1.2 BASIC TERMS AND THEIR DEFINITIONS

1. Pass of a compiler: Compilation is a complex process broken into different chunks or phases. These chunks are called passes of compiler. This process involves not only producing the executable codes but also include several other stages like preprocessing, assembling, linking and loading etc.

2. Single-pass compiler: When the compiler passes through each compilation unit or chunk exactly once, it is termed as the single pass compiler. The programming languages C, Pascal, FORTRAN etc fall into this category. Single pass compilers have the advantage of producing machine code faster, being efficient in case of short programs and memory requirement.

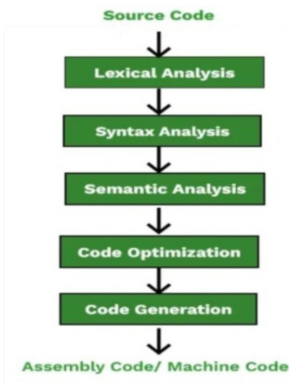


Fig. 1.1: Single-pass compiler

3. Multi-pass compiler: When the compiler passes the source code through several intermediate passes during compilation process is called multi-pass compiler. Each pass takes the output of its previous pass as input and produces some intermediate output. Programming language like C++, Java, Lisp etc use multi-pass compiler. It works well with complex programming languages. Since multiple passes produce more optimized code, code generation is better than single-pass.

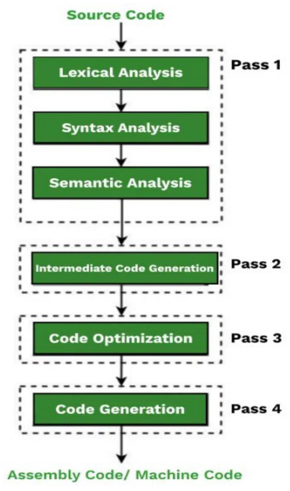


Fig. 1.2: A 4-pass compiler

4. Interpreter: It is also a kind of compiler which translates a program written in high level language into corresponding machine code. The only difference is that compiler converts a source code into equivalent target code. However, interpreter scans the program statement wise and then generates the machine executable format of the source. Programming languages like Python, Ruby, and JavaScript use interpreters.

5. Assembler: An assembler is system software that interprets a program written in assembly language into relocatable machine language. An assembly language is a low level program contains symbolically coded instructions. Assembler works by converting assembly language instructions into object code. An assembler may also be single-pass or multi-pass.

CHECK YOUR PROGRESS- I

1. A compiler is basically a _____________.
2. Apart from translation, compiler also produces ___ ______.
3. When does a compiler produce error messages?

4. Compilation is a complex process broken into different____.
5. Write down some advantages of a single pass compiler.
6. An _______ scans the program statement wise.
7. An assembler interprets a program written in assembly language into __ machine language.

1.3 ANALYSIS - SYNTHESIS MODEL OF COMPILATION

The process of compilation is mainly divided into two major parts- Analysis and Synthesis.

1.3.1 Analysis Phase: The analysis phase of a compiler mainly deals with analyzing codes of a program. The analysis phase performs the scanning of code, checks whether the codes conform to the rules or syntax of the language. This phase is important because accurate and efficient intermediate and target code will be generated once analysis phase correctly performs its task. Codes are scanned and checked if there is any lexical error. Once scanning is successful, codes are checked for syntactical representations. If they conform, successful generation of parse trees happen. Else error messages are generated. Successful syntactic representation may also lead to performing the semantic representation of codes. In semantic representation, mainly type checking is performed. Apart from these, this phase is mainly responsible for creating the symbol table. It is that part which mainly deals with performing the initial phases of compilation.

1.3.2 Synthesis Phase: The synthesis part is responsible for processing the target code. The output of analysis phase is captured by this phase. Generally, this portion does not depend on source language and mainly deals with efficient optimization of codes. Finally, generation of target codes is also a responsibility of this phase. Synthesis phase mainly deals with producing the target code.

1.4 PHASES OF A COMPILER

As already mentioned, a compiler operates in phases, each phase transforms a source program from one representation into another. Each phase does unique operations on the program, transforming into some intermediate representation. Each output representation goes to the next phase and again the transformation happens. Two other sub-phases also occur during compilation- symbol-table management and error handling. These two sub-phases interact with all the phases of the compiler.

The process of compilation mainly is divided into six major phases. They are- lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.

1.4.1 Lexical Analysis or scanning:

Lexical analysis or scanning is the first phase of a compiler. This phase reads the sequence of characters of the source program and finds the meaningful strings of characters. These strings are called lexemes and based. Based on its meaning, lexemes can be grouped into some categories. Thus, a lexeme can be identified as a keyword, identifier, number, operator, punctuation character, strings, comments or white spaces. Identifiers are the arbitrary sequences of letters, numbers or some special characters. Integers, floating point values, fractions and binary, octal or hexadecimal values fall under the numbers.

Each lexeme is represented using a token of form <*token-name, attribute-value*>. Here, *token-name* is an abstract symbol representing the lexeme and *attribute-value* is used to provide additional information about the lexeme. This *attribute-value* is optional. Also, suppose, there are two lexemes < and <= . Both lexemes are falls in the category *operator*. Now, tokens for the lexemes < and <= can be <*OP, LT*> and <*OP, LE*> respectively. For both the token *token-name* is same i.e. *OP*, because both the lexemes falls into same category *operator* and it is represented using abstract symbol *OP*. But for both the tokens, the attributes values are not same, because < and <= are different operator. Thus, *LT* denotes attribute value for the operator < (i.e. less than) and *LE* denotes attribute value for the operator <= (i.e. less than equal). Again, suppose, there are another two lexemes named as *add* and *pos,* which represents identifiers. Then the tokens for *add* and *pos* can be <*id, 1*> and <*id, 2*>. Here, *id* represents abstract symbol for *identifier* and *attribute-values* 1 and 2 represent the corresponding symbol table entries for the identifiers *add* and *pos* respectively. These symbol table entries hold the information about the identifiers i.e. type, size, scope etc. In case of operator, this symbol table entry is not required as the operator has no other attributes other than its category. Thus, its category is put directly in the place of *attribute-value*.

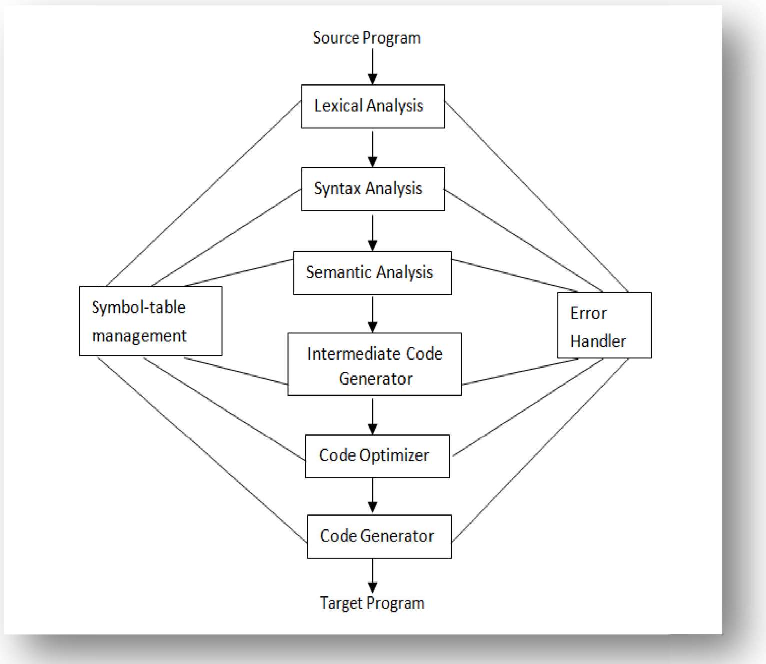


Fig. 1.3: Phases of compilation

For instance, consider the following C code:

```
void main()
{
int a, b;
a=10;
printf("The value of a is== %d",a);
}
```

This code contains the following lexemes:
'void', 'main', '(' , ')', '{',' int', 'a', ',', 'b', ';', 'a', '=', '10', ';', 'printf', '(', "The value of a is== %d", ',', 'a', ')', ';', '}'.

Here,

void, main, int, printf	-----> Keyword
(,), {, }, =	-----> Special character
int, printf	-----> Keyword
a, b	-----> Identifier
10	-----> Number
,, ;	-----> Separator
"The value of a is== %d"	-----> Literal

The lexer program i.e. lexical analyzer tries to detect the lexemes from the input code and for each lexeme corresponding token is also generated. These tokens are passed to the parsing phase in order to generate the grammatical structure from the code. Non-token elements like white spaces are eliminated during scanning and are excluded from the token count. Similarly, the string literals inside double inverted comma are always considered as a single token.

1.4.2 Syntax Analysis or Parsing:

One of the most important phases of compilation which performs all the syntax related analysis of the code. It checks whether the statements and expressions conform to the rules or syntax defined for the language as well as whether they are correctly formed. Parsing can begin once scanning is successful and there is no lexical error. It takes the tokens from the lexical analysis phase, groups them and tries to construct grammatical phrases. These grammatical phrases are later represented in terms of graphical structures called the parse trees. Parsing is used to determine if a string of tokens can be generated by a set of grammar rules. The grammar rules are represented in terms of Context Free Grammars (CFG). Parsing is successful if the string is accepted by CFG; else we may encounter a syntax error. Parsing falls into two classes- top-down and bottom-up. When parsing begins from the root and proceeds towards the leaves; it is termed as top-down parsing. Alternatively, in bottom-up parsing, construction begins at the leaves and proceeds towards the root.

1.4.3 Semantic Analysis:

Once a program is successfully parsed, it is checked for generation of semantic errors and gathering of type information. This is necessary for subsequent phases of code-generation. It uses the hierarchical structure generated by syntax analysis phase and identifies the operators and operands of expressions and statements. The function of semantic analysis phase is type checking. It finds out the type of each identifier and also checks whether each operator has operands permitted by the source language specification. Like for example, an array in C language cannot contain a real number as its index. Sometimes, situation may happen when a binary operator adds an integer and a floating point number. The compiler sometimes converts the integer into floating point equivalent and then performs the arithmetic. During type checking, the type of each identifier must be determined. Each variable may refer to a local storage, a global variable or a function parameter. Each variable definition is stored in symbol table. Name resolution attempts to solve the problem of variable referencing with the help of symbol table. Semantic analysis also deals with examining limits of arrays or bad pointer traversal.

For example, consider the following C-code

```
void main()
        {
        float a, b;
        a= b * 10;
        printf("The value of a is== %f",a);
        }
```

The semantic analyzer will find a type mismatch in the expression a= b * 10. Both a and b are floating point numbers and 10 is an integer. There is a difference between them in terms of storage allocation as integers require less space than floats. Representing 10 by 10.0 can compensate this mismatch. Therefore, the expression would be a= b * 10.0.

1.4.4 Intermediate Code Generator:

Some compilers generate an explicit intermediate representation of the source program. This representation must be easy to produce and easy to translate into target code. It is like the abstract structure of the source language which facilitates efficient optimization and target code generation. Sometimes, the intermediate code is generated in terms of "three-address code". The three-address code is a sequence of instructions which is like the assembly language with every memory location working like a register.

Consider the statement a= b * 10. In three-address code representation, it will be represented as-

```
t1 = float (10)
t2 = id2 * t1
id1 = t2
```

There are different approaches to intermediate representation-

- An **Abstract Syntax Tree (AST)**, which is simply a tree-like structure without much of the optimization done. A post-order traversal of the AST at each node can help generating the assembly level code.
- A **Directed Acyclic Graph (DAG)** is a simplified version of AST. Nodes are greatly simplified by eliminating the common sub expression. The values of DAGs do not change as common sub-expressions can be evaluated in any order. However, this assumption does not hold in case of conditional structures (like if or switch) or loop structures (like for, while) as repeated statements may modify some of the values.

1.4.5 Code Optimization:

One of the most important phases of compilation is code optimization. This phase attempts to improve the intermediate code. A considerable amount of compiler time is spent in generating a faster-running machine code. The transformation must take place in such a manner that it preserves the meaning of the program. Code improvisation also speeds up the program by a considerable amount; which in turn improves the running time of the program. Optimization may occur with or without flow control. Sometimes, optimization may happen in a straight line sequence, within a single block. This technique where no flow control is required is known as local optimization. Sometimes, changes may take place in the entire body of a function or procedure. This is global optimization where flow control is a must. Here, the optimization is complicated to some extent.

For instance, optimized code statement of the three-address code mentioned in Section 1.4.4 would be as follows:

```
id1 = id2 * 10.0
```

This representation is considered to be efficient in terms of memory requirement as well as CPU execution time.

1.4.6 Code Generation:

The final phase of compilation is code generation. It accepts the optimized code from the previous stage and produces executable machine code. The target code consists of absolute machine code, relocatable machine code or assembly code. The task of this phase includes translation of intermediate codes into machine codes. Absolute machine code has the advantage of being placed at its fixed location in memory. They can be executed immediately after they are produced. The relocatable machine codes of sub-programs need to be linked together and then loaded for execution. Though a considerable amount of time is consumed while linking and loading the sub-programs; they have the advantage of being compiled separately. This provides a great deal of flexibility to the process.

The benefit of assembly code is easy generation of symbolic instructions. Based on these symbolic notations, the assembler generates corresponding machine codes. The code generated in optimization phase can be represented as follows:

```
MOVF id2, R1
MULF 10.0, R1
MOVF R1, id1
```

F signifies that the statements deal with floating point numbers.

CHECK YOUR PROGRESS- II

4. The compilation process is mainly divided into _____ and _______ phases.
5. During semantic analysis _________ ____ is performed.
6. Target code is produced during _____________phase.
7. Two other sub-phases also occur during compilation are- _______and ______.
8. What is lexeme?
9. ______ are passed to the parsing phase in order to generate the grammatical structure form the code.
10. The graphical structure represented during syntax analysis phase is called ______.
11. The grammar rules are represented in terms of ____________.
12. In bottom-up parsing, construction begins at the ______and proceeds towards the________.
13. Rules have ________________at its left side and a combination of ____________ ____________at its right.
14. During ___________ ______________, the type of each identifier must be determined.

16. What are the approaches to intermediate representation?
17. Code optimization phase attempts to improve the intermediate code. State whether true or false.
18. Optimization may occur with or without flow control. State whether true or false.
19. What is the advantage of absolute machine code?

1.5 SUB-PHASES OF COMPILER

It is already mentioned in Section 1.4, Fig. 1.3, that each phase of compilation interacts with two sub-phases namely- symbol table management and error handler.

1.5.1 Symbol Table Management:

During program development, variable, function names are declared. These are called identifiers. Keeping track of the names as well as the attributes of these identifiers is very important. These attributes may include the type and size of the identifiers, their scope and in case of procedure calls, their names, the number and types of arguments and also their return type. These attributes need to be recorded somewhere so that their information may be retrieved when compilation proceeds. A symbol-table is a data structure which stores the records of each identifier declared in a program. During lexical analysis or scanning phase, the lexer (lexical analyzer) determines the identifiers. These identifiers are included into the symbol-table. However, the attributes are not yet determined. The remaining phases enter information about these identifiers into the symbol table. Like for example, the semantic analysis phase requires type information of the identifiers. Symbol table is indexed by the name of the identifier as its key field. It should be present inside the main memory throughout the compilation process because it is accessed every time an identifier is referenced. For example, the C statements

```
int x=10;
float y=15.2;
char z=’a’;
```

During tokenization, the lexical analyzer detects the variable names x, y and z as identifiers. The information about them is put into the symbol table as follows:

Table 1: Instance of a symbol table for variable

Symbol_Name	Kind	Type	Scope
x	var	int	Local
y	var	float	Local
z	var	char	Local

Again, consider the following statement-

void sum(int x, int y);

This statement has the function name **sum** as the identifier and which has two integer arguments x and y respectively. The symbol table contains the information of the function and the arguments in the following manner:

Table 2: Instance of a symbol table for function

Symbol_Name	Kind	Type
sum	func	int -> void

Table 3: Instance of a symbol table for arguments

Symbol_Name	Kind	Type
x	arg	int
y	arg	int

This way symbol tables are constructed for variables and function declarations.

1.5.2 Error Handler:

An integral part of compilation process is the handling of errors. Each phase may encounter errors. It must be able to deal with the error so that compilation can proceed. Lexical errors may happen if the scanner is unable to determine tokens in a program. Similarly, errors do not happen if a string of tokens can be generated by the grammar. If they violate the syntax rules, syntactic errors take place. Most of the errors are handled in syntax and semantic analysis phases. An Error handler must be able to detect error first. Once detected, reporting should be done and finally recovery mechanisms are employed. We shall discuss about error recovery techniques later on. Errors may be detected during the time when compilation occurs. Errors of such kind may be syntax errors or referring to non-existent file etc. They prevent the program form being compiled. Such errors are called compile-time error. Similarly, errors may be detected when the program is executed. Sometimes, reference to invalid data inputs or non-existent memory may be provided to a program. Sometimes, the program may experience lack of sufficient memory to run a program. Such errors are termed as run-time errors. Errors may be logical, which may occur when the codes are written. The program produces undesired output if errors happen in program logic development.

1.6 SUMMING UP

- A compiler is the system software that translates a program written in a source language into an equivalent target language. The output code produced by the compiler is termed as the object code.
- Compilation is a complex process broken into different chunks or phases. These chunks are called passes of compiler.
- When the compiler passes through each compilation unit or chunk exactly once, it is termed as the single pass compiler. When the compiler passes the source code through several intermediate passes during compilation process is called multi-pass compiler.
- The compiler converts a source code into equivalent target code. An interpreter scans the program statement wise and then generates the machine executable format of the source. An assembler interprets a program written in assembly language into relocatable machine language.
- The analysis phase of a compiler mainly deals with analyzing codes in terms of scanning or doing parsing of a program. The synthesis part is responsible for processing the target code.
- The process of compilation mainly is divided into six major phases. They are- lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.
- The lexical analysis phase reads the characters of the source program and groups them into tokens. A token may be a keyword, identifier, numbers, operators, punctuation character, strings, comments or white spaces. The sequence of characters forming a token is called as lexeme or lexical item.

- Syntax analysis phase checks whether the codes conform to the rules or syntax defined for the language. It takes the tokens from the lexical analysis phase, groups them and tries to construct grammatical phrases.
- During semantic analysis, a program is checked for generation of semantic errors and gathering of type information.
- Some compilers generate an explicit intermediate, easy to produce and easy to translate representation of the source program. It is like the abstract structure of the source language which facilitates efficient optimization and target code generation. Sometimes, intermediate code is generated in terms of "three-address code".
- Code Optimization phase attempts to improve the intermediate code. A considerable amount of compiler time is spent in generating a faster-running machine code. Code Optimization may be local or global.
- Code generation phase accepts the optimized code from the previous stage and produces executable machine code. The target code consists of absolute machine code, relocatable machine code or assembly code.
- A symbol-table is a data structure which stores the records of each identifier declared in a program. Their attributes may include the type and size of the identifiers, their scope and in case of procedure calls, their names, the number and types of arguments and also their return type.
- Each phase of compilation must be able to deal with errors. Errors may be compile-time, run-time or logical error.

1.7 ANSWER TO CHECK YOUR PROGRESS

1. translator
2. error messages
3. A compiler produces an error messages when it does not abide by the specification of the language.
4. Chunks or phases
5. Single pass compilers have the advantage of
   1. producing machine code faster
   2. being efficient in case of short programs and memory requirement.
6. Interpreter

7. Relocatable
8. Analysis, synthesis
9. type checking
10. synthesis
11. symbol-table management, error handling
12. The sequence of characters forming a token is called as lexeme or lexical item.
13. Tokens
14. parse tree
15. Context Free Grammars (CFG)
16. leaves, root
17. non-terminal, terminal or non-terminal
18. type checking
19. There are two approaches to intermediate representation-
    1. An Abstract Syntax Tree (AST), which is simply a tree-like structure without much of the optimization done.
    2. A Directed Acyclic Graph (DAG) is a simplified version of AST.
20. True
21. True
22. Absolute machine code has the advantage of being placed at its fixed location in memory. They can be executed immediately after they are produced.

1.8 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is a compiler? Answer in brief.
2. Why do we call a compiler a translator? Justify.
3. What do you understand by an interpreter?
4. What do you understand by an assembler?
5. Describe the functionality of synthesis phase.
6. What do you understand by tokens?
7. What do you understand mean by scanning in compiler?
8. What is lexical analyzer?
9. What is parsing? What is role of a parser?
10. What is parse tree? Describe in brief.
11. What do you mean by Context Free Grammar?
12. What is production rule? Discuss in brief.
13. What do you mean by syntax error?

14. What are the two major functions of semantic analysis phase?
15. What is three-address code? Explain in brief.
16. What are the different forms of intermediate representation?
17. What does the code optimization phase attempt to do?
18. What is absolute machine code? Discuss.

B. Long answer type questions.

1. What do you understand by pass of a compiler? What are its types? Explain each of them.
2. Explain the architectural framework of a typical 4-pass compiler.
3. How many major phases is the compilation process divided into? Explain each of them.
4. What are the two classes of parsing? Explain each of them with example.
5. What do you mean by type checking? Explain.
6. Explain each phases of compilation elaborately.
7. What are the two major sub phases of a compiler? Explain each of them.

1.9 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). *Compiler design: theory, tools, and examples*.
- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C* (pp. I-XVIII), Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: principles, techniques, and tools*, Second Edition.

×××

UNIT: 2
COMPILER OVERVIEW

Unit Structure

2.0 INTRODUCTION

The function of a compiler is to accept statements written in high level language and translate them into equivalent sequences of machine level instructions. For example, during the processing of statement a= b * c + d, the compiler does not directly perform the operations. Rather, issues a sequence of instructions that perform multiplication and addition operations. In broader sense, a compiler is a program that accepts as input a program written in high level language and produces an equivalent program in machine language. The input program is called as the source program and the machine level program is known as the target program. In this unit, we shall learn how a source program passes through different phases in order to generate an equivalent target program. There are six major phases and two sub-phases which we shall try to learn by considering some examples.

2.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Know the features of a language processor
- Differentiate between the functionalities between major and widely used language processors
- Understand the differences between high level language and machine level language
- Explain how each major phases of a compiler works

2.2 LANGUAGE PROCESSORS

A language processor is a program that translates a high level language source program into an equivalent target program. Apart from this, an important role of the compiler is to report error messages detected during translation process. The target program may be an executable machine language program. An interpreter is a language processor that executes the statements specified in the source program instead of producing the target code as a program. Therefore unlike compiler, an interpreter does not produce the whole program, rather produces the source program's output. The target program produced by the compiler is much faster than that of an interpreter. Moreover, interpreter gives better diagnostics to errors than a compiler. This happens because; the source program is executed statement by statement.

We have mentioned earlier that in a compiler, a large program may be chunked into pieces and relocatable machine codes are produced corresponding to these pieces. These codes are linked together with linkers and other library routines to produce codes that actually run on the machine. Finally, the loader loads all the executable object files into memory for execution.

Here, you might want to learn about high level language and machine level language. Unlike machine level languages, high level languages are easier to work with and maintain. They have much higher degree of machine independence and portability. They support data abstraction and program abstraction. But, using high level language may have some drawbacks. The compiler may generate inefficient machine codes. This in turn may require some additional software. Additionally, the programmer does not have control over machine resources such as registers, buffers or interrupts.

CHECK YOUR PROGRESS – I

1. A _____ translates a high level language source program into an equivalent target program.
2. The target program produced by the compiler is much _____than that of an interpreter.
3. ____________ loads all the executable object files into memory for execution.
4. High level languages have much higher degree of ________and ___________.
5. There are __________________ phases of a compiler.
6. The two sub-phases of a compiler are __________ and ______________.

2.3 PHASES OF A COMPILER

Now, let us see how the phases and sub-phases of a compiler (already discussed in previous unit) work together in order to produce an object code. As mentioned earlier a compiler passes through six major phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and final code generation. Two additional sub-phases symbol table management and error handler interact with these phases in order to produce an efficient machine code. A high level language program forms input to the first phase of the compiler, that is, lexical analysis. The output produced by this phase goes as the input to the next. Each phase produces a unique output and fed as an input to the next. In the following subsections we will discuss how each phase of the compiler converts the input it received into desired output.

2.3.1 Lexical Analysis

This is the first phase of a compiler. In this phase the lexical analyzer or scanner or tokenizer attempts to generate tokens from an input string. It does so by scanning the input from left to right one character at a time. A valid token is passed to the next phase of a compiler. As mentioned earlier, tokens are the other names of words. A word is a string of characters taken as a unit. Technically, a word is known as a lexeme or lexical item. There are some patterns defined in this phase against which each generated word is matched. If there is a match in the input, the token corresponding to the pattern is generated as output.

For example, in the statement:

sum = a + b * 24 + d

Lexemes are-

1. sum
2. =
3. a
4. +
5. b
6. *
7. 24
8. +
9. d

The corresponding tokens for the lexemes are-

1. <id, pointer to the symbol table entry for sum>
2. <=>
3. <id, pointer to the symbol table entry for a>
4. <+>
5. <id, pointer to the symbol table entry for b>
6. <*>
7. <Number, integer value 24>
8. <+>
9. <id, pointer to the symbol table entry for d>

Attribute values for the identifiers sum, a, b and d are entered into the symbol table in records 1, 2, 3 and 4 respectively and these attributes include identifier name, its type, size and scope. Similarly, token for the number 24 requires its type and value. Operators =, * and + do not require any additional attribute value.

2.3.2 Syntax Analysis or Parser

Tokens of lexical analysis phase are fed to the syntax analysis phase. This phase transforms these tokens into graphical structure termed as parse tree. But prior to doing so, this phase tries to check the syntax of the language structure. If the statements conform to the underlying grammatical structure of the language, it is treated as correct. Otherwise, the parser recognizes it as error and reports it. The graphical structure is termed as the syntax tree in which each interior node represents an operation and its child nodes represent the arguments or operands. Like for example, the interior node labeled * has <id, 3> as its left child and integer 24 as its right child. This node is identified by the value b. During execution of this statement, precedence rules for the language are followed. Therefore, multiplication is executed prior to addition. The tokens passed to this stage would be:

<id, 1> <=> <id, 2> <+> <id, 3> <*> <24> <+> <id, 4>

The grammatical structure is specified by context-free grammar and subsequent phases of the compiler use these structure. The graphical structure for the above set of tokens would be:

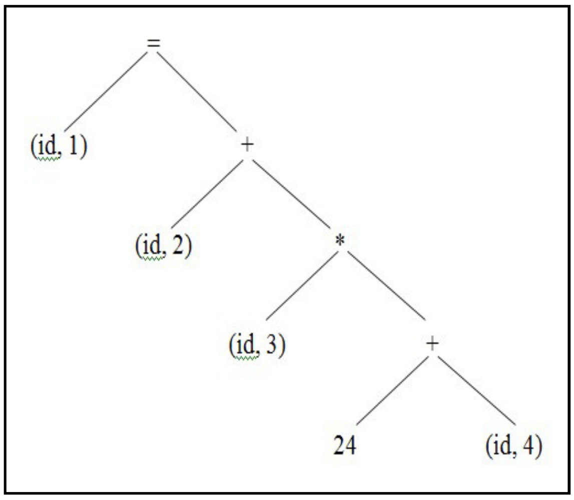


Fig 2.1: Syntax Tree of the statement sum = a + b * 24 + d

There are parsing algorithms which make the parsing decisions by analyzing the program and construct the graphical structure.

2.3.3 Semantic Analysis

The semantic analysis phase uses the syntax tree generated in the previous phase. It basically gathers the type information present in the source program statements. It saves this information in the symbol table. This phase checks to see whether the source program is semantically consistent with the language definition.

Semantic analysis mainly deals with type checking. This means the compiler checks whether each operator contains matching operands. For example, an array index is always required to be integers rather than real numbers. The compiler sees if the indices are integers or not. If it found to other than integers, an error should be reported. In our example statement, number 24 is multiplied with variable b. Here, if b is a floating point number, operator * is applied between an integer and a floating point number. Now, obviously digit 24 also has to be converted into floating point number. In C, the type casting method does this conversion explicitly. Thus, digit 24 becomes 24.0. The language specification which permits such type conversion is called coercions. In our example, we shall consider the variables as integers.

2.3.4 Intermediate Code Generation

The source program construct is translated into one or more intermediate representations. These representations may be syntax tree, Directed Acyclic Graph (DAG) or three-address codes. Intermediate codes must be easy to produce and easy to translate into target code. Syntax tree is a form of intermediate representation that is generated during syntax and semantic analysis phases. DAGs are the intermediate representations which may be thought of as the condensed form of syntax trees. Both syntax trees and DAGs are the graphical representations. In contrast to this, three-address codes are linear representations. Each statement contains three operands and at most one operator on the right side. Each operand act like a register. These instructions specify the order in which the operations are to be performed. Therefore, the sequence of three-address codes would be:

```
t1 =  24

t2  =  id3 * t1

t3  =  t2 + id4

t4  = id2 + t3

id1=  t4
```

The first three address instruction assigns integer 24 to t1. If identifiers sum, a, b or d are floating point numbers, then an explicit type conversion function inttofloat() will do the task. Thus, the statement would be:

```
t1 =  inttofloat(24)
```

As multiplication has more priority over addition, the intermediate code generator would first multiply 24 and b and then add d to it. Adding a to the output gives the final result of the statement. Now, assigning this result to the identifier id1 completes the statement.

2.3.5 Code Optimization

This phase attempts to improve the intermediate code generated in the previous phase. The generated code is machine-independent and gives faster target code. But better target code requires some other objectives to be fulfilled. These may include generation of target code that consumes less power or shorter code that consumes less memory. Now, referring to our example, this phase directly replaces number 24 by 24.0 if need arises. Also it tries to improve the intermediate code by eliminating unnecessary instructions and replacing them with one.

```
t1  =  id3 * 24

t2  =  t1 + id4

id1  = id2 + t2
```

Such representation saves a significant amount of time as well as space. This in turn significantly improves the running time required to execute the target code. Compilers which do such optimization are known as the Optimizing Compilers.

2.3.6 Code Generation

The optimized codes are fed into the code generation phase and later transformed into target code. The intermediate codes are translated into sequences of machine instructions. These sequences perform a specific task. Therefore, such instructions require memory locations or registers to store the variables. Here, the role of registers comes into play and crucial to decide which register will hold which variable. The equivalent set of machine codes for our example statement would be:

```
LD    R2,    id3

MUL   R2,    R2, #24

LD    R1,    id4

ADD   R1,    R1, R2

LD    R0,    id2

ADD   R0,    R0, R1

ST    id1, R0
```

The above code statement requires three registers to perform the translation. The first operand specifies the destination. The first instruction specifies that the contents of id3 are loaded in register R2. Then the multiplication operation multiplies R2 with integer 24. The result is stored back to R2. Similarly, contents of id4 is loaded into register R1 followed by addition between R1 and R2. The result is stored back to R1. Finally, id2 is loaded to R0 which is added with R1 so as to get the final result of the statement. Lastly, R0 is stored on id1, completing the execution of the whole statement. If the statement deals with floating point operations, an F must follow each instruction code; that is, LDF, MULF have to be used to specify the operation codes. A # used prior to 24 signifies that 24 is treated as immediate constant.

CHECK YOUR PROGRESS – II

7. A lexeme is a string of ______________________ taken as a unit.
8. Lexical analyzer puts the records of each identifier in a table called as _______________.
9. The internal nodes of a syntax tree represent an ___________________.

10. ____________________________ mainly deals with type checking.
11. Intermediate codes must be easy to ___________ and __________into target code.
12. Three-address codes are ________________________.
13. The code optimization phase generates _________________ running target code.
14. Variables are stored on registers or memory locations during _______________phase.
15. _____________are the fastest memory.

2.4 SUMMING UP

- A language processor is a program that translates a high level language source program into an equivalent target program. The target program may be an executable machine language program.
- Apart from this, an important role of the compiler is to report error messages detected during translation process.
- An interpreter is a language processor that executes the statements specified in the source program instead of producing the output as a program. Therefore unlike compiler, an interpreter does not produce whole program, rather produces the source program’s output.
- A compiler passes through six major phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and final code generation. Two additional sub-phases symbol table management and error handler interact with these phases in order to produce an efficient machine code.
- Lexical analyzer or scanner attempts to generate tokens from an input string by scanning the string from left to right. It does so by scanning the input from left to right one character at a time.
- Syntax analysis phase transforms these tokens into graphical structure termed as parse tree. But prior to doing so, this phase tries to check the syntax of the language structure. If

the statements do not conform to the underlying grammatical structure of the language, error is reported.

- Semantic analysis basically gathers the type information present in the source program statements. It saves this information in the symbol table. This phase checks to see whether the source program is semantically consistent with the language definition.
- The source program construct is translated into one or more intermediate representations. These representations may be syntax tree, Directed Acyclic Graph (DAG) or three-address codes. Intermediate codes must be easy to produce and easy to translate into target code.
- Code optimization phase attempts to improve the intermediate code. The generated code is machine-independent and gives faster target code. But better target code requires some other objectives to be fulfilled. These may include generation of target code that consumes less power or shorter code that consumes less memory.
- The intermediate codes are translated into sequences of machine instructions. These sequences perform a specific task. Therefore, such instructions require memory locations or registers to store the variables.

2.5 ANSWERS TO CHECK YOUR PROGRESS

1. language processor
2. faster
3. Loader
4. machine independence, portability
5. 6
6. symbol table management and error handler
7. characters
8. symbol table
9. operation
10. Semantic analysis
11. produce, translate
12. linear representations
13. faster
14. code generation
15. Registers

2.6 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is language processor?
2. How does a language processor play a significant role in programming?
3. What is compiler?
4. What is interpreter?
5. Differentiate between compiler and interpreter.
6. Write down functions of linker and loader.
7. What are high level languages? How do they differ from machine level language?
8. What do you mean by tokenization? Give example.
9. What is parser? Discuss.
10. What do you mean by optimizing compiler?
11. What is syntax tree? Explain with example.

B. Long answer type questions.

1. What is language processor? Explain.
2. Give some characteristic overviews on compiler and interpreter.
3. What are the major phases of a compiler? Consider an example of your own and explain how each phase functions in order to produce the target code?

2.7 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 3
FINITE STATE AUTOMATON AND REGULAR LANGUAGE

Unit Structure

3.0 INTRODUCTION

In the previous units, you have learnt compiler and its basic properties. You have also learnt definitions of some basic terminologies as well as functional characteristics of each phase and sub-phase of a compiler. The second unit covers a detailed discussion on overall functions and behavior of each phase of the compiler. Now, you all know that the process of compilation begins with lexical analysis. The lexical analyzer scans a string of characters to find the tokens. It employs finite state automaton to recognize the tokens. Therefore, a recognizer is a program that returns yes once a string is recognized and no otherwise. A finite state automaton (FSA) or a finite state machine (FSM) is an abstract machine which follows a pre-determined sequence of states to recognize a token. This unit mainly focuses on understanding how the finite state automaton works. It also discusses the various types of FSMs as well as their functionalities. Later in this block, we shall have in-depth discussion of some very essential topics related to lexical analysis.

3.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Define finite state automaton
- Understand basic concepts behind finite state automaton
- Know the different types of finite state automaton and their properties
- Know how to construct a finite state machine
- Understand the conversion process of NFA to DFA
- Know the minimization of DFAs

3.2 SPECIFICATION OF TOKENS

The lexical analysis phase is responsible for generating tokens from of a string of characters. We already have discussed in the previous unit that a pattern represents rule to identify a token. These patterns are specified using the regular expressions. Let’s start our discussion with some formal definition of some keywords used to specify the tokens.

1. **String:** The term alphabet denotes a finite set of symbols which may be either numbers or characters. For example, the set of alphabet may be denoted as {0, 1} or {a, b}.
   A string over an alphabet is a finite sequence of symbols generated from the alphabet. For e.g., a set of strings over the alphabet {0, 1} may be 0, 1, 00, 11, 01, 001, 011, 111, 1010, 1110010, 1110111 etc. The terms sentence or words are the synonyms of string. Number of operations can performed on strings. We can calculate its length, perform

concatenation on it. Some of the common terms used in strings along with their corresponding definitions are discussed below. We consider the string s= “Builder” here.

• Length of s	Length of a string is obtained by counting the symbols of the string. For e.g. |s| is of length 7
• Prefix of s	Prefix is obtained by removing zero or more trailing symbols from s. For e.g. Build is a prefix of Builder
• Suffix of s	Suffix is formed by removing zero or more leading symbols of s. For e.g. er is a suffix of Builder
• Substring of s	A string formed by removing a prefix and a suffix from s. For e.g. ild is a substring of Builder
• Proper prefix, suffix	A non-empty string x is a prefix, suffix or or substring of s substring of s, such that $s \neq x$
• Subsequence of s	A string generated by removing zero or more contiguous or non-contiguous symbols from s is a subsequence of s. For e.g. uild is a subsequence of Builder

Apart from these, there is one more term, called concatenation of strings. Consider string r = “String”. The concatenation os derived by appending s to r. Therefore, we have the concatenated string r.s = String Builder. A string of length 0 is termed as an empty string and is denoted by the symbol €.

2. **Language:** A set of strings formed over an alphabet is termed as language. A language may also contain an empty string or the set {€}. We already have mentioned that patterns are recognized by the regular expressions. Each pattern matches a set of strings and regular expressions give names to these set.

There are several operations that can be performed on languages. Table 1 showcases some of the permissible operations on languages. Upon application of these operations create new languages. We consider two languages L and D. L is the alphabet consisting of the set of letters {A, B, ………, Z, a, b, ………., z} and D is the alphabet consisting of the set of digits {0, 1, 2, ……, 9}.

Table 3.1 : Operations on languages

Operations	**Definition**
L ∪ D (union of L and D)	L ∪ D = {s | s is in L or s is in M}
LD (concatenation of L and D)	LD = {st | s is in L and t is in D}
L^* (Kleene closure of L)	$L^* = \bigcup_{i=0}^{\infty} L^i$ It denotes "zero or more concatenations of" L
L^+ (Positive closure of L)	$L^+ = \bigcup_{i=1}^{\infty} L^i$ It denotes "one or more concatenations of" L

- L ∪ D is the set of letters and digits.
- LD is the set of strings consisting of a letter and a digit
- L^* is the set of zero or more occurrences of letters
- L(L ∪ D)* is the set of all strings of letters and digits beginning with a letter
- D^+ is the set of one or more occurrences of digits

Operations union and concatenation are binary operations; however Kleene and positive closure are unary. For a language L over alphabet {0, 1},

- L^0 is {€}
- L^1 is {0, 1}
- L^2 is {00, 01, 10, 11}
- L^3 is {000, 001, 010, 011, 100, 101, 110, 111}
- L* is {€, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, 0010, ……….}

3. **Regular Expressions**: A regular expression is simply a sequence of letters that match with the same sequence of input letters. However, complex structures can also be generated through inclusion of special characters. The operations defined in point 2 are also expressed using some well-formed formula or notation. There are some lexical rules to form the regular expressions over any alphabet ∑. A

regular expression denotes a language. Therefore, if the expression is denoted by the symbol r, the language is expressed by L(r).

- If € is a regular expression, it denotes {€}, i.e., the set containing empty string.
- A symbol 'a' belonging to the set ∑, but not a meta character is a regular expression. It represents a regular expression containing only a single character. Therefore, a is the regular expression that denotes {a}, i.e. the set containing the string a.
- If r and s are the regular expressions; corresponding languages are L(r) and L(s). Then,
  a. (r) is the regular expression which denotes L(r).
  b. (r)(s) is the regular expression which denotes L(r).L(s).
  c. (r) | (s) is the regular expression which denotes L(r) ∪ L(s).
  d. (r)* is the regular expression which denotes L(r)*.

The language denoted by a regular expression is termed as regular set.

Let us make this concept clearer considering an example. Let the alphabet set ∑ = {a, b}.

a) The regular expression a | b represents the set {a, b}.
b) The regular expression (a | b) (a | b) or aa | ab | ba | bb represents the set {aa, ab, ba, bb}. It is the set of all strings of a's and b's having length two.
c) The regular expression a* denotes the set of all strings having zero or more occurrences of a. It represents the set {€, a, aa, aaa, aaaa,…………..}.
d) The regular expression (a | b)* represents the set of all strings containing zero or more occurrences of a or b. Therefore, it denotes the set of all strings of a's and b's and the corresponding regular expression is (a*b*)*.
e) The regular expression a | a*b denotes the set containing the string "a" and all strings consisting of zero or more instances a's followed by a b.

Let us see few examples considering these basic rules.

Regular Expression (r)	**Language L(r)**
hello	{ hello }
ro(s|p)e	{ rose, rope }
abb*	{ ab, abb, abbb, …….. }
(abb)*	{ €, abb, abbabb, abbabbabb, ………. }
a(a|b)*a	{ aa, aaa, aba, aaaa, aaba, abaa, …….. }

4. **Regular definition:** Regular expressions may be given names. It is represented more in a rule-like structure which inclines towards a grammatical approach. It is the other means of describing tokens. If ∑ is a set of alphabets, then the regular definition is of the form

$$\begin{aligned} \mathbf{d_1} &\longrightarrow r_1 \\ \mathbf{d_2} &\longrightarrow r_2 \\ \mathbf{d_3} &\longrightarrow r_3 \end{aligned}$$

Each $\mathbf{d_i}$ being the distinct name of the definition and r_i being the regular expression. Each r_i is a symbol of ∑ and previously defined names. The definition itself shows that it follows production-like specifiers with left side consisting of the symbols $\sum \cup \{ d_1, d_2, d_3 \}$. For instance,

keyword ⟶ long | int | double | while | for | if | then
digit ⟶ 0 | 1 | 2 |……….|9
digit _sequence ⟶ digit+
sign ⟶ + | -
relop ⟶ < | <= | = | <> | > | >=
id ⟶ letter (letter | number)*

Names are written in boldface to distinguish from symbols. The lexical analyzer recognizes the keywords as well as lexemes denoted by **relop, id, sign, digit _sequence** etc.

Let us have an elaborate discussion on how the LEX program works in order to generate a token.

CHECK YOUR PROGRESS – I

1. Compilation begins with __________analysis.
2. The lexical analyzer scans a string of characters to find _____.
3. A _____________over an alphabet is a finite sequence of symbols generated from the alphabet.
4. A set of strings formed over an alphabet is termed as ______.
5. An empty string is denoted as _______.
6. The language denoted by a regular expression is termed as ____________.

3.3 FINITE STATE AUTOMATON AND ITS BASIC CONCEPTS

A finite state automaton is a recognizer program that recognizes a certain string of tokens. It accepts a string x as input and answers "yes" if x is a sentence of the language and "no" otherwise. Regular expressions represent the lexemes. These regular expressions are translated into finite state automaton (FSA) or finite state machine (FSM). In other words, a finite state machine recognizes the regular expressions. We shall now try to understand different FSMs.

3.3.1 Types of Finite State Automaton

A finite automaton can be deterministic or non-deterministic. In Deterministic Finite Automata (DFA), on an input symbol, only one transition is possible from a given state. On the other hand, a Non-deterministic Finite Automaton (NFA) has more than one transition on the same input symbol. Another important feature of this automaton is here €-transition is possible. That is, a transition from one state to another is possible with no input symbol.

Therefore, an NFA mainly consists of 5-tuple (Q, $\sum$, δ, q_0, F):

- A finite set of states Q
- A set of input alphabets or symbols $\sum$

- A transition function (δ) that maps from one state to more than one state on an input symbol
- An initial/start state denoted by q_0
- A set of final states denoted by F

The transition always begins at initial state q_0. On reading an input symbol from the set ∑, the transition function δ moves from q_0 to another state, say q_1. In this way, all symbols of the input string are read and transition happens on each input until the final state is reached.

A DFA is a faster recognizer than NFA; but much bigger than an equivalent NFA. It is easy to implement in software or hardware. Both can recognize regular expressions. However, DFAs are difficult to construct than NFAs. DFA is also a special case of NFA in which

1. There is no €-transition, that is no transition on input €.
2. For each input ‘a’ and state S, there is at most one edge labeled ‘a’ from S.

In other words, it can be said that a DFA is unambiguous unlike NFAs. It also consists of 5-tuples with a difference in the transition function (δ).

3.3.2 Transition Diagram

A finite automaton is best described by a graphical construct. This diagrammatic representation is capable of recognizing the regular expressions. It is a labeled directed graph in which each state is represented in terms of labeled circles and each edge connecting the circles is represented as labeled arcs. Concatenation of the labels of the arcs produces the token to be recognized. A finite automaton consists of a finite number of states. The first state from where the generation of the automaton begins is called the Start State. It is always marked with an arrow pointing towards it. The last state of the recognizer is termed as the final or accepting state. In between the intermediate states are generated. Each state is represented by a circle with the name of the state inside it. The final state is represented using double concentric circles. Two adjacent states are joined by labeled arrows. This implies that upon encountering an input symbol from one state, a transition happens to another state.

Transitions take place until the final state is reached. Now, if we join the symbols of the arcs from start state to the final state, we get the string accepted by the finite automaton. Figure 1.1 gives a transition diagram of a finite automaton.

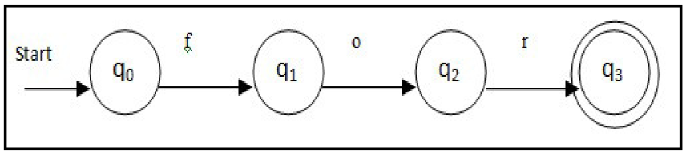


Fig 3.1: Finite automaton which accepts the keyword "for"

The finite automaton begins with the start state q_0. From q_0, upon encountering input symbol 'f', it moves to state q_1. A transition from q_1 to q_2 happens on input symbol 'o'. Finally, on input r, the finite automaton transits from q_2 to the final state q_3. Now, if we join the symbols from q_0 to q_3, we can conclude that the finite automaton accepts the string "for". It is worth mentioning here that a finite automaton may have more than one final state. It should also be noted that the machine may be either in an accepting or non-accepting state. If it is in accepting state, it can be concluded that the string is accepted by the finite state machine. Otherwise, it is not accepted by the machine. The set of all strings accepted by a finite state machine is termed as Regular Language.

The automaton is deterministic as single transition occurs from each state. The transition function maps from one state into another upon scanning of an input symbol. It works in the following manner.

$$\delta(q_0, f) = q_1$$
$$\delta(q_1, o) = q_2$$
$$\delta(q_2, r) = q_3$$

q_3 is in accepting state as end of string is reached.

3.3.3 Transition Table

A finite automaton is implemented using a transition table. It is a data structure which resembles a transition diagram. Like other tables, a transition table is also divided into rows and columns with row representing the states and column representing the input symbol. A transition function is resembled by a transition table. For instance, suppose a transition function from state q_0 on input 'a' reaches another state q_1. The entry for the row in the table is as shown in table 2.

Table 3.2 : Transition table

State	Input symbol
	a
q_0	$\{q_1\}$

Transition table provides faster access to the transitions from a state on an input character. But, it consumes a lot of space when the size of the input string is large.

Let us consider another example. The finite automaton starts at state q_0. On input 0, it moves to state q_1. On input 1, state q_0 is reached. From q_1, on input 0, the finite automaton loops itself and on input 1, moves to the next state q_2. State q_2 is the accepting state. The corresponding transition table would be as follows.

Table 3.3 : Transition table

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_2\}$
$*q_2$	-	-

In a transition table, the initial state is always denoted by an arrow preceding it. The final state is represented by a '*' symbol preceding it. This way we can construct transition table for any finite automaton.

CHECK YOUR PROGRESS – II

7. Regular expressions represent the_______________.
8. These regular expressions are translated into __________.
9. A finite automaton can be ______ or __________.
10. ______has more than one transition on the same input symbol.

11. ________________________ on an input symbol, only one transition is possible from a given state.
12. A DFA is a faster recognizer than NFA. State True or False.
13. Transition diagram is capable of recognizing the ____________________.
14. A finite automaton consists of a finite number of states. State True or False.
15. The final state is represented using ___________________________.
16. A ___________________ is also divided into rows and columns.

3.4 CONSTRUCTION OF FINITE AUTOMATON

Let us construct some deterministic finite automatons by considering regular expressions as examples.

3.4.1 Construction of DFA

Example 1: Draw a DFA with alphabet ∑ = {a, b} which accepts the string "aabb".

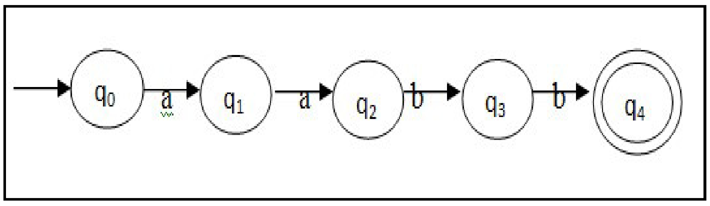


Fig 3.2: DFA that accepts the string "aabb"

The finite automaton begins with state q_0. On reading input symbol a, it reaches state q_1. From q_1, on input a, the finite automaton reaches q_2. On reading input b, q_3 is reached. Finally, from q_3, on reading input b, the accepting state q_4 is reached. Therefore, the finite automaton accepts the string "aabb". The transition function for the automaton can be defined as:

$$\delta\ (q_0, a) = q_1$$
$$\delta\ (q_1, a) = q_2$$

$\delta(q_2, b) = q_3$
$\delta(q_3, b) = q_4$

And the corresponding transition diagram would be

Table 3.4 : Transition table accepting string "aabb"

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	-
q_1	$\{q_2\}$	-
q_2	-	$\{q_3\}$
q_3	-	$\{q_4\}$
$*q_4$	-	-

Example 2: Draw a DFA with alphabet $\sum = \{a, b\}$ which accepts all strings starting with 'a'.

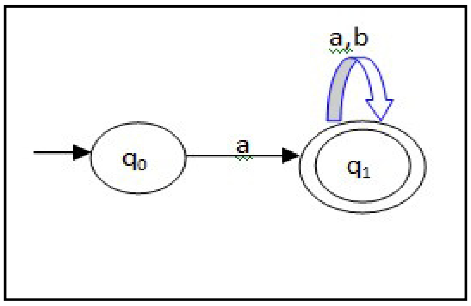


Fig 3.3: DFA that accepts all strings starting with 'a'

The finite automaton starts at state q_0. On input 'a', it moves to state q_1, which is an accepting or final state. This automaton accepts the string "a". Now, for any number of a's and b's, the q_1 loops itself resulting in acceptance of any string beginning with 'a' and any number of a's and b's. Therefore, the finite automaton accepts the string a(a | b)* with possible strings like a, aa,ab, aab, aba etc.

The corresponding transition functions would be:

$\delta(q_0, a) = q_1$
$\delta(q_1, a) = q_1$
$\delta(q_1, b) = q_1$

And the transition table would be:

Table 3.5 : Transition table accepting string "a(a | b)*"

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	-
$*q_1$	$\{q_1\}$	$\{q_1\}$

As there is no move on input b from q_0, you can draw a transition starting at q_0 with input b. This attains a new state q_2, which is a dead state.

Example 3: Draw a DFA with alphabet $\sum = \{0, 1\}$ which accepts all strings ending with '0'.

The automaton begins with state q_0. If the string consists only of the string {0}, it reaches the final state q_1. The string might also have many numbers of 1's and then finally a '0' (like 1110), state q_0 self loops on 1's and then to q_1 on '0'. The finite automaton may encounter 0's and 1's in between before ending up with a '0'.

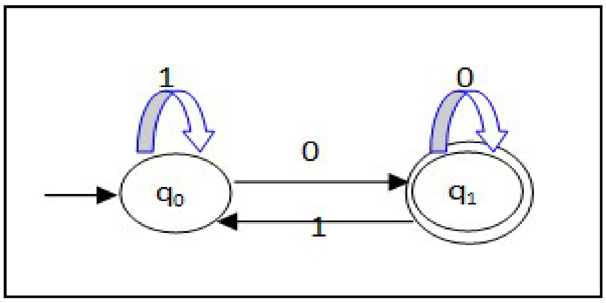


Fig 3.4: DFA that accepts all strings ending with '0'

The corresponding transition functions would be:

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

And the transition table would be:

Table 3.6 : Transition table accepting string which ends with '0'

State	Input symbol	
	0	1
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$
$*q_1$	$\{q_1\}$	$\{q_0\}$

Example 4: Draw a DFA for the language accepting strings ending with '00' over input alphabets ∑={0, 1}.

The string may contain only {00}. From the initial state, on two consecutive 0's, the final state is reached. However, the string might begin with 1's. Intermediate 1's may also exist.

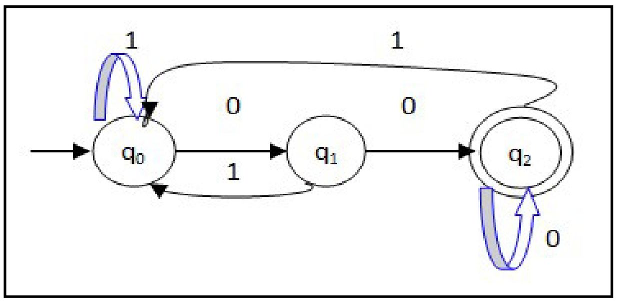


Fig 3.5: DFA that accepts all strings ending with '00'

The initial state for the DFA is q_0 and the final state is q2. Figure 1.5 depicts the transition diagram for the given string.

Accordingly, the transition functions would be:

$$\delta\ (q_0, 0) = q_1$$
$$\delta\ (q_0, 1) = q_0$$
$$\delta\ (q_1, 0) = q_2$$
$$\delta\ (q_1, 1) = q_0$$
$$\delta\ (q_2, 0) = q_2$$
$$\delta\ (q_2, 1) = q_0$$

And the transition table would be:

Table 3.7 : Transition table accepting string which ends with '00'

State	Input symbol	
	0	1
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_0\}$
$*q_2$	$\{q_2\}$	$\{q_0\}$

Example 5: Draw a DFA with alphabet $\sum = \{a, b\}$ which accepts all strings starting with 'a' and ending with 'b'.

The finite automaton starts with state q_0 and on input 'a', it reaches state q_1. The next input might be 'b'. On this input, the finite automaton moves to the final or accepting state q_2. This is applicable for the input string "ab". But, situation might arise when numbers of a's and b's might appear in between 'a' and 'b'. Therefore, the finite automaton might be of the following form.

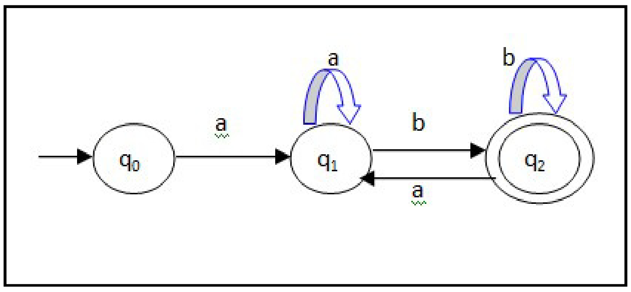


Fig 3.6: DFA that accepts strings starting with 'a' and ending with 'b'

The transition function would be given by

$$\delta (q_0, a) = q_1$$
$$\delta (q_1, a) = q_1$$
$$\delta (q_1, b) = q_2$$
$$\delta (q_2, a) = q_1$$
$$\delta (q_2, b) = q_2$$

And the corresponding transition table would be

Table 3.8 : Transition table accepting strings starting with 'a' and ending with 'b'

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	-
q_1	$\{q_1\}$	$\{q_2\}$
$*q_2$	$\{q_1\}$	$\{q_2\}$

Example 6: Draw a DFA with alphabet ∑ = {a, b} which accepts all strings having three consecutive a's.

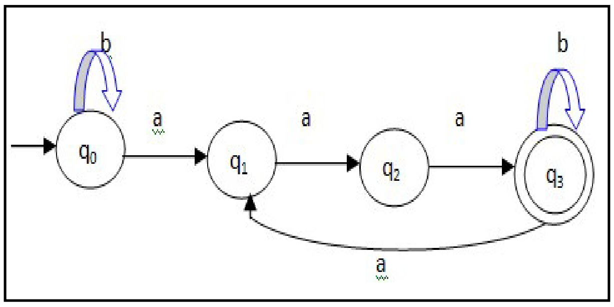


Fig 3.7: DFA that accepts all strings having three consecutive a's

The DFA begins with state q_0 which produces next state q_1 on input a. From q_1, the DFA eventually moves to states q_2 and q_3 on two consecutive a's. Therefore, the DFA accepts 000. However, strings might be of the form baaa, bbaaa, baaab, bbaaabbb, baaabbaaa……. State q0 self loops on any number of b's. Then after encountering three consecutive a's, any number of b's may take place. For that q_3 self loops. There is another possibility of encountering three or more numbers of 0's again. Transition q_3 to q_1 fulfills that.

The transition function would be given by

$$\delta\,(q_0, a) = q_1$$
$$\delta\,(q_0, b) = q_0$$
$$\delta\,(q_1, a) = q_2$$
$$\delta\,(q_2, a) = q_3$$
$$\delta\,(q_3, a) = q_1$$
$$\delta\,(q_3, b) = q_3$$

And the corresponding transition table would be:

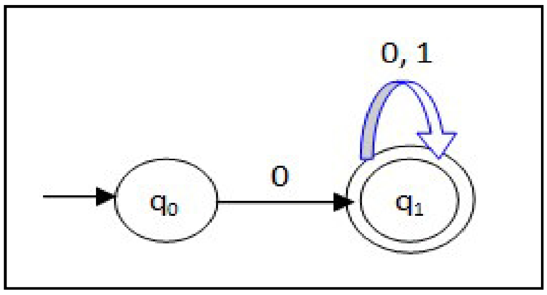
Table 3.9 : Transition table accepting strings with three consecutive a's

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	-
q_2	$\{q_3\}$	-
$*q_3$	$\{q_1\}$	$\{q_3\}$

Example 7: Draw a DFA which accepts all strings starting with '0' over alphabet $\sum = \{0, 1\}$.

If the string contains only a {0}, it immediately reaches the final state on input 0. If any other character comes after 0, the final state will self loop. The finite automaton would encounter a dead end if any other symbol is read from q_0.

Figure 1.8 depicts the finite automaton for the string.



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "0"] & q_1 \arrow[loop above, "0, 1"] \end{tikzcd}$$

Fig 3.8: DFA that accepts all strings starting with a

Similarly, the transition function for the DFA would be as follows:

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

And the corresponding transition diagram would be:

Table 3.10 : Transition table accepting strings starting with '0'

State	Input symbol	
	a	b
→q_0	{q_1}	-
*q_1	{q_1}	{q_1}

3.4.2 Construction of NFA

An NFA may attain more than one state on reading an input symbol. Let us again draw some non-deterministic finite automatons by considering regular expressions as examples.

Example 1: Design an NFA with ∑ = {a, b} which accepts all strings ending with ab.

The finite automaton begins with state q_0. It attains the final state on reading the string {ab}. If {ab} is preceded by any number of a's and b's, q_0 loops itself. The automaton reaches two states on input 'a'.

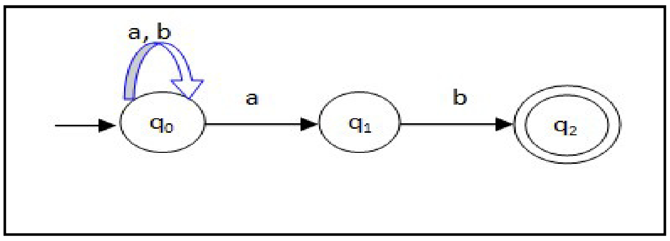


Fig 3.9: NFA that accepts all strings ending with ab

The corresponding transition functions would be given by:

$$\delta (q_0, a) = q_0, q_1$$
$$\delta (q_0, b) = q_1$$
$$\delta (q_1, b) = q_2$$

And, finally the transition table would be:

Table 3.11 : Transition table accepting strings ending with 'ab'

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	-	$\{q_2\}$
$*q_2$	-	-

Example 2: Design an NFA in which all the string containing a substring 1110 over the alphabet.

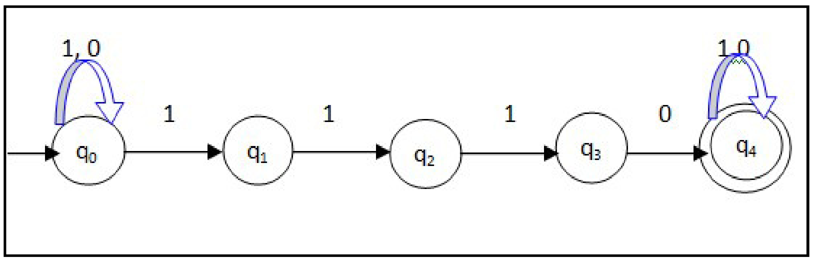


Fig 3.9: NFA that accepts all strings containing substring 1110

With the initial state q_0, the NFA goes through three intermediate states by reading three consecutive 1's. Finally, the NFA reaches the final state q_4, on input 0. For any consecutive 0's and 1's coming prior to the substring 1110, a self-loop occurs at state q_0. Similarly, sequences of 0's and 1's may appear following the string. Therefore, at state q_4 a self-loop occurs.

The corresponding transition functions would be given by:

$$\delta(q_0, 0) = \{q_0\}$$
$$\delta(q_0, 1) = \{q_0, q_1\}$$
$$\delta(q_1, 1) = \{q_2\}$$
$$\delta(q_2, 1) = \{q_3\}$$
$$\delta(q_3, 0) = \{q_4\}$$
$$\delta(q_4, 0) = \{q_4\}$$
$$\delta(q_4, 1) = \{q_4\}$$

The transition table for the finite automaton would be:

Table 3.12 : Transition table accepting strings containing substring 1110

State	Input symbol	
	0	1
→q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	-	$\{q_2\}$
q_2	-	$\{q_3\}$
q_3	$\{q_4\}$	-
*q_4	$\{q_4\}$	$\{q_4\}$

Example 3: Design an NFA with ∑ = {a, b} which accepts all strings of length 2.

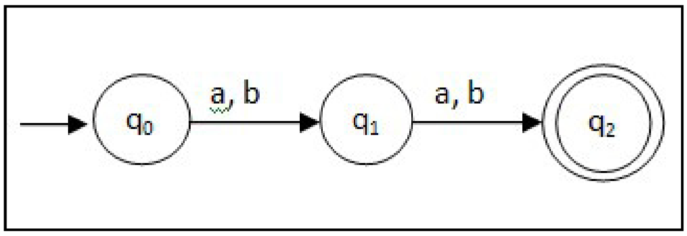


Fig 3.10: NFA that accepts all strings of length 2

Starting with state q_0, the NFA goes to state q_1 on input symbol a or b. The automaton reaches the final state q_2 from q_1 on input a or b. The corresponding transition functions for the NFA would be:

$$\delta(q_0, a) = \{q_1\}$$
$$\delta(q_0, b) = \{q_1\}$$
$$\delta(q_1, a) = \{q_2\}$$
$$\delta(q_1, b) = \{q_2\}$$

And the transition table would be:

Table 3.13 : Transition table accepting strings of length 2

State	Input symbol	
	a	b
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
$*q_2$	-	-

Example 4: Design an NFA with $\sum = \{0, 1\}$ which accepts all strings having second last element 1.

The probable set of strings accepted by the language

L= {01, 11, 110, 010, 011, 0010, 00110, 11010..........}

Therefore, the finite automaton would be

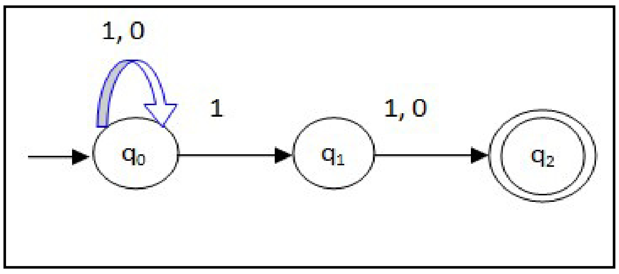


Fig 3.11: NFA that accepts all strings having second last element 1

The transition functions would be given by:

$$\delta (q_0, 0) = \{q_0\}$$
$$\delta (q_0, 1) = \{q_0, q_1\}$$
$$\delta (q_1, 0) = \{q_2\}$$
$$\delta (q_1, 1) = \{q_2\}$$

The transition table for the finite automaton would be:

Table 3.14 : Transition table accepting strings having second last element 1

State	Input symbol	
	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
$*q_2$	-	-

The finite automaton starts with the initial state q_0 and reaches the final state q_2.

Example 5: Design an NFA with $\sum = \{0, 1\}$ which accepts all strings having either 01 or 10 as substring.

The finite automaton begins with q_0 as the initial state. It contains two final states; one for accepting 01 as substring and another for 10 as substring. The finite automaton would be:

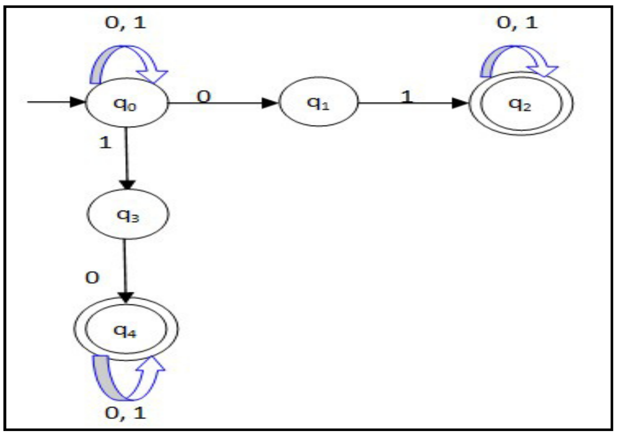


Fig 3.12: NFA that accepts all strings having substring either 01 or 10

The corresponding transition functions would be:

$$\delta(q_0, 0) = \{q_0, q_1\}$$
$$\delta(q_0, 1) = \{q_0, q_3\}$$
$$\delta(q_1, 1) = \{q_2\}$$
$$\delta(q_2, 0) = \{q_2\}$$
$$\delta(q_2, 1) = \{q_2\}$$

$\delta(q_3, 0) = \{q_4\}$
$\delta(q_4, 0) = \{q_4\}$
$\delta(q_4, 1) = \{q_4\}$

Finally, the transition table for the finite automaton would be:

Table 3.15 : Transition table accepting strings having either 01 or 10 as substring

State	Input symbol	
	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
q_1	-	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	-
$*q_4$	$\{q_4\}$	$\{q_4\}$

Example 6: Construct an NFA with $\sum = \{0, 1\}$ in which each string starts with "1" and ends with "0" to reach a final state.

The NFA starts with state q_0. The set of possible strings accepted by the language would be :

L= {10, 100, 110, 1110, 1100, 1010, 1111010, 1001110, 1001100…..}

The corresponding finite state automata would be:

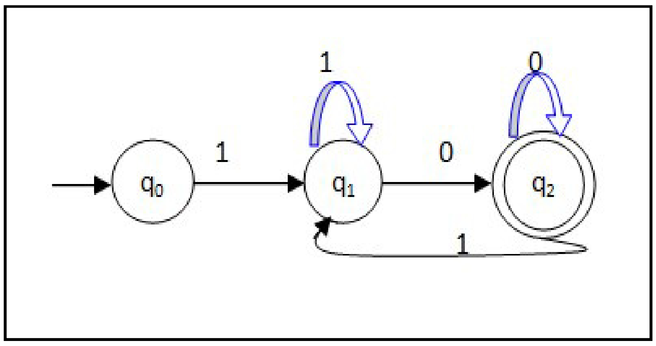


Fig 3.13: NFA that accepts strings starting with "1" and ending with "0"

The transition functions would be

$$\delta(q_0, 1) = \{q_1\}$$
$$\delta(q_1, 0) = \{q_2\}$$
$$\delta(q_1, 1) = \{q_1\}$$
$$\delta(q_2, 0) = \{q_2\}$$
$$\delta(q_2, 1) = \{q_1\}$$

And the transition table would be:

Table 3.16 : Transition table accepting strings starting with "1" and ending with "0"

State	Input symbol	
	0	1
$\rightarrow q_0$	-	$\{q_1\}$
q_1	$\{q_2\}$	$\{q_1\}$
$*q_2$	$\{q_2\}$	$\{q_1\}$

Example 7: Construct an NFA with $\sum = \{0, 1\}$ for the language L = $\{0^m1^n \mid m \geq 0 \text{ and } n \geq 1\}$.

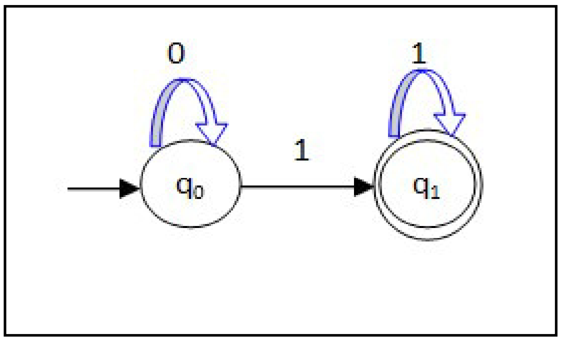


Fig 3.14: NFA that accepts the language L = $\{0^m1^n \mid m \geq 0$ and $n \geq 1\}$

The finite automaton has 0 or more occurrences of 0's and at least one 1 (Since n>=1).
The set of possible strings accepted by the language would be :

L= {1, 01, 011, 111, 001, 0111, 0011, 0001111…..}

Therefore, on any 0 input, state q_0 self loops. With input 1, q_0 moves to q_1. Then, for all other 1's, state q_1 self loops. The transition functions for the finite automaton would be:

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

And the transition table would be:

Table 3.17 : Transition table accepting language L = $\{0^m1^n \mid m \geq 0 \text{ and } n \geq 1\}$

State	Input symbol	
	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_1\}$
q_1	-	$\{q_1\}$

Example 8: Construct an NFA with $\sum = \{a, b\}$ for the language $L = \{(ab)^n \mid n \geq 1\}$.

The NFA begins with state q_0. The set of possible strings accepted by the language is:

L={ab, abab, ababab,……}

The transition diagram for the language would be:

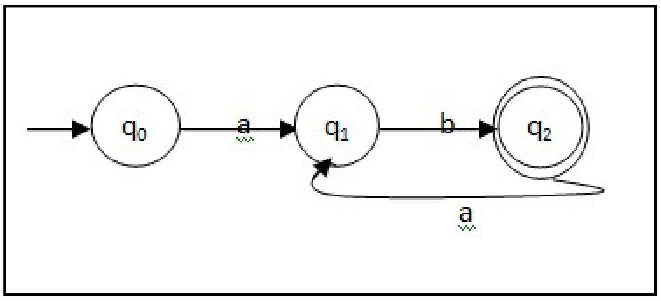


Fig 3.15: NFA that accepts the language L = {{(ab)^n | n≥1}

The transition functions corresponding to this finite automaton would be:

$$\delta(q_0, a) = \{q_1\}$$
$$\delta(q_1, b) = \{q_2\}$$
$$\delta(q_2, a) = \{q_1\}$$

And the transition table would be:

Table 3.18 : Transition table accepting language L = {0^m1^n | m ≥0 and n≥1 }

State	Input symbol	
	a	b
→q_0	{q_1}	-
q_1	-	{q_2}
q_2	{q_1}	-

3.4.3 Construction of NFA with ϵ-transitions

NFA can change its state without reading an input symbol. Such transition is depicted by labeling the appropriate arc with **ϵ. ϵ** does not belong to any input alphabet. Both NFA and ϵ-NFA recognize the same language. **ϵ**-transition does not extend the capability of the language. ϵ-closure of a state X consists of a set of states which are reachable from X with only null move including X. In other words, ϵ-closure for a state can be obtained by union operation of the states which can be reached from that state including the state itself.

Example 1: Draw a Non-deterministic Finite Automata which accepts the string "ab".

The finite automaton starts at state q_0. On input 'a', it moves to state q_1. From q_1, the finite automaton moves to state q_2 with no input; i.e. on **ϵ**-transition. Then from q_2, on input symbol 'b', the NFA moves to the final state q_3.

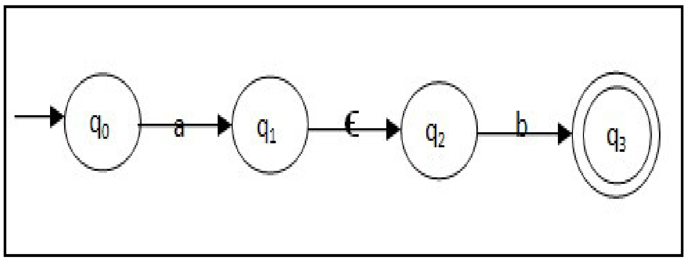


Fig 3.16: NFA with ϵ-transition that accepts the string 'ab'

Example 2: Draw a Non-deterministic Finite Automata which accepts the string "a | b".

The NFA begins with state q_0. Then, on ϵ-move, it moves to two different states q_1 and q_3. From q_1 and q_3, the NFA moves to q_2 and q_4 respectively on reading input symbols 'a' and 'b'. Then from q_2 and q_4, on ϵ-transition, it moves to final state q_5.

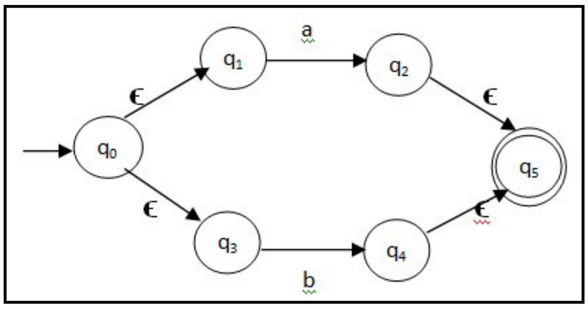


Fig 3.17: NFA with ϵ-transition that accepts the string 'a | b'

Example 3: Draw a Non-deterministic Finite Automata which accepts the string "a | b | c".

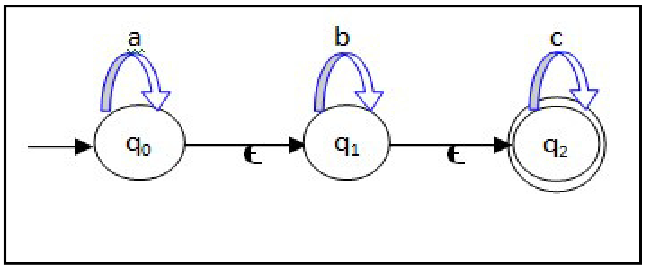


Fig 3.18: NFA with ϵ-transition that accepts the string 'a | b | c'

The NFA begins with state q_0. On input 'a', the automaton self loops. Then from q_0, the automaton moves to state q_1. Then on input 'b', q_1 self loops. Again, q_1 moves to q_2 on ϵ-transition. State q_2 is the final state which self loops on input 'c'.

Example 4: Draw a Non-deterministic Finite Automata which accepts the string "aa* | bb*".

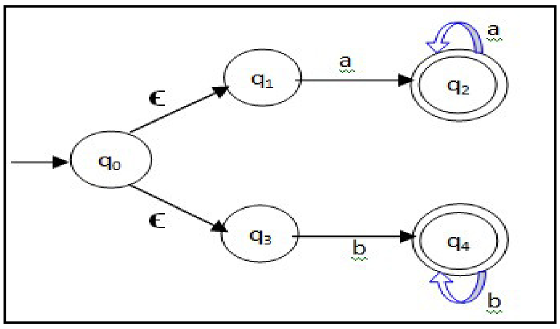


Fig 3.19: NFA with ϵ-transition that accepts the string ‘aa* | bb*’

The NFA starts with state q_0. With ϵ-transition, the NFA moves to states q_1 and q_3. From q_1, on input symbol ‘a’, a transition to the final state q_2 is made. Here, for any number of a’s, state q_2 self-loops. Similarly, from state q_3, on input symbol ‘b’, a transition to the final state q_4 takes place. Here also, for any number of b’s, state q_4 self-loops.

3.5 REGULAR EXPRESSION

Regular expression (or *regex*) is a pattern or a sequence of characters and meta characters that describes a set of strings. These strings match this pattern. A regex accepts certain set of strings and rejects the rest. A regular expression may compose of simple characters or a combination of simple and special characters. Patterns are written by enclosing between slashes. Therefore, a regex is constructed by combining simpler sub-expressions. Such patterns are matched to find the exact sequence of characters in strings. By default, matching is case sensitive. For example, the pattern /student/ matches the character combinations in strings only when the exact sequence “student” occurs. Special characters have special meanings in a regular expression. Such characters may include: dot (.), bracket [ ], parenthesis ( ), or (|), backslash (\), occurrence indicators (+, *, ?, { }), position anchors (^, $) etc.

The dot (.) operator matches any special character except new line. For instance, “….” matches four characters from a set of strings.

Just to recall, a finite automaton (FA) is an abstract machine that can be used to represent certain computations. A FA consists of a number of states and edges connecting these states. With each input symbol, the finite automaton moves to another state. An edge with the input label marks this transition. If the finite automaton reaches a final state after scanning all its inputs, it is said that the finite automaton accepts the string. Otherwise, it rejects the string.

A regular expression is defined by regular languages. There are some operations performed on these languages.

- **Union** of two sets includes all the elements of the sets. For example, the union of sets {ab, bc, ca} and {ca, cb, abc} is defined by the set {ab, bc, ca, cb, abc}.
- The **concatenation** of two sets can be defined by using "." operator between the sets. For example, the sets {ab, bc, ca} and {abc, €} is defined by as follows.

  {ab, bc, ca}.{abc, €} = {ab.abc, ab. €, bc.abc, bc. €, ca.abc, ca. €}

  = {ababc, ab, bcabc, bc, caabc, ca}

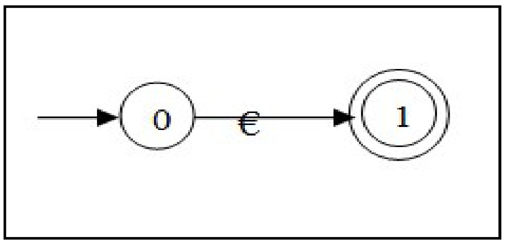
- The **Kleene *** operation is a unary operation and sometimes termed as closure. This operation generates zero or more concatenations of strings. Like, if L is language,
  L^0 = {€}
  $L^1 = L$
  $L^2 = L.L$
  $L^3 = L. L^2$
  .
  .
  $L^n = L.L^{n-1}$
  $L^* = L^0 + L^1 + L^2 + L^3 + ...$
  For instance, let the language be L = {0, 1}. Therefore,
  L^0 = {€}
  L^1 = {0, 1}
  L^2 = {0, 1}.{0, 1} = {00, 01, 10, 11}
  $L^3 = L. L^2$ = {0, 1}.{00, 01, 10, 11}
  = {000, 001, 010, 011, 100, 101, 110, 111}
  ....
  L^* = {€, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111}

This is the set of all strings of zeros and ones.

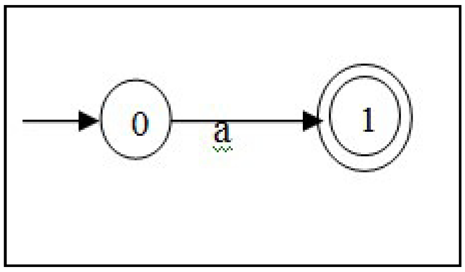
Regular languages and finite automaton are related to each other. For every regular language, there is a finite automata and vice versa.

Now, we shall try to convert regular expressions into their corresponding NFAs. There are certain rules to convert a regular expression into an NFA.

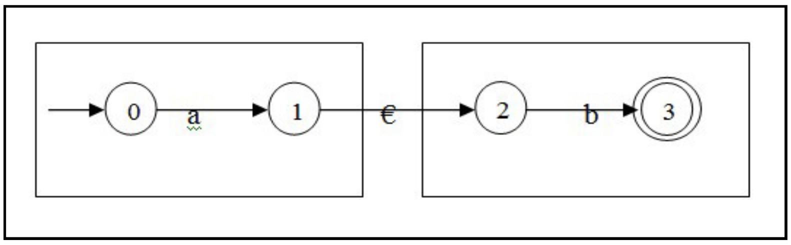
1. For the regular expression €, the NFA would have a start state. With input €, the start state moves to the final state. The NFA would be



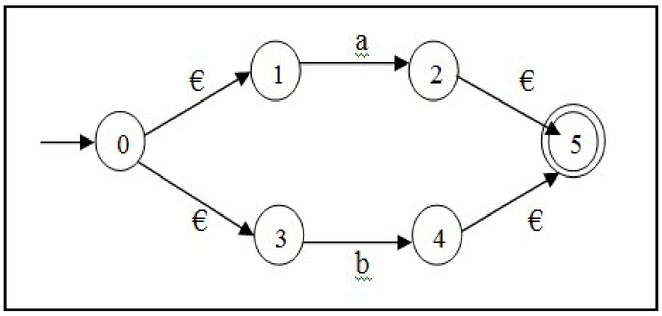
2. For the regular expression having only symbol a, the NFA would have a start state. With input €, the start state moves to the final state. The NFA would be



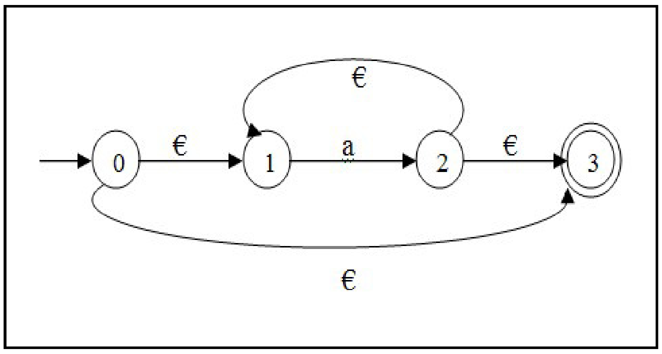
3. Let's consider another regular expression a.b which is a concatenation of two regular expressions a and b. Two separate NFAs are constructed for the inputs a and b and connected through €tart transition. The start state of NFA for a will be the start state of the combined NFA. The final state of NFA for b will be the final state of the combined NFA. Therefore, the NFA would be



4. Similarly, for the regular expression a + b or a | b can be expressed in terms of two automata. They are combined through a common start state and a final state.



5. For the regular expression a*, the corresponding NFA would be as follows.



Now, we shall try to construct some NFAs using these rules. Let us consider some regular expressions.

1. For the regular expression a(a + b)*ab, the NFA has an initial state 0. On input 'a', it moves to state 1. From 1, the NFA accepting (a+b)* is constructed according to rule 5. It reaches state 8. From 8, the NFA reaches the accepting state 10 on inputs 'a' and then 'b'. The final NFA would be as given in the following figure.

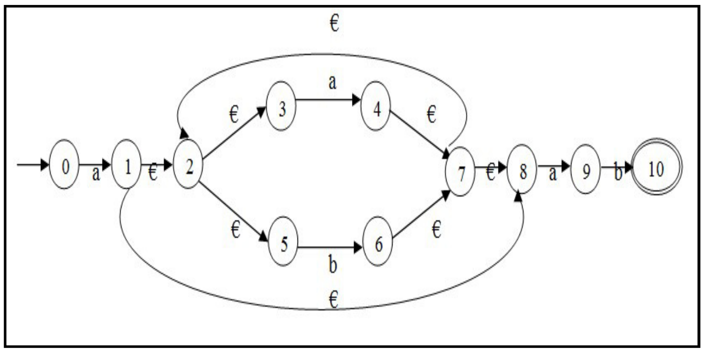


Fig 3.20: NFA accepting string a(a + b)*ab

2. Similarly, let us consider another regular expression a + b + ab and try to draw the NFA for the same.

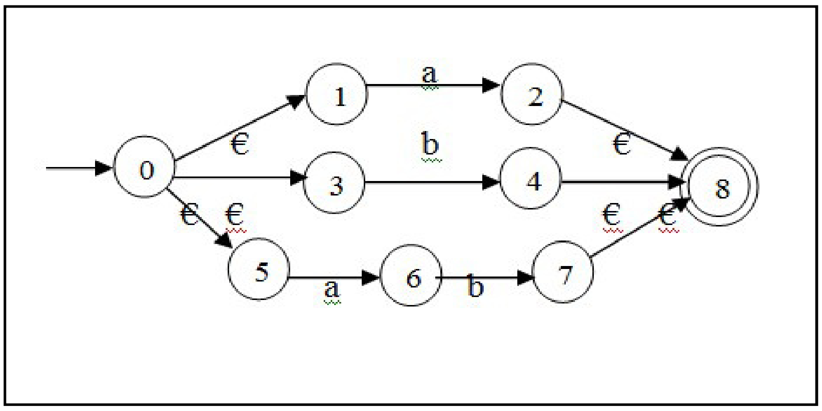


Fig 3.21: NFA accepting string a + b + ab

3.6 CONVERSION OF NFA TO DFA

NFAs are multivalued; that is, more than one transition may occur on reading one input symbol. However, computer simulation of such an NFA is really very hard. In other way, it can be said that multiple paths may exist from start state to an accepting state which defies the definition of acceptance. In order to find exactly one, we have to consider all that are available and then generate one leading to accepting state.

If an NFA is converted into its corresponding DFA, both will recognize the same language. Each entry of the transition table of an NFA contains a set of states; however, each entry of a DFA transition table is just a single state. An NFA to DFA conversion represents each DFA state corresponding to a set of NFA states. The DFA uses its state to keep track of all possible NFA states upon reading an input symbol. The algorithm that constructs a DFA from an NFA is termed as Subset construction.

Prior to going into the conversion process, let's understand some basic operations. Here, we consider **s** as an NFA state and T as a set of NFA states.

ϵ-closure(s): This set represents the NFA states reachable from a given state s on **ϵ**-transitions. For the NFA in figure 1.18, the **ϵ-**

closure(q_0) represents the set of all states reachable from q_0. Therefore, **ϵ-closure(q_0)= { q_0, q_1, q_2}**.

Similarly, let's derive the **ϵ-closure** set for figure 1.17.

ϵ-closure(q_0)= { q_1, q_3 }
ϵ-closure(q_2)= { q_5 }
ϵ-closure(q_4)= { q_5 }

ϵ-closure(T): The set of NFA states reachable from some state s in T on ϵ-transition alone.

Move(T, a): The set of NFA states to which there is a transition from some state s in T on an input symbol 'a'.

We consider an NFA N to convert into corresponding DFA D. For that purpose, a transition table ***Dtran*** is constructed for D. Each DFA set corresponds to a set of NFA states. Let's construct the DFA for the NFA in figure 1.19. Here, the set of alphabet is {a, b}. A state in DFA is an accepting state if it is a set of NFA states containing at least one accepting state of the NFA.

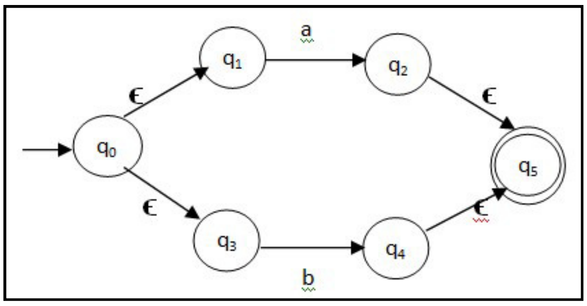

$$\begin{tikzcd}
 & q_1 \arrow[r, "a"] & q_2 \arrow[dr, "\epsilon"] & \\
\arrow[r] & q_0 \arrow[u, "\epsilon"] \arrow[d, "\epsilon"'] & & q_5 \\
 & q_3 \arrow[r, "b"'] & q_4 \arrow[ur, "\epsilon"'] &
\end{tikzcd}$$

Fig 3.22: NFA accepting string 'a | b**'**

Step 1: At first, the set **ϵ-closure(q_0)** is computed; **q_0** being the start state of the NFA. This set represents the start state of the DFA. We mark this set as A.

ϵ-closure(q_0)= { q_0, q_1, q_3 } = A

Step 2: We now compute **ϵ-closure(move(A,a))** set and move to state B. Then, we compute **ϵ-closure(move(A,b))** and name it state C.

The set, **move(A,a)** contains the states having transitions from the states of A on input 'a'. In this case, only **q_1** has transition on **'a'.**

Then, **move(A,b)** is computed; which contains the set of states having transitions from the states of A on input 'b'. In this case, only $\mathbf{q_3}$ has transition on **'b'.**

ϵ-closure(move(A, a)) = ϵ-closure({q_2})= { q_2, q_5 }

Thus, ***Dtran*[A, a]= B**

ϵ-closure(move(A, b)) = ϵ-closure({q_4})= { q_4, q_5 }

Therefore, ***Dtran*[A, b]= C**

Step 3: This process continues with the new unmarked sets of B and C. Finally, we reach a point where all the states of the set are marked.

Therefore, we shall continue by trying to deduce **ϵ-closure(move(B, a))** and **ϵ-closure(move(B, b))**. We can see that both sets do not have any transition from **{ q_2, q_5 }** either input '**a**' or '**b**'.

Similarly, we have to find the **ϵ-closure(move(C, a))** and **ϵ-closure(move(C, b))**. We can see that both sets do not have any transition from **{ q_4, q_5 }** either input '**a**' or '**b**'.

We have found three different states for the corresponding DFA.

Thus, the transition table for the DFA would be:

Table 3.20: Transition table Dtran for DFA

State	Input symbol	
	a	b
→ A	B	C
B	-	-
C	-	-

And the corresponding DFA would be:

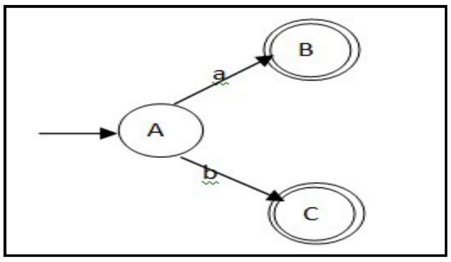


Fig 3.23: DFA accepting string 'a | b'

The DFA has two accepting states; one from the initial state A to the final state B on input ‘a’ and the other from A to the final state C on input ‘b’.

3.7 MINIMIZATION OF DFA

A regular expression is always recognized by a minimum-state DFA. This section will try to discuss on how to construct a minimum-state DFA by reducing the number of states in a given DFA. Reduction to minimum-state DFA should occur in such a way that it does not affect the language being recognized. We have a DFA M, a set of states S and a set of input symbols $\sum$. It starts in state s and accepts string w. Our algorithm follows the strategy for minimizing the DFA states by finding all groups of states distinguished by some input string. The group of states which cannot be distinguished is merged into a single state. Each set of states is partitioned with each partition consisting of a set of states that have not been distinguished from one another and any pair of states chosen from different groups has been found distinguishable by some input.

Initially, there are two groups in a partition: the accepting states and non-accepting states. A group of states, say $A=\{s_1, s_2, …., s_k\}$ is considered. On some input symbol a, we shall look at what transitions states $s_1, s_2, …., s_k$ will make. If these transitions fall in two or more different groups of the current partition, then A has to be split up. Suppose, on input a, states s_1 and s_2 move to states t_1 and t_2 of two different groups of the partition. Then A is splitted up into two subsets; one with s_1 and other with s_2.

This process of splitting groups is repeated until no more groups need to be split. Below we present the algorithm for minimizing the number of states of a DFA.

Algorithm: Minimizing the number of states of a DFA.

Input: A DFA M with a set of states S, set of input alphabets $\sum$. A set of transitions for all states, input symbol, a start set s_0 and a set of accepting states F.

Output: A DFA $M^{/}$ with minimum number of states accepting the same language as M.

Method:

1. An initial partition ∏ of the set of states is constructed having two groups: the accepting states F and the non-accepting states S-F.
2. for each group G of ∏ do begin
   Partition G into subgroups such that two states *s* and *t* of G are
   in the same subgroup if and only if for all input symbols a,
   states *s* and *t* have transitions on a to states in the same group ∏
   replace G in $\prod_{new}$ by the set of all subgroups formed
   end
3. If $\prod_{new} = \prod$, let $\prod_{final} = \prod$ and then continue with the next step. Else, repeat step 2. with $\prod: = \prod_{new}$
4. Choose one state in each group of the partition $\prod_{final}$ as the representative for that group. The representatives will be the states of the reduced DFA $M^{/}$. Let S be a representative state and suppose on input a, there is a transition of M from *s* to *t.* Let *r* be the representative of *t's* group (*r* may be *t*). Then $M^{/}$ has a transition from *s* to *r* on a. Let the start state of $M^{/}$ be the representative of the group containing the start state s_0 of M and let the accepting states of $M^{/}$ be the representatives that are in F. Each group of $\prod_{final}$ either consists of only states in F or has no states in F.
5. If $M^{/}$ has a dead state, that is a state *d* that is not accepting and that has transitions to itself on all input symbols, then remove *d* from $M^{/}$. Also remove any states not reachable from the start state. Any transitions to *d* from other states become undefined.

CHECK YOUR PROGRESS – III

11. NFA cannot change its state without reading an input symbol. State True or False.
12. A regular expression is always recognized by a __________________________.
13. Both NFA and ϵ-NFA recognize the same language. State True or False.

3.8 SUMMING UP

- A finite state automaton (FSA) or a finite state machine (FSM) is an abstract machine which follows a pre-determined sequence of states to recognize a token.
- A finite state automaton is a recognizer program that recognizes a certain string of tokens. It accepts a string x as input and answers “yes” if x is a sentence of the language and “no” otherwise. Regular expressions represent the lexemes. These regular expressions are translated into finite state automaton (FSA) or finite state machine (FSM).
- The lexical analysis phase is responsible for generating tokens from of a string of characters. A pattern represents rules to identify a token. These patterns are specified using the regular expressions.
- The term alphabet denotes a finite set of symbols which may be either numbers or characters. A string over an alphabet is a finite sequence of symbols generated from the alphabet.
- A set of strings formed over an alphabet is termed as language. A language may also contain an empty string or the set {€}.
- A regular expression is defined by regular languages.
- A regular expression is simply a sequence of letters that match with the same sequence of input letters. There are some lexical rules to form the regular expressions over any alphabet ∑.
- Regular expressions may be given names. It is represented more in a rule-like structure which inclines towards a grammatical approach. It is the other means of describing tokens.
- Finite state automaton is of two types- Non-deterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA).
- In Deterministic Finite Automata (DFA), on an input symbol, only one transition is possible from a given state. On the other hand, a Non-deterministic Finite Automaton (NFA) has more than one transition on the same input symbol. Another important feature of this automaton is here €-transition is possible. That is, an NFA can change its state without reading an input symbol.

- A finite automaton is best described by a graphical construct. This diagrammatic representation is capable of recognizing the regular expressions.
- A finite automaton consists of a finite number of states. It begins with the Start State. The last state of the recognizer is termed as the final or accepting state. In between the intermediate states are generated.
- A finite automaton is implemented using a transition table. It is a data structure which resembles a transition diagram.
- NFAs are multivalued; that is, more than one transition may occur on reading one input symbol. An NFA to DFA conversion represents each DFA state corresponding to a set of NFA states. If an NFA is converted into its corresponding DFA, both will recognize the same language.
- A regular expression is always recognized by a minimum-state DFA.

3.9 ANSWERS TO CHECK YOUR PROGRESS

1. lexical
2. Tokens
3. String
4. Language
5. €
6. regular set
7. lexemes
8. finite state automaton
9. deterministic, non-deterministic
10. Non-deterministic Finite Automaton
11. Deterministic Finite Automata
12. True
13. regular expressions
14. True
15. double concentric circles
16. transition table
17. False
18. minimum-state DFA
19. True

3.10 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is a Finite State Machine? Explain in brief.
2. What is finite automaton? Explain in brief.
3. What do you mean by string of alphabets?
4. What do you understand by string of language? Describe.
5. What is regular expression? Explain in brief.
6. Explain the functionalities of NFA.
7. Explain the functionalities of DFA.
8. Give a detailed discussion of transition diagram.
9. Explain transition table in detail.
10. What is NFA with ϵ-transition?

B. Long answer type questions.

1. Describe how tokens are specified.
2. Describe some of the operations performed on languages.
3. Give a detailed discussion of regular expression and regular definition.
4. What are the different types of finite automatons available? Explain each of them.
5. What are transition table and transition diagram? How are they related with each other?
6. Explain the process of constructing a DFA with an example.
7. Explain the process of constructing an NFA with an example.
8. Explain the rules for constructing NFAs with proper diagrams.
9. How do you convert an NFA into its corresponding DFA? Explain.
10. What do you understand by minimization of DFA?

3.11 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.

- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 4
LEX

Unit Structure:

4.0 INTRODUCTION

Lexical analysis is the first phase of a compiler. It takes the source program as input and breaks it into a stream of tokens. These tokens are later passed into the syntax analysis phase in order check whether it conforms to grammar or syntax of the language. Subsequent phases of compiler also generate intermediate forms of these inputs. Therefore, generation of accurate tokens play the most crucial role for successful compilation to proceed. In the previous units, we have got some basic ideas regarding different phases of a compiler and their functionalities. In this unit, we shall study how lexical analyzer functions in order to accurately generate tokens from a source program. We shall also study about lexical-analyzer generator simply known as lex or flex.

4.1 UNIT OBJECTIVES

After going through this unit, you will be able to

- Understand the functional characteristics of a lexical analyzer.
- Know the different types of lexical errors.
- Describe how the Lex-tool functions.
- Know the structure of a Lex program
- Conceptualize the ideas of writing a Lex program

4.2 LEXICAL-ANALYZER GENERATOR

As already discussed, a lexical analyzer is a program which reads stream of input characters from a source program and groups them into lexemes and produce a sequence tokens for each lexeme. These tokens are sent to the syntax analysis phase. A symbol table contains records of each identifier occurring in a program. When the lexical analyzer detects a lexeme representing an identifier, that lexeme is entered into the symbol table. The interaction between the lexical analyzer and the parser can be represented in terms of the following figure 4.1.

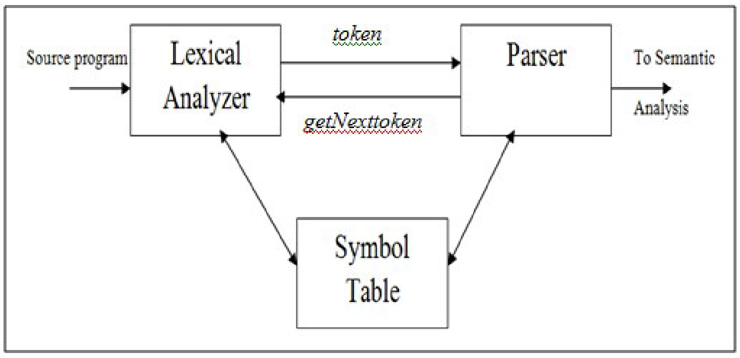
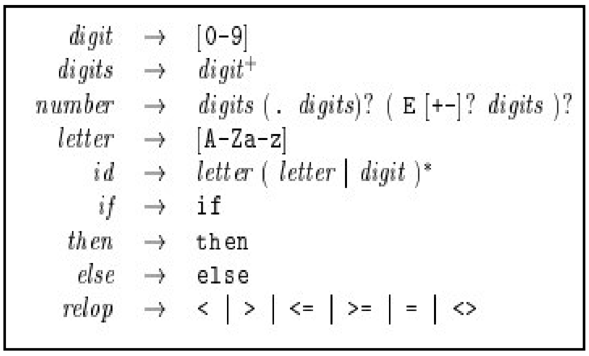


Fig 4.1: Interaction between the lexical analyzer and parser Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

While generating the tokens, the lexical analyzer does striping off comments and white spaces such as blank spaces, tabs, new lines etc. The lexical analyzer generates error messages when there is a violation of lexical rules in the source program. Some compilers may insert error messages at the positions where the error has occurred. While doing this, it makes an exact copy of the source program. Alternatively, the lexical analyzer may keep track of the new lines of the program so that they can be associated with the error messages. Expansion of macros is also a function of the lexical analyzer if it exists in the program.

During generation of tokens, the input program is scanned and checked whether the string being scanned conforms to some patterns. If the sequence of characters matches with the patterns, that sequence is considered as token. Therefore, patters can be thought of some lexical rules which are matched against the string. Like for each keyword, the pattern is the keyword itself. Now, if a sequence of characters matches with a pattern, then it is termed as lexeme. Lexemes can be termed as the instance of tokens. Finally, token consists of a token name and an optional attribute value. It can be regarded as the abstract symbol representing the lexicon. For example, any numeric constant 4.12, 5 or 3.23e2 can be represented by the token **number**. Similarly, any valid variable name can be recognized by **id**. For keywords, the tokens are specified by the corresponding keyword itself. For instance, the keyword "if" is identified by lexeme if and the corresponding token becomes **if**. Therefore, for each keyword, a corresponding token exists with pattern being the same with the keyword. Similarly, for operators like '+', '-', '/', '*' and '%', the token can be termed as **op.** And for the relational operators like '<', '>', '<=', '>=' or '!=', the corresponding token can be **relop.** A valid variable declaration with letters followed by letters and digits may be considered as token **id.** Therefore, for all identifiers, there is one corresponding token. On the other hand, there is one token corresponding tom each punctuation mark such as comma, semicolon or left/right parenthesis.

The above descriptions can be defined in terms of some regular definitions. These definitions are important to specify/recognize the tokens. They define the patterns for tokens which the lexical analyzer would eventually find out during scanning of inputs.



digit	$\rightarrow$	`[0-9]`
digits	$\rightarrow$	$digit^{+}$
number	$\rightarrow$	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	$\rightarrow$	`[A-Za-z]`
id	$\rightarrow$	*letter* (*letter* | *digit*)*
if	$\rightarrow$	`if`
then	$\rightarrow$	`then`
else	$\rightarrow$	`else`
relop	$\rightarrow$	< | > | <= | >= | = | <>

Fig 4.2: Regular definitions for specifying tokens
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

In addition, the white space may be recognized as the following definition:

$$\text{ws} \quad \rightarrow \quad (\text{blank} \mid \text{tab} \mid \text{newline})^{+}$$

The lexical analyzer strips out the white spaces occurring in the source program to find out the valid tokens

Sometimes, it becomes necessary to provide additional information to the subsequent compiler phases about which lexeme being matched; i.e. the value of the lexeme. This happens because more than one lexeme matches a particular pattern. For instance, the token **number** may match more than one number in the source code. This information is required for further processing as the code generator require this information about the lexeme being found. Therefore, the lexical analyzer sends this additional information about the attribute value along with the token being encountered to the parser of the compiler.

4.2.1 Lexical Errors

Errors can be detected during lexical analysis. The lexical analyzer must be able to report errors which occur when it is unable to find any valid token. Misspelling of tokens is a major issue which occurs frequently in a program and must be reported when detected. For instance, when characters of a keyword swap, the lexical analyzer must be able to detect it and report. Consider an example of the keyword **if.** When characters interchange, it becomes **fi.** The generator cannot detect it as an error; rather, it detects it as a valid variable declaration. Apart from this, in some programming languages, such declaration may be treated as a valid function name. Such kinds of problems occur during lexical analysis; but it is unaware of the fact, it treats the declaration as an identifier. Therefore, the subsequent phases may require detecting and reporting such errors.

Another error recovery strategy may be termed as the ‘panic mode’ recovery technique. This situation happens when the lexical analyzer cannot proceed because the prefix of the input does not match any of the patterns for tokens. The recovery strategy employs the technique of deleting the successive characters from the remaining input characters until a good token is found from the input being considered. The input may be repaired using some other actions. Some of these may include:

- Delete one character from remaining input
- Insert a missing character into the remaining input
- Replace one character by another
- Transposition of two adjacent characters

The purpose of such mechanism is to recognize the prefix of any input as valid lexeme which can later be transformed into well-formed token.

CHECK YOUR PROGRESS – I

1. _________ phase breaks the source program into a stream of tokens.
2. The lexical analyzer reads stream of input characters from a source program and groups them into __________.
3. When the lexical analyzer detects a lexeme representing an identifier, that lexeme is entered into the __________.
4. The lexical analyzer generates __________when there is a violation of lexical rules in the source program.
5. Expansion of ______ is a function of the lexical analyzer.

4.3 Lex-THE LEXICAL ANALYZER GENERATOR

Lex is a tool that allows specifying a lexical analyzer. It does so by describing regular expressions that represent patterns for tokens. The tool itself is known as the Lex Compiler. The Lex compiler transforms the input patterns into transition diagrams. The generated code simulating the transition diagram is known as the lex.yy.c. The input file to the Lex tool is referred to as Lex language. We already have discussed the translation of regular expressions into corresponding transition diagram.

4.3.1 How Lex Works

It is already mentioned previously that the input file to Lex is written in Lex language and is termed as lex.l. This file is fed into the Lex compiler which later transforms it into a C-file lex.yy.c. In the next phase, this C-program forms the input to a C-compiler that produces the output file a.out. This output file functions as the lexical analyzer which accepts input streams and produces its corresponding stream of tokens. This whole functionality of a Lex can be depicted using the following figure

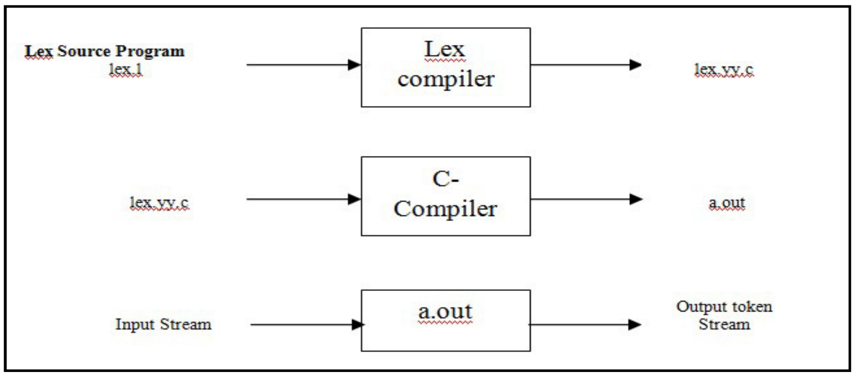


Fig 4.3: Work flow of a Lex
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

The compiled C-program a.out functions as a subroutine of the parser. This C-function returns an integer representing code for token names. And the attribute value is stored in the variable yylval. The attribute value may correspond to a numeric code, a pointer to the symbol table or it may contain no values. Variable yylval is shared between the lexical analyzer and the parser.

4.3.2 Structure of Lex Program

A typical Lex program is divided into sections. It consists of the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

Declarations of variables, constants or regular definitions are included in the declarations section. Constants resemble the values of variables or tokens.

Translation rules include patterns and their corresponding actions. Patterns represent regular expressions and may use the regular definitions used in the declaration section. Actions are the codes written using C. It consists of the following form:

Pattern {Action}

Auxiliary function includes the additional functions used in the actions part of the second section. They are compiled separately and loaded into the lexical analyzer.

Lex behaves in the following manner. When parser calls the lexical analyzer for a token, the lexical analyzer starts reading the input, one character at a time. When the longest prefix matches with a particular pattern, the corresponding action is performed. Now, this action returns the name of the token to the parser. But if it does not due to the presence of white spaces, then the lexical analyzer would search for additional lexemes until an action returns the name of a token to the parser. The returned value is stored in the variable yyval which is shared between the lexical analyzer and the parser. Now, consider the following program to understand the different code segments of a lexical analyzer.

```
                    %{
                            /* Declarations that manifest constants
                            LT, GT, LE, GE, EQ, NQ, IF, ELSE,
NUM, ID, RELOP */
                    }%
                            /* regular definitions */

                            delim           [ \t\n]

                            ws              {delim}+

                            letter          [A-Za-z]

                            digit           [0-9]

                            id
                            {letter}({letter}|{digit})*

                            number
                            {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

                    %%
                            {ws}            {/* no action and no
                    return */}
```

```
        if              {return(IF);}

        else            {return(ELSE);}

        {id}            {yylval     =      (int)
installID(); return(ID);}

        {number}        {yylval     =      (int)
installNum(); return(NUM);}

        "<"             {yylval      =      LT;
return(RELOP);}

        "<="            {yylval      =      LE;
return(RELOP);}

        "="             {yylval      =      EQ;
return(RELOP);}

        "<>"            {yylval      =      NQ;
return(RELOP);}

        ">"             {yylval      =      GT;
return(RELOP);}

        ">="            {yylval      =      GE;
return(RELOP);}

%%
int installID()
        {
        /* function to install a lexeme into the
symbol table and return a pointer
        thereto. The first character of the
        lexeme is pointed to by yytext, and its
        length is returned to yyleng */
        }
int installNum()
        {
        /* It is similar to installID, but it puts
        numerical constants into a separate
        table */
        }
```

Fig 3.4: Lex program describing a simple form of branching statements and conditional expressions
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

The declaration section includes a pair of brackets %{ and }%. The definitions that manifest constants are used here. Each constant is associated with a unique integer through a C #define statement. These declarations are directly copied to lex.yy.c. These are not treated as regular definitions.

Regular definitions are later used in the translation section to define the patterns and are surrounded by curly brackets. The translation rules for keywords return the keyword itself. White spaces require no actions to perform.

The auxiliary function section includes two functions: installID() and installNum(). Like declaration, anything within auxiliary function section is directly copied to lex.yy.c. The action taken when a pattern for id is matched is as follows:

1. First, function installID() places the lexeme into the symbol table.
2. This function also returns a pointer to the symbol table and places it in the variable yylval which can later be used by the parser. Additionally, two variables are generated by the function: yytext is function which is a pointer to the beginning of the lexeme and yyleng which contains the length of the lexeme.
3. The token name ID is returned to the parser.

Similarly, the action for number is performed by the auxiliary function installNum().

In a Lex program, during matching procedure, it is always preferable to consider a longer prefix to a smaller one. And if the longest prefix matches more than one pattern, it is desirable to consider the first pattern listed in the Lex program.

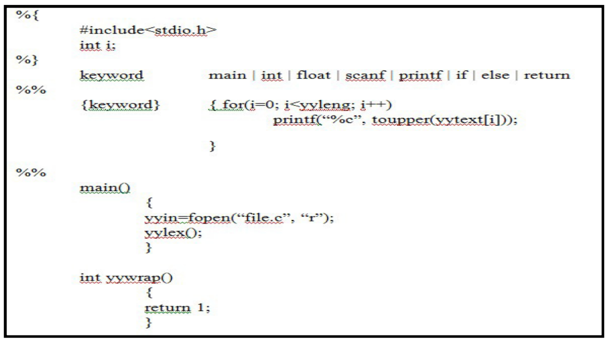
CHECK YOUR PROGRESS – II

6. _______ is a tool that allows specifying a lexical analyzer.
7. The Lex compiler transforms the input patterns into_________.
8. The input file to the Lex tool is referred to as___________.
9. Lex compiler transforms a lex file into a C-file termed as____.

10. The attribute value of a token is stored in the variable ____________.
11. Translation rules in Lex program include ______________ and their corresponding ______________.
12. The variable yyval which is shared between ___________________and___________.
13. The declaration section includes a pair of brackets ____________.
14. The function ___________ places the lexeme into the symbol table.
15. The function _________ contains the length of the lexeme.

Now, let us see some examples of simple Lex programs and try to conceptualize how they function.

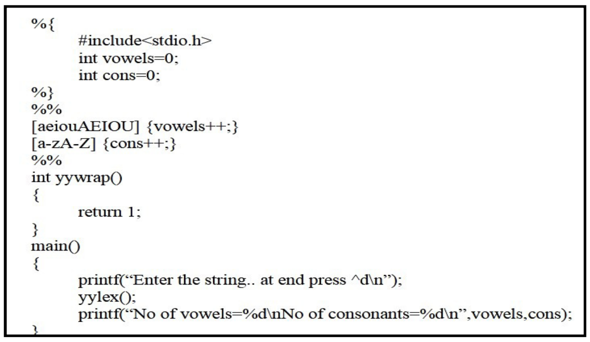
1. Program to convert keywords into uppercase.



```
%{
        #include<stdio.h>
        int i;
%}
        keyword         main | int | float | scanf | printf | if | else | return
%%
        {keyword}       { for(i=0; i<yyleng; i++)
                                printf("%c", toupper(yytext[i]));

                        }

%%
        main()
                {
                yyin=fopen("file.c", "r");
                yylex();
                }

        int yywrap()
                {
                return 1;
                }
```

Fig 3.5: Lex program to convert keywords into uppercase.

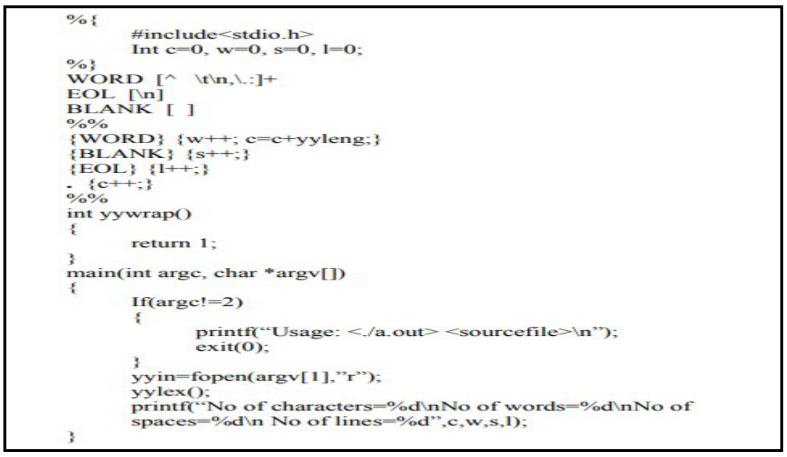
2. Program to count the number of vowels and consonants in a given string.



```
%{
        #include<stdio.h>
        int vowels=0;
        int cons=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {cons++;}
%%
int yywrap()
{
        return 1;
}
main()
{
        printf("Enter the string.. at end press ^d\n");
        yylex();
        printf("No of vowels=%d\nNo of consonants=%d\n",vowels,cons);
}
```

Fig 3.6: Lex program to count the number of vowels and consonants in a given string.

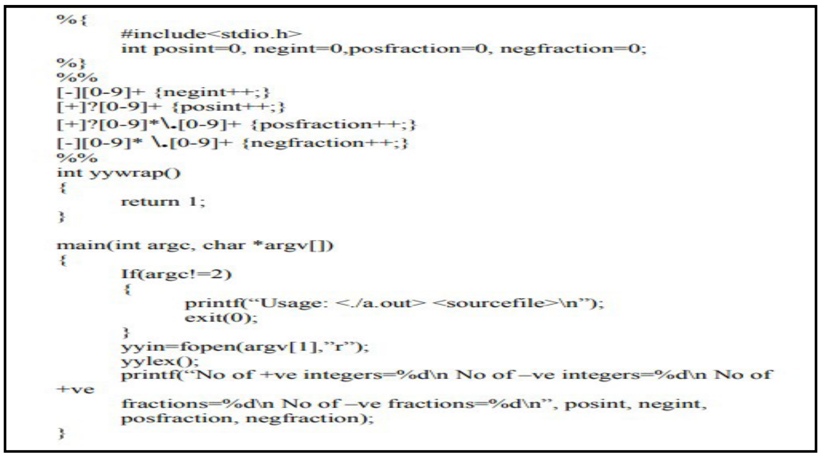
3. Program to count the number of characters, words, spaces, end of lines in a given input file.



```
%{
        #include<stdio.h>
        Int c=0, w=0, s=0, l=0;
%}
WORD [^  \t\n,\.:]+
EOL [\n]
BLANK [ ]
%%
{WORD} {w++; c=c+yyleng;}
{BLANK} {s++;}
{EOL} {l++;}
. {c++;}
%%
int yywrap()
{
        return 1;
}
main(int argc, char *argv[])
{
        If(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        yylex();
        printf("No of characters=%d\nNo of words=%d\nNo of
        spaces=%d\n No of lines=%d",c,w,s,l);
}
```

Fig 3.7: Lex program to count the number of characters, words, spaces, end of lines in a given input file

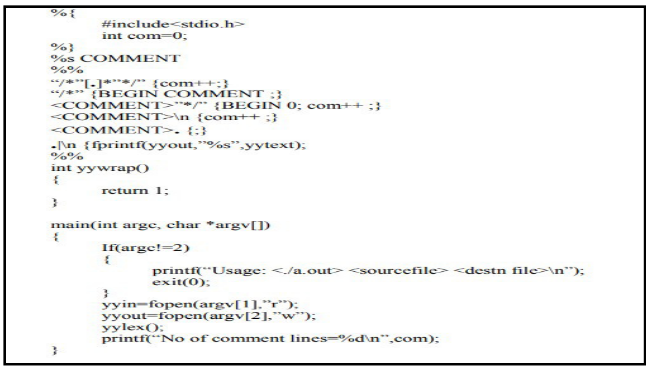
4. Program to count no of:
   a) +ve and –ve integers
   b) +ve and –ve fractions



```
%{
        #include<stdio.h>
        int posint=0, negint=0,posfraction=0, negfraction=0;
%}
%%
[-][0-9]+ {negint++;}
[+]?[0-9]+ {posint++;}
[+]?[0-9]*\.[0-9]+ {posfraction++;}
[-][0-9]* \.[0-9]+ {negfraction++;}
%%
int yywrap()
{
        return 1;
}

main(int argc, char *argv[])
{
        If(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        yylex();
        printf("No of +ve integers=%d\n No of –ve integers=%d\n No of
+ve
        fractions=%d\n No of –ve fractions=%d\n", posint, negint,
        posfraction, negfraction);
}
```

Fig 3.8: Lex program to count the number of +ve and –ve integers/+ve and –ve fractions

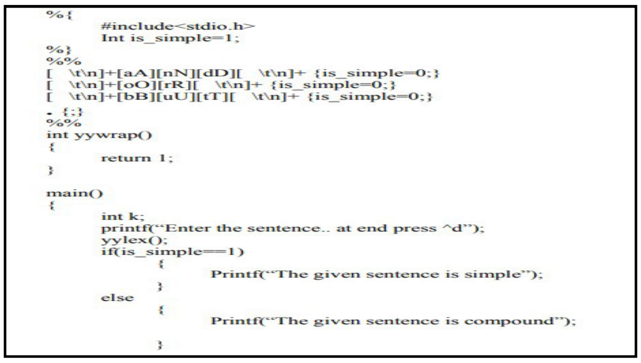
5. Program to count the number of comment line in a given C program. Also eliminate them and copy that program into separate file.



```
%{
        #include<stdio.h>
        int com=0;
%}
%s COMMENT
%%
"/*"[.]*"*/" {com++;}
"/*" {BEGIN COMMENT ;}
<COMMENT>"*/" {BEGIN 0; com++ ;}
<COMMENT>\n {com++ ;}
<COMMENT>. {;}
.|\n {fprintf(yyout,"%s",yytext);
%%
int yywrap()
{
        return 1;
}

main(int argc, char *argv[])
{
        If(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile> <destn file>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        yyout=fopen(argv[2],"w");
        yylex();
        printf("No of comment lines=%d\n",com);
}
```

Fig 3.9: Lex program to count the number of comment line in a given C program

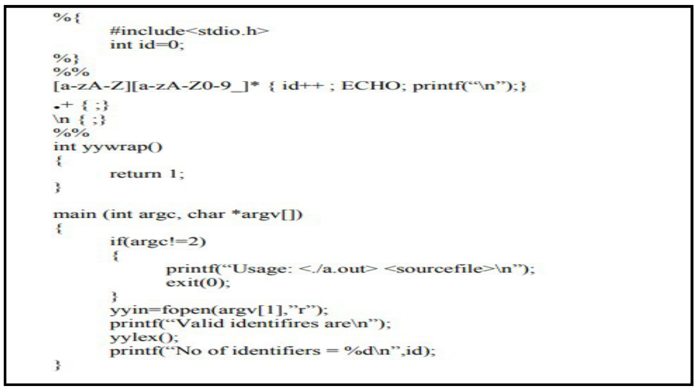
6. Program to recognize whether a given sentence is simple or compound.



```
%{
        #include<stdio.h>
        Int is_simple=1;
%}
%%
[  \t\n]+[aA][nN][dD][  \t\n]+ {is_simple=0;}
[  \t\n]+[oO][rR][  \t\n]+ {is_simple=0;}
[  \t\n]+[bB][uU][tT][  \t\n]+ {is_simple=0;}
. {;}
%%
int yywrap()
{
        return 1;
}

main()
{
        int k;
        printf("Enter the sentence.. at end press ^d");
        yylex();
        if(is_simple==1)
                {
                        Printf("The given sentence is simple");
                }
        else
                {
                        Printf("The given sentence is compound");

                }
```

Fig 3.9: Lex program to recognize whether a given sentence is simple or compound

7. Program to recognize and count the number of identifiers in a given input file.



```
%{
        #include<stdio.h>
        int id=0;
%}
%%
[a-zA-Z][a-zA-Z0-9_]* { id++ ; ECHO; printf("\n");}
.+ { ;}
\n { ;}
%%
int yywrap()
{
        return 1;
}

main (int argc, char *argv[])
{
        if(argc!=2)
        {
                printf("Usage: <./a.out> <sourcefile>\n");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        printf("Valid identifires are\n");
        yylex();
        printf("No of identifiers = %d\n",id);
}
```

Fig 3.10: Lex program to recognize and count the number of identifiers in a given input file

It is already discussed in figure 3.4 that a Lex file always has the extension .l. Therefore, each file is saved as file_name.l. The corresponding C-source code is generated through the command lex. Therefore, the file lex.yy.c is produced which in turn is compiled using the Lex compiler. This function creates an executable file a.out which can directly run the program. These whole tasks can be grouped into the following commands.

```
lex <pgm_name.l>

cc lex.yy.c –ll

./a.out
```

The Lex library is invoked using –ll option in command cc lex.yy.c. The above commands do tokenization of a source code.

4.4 SUMMING UP

- Lexical analyzer is a program which reads stream of input characters from a source program and groups them into lexemes and produce a sequence tokens for each lexeme.
- During generation of tokens, the input program is scanned and checked whether the string being scanned conforms to some patterns. If the sequence of characters matches with the patterns, that sequence is considered as tokens. Therefore, patters can be thought of some lexical rules which are matched against the string.
- While generating the tokens, the lexical analyzer does striping off comments and white spaces such as blank spaces, tabs, new lines etc. The lexical analyzer generates error messages when there is a violation of lexical rules in the source program.
- Lex is a tool that allows specifying a lexical analyzer. The tool is known as the Lex Compiler. The Lex compiler transforms the input patterns into transition diagrams. The generated code simulating the transition diagram is known as the lex.yy.c. The input file to the Lex tool is referred to as Lex language.
- A Lex-program is divided into three sections- declarations, translation rules and auxiliary functions.

- Declarations of variables, constants or regular definitions are included in the declarations section. Translation rules include patterns and their corresponding actions. Patterns represent regular expressions and may use the regular definitions used in the declaration section. Auxiliary function includes the additional functions used in the actions part of the second section.
- Each Lex file is saved as file_name.l. The corresponding C-source code is generated through the command lex. Therefore, the file lex.yy.c is produced which in turn is compiled using the Lex compiler. This function creates an executable file a.out which can directly run the program.

4.5 ANSWERS TO CHECK YOUR PROGRESS

1. Lexical analysis
2. Lexemes
3. symbol table
4. error messages
5. macros
6. Lex
7. transition diagrams
8. Lex language
9. lex.yy.c
10. yylval
11. patterns, actions
12. lexical analyzer, parser
13. %{ and }%
14. installID()
15. yyleng

4.6 POSSIBLE QUESTIONS

A. Short answer type questions.

1. Why does generation of tokens play important role during compilation?
2. What is lexical-analyzer generator? Discuss in brief.

3. How does the lexical-analyzer generator derive tokens from a stream of characters?
4. How do the lexical analyzer and the parser interact with each other? Describe in brief.
5. What is Lex? Discuss in brief.

B. Long answer type questions.

1. Give a detailed discussion on lexical-analyzer generator.
2. Explain the process of generation of tokens from a stream of characters.
3. What are lexical errors? How is recovery from errors done during lexical analysis? Explain.
4. What is Lex? How does it work? Describe.
5. Describe the structure of a Lex program.

4.7 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 5
CONTEXT FREE GRAMMARS

Unit Structure:

5.1 INTRODUCTION

In the previous unit, we have studied some basic understandings of lexical analysis phase. We also have got ideas related to how lex programs are developed using C programming language. The lexical analysis phase of a compiler is responsible for breaking the program into constituent pieces of stream of characters and generates tokens for the same. These tokens are passed to the parsing phase in order to construct the grammatical structure of the language. These grammatical structures describe the syntax of the language. For specifying the syntax, a widely accepted notation termed as Context-Free Grammar (CFG) or Backus Naur Form (BNF) is used. Besides specifying the syntax of the language, CFGs also guide in a grammar oriented compiling technique called Syntax Directed Translation. During this phase, a hierarchical structure called syntax tree is constructed. In this usit, we will study how CFGs are formed. We shall also study how they take part in forming parse trees of different classes of grammars.

5.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Learn the basic concepts of Context-Free Grammar.

5.3 CONTEXT FREE GRAMMARS

To specify the rules or syntax of a language, grammars are used. We use language to communicate. While communicating we use some naturally occurring languages. And when we talk to each other via the language, knowingly or unknowingly we follow some language rules. Communication follows from these rules which human beings develop from childhood. Therefore, in other words, we gather knowledge of a language and accordingly we apply this knowledge to communicate. In other words, syntax can be regarded as the grammar of a language. Similar is the case with computer languages. There are some language specific rules which are incorporated into the language itself. And when the programmer writes codes, he/she has to write according to these rules or syntax. If a statement does not conform to the syntax of the language, the compiler shows a syntax error message. Therefore, syntax plays a crucial role in a programming language.

Grammar describes the structure of a language i.e., syntax of a language is specified by the grammar. The grammar or syntax of a programming language is described in terms of production rules.

A context Free Grammar G is a 4-tuple (V, T, P, S) which consists of the followings:

- V is a set of non-terminals or variables. A non-terminal always forms the left side of the production rule and the right side of the same may consist of a combination of terminals and non-terminals.
- T is a set of terminals, which are referred to as tokens.
- P is the set of production rules of the language. A production mainly consists of two parts. They are separated by an arrow "→" mark. Like for example, a production may be of the form $A \rightarrow aBC$, where A is the left side or head of the production and aBC is the right side or body of the production. The left side always consists of non-terminals

and right side may have only terminals, only non-terminals or the combination of both. It needs special mention here that the body of a production may contain an empty string or ϵ.

- S is the start symbol of the grammar from where derivation of strings takes place.

There are some other notational conventions also exist through which these terminals and non-terminals can be distinguished.

Terminals:

- Lowercase letters of the alphabet like a, b, c etc.
- Operators such as +, * etc.
- Digits such as 0, 1, 2, …….., 9.
- Punctuation markers such as parenthesis, comma and so on.
- Keywords such as if, while, for etc.

Non-terminals:

- Uppercase letters of the alphabet such as A, B, C etc.
- Lowercase but italicized names such as *expr or stmt*.
- Letter S, which is treated as the start symbol of the grammar.
- While defining grammar of a language in terms of production rules, uppercase letters such as E, T etc may appear. These are also treated as non-terminals.
- Generally, the head of the first production of a grammar is always the start symbol.

Again, a set of productions may have same non-terminal on the left. For example, consider the productions

$E \rightarrow E + T$

$E \rightarrow E * T$

Here, the heads of these two productions consist of same non-terminal E. Therefore, the two productions can be grouped together by incorporating a vertical bar in between the bodies of the productions. Thus, the two productions can be rewritten as

$E \rightarrow E + T \mid E * T$

CHECK YOUR PROGRESS – I

1. The lexical analysis phase of a compiler is responsible for producing __________.
2. For specifying the syntax of a language, the notation used is termed as _________.
3. The grammar oriented compiling technique is called _________.
4. Syntax can be regarded as the _________of a language.
5. If a statement does not conform to the syntax of the language, the compiler shows a _______message.
6. The left side of a production is termed as ____ and the right side is termed as ____.
7. Terminals are written using ________ letters.
8. Non-terminals are written using ________ letters.

Context free grammars are capable of describing most of the syntax of the programming languages, which makes it suitable for used in parsing. There are two major types of parsing techniques available- top-down and bottom-up. Top-down parsing begins at the start symbol or root and it proceeds until the leaf nodes contain terminal symbols. Conversely, bottom–up parsing starts at the leaf nodes and it proceeds till we get the start symbol. We shall study these two techniques in later chapters. Parser which is used in top down parsing is called LL parser and parser which is used in bottom up parsing is called LR parser.

5.4 LL PARSER

In the previous section, we got to know that during parsing, scanning of input symbols takes place from left to right. Once a symbol is read, corresponding production rule is applied so that parsing happens efficiently. The LL(1) class is a predictive parsing grammar. The first "L" stands for scanning the input from left to right and the second "L" is for doing the leftmost derivation. During each parsing step, the look ahead symbol is one input symbol and it is designated as 1. LL(1) is a strong class of grammar which can represent most of the programming language constructs. No left recursive or ambiguous grammar can be LL(1).

We have two distinct productions $A \rightarrow \alpha \mid \beta$ of grammar G. G is termed as LL(1) if the following conditions hold.

1. For no terminal a, both α and β derive strings beginning with a.
2. At most one of α and β can derive an empty string.
3. If $\beta \overset{*}{\Rightarrow} €$, then α does not derive any string beginning with a terminal in FOLLOW(A). Similarly, if $\alpha \overset{*}{\Rightarrow} €$, then β does not derive any string beginning with a terminal in FOLLOW(A).

Predictive parsers can be constructed using LL(1) grammar. By looking at the current input symbol, the appropriate production rule for a non-terminal has to be applied. For instance, the flow-of-control constructs can satisfy LL(1) conditions. For an LL(1) grammar, a parsing table is constructed. Each entry of the parsing table uniquely determines which production rule has to be applied on the current input symbol. Otherwise, it signals an error. Sometimes, the parse table can have multiple entries for the same input. This happens when the grammar is ambiguous or left recursive. However, some grammar can never be LL(1) although ambiguity and left recursion are eliminated. We shall study about this elaborately later.

5.5 LR PARSER

Unlike LL parser, LR parser is a bottom-up parser. This class of grammar is termed as LR(*k*) where L stands for "L" stands for left to right scanning of the input and "R" stands for rightmost derivation in reverse and k is the number of lookahead input symbols. The default value of k is 1. This is a simple and efficient approach to constructing shift-reduce parser and is termed as the Simple LR parser. Apart from this, there are two complex structures: **Canonical LR and LALR**. Like the non-recursive LL parsers, LR parsers are also table-driven parsers. An LR parser must be able recognize the handles appearing at the top of the stack. LR parsing possesses some important characteristics which make the parser attractive as well as efficient.

- LR parsers can recognize virtually most of the programming language constructs for which context free grammars can be defined.
- It is the most general non-backtracking shift-reduce parsing method.
- As it scans inputs from left to right, it is possible to early detect the syntax error by the parser.
- The LR grammar can recognize more programming constructs than the LL grammar. Therefore, the class of grammars parsed by the LR grammar class is a proper superset of the class of grammars parsed by the predictive or LL grammars. The parser must be able to recognize the right hand side of a production occurring in a right sentential form with k-lookahead symbols. And therefore, the requirement for a parser to be LR is less strict compared to that of an LL(1) grammar.

However, this method has a serious drawback of involving much more work when it comes to the costruction of parser. There are two major operations involved in this parser: *Shift* and *Reduce*. The parser has to take a decision on when to shift and when to reduce. The parser also maintains states in order to represent the halting positions in which the parser might be in. The states represent a set of items that the parser generated during parsing. An item in LR(0) is derived by placing a dot at some position of a production's body. Thus, the set of items of a production X → ABC would be:

X → .ABC

X → A.BC

X → AB.C

X → ABC.

And if the production is of the form X → €, then the item would be X → . . The item X → .ABC indicates that we would to get a string derivable from ABC next on input. Similarly, the item X → A.BC indicates that we already have seen a string derivable from A and expect to see another derivable from BC. Item X → ABC. indicates that ABC exists in the body and it can be reduced to X. Such collection of set of items is called as *canonical LR(0)* items. These items are later used to form a Deterministic Finite Automaton

(DFA) and the automaton is termed as *LR(0) automaton*. Each state of the automaton represents a set of items in the canonical LR(0) collection of items. The collection is constructed using two functions: CLOSURE and GOTO. A detailed discussion of the construction of Simple LR (SLR) or LR(0) parser will be done later in a chapter.

Two more powerful LR parsers are Canonical LR and LALR parsers. The Canonical LR method works on a large set of items and it is termed as LR(1) parser. On the other hand, the LALR or Lookahead LR method contains fewer states than the LR(1) parsers. It is based on LR(0) items and can handle more grammars.

CHECK YOUR PROGRESS – IV

9. The LL(1) class is a __________________ grammar.
10. No left recursive or ambiguous grammar can be ___________.
11. LR parser is a ________________ parser.
12. LR parser is an efficient approach to constructing a ________ parser.
13. Both LL and LR parsers are __________ parsers.
14. The canonical LR(0) items form a ________________.
15. The canonical LR(0) items are constructed using two functions: ______ and ______.

5.6 SUMMING UP

- The lexical analysis phase of a compiler is responsible for breaking the program into constituent pieces of stream of characters and generates tokens for the same.
- Tokens are passed to the parsing phase in order to construct the grammatical structure or syntax of a language. For specifying the syntax, a widely accepted notation termed as Context-Free Grammar (CFG) or Backus Naur Form (BNF) is used.
- A context Free Grammar G is a 4-tuple (V, T, P, S), V is the set of variables or non-terminals, T is the set of terminals, P

states the set of production rules and S is the start symbol of G.

- Terminals include lowercase letters, operators, digits, punctuation marks and keywords. Non-terminals are designated in terms of uppercase letters.
- A grammar derives strings of terminals. The strings are derived by beginning with the start symbol. At each derivation step, each non-terminal is repeatedly replaced by the body of the production. Derivation stops when the string of terminals is finally generated.
- The set of all strings generated by a grammar is called the language of the grammar. A language L for the set of grammars G is denoted as L(G).
- Grammars are more powerful notation than regular expression. They describe a language more precisely. A programming language construct that is described by a regular expression can also be described by a grammar. However, the reverse is not possible. Therefore, every regular expression is a context free grammar but not vice-versa.
- There are two major types of parsing techniques available- top-down and bottom-up. Top-down parsing begins at the start symbol or root and it proceeds until the leaf nodes contain terminal symbols. Conversely, bottom–up parsing starts at the leaf nodes and it proceeds till we get the start symbol.
- The LL(1) class is a predictive parsing grammar. The first "L" stands for scanning the input from left to right and the second "L" is for doing the leftmost derivation. During each parsing step, the look ahead symbol is one input symbol and it is designated as 1. No left recursive or ambiguous grammar can be LL(1).
- Unlike LL parser, LR parser is a bottom-up parser. This class of grammar is termed as LR(*k*) where L stands for "L" stands for left to right scanning of the input and "R" stands for rightmost derivation in reverse and *k* is the number of look ahead input symbols. The default value of *k* is 1. This is a simple and efficient approach to constructing shift-reduce parser and is termed as the Simple LR parser.
- There are two complex and powerful structures of LR parsing: **Canonical LR and LALR**. Like the non-recursive

LL parsers, LR parsers are also table-driven parsers. LR grammar can recognize more programming constructs than the LL grammar.

- There are two major operations involved in this parser: *Shift* and *Reduce*. The parser has to take a decision on when to shift and when to reduce.
- An item in LR(0) is derived by placing a dot at some position of a production's body. Such collection of set of items is called as *canonical LR(0)* items. These items are later used to form a Deterministic Finite Automaton (DFA) and the automaton is termed as *LR(0) automaton*.

5.7 ANSWERS TO CHECK YOUR PROGRESS

1. Tokens
2. Context-Free Grammar (CFG)
3. Syntax Directed Translation
4. Grammar
5. syntax error
6. head, body
7. lowercase
8. uppercase
9. predictive parsing
10. LL(1)
11. bottom-up
12. shift-reduce
13. table driven
14. Deterministic Finite Automaton (DFA)
15. CLOSURE, GOTO.

5.8 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is Context-Free Grammar?
2. Write down the functions of syntax analysis phase?
3. What is grammar? Define.
4. What do you mean by language of a grammar?
5. How language and syntax are related?
6. What is production? Describe in brief.

7. Describe the notational conventions of terminals and non-terminals.
8. What is look ahead symbol? Discuss in brief.
9. What is LR(0) parsing technique? Discuss.
10. What is item in LR(0) parsing technique?

B. Long answer type questions.

1. Give some characteristic overview of the LL(1) parser.
2. Give some characteristic overview of the LR parser.
3. Why are LR parsers considered to be an efficient parser?

5.9 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). *Compiler design: theory, tools, and examples*.
- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C (pp. I-XVIII)*. Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). *Compilers: principles, techniques, and tools*, Second Edition.

×××

UNIT: 6
PARSE TREE AND AMBIGUITY

Unit Structure:

6.0 INTRODUCTION

The task of parsing or syntax analysis phase is to obtain a stream of tokens generated by the lexical analyzer. A parser tries to verify whether a string of tokens can be generated by the grammar of the language. The parser also reports any syntax errors if the string of tokens do not conform to the grammar or the language rules. During parsing, a tree is constructed. It is a graphical representation that shows how a start symbol of the grammar derives a string of the language. This tree is termed as parse tree. The syntax of the language is specified using the notation called Context Free Grammar (CFG) or simply grammars. They are basically represented in terms of production rules. We already have discussed production rules and their structures have been discussed in the last chapter. In this chapter, we shall discuss the derivation of strings using production rules defined for the language. Later, we shall discuss how parse trees are constructed and also various issues pertaining to the derivation of strings and construction of parse trees.

6.1 UNIT OBJECTIVE

After going through this unit, you will be able to

- Define derivation of strings
- Understand different types of derivations
- Know the concepts of parsing and its types
- Know how parse trees are constructed
- Explain the relation between derivation and parse tree
- Describe the ambiguity of grammars
- Know what is parsing

6.2 DERIVATIONS

The set of all strings generated by a grammar is called the language of the grammar. A language L for the set of grammars G is denoted as L(G).

Now, let us see how grammars take part in derivation of strings.

Production rules of a grammar are applied to derive strings of terminals. The strings are derived by beginning with the start symbol. At each derivation step, each non-terminal is repeatedly replaced by the body of the production. Derivation stops when the string of terminals is finally generated.

Consider two productions of the form $B \rightarrow \alpha A\beta$ and $A \rightarrow \gamma$. Now, in one step derivation, the right side of the production can be rewritten as $\alpha A\beta \Rightarrow \alpha\gamma\beta$ where α and β are the grammar symbols. The symbol $\Rightarrow$ means derivation in one step. Sometimes, derivation takes place in zero or more steps. It is denoted as $\overset{*}{\Rightarrow}$. Likewise the notation $\overset{+}{\Rightarrow}$ derives strings in one or more steps. Suppose, a string is derived from the start symbol in zero or more steps, we write it as

S⇒α; α is called the sentential form, which is a combination of grammar symbols like terminals and non-terminals.

During the process of derivation, the input symbols of the string to be parsed are scanned from left to right. Then a decision has to be taken about which production rule has to be applied based on the symbol read. Once a symbol is read, accordingly a production to be applied has to be decided. It should be based on the condition that the scanned symbol must exist at the first position of the right side production. But sometimes, situation may arise when more than one production may have that scanned symbol at the right side. Then, a firm decision has to be made which production would best fit that situation.

We shall now try to derive a simple string 8 + 7 – 5 using the following grammar.

list → list – digit

list → list + digit

list → digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 +

The derivation steps would be:

list ⇒ list – digit

⇒ list + digit – digit

⇒ digit + digit – digit

⇒ 8 + digit – digit

⇒ 8 + 7 – digit

⇒ 8 + 7 – 5

Derivation begins by applying the production with the start symbol. Here, the production *list → list – digit* is applied. In the next step, variable *list* is substituted by *list + digit* by using the right side of the production *list →list + digit* and finally the production *list → digit* is applied. Now, the right side contains *digit + digit – digit* and the production with head *digit* is applied. Therefore, digits are replaced and finally we get the desired string of terminals.

Let us again take another set of production rules and try to derive a string (id + id * id).

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$

$E \Rightarrow (E)$

$\Rightarrow (E * E)$

$\Rightarrow (E + E * E)$

$\Rightarrow (id + id * id)$

But, sometimes the grammar or the productions may not be able to produce the string. This in turn deduces the fact that the string does not belong to the language.

Derivation takes place in two different techniques: leftmost derivation and rightmost derivation. Sometimes, the leftmost non-terminal is expanded in each step. In the previous example, the leftmost E is expanded to generate the string (id + id * id). This is termed as leftmost derivation. During the first derivation step, E is the only non-terminal. It is expanded by its right side E * E. In the next step, the leftmost E is expanded with the body E + E. Now, let us reconsider the same example and see how leftmost derivation derives the desired string of terminals.

$E \overset{lm}{\Rightarrow} (E)$

$\overset{lm}{\Rightarrow} (E * E)$

$\overset{lm}{\Rightarrow} (E + E * E)$

$\overset{lm}{\Rightarrow} (id + E * E)$

$\overset{lm}{\Rightarrow} (id + id * E)$

$\overset{lm}{\Rightarrow} (id + id * id)$

Finally, $E \rightarrow id$ is applied to each leftmost E.

On the other hand, the rightmost derivation is another form of derivation. In each derivation step, the rightmost non-terminal is expanded. The same example can be used to do rightmost derivation.

$E \overset{rm}{\Rightarrow} (E)$

$\overset{rm}{\Rightarrow}$(E + E)

$\overset{rm}{\Rightarrow}$(E + E * E)

$\overset{rm}{\Rightarrow}$(E + E * id)

$\overset{rm}{\Rightarrow}$(E + id * id)

$\overset{rm}{\Rightarrow}$(id + id * id)

This is similar to leftmost derivation except the fact that in every derivation step, the rightmost non-terminal gets expanded. During the first step, the only non-terminal E is replaced with the right side; that is E + E. In the next step, the rightmost E is again expanded and gets replaced by E * E. Again, the production E → id is applied on the rightmost E and the sentential form becomes (E + E * id). Eventually, all the rightmost non-terminals are expanded and finally we get the string (id + id * id).

In both the cases, choice of production to be applied becomes crucial aspect because it has to be correctly decided which production rule has to be applied in a particular derivation step.

CHECK YOUR PROGRESS – I

1. Strings are derived by beginning with ________________ symbol.
2. A language L for the set of grammars G is denoted as ___________.
3. The symbol _________ means derivation in one step.
4. The notation ________ derives strings in one or more steps.
5. The notation _______ derives strings in one or more steps.
6. A ________form is a combination of terminals and non-terminals.
7. Derivation takes place in two different techniques: ____and ____derivations.

6.3 PARSE TREE

It is worth to mention here that each derivation step represents corresponding level of a parse tree. Therefore, each step of string derivation can be represented during construction of a parse tree. Just like derivation, parse tree construction also begins at the start symbol. Therefore, production *list* → *list* – *digit* can be represented in terms of parse tree nodes as follows.

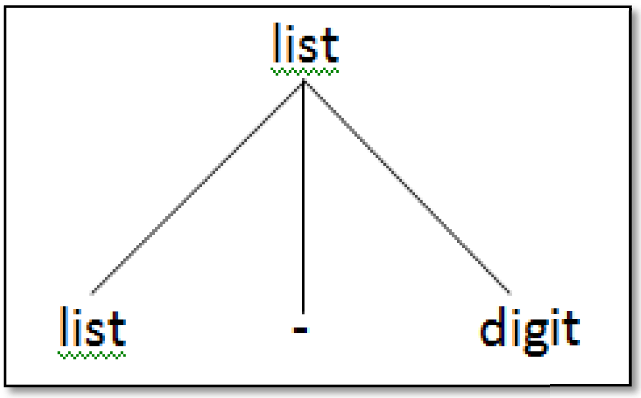


Fig 6.1: Parse tree representation of a derivation step

In parse tree, the start symbol of the grammar is termed as the root of the tree. Root has no parents. The parse tree consists of one or more nodes. The head and the body of a production represent the parent and children respectively. Each grammar symbol of the body of a node appears as the child node of the tree. Edges come out of the parent to its children. Nodes *list*, - and *digit* appear as the children of *list* and are also called as siblings. They are drawn from left to right. If a node has no children, it is called as terminal node. An interior node can have child node(s). A descendant of a node is the node itself, its child node, child of child node and so on. If a node has a descendant node, then it is termed as the ancestor of the descendant. A special case of production A→ ϵ, then a node is labeled with A and has a single child ϵ.

The derivation steps shown in Section 6.2 for the string 8+7-5 can be represented in terms of parse trees as follows-

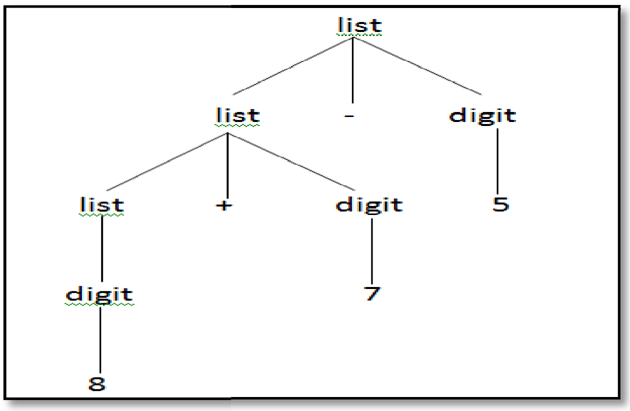


Fig 6.2: Parse tree corresponding to derivation of string 8 + 7 – 5

6.4 AMBIGUITY

During parsing, parse trees are generated. These parse trees are constructed to see if a string of tokens can be generated using a set of grammatical constructs. Now, sometimes situation may arise when the same grammar may produce more than one parse tree for the same string. Such grammar is termed as the ambiguous grammar. An ambiguous grammar can produce more than one leftmost derivation or more than one rightmost derivation for the same string of tokens. Such kind of situation is undesirable as we cannot decide which parse tree to select for a sentence. Therefore, it is desirable to eliminate the ambiguity of grammars. There are disambiguating rules which discard or reject the undesirable parse trees and keeps only one.

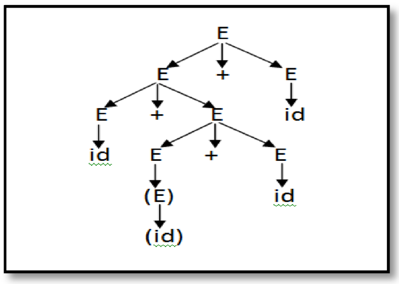
Now, let us see with an example how ambiguous parse trees are produced. We consider the following set of grammars which will produce more than one parse tree.

$$E \longrightarrow E + E$$

$$E \longrightarrow (E)$$

$$E \longrightarrow id \mid €$$

Let us try to derive the string “id+(id)+id+id” using this set of productions.



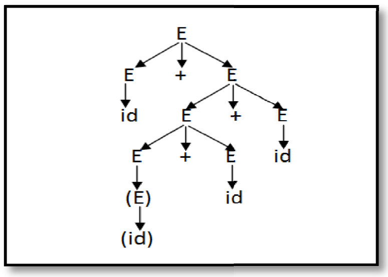


Fig 6.3: Parse tree for string "id+(id)+id+id"

Fig 6.4: Parse tree for string "id+(id)+id+id"

Figures 6.3 and 6.4 derive the same string. But the parse trees are different. Therefore, this grammar can be termed as an ambiguous grammar.

Figure 6.3 involves the following leftmost derivation steps:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E + E \\ &\Rightarrow id + E + E \\ &\Rightarrow id + E + E + E \\ &\Rightarrow id + (E) + E + E \\ &\Rightarrow id + (id) + E + E \\ &\Rightarrow id + (id) + id + E \\ &\Rightarrow id + (id) + id + id \end{aligned}$$

Figure 6.4 involves the following leftmost derivation steps:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E + E \\ &\Rightarrow id + E + E + E \\ &\Rightarrow id + (E) + E + E \\ &\Rightarrow id + (id) + E + E \\ &\Rightarrow id + (id) + id + E \\ &\Rightarrow id + (id) + id + id \end{aligned}$$

There is no particular algorithm to eliminate the ambiguity of grammar. Rather, we can rewrite the grammar such that only one derivation or parse tree for the string can be produced. A left recursive and right recursive grammar can be ambiguous. We shall discuss recursive grammars and elimination of recursive grammars in the later chapter.

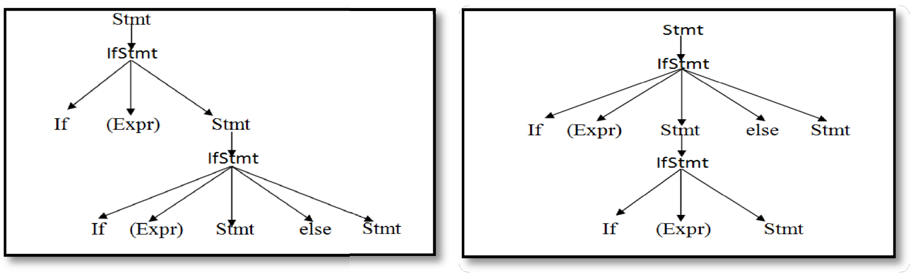
One very common form of ambiguity in programming language is conditional statement. An example of such kind might be the following “if-else” grammar:

Stmt $\rightarrow$ IfStmt

IfStmt $\rightarrow$ If (Expr) Stmt

IfStmt $\rightarrow$ If (Expr) Stmt else Stmt

Now, let’s start deriving the string “If (Expr) If (Expr) Stmt else Stmt”. Derivation begins with the non-terminal Stmt as root. It is a statement which includes another statement with If. The expression Expr evaluates to either true or false.



6.5: Derivation of two parse trees for the string If (Expr) If (Expr) Stmt else Stmt

Source: Compiler Design_ Theory Tools and Examples by Seth D. Bergmann

This problem can be resoved by associating ‘elses’ with the closest previous unmatched ‘ifs’. Therefore, the second tree corresponds to the correct interpretation. The grammar can be rewritten so as get an equivalent as well as unambiguous grammar.

Stmt $\rightarrow$ IfStmt

IfStmt $\rightarrow$ Matched

IfStmt $\rightarrow$ Unmatched

Matched $\rightarrow$ If (Expr) Matched else Matched

Matched $\rightarrow$ Other

Unmatched $\rightarrow$ If (Expr) Stmt

Unmatched $\rightarrow$ If (Expr) Matched else Unmatched

Now, this grammar generates the same string but allows only one parsing.

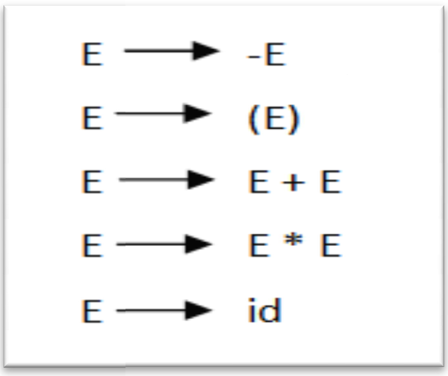
There is a correspondence between context free grammars and regular expressions. Grammars are more powerful notation than regular expression. They describe a language more precisely. A programming language construct that is described by a regular expression can also be described by a grammar. However, the reverse is not possible. Therefore, every regular expression is a context free grammar but not vice-versa.

CHECK YOUR PROGRESS – II

8. Parse tree construction begins at the ________symbol.
9. _____ has no parents.
10. A ________ node has no children.
11. An _________node can have child node(s).
12. Every regular expression is a _________.
13. __________is the process of generating a string of terminals by applying a set of production rules.
14. Derivation and __________ ________are interrelated.
15. An ambiguous grammar can produce more than one _________ derivation or _______ derivation for the same string of tokens.
16. Each interior nodes of a parse tree is labeled with some__________.
17. It is desirable to eliminate the _________of grammars.

6.5 PARSING

During syntax analysis or parsing, characters or tokens are grouped into hierarchical format. This hierarchical format is represented in terms of tree structure. This tree structure is called as the parse tree. Parsing involves grouping the tokens of the source program into grammatical structure. Parsing begins with the start state as its root. Production rules are applied to each node if it is a non-terminal. However, during parsing, it has to be decided exactly which production rule will produce the required string. Sometimes, it may happen that there are more than one production rules available with the same non-terminal. Exactly which production rule will give the desired string of terminals must be decided during each parse tree construction step. Let us see the generation of a parse tree by considering the following production rules.



$$
\begin{aligned}
E &\longrightarrow -E \\
E &\longrightarrow (E) \\
E &\longrightarrow E + E \\
E &\longrightarrow E * E \\
E &\longrightarrow id
\end{aligned}
$$

Fig 6.6: Production rules for generation of parse tree

The string to be derived is : - (id + id). Construction begins at root node E. As the string begins with the symbol '–', the only applicable production is E → -E. During each generation step, the next input symbol is read and accordingly, productions are applied. This time, it is '('; therefore, production E → (E). This way we keep on applying productions and expanding the parse tree untill we get a string consisting of all leaf nodes as terminal nodes. Here, we stop the parse tree construction. Finally, we will concatenate all the leaf nodes in order to generate the string of terminals. In a parse tree, all interior nodes are non-terminals and all leaf nodes are terminals.

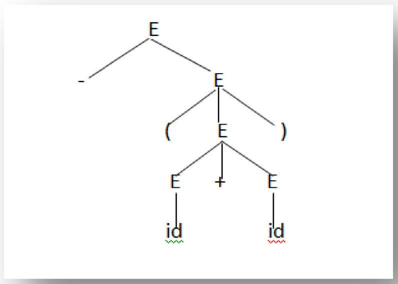


Fig 6.7: Parse tree for string '- (id + id)'

Most commonly, parsing falls in two major categories – top-down parsing and bottom-up parsing. These categories are based on the manner or order in which nodes of the parse tree are constructed.

6.5.1 Top-down parsing

Sometimes, the order of derivation of parse tree begins at the root. Construction proceeds by applying production at the root node of the tree. Subsequently, rules are applied to the non-terminals appearing at the nodes of the tree. This way, the tree gets expanded and at one point of time, we shall have leaf nodes consisting only of terminals. Finally, we will concatenate the leaf nodes in order to generate the string of terminals. We shall consider the following set of production rules and try to derive the string "aabbcbbaa".

$S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow c$

The parse tree generation becomes as follows:

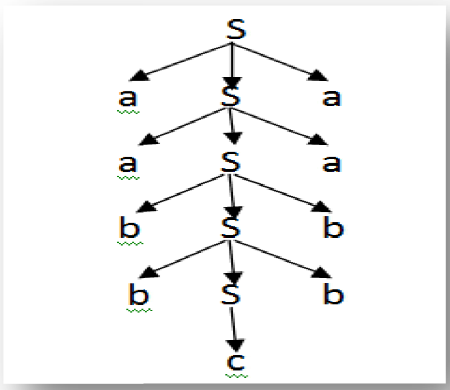


Fig 6.8: Top-down parsing for generation of string “aabbcbbaa”

The parser begins by scanning the string from left with one character at a time. The first character of the string is ‘a’ and there is only one production which starts with ‘a’, S → aSa. So, it is applied from the start state S. Next input symbol is ‘a’ which is also the lookahead symbol of the string. Therefore, the same production is applied. Now the lookahead symbol is ‘b’. Therefore, production S → bSb is applied as it is the only production which starts with ‘b’. For the next character, again the same production is applied. The string that we have got is “aabbSbbaa”. Finally, we apply S → c in order to generate the string “aabbcbbaa”.

It is worth mentioning here that sometimes more than one production may match the lookahead symbol. We may try to apply such production. However, such application may not eventually lead to derive the string that we desire or parsing is incomplete. In that case, we may need to backtrack and try to apply other productions. This eventually leads to consume much time. Therefore, a special case of top-down parsing has been introduced which is termed as the predictive parsing.

A recursive descent parsing is a top-down parsing method in which a set of recursive procedures are executed. A procedure is associated with each non-terminal of a grammar. A special case of recursive-descent parsing is predictive parsing. Here, each lookahead symbol can determine exactly which production has to be applied in every step of parsing. Here, a matching procedure unambiguously deteremine the procedure to be selected for each non-terminal.

Let us consider the following set of rules which derives the same string "aabbcbbaa".

$S \rightarrow aSa \mid Sa$

$S \rightarrow bSb \mid Sb$

$S \rightarrow c$

The lookahead symbol is initially set to 'a'. Parsing begins with a call to the procedure for the starting non-terminal S. There are two productions starting with S. The code is executed as follows:

```
            match(a); S; match(a)                --------------(1)
and         S; match(a)                          -------------- (2)
```

Each terminal in the right side of the production is matched against the lookahead symbol. Each non-terminal again calls its procedure. 2 does not match the lookahead. However, 1 does. Next, S is expanded. The next looahead is 'a'. Accordingly, production S with right side aSa is applied. Therefore, the same code like 1 is executed. Next time again S is expanded with the following code and the lookahead symbol is 'b'.

```
            match(b); S; match(b)                -------------- (3)
```

Once 'b' is matched, again the procedure for S is to be executed. The same set of codes like 3 will be executed. Therefore, the next lookahead symbol 'b' is matched against match(b) procedure and finally code for S is again executed. This time the code to be executed is

```
            match(c)                             -------------- (4)
```

This time, the lookahead symbol is ‘c’. This matches with the matching procedure 4. The lookahead symbol guides the selection of production to be used. The succeeding lookahead symbols are already generated and no more non-terminals are to be expanded.

6.5.2 Bottom-up parsing

Bottom-up parsing is the reverse of top-down parsing. Here, parsing begins at the leaf nodes. At every derivation step, grammar rules are applied and ultimately we reach a point where only the start state is attained. Bottom up parsing always begins with an empty stack. One or more input symbols are pushed onto the stack. Now, according to the grammar rules, these symbols are replaced by non-terminals. Parsing terminates when all the input symbols have been read and we get the start symbol that is, the root node left in the stack. This whole process can be thought as reducing a string to the start symbol of the grammar. At every reduction step, a particular substring matching the right side of a production is replaced by the left side of the production. It can be observed here that at every step, the rightmost derivation occurs in reverse. The substring of the string which matches the right side of a production is also termed “handle” of the string. The handle is reduced to the non-terminal at the left side of the production. This represents one step along the reverse of a rightmost derivation. Bottom-up parsing is also termed as shift-reduce parsing.

Now, let us see how bottom-up parsing happens while parsing the same string “aabbcbbaa”.

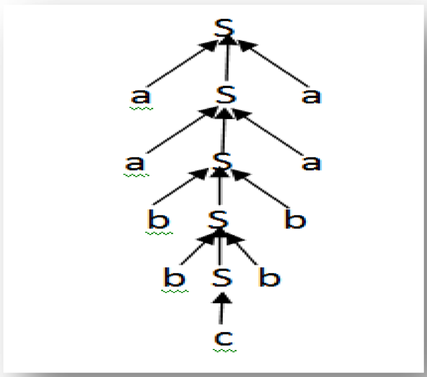


Fig 6.9: Bottom-up parsing for generation of string “aabbcbbaa”

The derivation step proceeds as follows:

aabbcbbaa
aabbSbbaa
aabSbaa
aaSaa
aSa
S

The stack contains a $ and the input string is aabbcbbaa$. When finally parsed, the stack will contain $S (S being the start symbol) and the input string will contain $. The following table represents each move of the parser and shows how the shift-reduce parser behaves during parsing.

Table 1 : Shift-reduce parsing on input “aabbcbbaa”

Step	Stack	Input	Action
1	$	aabbcbbaa$	Shift
2	$a	abbcbbaa$	Shift
3	$aa	bbcbbaa$	Shift
4	$aab	bcbbaa$	Shift

5	\$aabb	cbbaa\$	Shift
6	\$aabbc	bbaa\$	Reduce to S $\rightarrow$ c
7	\$aabbS	bbaa\$	Shift
8	\$aabbSb	baa\$	Reduce to S $\rightarrow$ bSb
9	\$aabS	baa\$	Shift
10	\$aabSb	aa\$	Reduce to S $\rightarrow$ bSb
11	\$aaS	aa\$	Shift
12	\$aaSa	a\$	Reduce to S $\rightarrow$ aSa
13	\$aS	a\$	Shift
14	\$aSa	\$	Reduce to S $\rightarrow$ aSa
15	\$S	\$	Accept

The shift-reduce parser involves four possible actions: 1. Shift 2. Reduce 3. Accept 4. Error

1. In Shift action, the next input symbol is shifted to the top of the stack
2. In Reduce action, the parser identifies the top of the stack as the right end of the handle. The handle is then replaced with the non-terminal at the right.
3. In Accept action, the parser announces successful completion of parsing.
4. In Error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

As already mentioned, a "handle" of a string is the substring that matches the right side of a production. The handle is reduced to the non-terminal on the left side of the production. During step 6, "c" is the handle and is reduced to S. During steps 8 and 10, "bsb" is the handle generated and is reduced to S. Similarly, steps 12 and 14 reduce handle "aSa" to the non-terminal S.

CHECK YOUR PROGRESS- III

18. Parsing involves grouping the tokens of the source program into __________.
19. The hierarchical structure created during parsing is called ____________.
20. During top-down parsing, construction begins at the ____________ of the tree.
21. A top-down parsing method in which a set of recursive procedures are to be executed is termed as __________ ____________ _____________.
22. What is predictive parsing?
23. __________ ____ parsing begins at the leaf nodes and ends at _________.
24. What do you understand by handle of a string?
25. Bottom-up parsing is also termed as ________________.
26. What are the actions involved in Shift-Reduce parsing?

STOP TO CONSIDER

A **handle** of a string is a substring that matches the right side of a production. During shift-reduce parsing, reduction to the non-terminal on the left side of a production takes place. This happens when we have a right sentential form α and a production A → β, with β matching a substring at a particular position of α. In that case, β is replaced by A and eventually the whole string reduces to the start symbol of the grammar. β is termed as the handle of the string and each reduction represents one step along the reverse of a rightmost derivation.

Reducing β to A in α can be regarded as **handle pruning**. This is another way of removing the children of A from the parse tree.

6.6 SUMMING UP

- A grammar derives strings of terminals. The strings are derived by beginning with the start symbol. At each derivation step, each non-terminal is repeatedly replaced by the body of the production. Derivation stops when the string of terminals is finally generated.

- The set of all strings generated by a grammar is called the language of the grammar. A language L for the set of grammars G is denoted as L(G).
- Derivation may take place in zero or more steps or in one or more steps. A sentential form is a combination of grammar symbols like terminals and non-terminals. During the process of derivation, the input symbols of the string to be parsed are scanned from left to right.
- Each step of string derivation can be represented during construction of a parse tree. Just like derivation, parse tree construction also begins at the start symbol or root. Root has no parents.
- A tree consists of one or more nodes. The head and the body of a production represent the parent and children respectively. Edges come out of the parent to its children.
- Grammars are more powerful notation than regular expression. They describe a language more precisely. A programming language construct that is described by a regular expression can also be described by a grammar. However, the reverse is not possible. Therefore, every regular expression is a context free grammar but not vice-versa.
- Each derivation step can be represented as a parse tree level in top-down fashion. Parse tree is a depiction of derivation.
- An ambiguous grammar can produce more than one leftmost derivation or more than one rightmost derivation for the same string of tokens.
- Durng ambiguity, we cannot decide which parse tree to select for a sentence. There are disambiguating rules which rewrite the grammar and discard the undesirable parse trees and keeps only one
- Parsing falls in two major categories – top-down parsing and bottom-up parsing.
- In top-down parsing, construction begins at the root node of the tree. Subsequently, rules are applied to the non-terminals appearing at the nodes of the tree. Construction stops when leaf nodes consisting only of terminals are generated.

- Bottom-up parsing is the reverse of top-down parsing. Parsing begins at the leaf nodes. At every derivation step, grammar rules are applied by shifting and reducing and parsing stops when only the start state is attained.
- Bottom up parsing always begins with an empty stack. Symbols are pushed onto the stack. These are reduced by non-terminals according to the grammar rules.
- At every reduction step, a particular substring matching the right side of a production is replaced by the left side of the production. The substring of the string which matches the right side of a production is also termed "handle" of the string. The handle is reduced to the non-terminal at the left side of the production.

  .

6.7 ANSWERS TO CHECK YOUR PROGRESS

1. Start
2. L(G)
3. $\Rightarrow$
4. $\Rightarrow^{*}$
5. $\Rightarrow^{+}$
6. sentential form
7. leftmost, rightmost
8. start
9. Root
10. terminal
11. interior
12. context free grammar
13. Derivation
14. parse trees
15. leftmost, rightmost
16. non-terminals
17. ambiguity
18. grammatical structure.
19. parse tree
20. root
21. recursive descent parsing
22. A special case of recursive-descent parsing is called the predictive parsing. Here, each look ahead symbol can

determine exactly which production has to be applied inat every step of parsing.

23. Bottom-up, root
24. The substring of the string which matches the right side of a production is termed as "handle" of the string. The handle is reduced to the non-terminal at the left side of the production.
25. shift-reduce parsing
26. The shift-reduce parser involves four possible actions: 1. Shift 2. Reduce 3. Accept 4. Error

6.9 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What are the roles of a parser?
2. What is parse tree? Discuss.
3. How are the production rules represented?
4. What is leftmost derivation? Discuss.
5. What is rightmost derivation? Discuss.
6. Show with an example, the ambiguity in conditional statements?
7. What is recursive descent parsing?
8. What is predictive parsing?
9. How are derivations and parse trees related?

B. Long answer type questions.

1. What is derivation? Explain with examples.
2. What are the two approaches of derivations? Explain each of them elaborately.
3. What is ambiguity? Explain with example.
4. Explain the process of parsing with examples.
5. What is parsing? What are its types? Explain each of them elaborately.
6. Describe predictive parsing in detail.
7. What is Shift-Reduce parsing? Explain with example.

6.9 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.

- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C* (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 7
DERIVING FIRST & FOLLOW SETS

Unit Structure

7.0 INTRODUCTION

Grammars describe the syntax of programming languages. In the last unit, we discussed that an ambiguous grammar can produce more than one parse tree for the same set of grammar rules. In this unit we shall discuss how left recursive grammars are eliminated for effective handling of top-down parsing. Another important grammar transformation termed as left factoring is suitable for predictive parsing. We shall discuss left factoring of grammars in this unit.

7.1 UNIT OBJECTIVE

After going through this unit, you will be able to

- Understand left recursion and its possible elimination
- Know how to do left factoring of grammars
- Derive FISRT and FOLLOW sets

7.2 ELIMINATION OF LEFT RECURSION

Context Free Grammars (CFG) describe the syntax of a language. They are the set of recursive rules used to generate patterns of strings of the language. In the language theory, an LL grammar is a context free grammar which parses the input from left to right and constructs the leftmost derivation of the sentence. When scanning happens from left to right with one input symbol as look ahead at each step of parsing, it is termed as the LL(1) grammar. An LL(1) grammar has to be unambiguous. It must not contain any common left prefixes with no left-recursion. Later, the LL(1) parsing table is constructed by generating the FIRST and FOLLOW sets of the grammar. We already have had some idea regarding what ambiguity is. Now, let us talk about left recursion. A grammar is said to be left recursive if it is of the following form:

$A \rightarrow A\alpha$ --------------- (1)

Apart from this, any grammar rule of the form $A \rightarrow B\beta$ ---------------- (2) such that $B \Rightarrow A\gamma$ by some sequences of derivations. The leftmost non-terminal appears as the first symbol on the right side of the production. It can create infinite loop, creating errors as well as leading significant decrease in performance.

A production rule of the form Expr → Expr + Term can be considered as a left recursive grammar as non-terminal Expr repeats at the first place of the right side. We can resolve this problem simply by rewriting the production as Expr → Term + Expr. But this may create a new problem of common left prefix. This would also result in the right associative plus operator.

More formally, a grammar of the form

$A \quad A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots..$ ----------- (3)

Can be termed as the left recursive grammar and can be rewritten by introducing a new variable or non-terminal A′ in the following manner:

$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots..$

$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots.. \mid €$

Let us understand left recursion with an example. Consider the following grammar and eliminate left recursion from it.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Productions $E \rightarrow E + T$ and $T \rightarrow T * F$ are left recursive and need to be resolved. We introduce two new variables E′ and T′ and rewrite the productions as follows:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid €$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid €$

$F \rightarrow (E) \mid id$

We shall see another left recursive grammar and rewrite it by eliminating the same.

$A \rightarrow ABd / Aa / a$

$B \rightarrow Be / b$

The new grammar would be as follows:

$A \rightarrow a A'$

$A' \rightarrow BdA' \mid AA' \mid €$

$B \rightarrow bB'$

$B' \rightarrow eB' \mid €$

Similarly, another set of productions

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid €$

Non-terminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$. However, it is not immediately left recursive.

Considering another example which is indirectly left recursive and try to eliminate left recursion from it.

$S \rightarrow A\alpha \mid \beta$

$A \rightarrow Sd$

Now, the grammar can be transformed into the following left recursive form:

$$S \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow A\alpha d \mid \beta d$$

Now, eliminating left recursion will produce the following productions:

$$S \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow \beta d A'$$

$$A' \longrightarrow \alpha d A' \mid €$$

7.3 LEFT FACTORING

Sometimes, we might have multiple grammars with the same non-terminal at left side. The right side contains a common prefix of tokens. We simply rewrite these productions and replace them with another production. Transformation of such kind is useful for producing a grammar suitable for predictive parsing. Formally, the productions of the form:

$A \quad \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \rightarrow \ldots\ldots \mid \alpha\, \beta_n \mid \gamma$ ------------ (4)

are n-productions with a common prefix α. γ represents all kinds of productions which do not begin with α . We introduce a new variable A′ and rewrite the productions as follows:

$$A \longrightarrow \alpha\, A' \mid \gamma$$

$$A' \longrightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$$

Such equivalent representations are termed as left factoring of grammars.

Let us explain left factoring with a suitable example.

$$S \longrightarrow iEtS \mid iEtSeS' \mid b$$

$$E \longrightarrow a$$

Here, i, t and e stand for if, then and else. E and S both stand for expression and statement. Now, the left factored grammar would be:

$S \longrightarrow iEtSS' \mid b$

$S' \longrightarrow eS' \mid €$

$E \longrightarrow a$

S′ be the new variable used during transformation. Left factorization eliminates common prefixes from a set of productions.

CHECK YOUR PROGRESS- I

1. An LL(1) grammar has to be ______________.
2. LL(1) parsing table is constructed by generating the ____________ and __________ sets of the grammar.
3. A left recursive grammar creates ______________.
4. In a ____________ ______________ grammar, it is desirable to eliminate the common prefix of tokens.

7.4 DERIVE THE FIRST AND FOLLOW SETS

In the following section, we shall try to derive the FIRST and FOLLOW sets for any LL grammar G, which will be used for construction of effective predictive parsing table. Construction of predictive parsing table has been discussed in next chapter. Note that, the grammar G should be left factored as well as left recursion should be eliminated from the productions which are left recursive.

FIRST(α), for a string of grammar symbols α, is a set of terminals that begin the strings derived from α. If $\alpha \overset{*}{\Rightarrow} €$, then € is in FIRST(α).

To derive the FOLLOW(A) for a non-terminal A, the set of terminals appearing immediately to the right of A must be found out. For a derivation of the form $S \overset{*}{\Rightarrow} \alpha Ab\beta$, where αAbβ be a sentential form, FOLLOW(A) is derived by the set of terminals b. If A is the rightmost symbol of the sentential form, then FOLLOW(A) consists of only of $.

Now, let us understand the rules for computing the set FIRST(A) for the grammar symbol A. We apply these rules untill no more terminals or € can be added to the FIRST set.

1. If A is the terminal, then FIRST(A) consists only of the set {A}.
2. If $A \rightarrow €$ is a production, then FIRST(A) contains the element €.
3. Let $X \rightarrow Y_1Y_2Y_3....Y_n$ be a production for the non-terminal X, then a is in FIRST(X) if for some i, a is in FIRST(Y_i) and € is in all FIRST(Y_1),…….., FIRST(Y_{i-1}). If € is in FIRST(Y_j) for all j=1,2,….,n, then add € to FIRST(X). If Y_1 does not derive €, then FIRST(X) consists of the set FIRST(Y_1). But if $Y_1 \Rightarrow €$, then FIRST(Y_2) is added to FIRST(X) and so on.

Now, the FIRST for a string $Y_1Y_2Y_3....Y_n$ is computed by adding all non-€ symbols of FIRST(Y_1). If € is in FIRST(Y_1), then the non-€ symbols of FIRST(Y_2) is added. Similarly, if FIRST(Y_2) contains €, then the non-€ symbols of FIRST(Y_3) will be added to the FIRST set. Finally, for all i, if FIRST(Y_i) contains €.

Now, to compute the FOLLOW(A) set for a non-terminal A, we must apply the following rules:

1. Place \$ to the FOLLOW(A), where A is the start symbol and \$ is the right endmarker.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except € is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$, where FIRST(β) contains €, then everything in FOLLOW(A) is in FOLLOW(B).

Example 1: Consider the following grammar and find the FIRST and FOLLOW sets.

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid €$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid €$

$F \rightarrow (E) \mid id$

We know that,

FIRST (E)=FIRST(T)=FIRST(F)={ (, id }

FIRST (E′)={ +, € }

FIRST (T′)={ *, € }

FOLLOW (E)= FOLLOW(E′)={), $ }

FOLLOW (T)= FOLLOW(T′)={ +,), $ }

FOLLOW (F)= { *, +,), $ }

As per rule 3 of derivation of FIRST set, FIRST (E), FIRST (T) and FIRST (F) are equal. We can derive FIRST (F) set having terminals { (, id }. Similarly, FIRST (E′) will contain the set { +, € } as per the production E′ → + T E′ | €. And FIRST (T′) will be derived by the set { *, € } as per the production T′ → * F T′ | €.

Now, derivation of FOLLOW (A) sets require $ to be placed in the set where A is the start symbol. Here, the start symbol is E. Therefore, FOLLOW (E) will contain $ in the set. Apart from this, the FOLLOW (E) also contains ')' in the set as it is the terminal which follows E. Now, according to rule 3 everything in FOLLOW (E) is in FOLLOW (E′). Therefore, FOLLOW (E′) will contain the same set. Derivation of FOLLOW (T) require the set FIRST (E′)={ +, € }. Therefore, '+' is in the set. Since, E′ => €, therefore according to rule 2 everything in FOLLOW(E′) is in FOLLOW(T). Thus, FOLLOW (T) is in { +,), $ }. Rule 3 implies FOLLOW (T′) will contain the same elements that FOLLOW (T) contains. In the same way, FOLLOW (F) will be derived by the set FIRST (T′) which is { *, € }. Now, FOLLOW (F) will contain the element '*' in the set. Since, E′ => €, everything in FOLLOW (T′) is in FOLLOW (F). Therefore, FOLLOW (F) will have the set { *, +,), $ }.

FIRST and FOLLOW sets are essential for construction of predictive parsing table. Any grammar can derive these sets in order to produce its corresponding parsing table. However, if the grammar is left-recusrive or ambiguous, the table will have multiple entries.

We shall see how to construct predictive parsing tables based on these sets in the next unit.

CHECK YOUR PROGRESS- II

5. The top-down parser which involves backtracking is called the __________ __________ parser.
6. A special form of top down parser which does not involve backtracking is termed as _________ parsing.
7. ________ parsers can be constructed using a class of LL(1) grammars.
8. A predictive parser program consists of procedures for each __________.

7.5 SUMMING UP

- Context Free Grammars (CFG) describe the syntax of a language. They are the set of recursive rules used to generate patterns of strings of the language. An LL grammar is a context free grammar which parses the input from left to right and constructs the leftmost derivation of the sentence.
- When scanning happens from left to right with one input symbol as look ahead at each step of parsing, it is termed as the LL(1) grammar. An LL(1) grammar has to be unambiguous. LL(1) parsing table is constructed by generating the FIRST and FOLLOW sets of the grammar.
- Sometimes, we might have multiple grammars with the same non-terminal at left side. The right side contains a common prefix of tokens. We simply rewrite these productions and replace them with another production in order to make it suitable for predictive parsing.
- Construction of predictive parsing involves two functions- construction of FIRST and FOLLOW sets. They fill entries of the parse table.

7.6 ANSWERS TO CHECK YOUR PROGRESS

1. Unambiguous
2. FIRST and FOLLOW

3. infinite loops
4. left factored
5. recursive descent
6. predictive
7. Predictive
8. non-terminal

7.7 POSSIBLE QUESTIONS

A. Short answer type questions.

1. Why is it required to eliminate left recursion from a grammar? Discuss.
2. What is left recursive grammar? Describe.
3. What are the tasks performed by a predictive parser program?
4. Why is it necessary to derive the FIRST and FOLLOW sets of a grammar?

B. Long answer type questions.

1. Describe how left recursion is eliminated from a grammar?
2. What is left factoring of grammar? Explain with an example.
3. Describe the rules for deriving the FIRST and FOLLOW sets of a grammar.

7.8 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C* (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 8
TOP DOWN PARSING

Unit Structure

8.0 INTRODUCTION

We already have understood what parsing is. We also know that parsing is of two types- top down and bottom up. In top down parsing, construction of parse tree begins at the root and it expands until the leaf nodes generate a string of terminals. Parsing of such kind may or may not require backtracking. Backtracking parsers need repeated scanning of the input during parse tree construction. And when a grammar has more than one production with the same left hand side, then it may be required to backtrack to the previous input once application of a production does not lead to generate the desired string of terminals. Another form of top down parser which does not require backtracking is the predictive parsing. In this technique, from the set of productions mentioned above, the parser is able to select the most eligible one, which leads to generation of desired string of terminals. In the last unit, we had tried to generate the FIRST and FOLLOW sets; the two major functions of the predictive parsing table. In this unit, we shall discuss the different types of top down parsers. We shall also discuss how to generate parsing tables for all these grammars. We shall learn to implement these two sets in order to unambiguously determine which production rule to be applied in every generation step.

8.1 UNIT OBJECTIVE

After going through this unit, you will be able to:

- Define top down parsing
- Understand the working of a top-down parsing
- Explain the functionality of an LL(1) parser
- Know the working principle of a recursive descent parser
- Explain the working of Non-recursive Predictive Parsing

8.2 TOP DOWN PARSING

The parsing problem can be defined as: given a grammar and a string, we need to determine whether the string belongs to the language of the grammar. Parsing algorithms define the structure in which the derivation tree is built or traversed. Based on this, parsing algorithms are classified into top down and bottom up parsing. Top down parsing corresponds to application of rules in a general top down fashion. The algorithm scans one symbol at a time and applies production rules and decides whether the string can be derived. Parsing always begins with the starting non-terminal. It tries to decide which production rule to be applied at every derivation step. The first symbol on the right side of rules is compared with the scanned input symbol. If they match, the rule is applied. Top down parsing attempts to find the leftmost derivation of an input string. Eventually, the parse tree is constructed from the root and nodes are generated in preorder fashion.

Methods are written for each non-terminal of the grammar. The allowable format of the grammar is A → α; where α is a string of terminals and non-terminals.

8.2.1 LL(1) Grammar

We have got a brief overview on LL(1) grammars in the previous unit. Language structures must be expressed in order ensure that it belongs to the LL(1) community. However, not all languages can be expressed in terms of LL(1) grammar. They are the subset of Context Free Grammars (CFGs) that can be parsed with any simple parsing algorithm. It can be parsed by considering only one non-terminal and the next token in the input stream. However, it must be remembered that an ambiguous grammar can never be an LL(1) grammar. Therefore, ambiguity must be removed from it. Apart from this, a left recursive grammar cannot be LL(1) also. Same is the case with grammars with common left prefixes. Elimination of left recursion is must for ensuring a grammar to be LL(1). Similarly, common left prefixes must also be removed. Once these three are done, we can go for generating the FIRST and FOLLOW sets of the transformed grammar anf finally construct the LL(1) parsing table. We already have learnt to eliminate ambiguity of grammar in the last unit. We also have learnt to eliminate left recursion as well as common left prefixes.

Scanning happens from left to right and produces the leftmost derivation of the input with one lookahead symbol at each step of parsing. LL(1) grammars can also be parsed with tables. The parsing table has two entries rows and columns- non-terminals and terminals respectively. A table-driven parser requires a grammar, a parse table and a stack to represent the non-terminals of the grammar. During each step, the top of the stack and the next input token are considered. If they match, stack is popped and the token is accepted. If they do not, the parse table is consulted and the next rule is applied. This process continues until the end of string is reached and parser successfully halts. The FIRST and FOLLOW sets are considered to play the crucial role in this parsing method.

CHECK YOUR PROGRESS- I

1. What is top-down parsing?
2. __________parsers need repeated scanning of the input during parse tree construction.
3. Another form of top down parser which does not require backtracking is the __________ ________.
4. _____grammar are the subset of Context Free Grammars.
5. An _________ and _______________ grammar can never be an LL(1) grammar.
6. What are the three prerequisites for generating an LL(1) grammar?
7. LL(1) grammars can also be parsed with__________.
8. An LL(1) parsing table contains ___________ as rows and __________ as columns.

8.3 RECURSIVE DESCENT PARSING

A general approach to top down parsing is the recursive descent parsing. It involves backtracking which eventually leads to repeated scan of the input. A simple function is associated with each non-terminal of the grammar. The definition of the function corresponds to the right side of the grammar rule. When a non-terminal is contained in the right side, again the function corresponding to the non-terminal is called. And if terminals are encountered; then the next token is considered. The first terminal of each rule works as a catalyst to decide which rule must be used for parsing. Each succeeding symbol of the rule must be handled separately; terminals for reading the next input symbol and non-terminals by calling to the function or procedure associated with that terminal.

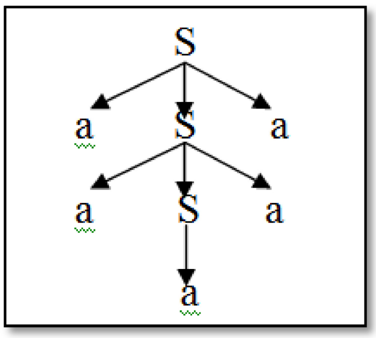
Parsing begins with the first symbol of the input string. Then the procedure for the starting non-terminal S is employed. This way the entire input string is read with one symbol at a time and the corresponding rule defining a non-terminal is applied. When control returns to the parse method after reading the entire input string, parsing is assumed to be completed. The two states accept and reject indicate whether the input string is in the language or not.

Let us consider the following grammar to understand the process better.

$$S \longrightarrow aSa \mid aS$$

$$S \longrightarrow a \mid b$$

Now, let us apply these rules to derive the string “aaaaaa”. Initially, the production S → aSa is applied as shown in figure 8.1. Now, there are two pointers: one points to the first symbol of the string; i.e. ‘a’. When expanded, the second pointer points to the first symbol of the right side of the production; i.e. ‘a’. They are matched for simiality. Since they match, both the pointers move



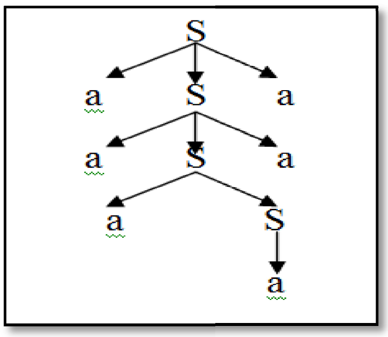


Fig: (a) **Fig: (b)**

Fig 8.1: Steps in a recursive descent parser

one step forward. Pointer 1 moves to the second token of the string whereas non-terminal S will be expanded. Now, procedure S is called recursively and the same production is applied. The second pointer now moves to ‘a’ of the tree. Both the pointers match and pointer 1 now moves to the third ‘a’ of the string. S is again expanded with the rule S → a. Pointer 2 moves to this ‘a’; eventually leading to a match and forwarding Pointer 1 to the fourth ‘a’ of the string. In the parse tree, pointer 2 moves to the fourth ‘a’ and they match again. This forwards both the pointers to refer to the fifth ‘a’s of the string as well as the tree. They match again and the first pointer moves to the sixth element of the string. But, pointer 2 halts and parsing is unsuccessful. Here, backtracking occurs to the last non-terminal being expanded. So, we will move back to S and try to apply another alternative S → aS. Finally, S is again expanded using production S → a. Now, this process produces the string we want to parse. This is how, recursive descent parsing backtacks to a faulty node and tries to decide whether a string belongs to a language or not.

Recursive descent parser result in inefficiencies as it may involve backtracking as well as ambiguities.

8.4 PREDICTIVE PARSING

Another form of top down parsing which does not require backtracking is the predictive parsing. The parser can effectively predict which grammar rule can be unambiguously applied on a particular input token. Predictive parsing can be obtained by carefully eliminating left recursion from the grammar and also left factoring it. The new grammar can be parsed by a recursive descent parser with no backtracking.

8.4.1 Non-recursive Predictive Parsing

Predictive parsing can be effectively implemented using a stack rather than recursive procedure calls. The main problem with predictive parsing is to determine the production to be applied for a non-terminal. The non-recursive predictive parser looks for a production to be applied in a parsing table. The model of a non-recursive predictive parser has been shown in the following figure.

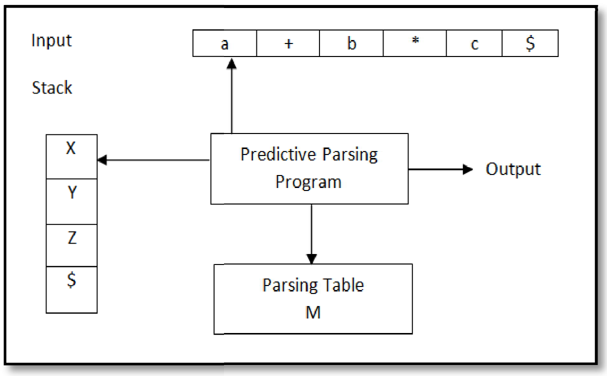


Fig 8.2 : Model of a non-recursive predictive parser

The table driven predictive parser contains an input buffer, a stack, a parsing table and an output stream. The bottom of the stack is indicated by a $. Similarly, the same indicates the right end marker of the input buffer. The parsing program controls the overall functioning of the parser. The program consults the parsing table and based on this, it determines the rule to be applied to a non-terminal. The parsing program considers the top of the stack X and the current input symbol a. The action of the parser depends on these two symbols ans is determined by as follows:

1. If X=a=$, the parser halts with a successful completion of parsing.
2. If X=a≠$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is non-terminal, the parser program consults entry M[X, a] of the parsing table M. The entry will be either an X-production or an error entry. If M[X, a] = { X → PQR}, the parser replaces X by RQP with P at the top. If M[X, a] = error, the parser calls for the error recovery routine.

The algorithm for implementing non-recursive predictive parsing has been shown below.

Algorithm 8.1: 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 error.

Method: The parser is initialized with $S as the only element on the stack, where S is the start symbol of G and w$ in the input buffer. It utilizes the predictive parsing table M to produce the parse tree for the input in the following manner.

```
Set ip to point to the first symbol of w$
repeat
let X be at the top of the stack and a be the symbol pointed to by ip
if X is a terminal or $ then
if X = a then
pop X off the stack and advance ip
else error()
    else
    if M[X, a] = X → $Y_1Y_2Y_3$.....$Y_k$ then begin
    pop X off the stack
    push $Y_kY_{k-1}$ .........$Y_2Y_1$ onto the stack with $Y_1$ on the top
```

```
output the production $X \rightarrow Y_1Y_2Y_3.....Y_k$
        end
        else error()
        until X = $       /* stack is empty */
```

Prior to applying the algorithm, we must learn to create the parsing table for the grammar. Construction of predictive parsing table follows an algorithm which we will discuss now. In the previous chapter, we had gained some ideas regarding the derivation of FIRST and FOLLOW sets. It is worth to mention at this point that these two sets will be applied during the construction of predictive parsing table.

Algorithm 8.2: Construction of predictive parsing table

Input: Grammar G

Output: Parsing table M

Method:

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

We shall apply this algorithm to derive the parsing table after eliminating left recursion from the grammar and further left factoring it.

Let us again consider the following grammar G derived in section 7.5 of the previous unit.

G: $E \longrightarrow T E'$

$E' \longrightarrow + T E' \mid €$

$T \longrightarrow F T'$

$T' \longrightarrow * F T' \mid €$

$F \longrightarrow (E) \mid id$

We know that,

FIRST(E)=FIRST(T)=FIRST(F)={ (, id }

FIRST(E′)={ +, € }

FIRST(T′)={ *, € }

FOLLOW(E)= FOLLOW(E′)={), $ }

FOLLOW(T)= FOLLOW(T′)={ +,), $ }

FOLLOW(F)= { *, +,), $ }

Now, let us apply algorithm 8.2 to generate the predictive parsing table. The input symbols are id, +, *, (,) and $. The non-terminals are E, E′, T, T′ and F.

Table 8.1: Predictive parsing table M
Source: Compilers: Principles, techniques and tools by Aho, Sethi and Ullman

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→ T E′			E→ T E′		
E′		E′→ +T E′			E′→ €	E′→ €
T	T→ F T′			T→ F T′		
T′		T′→ €	T′→ * F T′		T′→ €	T′→ €
F	F→ id			F→ (E)		

The FIRST sets of non-terminal E, T and F consists of set { (, id }. Therefore, the parse table entries M[E, id], M[T, id] and M[F, id] will contain the productions E→ T E′, T→ F T′ and F→id respectively. Similarly, the FIRST set of E′ consists of { +, € }. So, the production E′→ +T E′ will fill the entry M[E′, +] of the parsing table M. Now, since the set also contains €, therefore we shall consider the FOLLOW set of E′, which is {), $ }. The entries M[E′,)] and M[E′, $] will be filled up with E′→ €. The FIRST set of T′ consists of { *, € }. Therefore, M[T′, *] entry contains the production T′→ * F T′. Similarly, for € the FOLLOW(T′) set will be used which is {+,), $ }. So, the entries M[T′, +], M[T′,)] and M[T′, $] will be filled up with T′→ €.

Now, Let's implement algorithm 8.1 on input string "id + id * id" to trace the sequence of moves using this table. Initially, the input pointer points to the leftmost symbol of the string. We put a $ as the right end marker of the input string. Similarly, the stack will also contain $E; where $ indicates the bottom of the stack and E is the start symbol of the grammar.

Table 8.2: Moves made by predictive parsing table M on input id + id * id
Source: Compilers: Principles, techniques and tools by Aho, Sethi and Ullman

Stack	Input	Output
$E	id + id * id $	
$ E′ T	id + id * id $	E→ T E′
$ E′ T′ F	id + id * id $	T→ F T′
$ E′ T′ id	id + id * id $	F→ id
$ E′ T′	+ id * id $	
$ E′	+ id * id $	T′→ €
$ E′ T +	+ id * id $	E′→ +T E′
$ E′ T	id * id $	
$ E′ T′ F	id * id $	T→ F T′
$ E′ T′ id	id * id $	F→ id
$ E′ T′	* id $	
$ E′ T′ F *	* id $	T′→ * F T′
$ E′ T′ F	id $	
$ E′ T′ id	id $	F→ id
$ E′ T′	$	
$ E′	$	T′→ €
$	$	E′→ €

Parsing begins with E as the top of the stack and the input symbol id. The parser looks up the table to find the entry E→ T E′ and replaces E with E′T with T at the top. Next, the parser finds the entry M[T, id] as T→ F T′. T is now replaced with T′ F with F at the top of the stack. Now, entry M[F, id] is searched and found to be F→ id. The stack now cntains $ E′ T′ id with id at the top. The input pointer also points to id. Therefore, according to algorithm 8.1 the top of the stack must be popped and the input pointer is advanced to '+'. The stack now contains $ E′ T′. The parser again tries to find out the entry M[T′, +] which is T′→ €. This replaces T′ by € leaving the stack element $ E′. In this way, the parser makes different moves according to the algorithm. If finally we get an empty stack as well as input string having the only symbol '$', parsing comes to a halt after accepting the string.

Sometimes, it happens that the predictive parsing table M contains multiple entries. If grammar G1 is left recursive or ambiguous, it will have at least one multiply-defined entry. Let us consider the grammar

G1: S → iEtSS′ | a

S′ → eS | €

E → b

Now, we derive the FIRST set of G.

FIRST(S) = {a, i}

FIRST(S′) = {e, €}

FIRST(E) = {b}

Next, we derive the FOLLOW set of G.

FOLLOW(S) = {e, $}

FOLLOW(S′) = {e, $}

Table 8.3: Predictive parsing table M
Source: Compilers: Principles, techniques and tools by Aho, Sethi and Ullman

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S → a			S →		

				iEtSS′		
S′			S′ → eS S′ → €			S′ → €
E		E → b				

The table rows are represented by non-terminals S, S′ and E and the columns are represented by terminals a, b, e, i, t and $. According to algorithm 8.2, the parse table entries for S non-terminals correspond to FIRST(S). Similarly, for S′, the FIRST(S′) will be considered. Accordingly, production S′ → eS will be entered in M[S′, e]. Since, FIRST(S′) contains €, FOLLOW(S′) set will be considered, which is {e, $}. Therefore, S′ → € will be put in the entries M[S′, e] and M[S′, $]. Lastly, the entry M[E, b] will contain the production E → b. The entry M[S′, e] contains more than one entry. A grammar whose parsing table does not contain multiple-defined entries is termed as LL(1) grammar. A grammar G is LL(1) if and only if for any two distinct production A → α | β of G:

1. Both α and β should not derive strings beginning with terminal a.
2. At most one of both α and β derive the empty string €.
3. If $\beta \overset{*}{\Rightarrow}$ €, then α does not does not derive any string beginning with a terminal in FOLLOW(A).

Clearly, G1 is not an LL(1) grammar. Therefore, question arises what to do with a parsing table having multiple-entries. Of course, one immediate action that should be taken is to eliminate left recursion and then left factoring the grammar. However, some grammars can never produce an LL(1) parser. In general, there is no universal rule which can transform a multiple-defined entry of a grammar into a single valued one without affecting the language recognized by the parser.

CHECK YOUR PROGRESS- II

9. Recursive descent parsing is a bottom-up parsing. State whether true or false.
10. Recursive descent parsing does not involve backtracking. State whether true or false.
11. Why does a recursive descent parser result in inefficiencies?
12. A predictive parser involves backtracking. State whether true or false.
13. Predictive parsing can be effectively implemented using a ________rather than recursive procedure calls.
14. If grammar G is left recursive or ambiguous, it will have at least one ___________ entry.

8.5 SUMMING UP

- Derivation may take place in zero or more steps or in one or more steps. A sentential form is a combination of grammar symbols like terminals and non-terminals. During the process of derivation, the input symbols of the string to be parsed are scanned from left to right.
- Each step of string derivation can be represented during construction of a parse tree. Just like derivation, parse tree construction also begins at the start symbol or root. Root has no parents.
- A tree consists of one or more nodes. The head and the body of a production represent the parent and children respectively. Edges come out of the parent to its children.
- Grammars are more powerful notation than regular expression. They describe a language more precisely. A programming language construct that is described by a regular expression can also be described by a grammar. However, the reverse is not possible. Therefore, every regular expression is a context free grammar but not vice-versa.
- In top down parsing, construction of parse tree begins at the root and it expands until the leaf nodes generate a string of terminals. Parsing always begins with the starting non-

terminal. The parsing algorithm scans one symbol at a time and decides which production rules will be applied in order to derive a string.

- Top down parsing may or may not require backtracking. Backtracking parsers need repeated scanning of the input during parse tree construction. The two states of parsing are accept and reject which indicate whether the input string is in the language or not.
- LL(1) grammar is the subset of Context Free Grammars (CFGs) that can be parsed with any simple parsing algorithm. It can be parsed by considering only one non-terminal and the next token in the input stream. Scanning happens from left to right and produces the leftmost derivation of the input with one lookahead symbol at each step of parsing.
- The FIRST and FOLLOW sets are the two major functions that play important role in construction of the predictive parsing table. Prior to generating the FIRST and FOLLOW sets, ambiguities, left recursion and common left prefixes must be removed. Then, one can go for generating LL(1) parsing table.
- The recursive descent parsing involves backtracking which eventually leads to repeated scan of the input. A simple function is associated with each non-terminal of the grammar. The definition of the function corresponds to the right side of the grammar rule. Recursive descent parser results in inefficiencies as it may involve backtracking as well as ambiguities.
- The predictive parser can effectively predict which grammar rule can be unambiguously applied on a particular input token. Parsing can be obtained by carefully eliminating left recursion from the grammar and also left factoring it.
- Predictive parsing can be effectively implemented using a stack rather than recursive procedure calls. The main problem with predictive parsing is to determine the production to be applied to a non-terminal.
- A table driven predictive parser contains an input buffer, a stack, a parsing table and an output stream. . If a grammar is left recursive or ambiguous, it will have at least one multiply-defined entry in the table.

8.6 ANSWERS TO CHECK YOUR PROGRESS

1. In top down parsing, construction of parse tree begins at the root and it expands until the leaf nodes generate a string of terminals.
2. Backtracking
3. predictive parsing.
4. LL(1)
5. Ambiguous, left recursive
6. 1. Removal of ambiguity 2. Removal of left recursion and 3. Elimination of common left prefixes.
7. Tables
8. non-terminals, terminals
9. False
10. False
11. Recursive descent parser result in inefficiencies as it may involve backtracking as well as ambiguities.
12. False
13. stack
14. multiply-defined

8.7 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What do you mean by parsing? Explain its types in brief.
2. What is top-down parsing? Explain with an example.
3. What is LL(1) parsing? Discuss.
4. How does the recursive descent parser lead to inefficient parsing?
5. What is predictive parsing? Explain in brief.
6. Why are FIRST and FOLLOW sets required during parsing?

B. Long answer type questions.

1. Explain how top-down parsing is implemented.
2. Describe how LL(1) parsing is implemented.
3. What is recursive descent parsing? Describe how backtracking takes place in recursive descent parsing.
4. Describe the model of non-recursive predictive parsing.
5. How do you construct a predictive parsing table? Explain with an example.

6. How do you derive the FIRST and FOLLOW sets from a grammar? Also describe the steps to construct parsing table using these two sets.

8.8 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C* (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 9
BOTTOM UP PARSING

Unit Structure

9.0 INTRODUCTION

Bottom-up parsing is a strategy used by compilers to analyze the structure of programming language statements and construct a parse tree from a sequence of input tokens. This parsing technique starts from the leaves (the terminal symbols or tokens) and works its way up to the root of the parse tree. It builds the parse tree by identifying the lowest-level elements first and then applying production rules in reverse (from right to left) to integrate these elements into higher-level structures, until the entire input is reduced to the start symbol of the grammar.

9.1 TECHNIQUES USED IN BOTTOM UP PARSING

9.1.1 Shift-Reduce Parsing Technique

One of the most common bottom-up parsing methods is the Shift-Reduce parsing technique, which uses a stack to hold grammar symbols and a buffer for the input tokens. The parser shifts tokens from the input buffer to the stack and tries to match the items at the top of the stack with the right-hand side of grammar rules. When a match is found, the parser performs a reduce operation by replacing the items on the stack with the corresponding non-terminal symbol from the left-hand side of the rule.

- **Example:**

Consider a simple grammar for arithmetic expressions:

1. "E → E + T"
2. "E → T"
3. "T → T * F"
4. "T → F"
5. "F → (E)"
6. "F → id"

And an input expression to be parsed: “id + id * id”

The bottom-up parsing process for this expression could look something like this:

- **Initial State**:
  - Stack: "empty"
  - Input: "id + id * id$" ($ signifies the end of the input)

- **Shift "id":**
  - Stack: "id"
  - Input: "+ id * id$"

- **Reduce by rule 6 (F → id):**
  - Stack: "F"

- Input: "+ id * id$"

- Reduce by rule 4 (T → F):
  - Stack: "T"
  - Input: "+ id * id$"

- Reduce by rule 2 (E → T):
  - Stack: "E"
  - Input: "+ id * id$"

- Shift "+":
  - Stack: "E +"
  - Input: "id * id$"

- Shift "id":
  - Stack: "E + id"
  - Input: "* id$"

- Reduce by rule 6 (F → id):
  - Stack: "E + F"
  - Input: "* id$"

- Reduce by rule 4 (T → F):
  - Stack: "E + T"
  - Input: "* id$"

- Shift "*":
  - Stack: "E + T *"
  - Input: "id$"

- Shift "id":
  - Stack: "E + T * id"
  - Input: "$"

- Reduce by rule 6 (F → id):
  - Stack: "E + T * F"
  - Input: "$"

- Reduce by rule 3 (T → T * F):
  - Stack: "E + T"
  - Input: "$"

- Reduce by rule 1 (E → E + T):
  - Stack: "E"
  - Input: "$"

At this point, the entire input has been reduced to the start symbol "E", indicating that the input string is syntactically correct according to the given grammar. The parser has constructed the parse tree for the input expression from bottom to top, confirming its structure based on the grammar rules.

9.1.1 Operator Precedence Parsing

Operator precedence parser is kinds of shift reduce parsing method. Thus, Operator Precedence Parsing is a bottom-up parsing technique used to parse the input where operators have different precedences. Unlike other parsing methods, it doesn’t require a parse tree. Instead, it relies on a precedence table to make parsing decisions. The table defines the precedence and associativity of operators, allowing the parser to decide when to shift (read more input) and when to reduce (combine tokens into expressions based on the operators" precedence).

The main idea behind Operator Precedence Parsing is to handle infix expressions where operators have different levels of precedence and may have left-to-right (left associativity) or right-to-left (right associativity) evaluation order.

- **Example:**

Consider a simple expression grammar that includes addition ("+"), multiplication ("*"), and parentheses to alter precedence:

a) Operators: +, *
b) Operands: "id" (identifiers or numbers)
c) Parentheses: "(", ")"

Precedence and associativity rules:
a) * has higher precedence than +.
b) Both "*" and "+" are left associative.
c) Parentheses "(" and ")" can alter the normal precedence.

Let's parse the expression: "id + id * id"

First, construct the precedence table based on the grammar rules:

Operator	"<" (less precedence)	"=" (equal precedence)	">" (greater precedence)
"+"	"id", "("	"+"	"*", ")"
"*"	"id", "(", "+"	"*"	")"
"id"	"+", "*"		")", "+", "*"
"("	"id", "("		")"
")"	"+", "*"		")", "+", "*"

Now, let's parse the expression **"id + id * id":**

1. **Initial State:** Start with an empty stack and the input expression in front of you.

2. **Step 1:** Shift "id" onto the stack because there's no precedence relation on an empty stack.

3. **Step 2:** Upon seeing "+", compare it with "id" on the stack. According to the table, "id > +", so shift "+" onto the stack.

4. **Step 3:** Shift the next "id" because "+ < id".

5. **Step 4:** Upon seeing "*", compare it with "id" on the stack. Since "id < *", shift "*" onto the stack.

6. **Step 5:** Shift the last "id".

7. **Reduction Steps:** Now, you'll start reducing based on the precedence. The top of the stack is "id", which is greater in precedence than the operators "+" and "*" below it, indicating that these tokens can be combined according to the rules of precedence ("id * id" gets combined first due to higher precedence of "*" over "+", then the result is combined with the leading "id" via "+").

The process involves looking at the stack and the next input symbol, then deciding whether to shift (put the symbol on the stack) or reduce (apply a grammar rule to reduce the stack's top symbols to a non-terminal) based on the precedence relations.

This explanation simplifies the actual parsing steps and omits the construction of syntax trees or the detailed handling of end-of-input scenarios. Operator precedence parsing requires a careful design of the precedence table and clear rules for handling each operator and operand to ensure correct parsing and evaluation of expressions.

9.2 INTRODUCTION TO LR PARSING

LR parsing, which stands for Left-to-right, Rightmost derivation parsing, is a bottom-up method for analyzing the syntax of a given input string against a set of production rules in a context-free grammar. It builds a parse tree from the leaves (the input symbols) up to the root (the start symbol). This method is particularly efficient and widely used for parsing programming languages due to its ability to parse a vast class of grammars, known as LR grammars, which includes most programming languages.

9.2.1 Importance of LR Parsing

LR parsers have several key advantages that make them suitable for compiler design and syntax analysis in general:

1. **Efficiency**: They can parse many constructs in programming languages in linear time relative to the length of the input.
2. **Power:** LR parsing techniques can handle a wide range of grammars, including those that are not suitable for simpler parsers like recursive descent parsers without backtracking.
3. **Detecting Syntax Errors Early**: LR parsers can detect syntax errors as soon as it is mathematically possible to do so in a left-to-right scan of the input.
4. **Automatic Parser Generation**: Tools like YACC (Yet Another Compiler Compiler) and Bison can automatically generate LR parsers from a grammar specification, making the development of compilers and interpreters easier and less error-prone.

- **Example of LR Parsing**

Consider a simple grammar for arithmetic expressions:
E → E + T | T
T → T * F | F
F → (E) | id

Where "E" is an expression, "T" is a term, "F" is a factor, and "id" represents identifiers (like variable names or numbers).

To parse the input string "id + id * id" using LR parsing, we would proceed as follows:
1. Shift the first "id" onto the stack. The stack now contains "id".
2. Since "id" matches the right side of the production "F → id", we reduce it to "F". The stack now contains "F".
3. The "F" can be further reduced to "T" ("T → F"), and then to "E" ("E → T"). Now, the stack contains "E".
4. Next, we shift the "+" and the following "id" onto the stack, giving us a stack of "E + id".

5. The "id" is reduced to "F", then "T", and since we have "E + T" on the stack, we can reduce it to "E" using the rule "E → E + T".
6. Now, the stack once again contains "E". If this was the entire input, we would be done, and the input would be successfully parsed.

This process illustrates how an LR parser reads the input, shifts symbols onto a stack, and applies reductions according to the grammar’s production rules. The parser continues this process until the entire input is consumed and reduced to the start symbol, indicating that the input conforms to the grammar’s rules.

LR parsing’s ability to efficiently handle complex grammars with a systematic approach makes it a cornerstone of modern compiler design, allowing for robust, error-resistant parsers that can be automatically generated from a language’s grammar.

9.3 LR(0) AUTOMATON

LR(0) automaton is a foundational concept in the theory of LR parsing, which is used to recognize whether a string belongs to a certain grammar. It forms the basis for more complex LR parsing strategies, like SLR(1), LALR(1), and LR(1), which are widely used in compiler design for syntax analysis. An LR(0) automaton is constructed from the grammar of a programming language and is used to generate the parsing table that guides the parsing process.

9.3.1 Working of LR(0) Automaton

The LR(0) automaton is a state machine where each state represents a set of "items," and each item represents a possible position in the parsing process of a production rule from the grammar. An item in this context is a production rule with a dot indicating how much of the rule has been seen (or parsed) at any point in time.

9.3.1.1 Construction of LR(0) Automaton

1. Start Item: Begin with the start symbol of the grammar augmented with a new start rule (e.g., "S" → S" for start symbol "S") and a dot at the beginning (e.g., "S" → .S"). This constitutes the initial state of the automaton.
2. Closure Operation: For each state, apply the closure operation to include all items that could potentially be reached from the current items in the state. This means if an item has a dot before a non-terminal, add new items for each production of that non-terminal, with the dot at the beginning.
3. Transition Operation: For each state and each grammar symbol (terminal or non-terminal), create a transition to a new state by moving the dot past that symbol in all items where it appears directly before the symbol. This represents the parser recognizing that symbol and moving forward in the input.
4. Repeat: Apply the closure and transition operations until no new states can be added.

- **Example:**

Given a simple grammar:
S → A
A → aA | b

The LR(0) items would include:

- "S" → .S" (the augmented start rule)
- "S → .A"
- "A → .aA"
- "A → .b"

From these items, you would construct states and transitions based on where the dot can move given the input symbols ("a" or "b"). For instance, moving the dot over "A" in "S → .A" would lead to a new state with the item "S → A." indicating that an "A" has been fully parsed as expected by the rule for "S".

9.3.2 Importance of LR(0) Automaton

The LR(0) automaton is important because:

a) It provides a systematic way to construct parsing tables for LR(0) parsers, which can then be used to parse languages defined by LR(0) grammars.

b) It helps in understanding how parsers recognize language constructs and manage ambiguities or errors in syntax.

- It lays the groundwork for understanding more complex LR parsing methods that deal with a broader range of grammars by introducing concepts like lookaheads and merging states to handle conflicts.

However, LR(0) parsers are limited by their inability to handle grammars with certain kinds of ambiguity or lookahead requirements, which are overcome by SLR(1), LALR(1), and LR(1) parsers that build on the concept of the LR(0) automaton with additional features.

9.4 SLR PARSING TABLE

Simple LR (SLR) parsing is an enhancement of LR(0) parsing that aims to resolve some of the limitations associated with LR(0) parsers, particularly their inability to handle grammars with certain conflicts. SLR parsing uses a parsing table constructed from an LR(0) automaton but with an added mechanism for deciding when to reduce, based on lookahead tokens. This additional lookahead information allows SLR parsers to handle a wider range of grammars by reducing the number of conflicts (such as shift/reduce and reduce/reduce conflicts) encountered during parsing.

9.4.1 Components of the SLR Parsing Table

An SLR parsing table consists of two main parts:

1. Action Table: Determines the parser's actions (shift, reduce, accept, or error) based on the current state and the lookahead token from the input. The actions are:

- **Shift:** Move the parser to a new state, "shifting" the lookahead token onto the stack.
- **Reduce:** Apply a grammar rule to replace a sequence of symbols on the stack with the rule's left-hand side, moving the parser to a state that reflects this reduction.
- **Accept:** Indicate that the input string has been successfully parsed.
- **Error:** Indicate a syntax error.

2. Goto Table: Used after reductions to determine the next state based on the non-terminal that has just been introduced onto the stack by a reduction.

9.4.2 Construction of the SLR Parsing Table

To construct an SLR parsing table, follow these steps:

1. Construct the LR(0) Automaton: Begin by constructing the LR(0) automaton for the given grammar, as described previously.

2. Identify Follow Sets: For each non-terminal in the grammar, compute the "follow set," which is the set of terminals that can appear immediately to the right of that non-terminal in some "sentential form" of the grammar. The sentential form includes both complete sentences in the language and partial sentences that could be completed into full sentences.

3. Fill in the Action Table:

- For each state in the LR(0) automaton, if there is a transition on a terminal symbol, add a "shift" action to the corresponding cell in the action table.
- If a state includes an item indicating a completed rule (e.g., "A → α."), add a "reduce" action for that rule in all cells under terminals in the follow set of "A". If the item is for the augmented start rule (indicating the entire input has been parsed), add an "accept" action for the end-of-input marker.

4. Fill in the Goto Table: For each state that has a transition on a non-terminal, record the resulting state in the goto table.

- **Example:**

Given a grammar:

1. S → L=R
2. S → R
3. L → *R
4. L → id
5. R → L

Without delving into the specifics of constructing the entire LR(0) automaton and follow sets, the key concept here is how the action and goto tables are filled based on the automaton's states and the grammar's follow sets. For instance, if the follow set of "L" includes "=", and there's a state in the automaton with "L → *R.", indicating that rule 3 can be reduced, then in the action table, for that state, under the "=" column (because "=" is in the follow set of "L"), you would add a "reduce by rule 3" action.

9.4.3 Importance of SLR Parsing

SLR parsers are more powerful than LR(0) parsers because they can handle a broader class of grammars thanks to their use of follow sets to resolve conflicts. They strike a balance between the simplicity of LR(0) parsers and the power of more complex parsers like LALR(1) and LR(1), making them suitable for many practical parsing tasks. However, SLR parsing still has limitations and can encounter conflicts with certain grammars, leading to the development of even more sophisticated parsing techniques.

9.5 CHECK YOUR PROGRESS

1. What is bottom-up parsing?
   - A) A parsing technique that starts from the root of the parse tree and works its way down to the leaves.
   - B) A parsing technique that starts from the leaves of the parse tree and works its way up to the root.
   - C) A parsing technique that only uses recursive descent parsing methods.
   - D) A parsing technique that does not use any stack for parsing.

2. Which of the following is a common bottom-up parsing method?
   - A) Predictive parsing
   - B) Recursive descent parsing
   - C) Shift-Reduce parsing
   - D) Top-Down parsing

3. In the context of Shift-Reduce parsing, what does the "shift" operation do?
   - A) It replaces items on the stack with a non-terminal symbol.
   - B) It moves tokens from the input buffer to the stack.
   - C) It discards tokens from the input without processing.
   - D) It merges two adjacent non-terminals into a single terminal.

4. What is the purpose of the "reduce" operation in Shift-Reduce parsing?

- A) To add more tokens to the input buffer.
- B) To move tokens from the stack back to the input buffer.
- C) To replace items on the stack with the corresponding non-terminal symbol from the left-hand side of the grammar rule.
- D) To shift the focus of parsing from left to right in the input.

5. Based on the example given, which rule is applied first in the parsing process of the input "id + id * id"?

- A) E → E + T
- B) E → T
- C) T → T * F
- D) F → id

6. Which of the following statements best describes the final state of the stack when the input "id + id * id" is successfully parsed?

- A) The stack contains multiple instances of terminals and non-terminals mixed together.
- B) The stack is empty because all symbols have been processed.
- C) The stack contains just the start symbol of the grammar.
- D) The stack contains all the input tokens in reverse order.

7. How does bottom-up parsing confirm the syntactic structure of an input according to a given grammar?

- A) By matching input tokens with grammar rules from left to right.
- B) By applying grammar rules in reverse to reconstruct the input from the start symbol.
- C) By predicting which grammar rule to apply next based on lookahead tokens.

- D) By eliminating tokens that do not match any grammar rule.

8. What is the main purpose of Operator Precedence Parsing?
   - A) To evaluate arithmetic expressions based on operator precedence and associativity.
   - B) To directly execute arithmetic operations without parsing.
   - C) To parse programming languages entirely, including control structures.
   - D) To eliminate the need for a grammar in parsing expressions.

9. Which of the following is true about Operator Precedence Parsing?
   - A) It does not require a precedence table for parsing.
   - B) It only applies to left-associative operators.
   - C) It uses a precedence table to resolve conflicts between operators.
   - D) It treats all operators as having equal precedence.

10. In Operator Precedence Parsing, what does a "shift" action imply?
    - A) Immediate execution of an operation based on operator precedence.
    - B) Reading more input to decide the parsing action based on the precedence table.
    - C) Reduction of the current input without considering further tokens.
    - D) Ignoring operator precedence and associativity rules.

11. What role does the precedence table play in Operator Precedence Parsing?
    - A) It specifies the syntax of the programming language.

- B) It determines the order of operations based on the precedence and associativity of operators.
- C) It eliminates the need for operators in expressions.
- D) It provides a way to bypass grammar rules for faster parsing.

12. How does Operator Precedence Parsing handle the expression "a + b * c"?
- A) It groups "a + b" together first because addition comes first in the expression.
- B) It groups "b * c" together first because multiplication has higher precedence than addition.
- C) It treats addition and multiplication as having equal precedence.
- D) It requires additional input from the user to decide the order of operations.

9.6 ANSWERS TO CHECK YOUR PROGRESS

1. B) A parsing technique that starts from the leaves of the parse tree and works its way up to the root.
2. C) Shift-Reduce parsing
3. B) It moves tokens from the input buffer to the stack.
4. C) To replace items on the stack with the corresponding non-terminal symbol from the left-hand side of the grammar rule.
5. D) F → id
6. C) The stack contains just the start symbol of the grammar.
7. B) By applying grammar rules in reverse to reconstruct the input from the start symbol.
8. A) To evaluate arithmetic expressions based on operator precedence and associativity.
9. C) It uses a precedence table to resolve conflicts between operators.

10. B) Reading more input to decide the parsing action based on the precedence table.
11. B) It determines the order of operations based on the precedence and associativity of operators.
12. B) It groups "b * c" together first because multiplication has higher precedence than addition.

9.7 CANONICAL LR PARSING TABLE

A canonical LR (CLR) table is a tool used in compiler design to implement an LR(1) parser. An LR(1) parser is a type of bottom-up parser that reads input from left to right and constructs a rightmost derivation in reverse. It uses a deterministic finite automaton to handle parsing decisions based on lookahead symbols.

The canonical LR table consists of two main parts:

1. **Action Table**: Dictates the parsing actions (shift, reduce, accept, or error) to be taken.
2. **Goto Table**: Guides the parser on state transitions based on non-terminal symbols.

9.7.1 Construction of a Canonical LR Table

1. **Grammar:** Begin with context-free grammar. For example:

   S' → S
   S → CC
   C → cC | d

2. **Augmented Grammar:** Add an augmented start production `S' → S`.
3. **Item Sets:** Construct the LR(1) item sets (states) and the transitions between them. Each item set includes items of the form `[A → α•β, a]`, where `•` indicates the current position in the production, and `a` is the lookahead symbol.

4. **Action and Goto Tables:** Based on the item sets and transitions, fill in the action and goto tables.

- **Example:**

Consider the grammar:

S' → S
S → CC
C → cC
C → d

Step 1: Augmented Grammar

S' → S
S → CC
C → cC
C → d

Step 2: Construct LR(1) Item Sets

Item Set 0
S' → •S, $
S → •CC, $
C → •cC, c/d
C → •d, c/d

Item Set 1 (After shift on `S`)
S' → S•, $

Item Set 2 (After shift on `C`)
S → C•C, $
C → •cC, $
C → •d, $

Item Set 3 (After shift on `c`)

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

Item Set 4 (After shift on `d`)

C → d•, c/d

(Additional sets would be constructed similarly.)

Step 3: Construct Action and Goto Tables

The action table indicates whether to shift, reduce, or accept, based on the current state and input symbol. The goto table indicates state transitions based on non-terminal symbols.

Action Table:

State	c	d	$
0	S3	S4	
1			Acc
2	S3	S4	
3	S3	S4	
4	R4	R4	R4

Goto Table:

State	S	C
0	1	2
2		5
3		6

In the action table:

- `S` indicates shift followed by the state number.
- `R` indicates reduction followed by the production number.
- `Acc` indicates acceptance.

In the goto table:
- Entries indicate the state to transition to after a reduction.

- **Parsing Example**

To parse the input string `cd$`:

1. Initial State : (State 0, `cd$`)

2. Shift `c` : (State 0, `cd$`) → Shift to State 3 → (State 3, `d$`)
3. Shift `d` : (State 3, `d$`) → Shift to State 4 → (State 4, `$`)
4. Reduce `C → d` : (State 4, `$`) → Reduce to State 2 (from 0) → (State 2, `$`)
5. Shift `C` : (State 2, `$`) → Shift to State 5 → (State 5, `$`)
6. Reduce `C → cC` : (State 5, `$`) → Reduce to State 0 → (State 0, `$`)
7. Shift `C` : (State 0, `$`) → Shift to State 1 → (State 1, `$`)
8. Accept : (State 1, `$`)

By following these steps, the parser determines that `cd` is a valid string according to the grammar.

This process illustrates how the canonical LR table guides the parser through each decision to construct the parse tree or validate the input string.

9.8 LALR PARSING TABLE

A Look-Ahead LR (LALR) table is a type of parsing table used in LALR parsers, which are a simplified form of canonical LR parsers. The LALR parser combines states with identical cores (the same items without considering lookahead symbols) from the LR(1) parsing table, making it more memory efficient while still being powerful enough to parse many practical programming languages.

9.8.1 Construction of an LALR Table

Let's consider a simple grammar:

$$
\begin{aligned}
S' &\rightarrow S \\
S &\rightarrow CC \\
C &\rightarrow cC \\
C &\rightarrow d
\end{aligned}
$$

Step 1: Construct LR(1) Item Sets

First, create the LR(1) item sets for the given grammar. This step is the same as for constructing a canonical LR table.

- Item Set 0

S' → •S, $
S → •CC, $
C → •cC, c/d
C → •d, c/d

- Item Set 1 (After shift on `S`)

S' → S•, $

- Item Set 2 (After shift on `C`)

S → C•C, $
C → •cC, $
C → •d, $

- Item Set 3 (After shift on `c`)

C → c•C, c/d
C → •cC, c/d
C → •d, c/d

- Item Set 4 (After shift on `d`)

C → d•, c/d

Step 2: Merge LR(1) Item Sets with the Same Core

Combine item sets that have the same core (items without lookahead symbols).

Merged Item Sets
- Core of Set 2 and 3 :
- Set 2: {S → C•C, $}
- Set 3: {C → c•C, c/d}

- Merged: {S → C•C, $}, {C → c•C, c/d}

Step 3: Construct the LALR Action and Goto Tables

Based on the merged item sets, construct the action and goto tables.

Action Table:

State	c	d	$
0	S3	S4	
1			Acc
2	S3	S4	
3	S3	S4	
4	R4	R4	R4
5			R2

Goto Table:

State	S	C
0	1	2
2		5
3		6
5		6

- **Parsing Example**

To parse the input string \`cd$\`:

1. Initial State : (State 0, \`cd$\`)
2. Shift \`c\` : (State 0, \`cd$\`) → Shift to State 3 → (State 3, \`d$\`)
3. Shift \`d\` : (State 3, \`d$\`) → Shift to State 4 → (State 4, \`$\`)
4. Reduce \`C → d\` : (State 4, \`$\`) → Reduce to State 2 (from 0) → (State 2, \`$\`)
5. Shift \`C\` : (State 2, \`$\`) → Shift to State 5 → (State 5, \`$\`)
6. Reduce \`C → cC\` : (State 5, \`$\`) → Reduce to State 0 → (State 0, \`$\`)
7. Shift \`C\` : (State 0, \`$\`) → Shift to State 1 → (State 1, \`$\`)
8. Accept : (State 1, \`$\`)

By following these steps, the parser determines that `cd` is a valid string according to the grammar.

9.8.2 Advantages of LALR Parsers

1. **Efficiency:** LALR parsers are more memory-efficient than canonical LR parsers because they combine states with identical cores.
2. **Practicality:** LALR parsers are used in many parser generators (like Yacc and Bison) due to their balance between power and efficiency.

9.8.3 Comparison with LR(1) Parsers

1) **State Count:** LALR parsers have fewer states than LR(1) parsers because they merge states with the same core.
2) **Lookahead:** Both use lookahead symbols to make parsing decisions, but LALR parsers have simplified tables due to state merging.

LALR tables thus provide a practical and efficient approach to syntax analysis in compilers, making them a popular choice in real-world applications.

9.9 ERROR DETECTION AND RECOVERY

Error detection and recovery are crucial aspects of parser design, especially in compilers, to ensure that programs can be analysed and executed even if they contain syntax errors. Here's an explanation of error detection and recovery in the context of LALR parsers, along with various methods and examples.

9.9.1 Error Detection

Error detection involves identifying syntactical mistakes in the input string that do not conform to the grammar rules. In an LALR parser, errors are typically detected during the parsing process when no valid action is defined for a given state and input symbol combination.

9.9.2 Error Recovery

Once an error is detected, error recovery techniques are used to handle the error and continue parsing. This helps in providing meaningful error messages and allows the parser to process the remaining input, which is particularly useful in interactive development environments.

9.9.3 Methods of Error Recovery

1. Panic Mode Recovery
2. Phrase Level Recovery
3. Error Productions
4. Global Correction

1. Panic Mode Recovery

This method involves skipping input symbols until a synchronizing token (often a statement terminator like a semicolon) is found. The goal is to skip past the error and resume parsing from a known state.

Example: Suppose the grammar expects an assignment statement:

S → id = E ;
E → id | num

Input: id = num id = num ;

Error detected: After parsing id = num, the parser encounters \`id\` instead of the expected \`;\`.
Recovery: Skip tokens until a semicolon is found, then resume parsing.

Parser might skip `id = num` and start fresh from `;`, resulting in:
id = num ; (valid statement)

2. Phrase Level Recovery
In this method, the parser attempts to replace or insert a small number of tokens to correct the error and continue parsing. This involves local corrections and can provide better recovery in some cases.

Example: Consider the same grammar:

S → id = E ;
E → id | num

Input: `id = id num ;`

Error detected: The parser expects a single token after `=`, but finds `id num`.

Recovery: Insert a semicolon or remove extra tokens to correct the input.

Correction could be:

id = id ; num ;

or

id = num ;

3. Error Productions
This method involves adding special error productions to the grammar. When these productions are used, the parser can provide specific error messages and attempt to continue parsing.

Example: Extend the grammar with an error production:

S → id = E ;
E → id | num | error

Input: `id = + ;`

Error detected: `+` is not a valid token for `E`.

Recovery: The parser uses the `error` production, outputs an error message, and skips to the next statement.

Error: Invalid token `+`.
Resuming parsing...

4. Global Correction

This method attempts to make the minimal number of changes (insertions, deletions, substitutions) to the entire input to make it syntactically correct. It's more sophisticated and computationally expensive, typically used in advanced development environments.

Example: For the grammar:

S → id = E;
E → id | num

Input: `id = num id = num;`
Error detected: The sequence `id = num id = num ;` is incorrect due to missing semicolons.

Recovery: The parser suggests the minimal changes:

id = num ; id = num ;

9.9.4 Summary

Error detection and recovery are essential to robust parsing. LALR parsers can use several strategies:

1. Panic Mode Recovery: Quickly skip to a synchronising token.
2. Phrase Level Recovery: Make local corrections to continue parsing.
3. Error Productions: Extend the grammar to handle common errors.
4. Global Correction: Make minimal global changes to correct the input.

These techniques allow parsers to handle errors gracefully, providing meaningful feedback to developers and allowing the continuation of the parsing process even in the presence of syntax errors.

9.10 CHECK YOUR PROGRESS

1. Which part of the canonical LR table dictates the parsing actions to be taken?

A. Goto Table
B. Action Table
C. Item Sets
D. Transition Table

2. What does the `R` symbol in the Action Table indicate?

A. Shift
B. Reduce
C. Accept
D. Error

3. In the context of LR parsing, what does the `•` symbol represented in an item?

A. The start of a production
B. The end of a production
C. The current position in the production
D. The lookahead symbol

4. What is the purpose of adding an augmented start production in the grammar?

A. To reduce the complexity of the grammar
B. To simplify the parsing table
C. To clearly define the start state for parsing
D. To remove ambiguities in the grammar

5. Consider the grammar: `S → CC`, `C → cC`, `C → d`. Which of the following is NOT a valid item set for this grammar?

A. `S' → •S, $`

B. `S → C•C, $`

C. `C → d•, c/d`

D. `C → c•, $`

6. In the canonical LR parsing table, what does the `Acc` symbol indicate?

A. Shift

B. Reduce

C. Accept

D. Error

7. What is the lookahead symbol used for in an LR(1) item?

A. To indicate the current position in the production

B. To determine the next action in the parsing process

C. To mark the end of a production

D. To define state transitions

8. *Item Set:*

S' → •S, $

S → •CC, $

C → •cC, c/d

C → •d, c/d

Given the following item set, which transition is NOT possible?

A. Shift on `c`

B. Shift on `d`

C. Reduce on `C`

D. Accept on `$`

9. Which of the following states would indicate the final acceptance in an LR(1) parser?

A. State 0

B. State 1

C. State 3

D. State 4

10. In the goto table, which symbols are used to guide state transitions?

A. Terminal symbols
B. Non-terminal symbols
C. Both terminal and non-terminal symbols
D. Lookahead symbols

11. What does LALR stand for in the context of parsers?
A. Look-Ahead Left-to-Right
B. Look-Ahead LR
C. Look-Ahead Right-to-Left
D. Look-Ahead Recursive

12. What is the main advantage of LALR parsers compared to canonical LR parsers?
A. They are faster
B. They have fewer states
C. They are easier to implement
D. They support more complex grammars

13. Which part of the LALR parsing table is used to guide state transitions based on non-terminal symbols?
A. Action Table
B. Goto Table
C. Lookahead Symbols
D. Core Table

14. In an LALR parser, what does the merging of LR(1) item sets depend on?
A. Identical lookahead symbols
B. Identical core items
C. Identical action entries
D. Identical goto entries

15. Consider the grammar: `S → CC`, `C → cC`, `C → d`. What will be the merged item set for the core items `{C → c•C}` and `{C → c•C, $}`?
A. `{C → c•C, c}`
B. `{C → c•C, d}`
C. `{C → c•C, c/d}`
D. `{C → c•C, c/d/$}`

16. Which parser generator commonly uses LALR parsing tables due to their balance between power and efficiency?

A. ANTLR

B. Yacc

C. Bison

D. Both B and C

17. In the context of LALR parsers, what does the `Acc` symbol in the action table signify?

A. Accept the input string

B. Shift the input symbol

C. Reduce the production

D. Transition to a new state

18. What is the result of merging LR(1) item sets with identical cores?

A. Increased parsing speed

B. Simplified grammar rules

C. Reduced number of states

D. Enhanced lookahead capability

19. In the given example, which state in the LALR table indicates a reduction by the production `C → d`?

A. State 0

B. State 3

C. State 4

D. State 5

20. Which part of the LALR table helps the parser determine the next action to take based on the current state and input symbol?

A. Goto Table

B. Action Table

C. Core Table

D. Lookahead Table

9.11 ANSWERS TO CHECK YOUR PROGRESS

1. B. Action Table
2. B. Reduce
3. C. The current position in the production
4. C. To clearly define the start state for parsing
5. D. `C → c•, $`

6. C. Accept
7. B. To determine the next action in the parsing process
8. D. Accept on `$`
9. B. State 1
10. B. Non-terminal symbols
11. B. Look-Ahead LR
12. B. They have fewer states
13. B. Goto Table
14. B. Identical core items
15. C. `{C → c•C, c/d}`
16. D. Both B and C
17. A. Accept the input string
18. C. Reduced number of states
19. C. State 4
20. B. Action Table

9.12 REFERNCES AND SUGGESTED READINGS

1. Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
2. "Introduction to Automata Theory, Languages, and Computation" by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman
3. "Parsing Techniques: A Practical Guide" by Dick Grune and Ceriel J.H. Jacobs

9.13 ADDITIONAL RESOURCES

Websites:

- **"Compiler Explorer"**
  Compiler Explorer
  An interactive compiler with real-time parsing and code generation visualization.

Tools:

- **JFLAP**
  JFLAP - Interactive Learning Tool for Automata Theory and Formal Languages
  A tool that helps visualize different parsing algorithms and automata.

UNIT: 10
YACC

Unit Structure:

10.0 INTRODUCTION

YACC, which stands for "Yet Another Compiler-Compiler," is a powerful tool used for generating parsers, essential components in the compiler construction process.It was developed at the beginning of the 1970s by Stephen C. Johnson for the Unix operating system. A parser is responsible for converting source code into a structure that the machine can understand and execute. YACC is particularly recognized for generating LALR (1) parsers (Look-Ahead Left-to-Right parsers with 1 token of look-ahead), which are efficient in both memory usage and execution time. These parsers are widely used in compiling programming languages due to their balance of simplicity and performance.

10.1 UNIT OBJECTIVES

- Analyze and construct context-free grammars for language specifications.

- Understand the fundamental concepts of compiler design, focusing on lexical analysis and syntax analysis.
- Learn how to use Lex for lexical analysis and Yacc for syntax parsing to build a basic compiler.
- Explore the process of defining grammar rules and lexical patterns for different types of programming constructs.
- Develop the ability to integrate Lex and Yacc to build a functioning compiler that can process input code.

10.2 YACC BASICS AND WORKFLOW

YACC operates by taking a formal grammar as input, which specifies the syntax rules of a given programming language. These grammar rules are typically written in Context Free Grammar, a notation used to describe the structure of languages. Each rule may be associated with small snippets of C code, known as actions. When YACC processes the grammar, it generates a C program that serves as the parser. This parser reads the source code and analyses it based on the grammar, facilitating the translation of code into an internal representation. The parser asks the lexical analyzer (often generated by a tool like Lex) for the next token, and continues parsing based on that token. When a grammar rule is recognized, YACC performs a reduction, meaning it replaces a sequence of tokens with the corresponding nonterminal and executes the C code attached to that rule.

Before discussing how a YACC and a LEX program works, let us discuss a Context-Free Grammar (CFG) for arithmetic expression. A Context-Free Grammar for arithmetic expressions is a set of production rules that describe the syntax of valid arithmetic expressions. In this case, the CFG will be capable of generating expressions involving numbers, addition, subtraction, multiplication, division, and parentheses. Below is an example of such a CFG:

Terminals: These are the basic symbols or tokens in the grammar, such as numbers and arithmetic operators. {+, -, *, /, (,), ID,NUMBERS, NL}

Nonterminal: These are syntactic categories or variables that represent different types of expressions. {exp, stmt}

Start Symbol: The starting point of the grammar. Here, exp is the start symbol, representing a complete arithmetic expression.

Production Rules: These define how each nonterminal can be expanded using other terminals or nonterminal.

stmt->exp NL

exp->exp + exp | exp–exp | exp * exp | exp / exp | (exp) | ID | NUMBER

Using the above mentioned CFG we can parse an arithmetic expression involving numbers, addition, subtraction, multiplication, division, and parentheses. Now we can implement the above grammar in YACC to parse a given string. To implement the parser in YACC we also need a lexical analyzer which should generate tokens for the parser. If we notice the CFG we can see that to parse a given string (expression) the lexical analyzer (the LEX program) should able to return three tokens- NUMBER, ID, and NL which represent number, identifier and new lines respectively.

To understand the working of LEX and YACC let us consider the following LEX and YACC program-

```
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
```

```
stmt: exp NL { printf("Valid Expression"); exit(0);}
;
exp: exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| '(' exp ')'
| ID
| NUMBER
;
%%
int yyerror(char *msg)
{
printf("Invalid Expression\n");
exit(0);
}
main ()
{
printf("Enter the expression\n");
yyparse();
}
```

Fig. 10.1: A sample YACC program to heck the syntax of a simple expression.

```
%{
#include "y.tab.h"
%}
%%
[0-9]+    {return NUMBER; }
[a-zA-Z][a-zA-Z0-9_]*{ return ID; }
\n{return NL;}
.{return yytext[0]; }
%%
```

Fig. 10.2: The LEX program to to generate tokens for the YACC program.

In the above examples the LEX program is the lexical analyzer and the YACC program is the syntax analyzer.

Now let us discuss how these two programs works together-

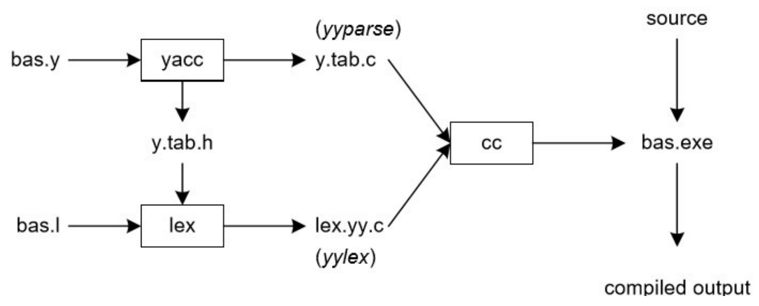


Figure 3: How a YACC program works with a LEX program.

As mention in Figure 3, let us name the YAAC program as **bas.y** and the LEX program as bas.l. YACC processes the grammar descriptions provided in bas.y and generates a syntax analyzer (parser). A YACC program has three sections. Each section is separated by %% symbol. As shown in Fig. 10.1, the declaration section (Line number 1, 2 and 3) declares different tokens. Based on these declarations the y.tab.h file is created. % left, defines how YACC will solve repetition of operators in case you have. i.e. it specifies the associativity of an operator. The associativity of an operation determines which of two operations of the same precedence level is carried out first.The second section of a YACC program contains the grammar description. In the above YACC program from line number 5 to 14 contains the grammar representation. The third section of a YACC program is the routine section.

The default name of the parser is y.tab.c, which is C source file. Asshown in Figure 3, it will also generate a header file called y.tab.h which contains definitions for tokens declared in bas.y file. In our case tokens are NUMBER, ID, and NL (line number 1 of Fig 10. 1). LEX includes the y.tab.h file, reads the pattern descriptions (line number 5, 6 and 7 in Fig. 10.2) from bas.l, and generates a lexical analyzer, which contains the yylex function, and stores it in another C source file lex.yy.c.

Afterward, both the lexical analyser and parser are compiled and linked to produce the executable file. In the program's main function, yyparse() is invoked to run the compiler which is present in y.tab.c. During this process, yyparse() automatically calls yylex() to retrieve each token. Table 10.1 shows how to compile and execute a parser and a lexical analyser written in YACC and LEX.

Steps	**Command**	**Action**
Step 1	yacc –d bas.y	create y.tab.h andy.tab.c
Step 2	lexbas.l	create lex.yy.c
Step 3	cc lex.yy.cy.tab.c –o exe	compile/link and create an executable file
Step 4	./exe	execute

Table 10.1: Steps to execute the parser

10.3 CONCLUSION

In this unit, we explored the fundamental aspects of compiler design with a focus on syntax analysis using YACC programing. We learned how to define tokens and grammar rules, builda basic parser, and integrate lexical and syntax analysis processes. Through this approach, we observed how a compiler translates source code into a structured format that a machine can process. This foundation provides essential skills for developing more advanced compiler components, preparing us for further exploration into semantic analysis, code generation, and optimization in subsequent chapters.

10.4 CHECK YOUR PROGRESS

1. What does YACC stand for?
   A) Yet Another Compiler-Compiler
   B) You Always Code Carefully
   C) Your Algorithm Code Compiler
   D) Yield And Compute Code

2. What is the primary function of YACC in compiler construction?
   A) To perform lexical analysis
   B) To generate machine code
   C) To create parsers from grammar specifications
   D) To optimize assembly code

3. In YACC, what does the yyparse() function do?
   A) It performs lexical analysis
   B) It initializes the compiler
   C) It optimizes the code generated by the parser
   D) It parses input according to the grammar defined

4. What type of parsers does YACC generate by default?
   A) LR(1) parsers
   B) LL(1) parsers
   C) LALR(1) parsers
   D) Recursive descent parsers

5. What option is used in YACC to generate a header file with token definitions?
   A) -l
   B) -o
   C) -d
   D) -g

6. In YACC, how are errors handled during parsing?
   A) Using the yyfail() function
   B) Using the yyerror() function
   C) Using the yyexit() function
   D) Using the yyrecover() function

7. Which of the following is used to define grammar in a YACC file?
   A) Regular Expressions
   B) Context-Free Grammar (CFG)
   C) Pushdown Automaton
   D) None of the above

8. Which of the following file extensions is typically associated with YACC grammar files?
   A) .lex
   B) .l
   C) .y
   D) .c

9. In YACC, what are actions associated with a grammar rule written in?
   A) C Code
   B) Assembly code
   C) Java code
   D) Python code

10. Which of the following functions does YACC call to retrieve tokens from the lexical analyzer?
   A) yyparse()
   B) yylex()
   C) yyerror()
   D) main()

10.5 ANSWERS TO CHECK YOUR PROGRESS

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
A	C	D	C	C	B	B	C	A	B

10.6 SHORT ANSWER TYPE QUESTIONS

1. What is YACC, and what does it stand for?
2. What type of parsers does YACC generate?
3. Explain the purpose of a yyparse() function in YACC.
4. Define the yylex() function and its role in YACC programs.
5. Explain the role of yyerror() in YACC.

10.7 LONG ANSWER TYPE QUESTIONS

1. Describe the workflow of a YACC program, from grammar specification to generating a parser.
2. Explain the structure of a YACC input file and describe each section in detail.
3. What is the difference between shift and reduce actions in a YACC-generated parser?
4. Discuss the integration of Lex and YACC, highlighting how they work together in a compiler.

5. Describe a real-world application where YACC could be effectively utilized and explain why.

10.8 REFRENCES AND SUGGESTED READINGS

1. Das, Vinu V. Compiler Design Using Flex and Yacc. PHI.

×××

BLOCK- II

UNIT 11: SYMBOL TABLE MANAGEMENT

Unit Structure

11.0 INTRODUCTION

In programming languages, identifiers play an important role as they refer to names of the variables, arrays and procedures. It is very essential for a compiler to record the information about the attributes of an identifier used in a source program. Usually, the attributes provide information about the type, scope and storage allocated to an identifier. In case of procedure names, the attributes may be like number and type of its arguments, method of passing each argument and the return type of the procedure. Such information is essential for transforming a program written in source language construct into a target language equivalent. The analysis phase of the compiler collects these information and the later phases will use them to generate the target code. During scanning or lexical analysis phase, lexemes forming identifiers are found out and saved as symbol table entry. Later phases of the compiler would extract more information and add to the table so that proper target code corresponding to the source code could be generated.

11.1 UNIT OBJECTIVE

After going through this unit, you will be able to:

- Define type checking
- Understand the basic functionalities of symbol table
- Know the prerequisite features of symbol table
- Describe the structure of a symbol table
- Explain an overview of the type checking system
- Understand the B-Minor type checker

11.2 SYMBOL TABLE

In programming, we frequently perform type checking. This leads to the need of identifying the type of the identifier beforehand. A variable could be a local one, global one or parameters to functions. The problem of identifying the scope of a variable is solved by the technique called name resolution. It refers to a table called the symbol table which contains the attributes of each variable.

A symbol table is a large data structure which is indexed by a symbol name or lexeme. The lexical analyzer works in a manner similar to pattern recognition. It tries to identify the tokens as well as their associated lexemes just by scanning the statements. During this phase, the symbol table is created. The same identifier may be declared in different locations or procedures. All these instances must be recorded in the symbol table. This can be accomplished by setting up different symbol tables for each block or by keeping pointers to blocks within a single symbol table. A simple way to represent symbol table is in terms of array of structures where each structure represents a record of an identifier. Type definition and declarations of constant may be found in them.

11.2.1 Symbol Table Requirements

A computer program basically consists of three major parts: declarations, expressions and statements. The output is produced by separate logical sections of the input program. Similarly, a subroutine is always prefixed by declaration and its parameter list. Then the body of the subroutine comes, which contains the statements and expressions. And finally, the code to return ends the subroutine.

The symbol table is like a database that contains records of each subroutine or variables declared in a program. It is indexed by the key field- generally the subroutine or variable name. Apart from this, the structure also contains fields like numeric value assigned to each symbol as well as symbol's type or the subroutine's return value. The declaration process enters records in the database and also is deleted from the same when the scoping rules determine that the object is out of scope and is no longer referenced. For example, the local variables declared in C-programs, which are out of scope are deleted once the compiler finishes working on it.

The symbol table communicates with other phases of compiler as well. At the very basic, it interacts with the lexical analysis phase specially when type definition and constant declarations are encountered. A ***typedef*** creates a symbol table entry for a new type like other variable names. The type definitions are indicated by setting a bit in the symbol table entry. There are some characteristics that a symbol table is required to possess for an efficient compilation process to proceed.

a. **Speed**: Table look-up must be as fast as possible. This is because; the symbol table is accessed everytime an identifier is referenced. The entire table must be in main memory when compilation proceeds. This approach has one major limitation on the maximum memory allotted to the symbol table. The input size of the program is limited so that the symbol table can accommodate records corresponding to each identifier declared within the program.
b. **Ease of maintenance**: A very basic criteria required in a symbol table is that- it should be easy to maintain. The functions must be organized in such a manner that anybody other than the compiler writer should be able to maintain the program.
c. **Flexibility**: The symbol table must be able to represent variables of arbitrary type. For example, the C-programming language does not impose any limitations on variable declarations. Moreover, the symbol table must be able to scale; i.e., new records must be inserted easily as and when new identifiers are encountered.
d. **Support of redundant entries**: In most programming languages, there is a provision of declaring the same variable name at different places of the program. This is possible, when the same variable name has different scopes. In-spite of having same name, they are treated as different variables. Correspondingly, the entries for each of these variables are also considered as distinct. Their scopes determine which variable is active at one moment. This is termed as shadowing as the active variable shadows the inactive one.

e. **Ease of deletion**: The symbol table manager must be able to efficiently delete the local variables declared inside a block. The deletion must take place in an arbitrary manner without having to look each element separately.

The symbol table must consist of two basic layers. The outer layer is termed as the maintenance layer. It deals with functions such as creating data structures for specific symbols, inserting such structures into the table and managing the table. This layer also performs deletion operation in order to delete structures of symbols. The insertion and deletion operations can be performed by executing low level subroutine call. There is also an inner layer which handles the actual table maintenance tasks at the physical level; like inserting new entries in the table, searching them as well as deleting them.

11.2.2 Symbol Table Structure

As symbol table contains the information about each and every variable or subroutine declared in a program. They must be represented in a manner similar to keeping records in a database table. For that purpose, ***structure*** is used. Each entry in the table is represented by the statement **struct symbol** which creates a structure for the symbol or variable. The symbol structure would be as follows:

```
struct symbol   typedef enum
{               {
symbol_t kind;
SYMBOL_LOCAL,
struct type *type;
SYMBOL_PARAM,
char *name;
SYMBOL_GLOBAL
int which;      } symbol_t;
};
```

(Source: Introduction to Compilers and Language Design by Prof. Douglas Thain)

The kind field indicates whether the variable is local, global or parameter to a function. The type field is a pointer to the type structure indicating the type of the variable. The name of the variable is indicated by the name field and which field indicates the ordinal positions of the variables and parameters.

It may be required to support multiple declarations for the same identifier declared in a program. Let us consider an example and see how symbol tables can be constructed.

```
int x;
void proc(int m)
{
 float x, y;
 {
        int a, b;
 }
}
 int func(int n)
{
      bool t;
}
```

The program uses two functions: proc() and fun(), each having own parameters. The symbol table representation would be as follows:

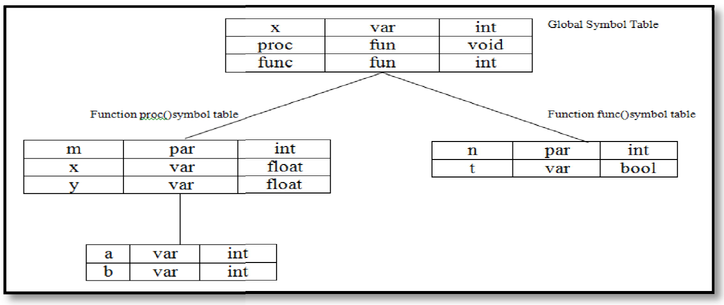


Fig. 11.1: Organization of symbol table

A suitable symbol structure is created for each variable declaration and entered into the symbol table. Therefore, in other words, a symbol table can be regarded as the mapping between the variable name and its symbol structure. It is already mentioned that multiple declarations with the same variable name is possible in most programming languages. But, each declaration must have distinct scope. The scopes might be global scope, scope for function parameters and local scopes and also nested scopes. These multiple definitions are organized as a stack of hash tables. This technique allows a variable with multiple scopes to exist by mapping each hash table to their corresponding symbols. A push operation enters a hash table everytime a new scope is encountered and a pop operation deletes the table once its scope is left.

CHECK YOUR PROGRESS- I

1. The problem of identifying the scope of a variable is solved by the technique called _________.
2. A symbol table contains _______ of _______.
3. A symbol table is a data structure indexed by a ______.
4. A simple way to represent symbol table is in terms of_________.
5. In a symbol table, each structure represents a record of an identifier. State whether true or false.
6. A symbol table must have the ability to_________.
7. A computer program basically consists of three major parts: ___, ____ and______.
8. The symbol table must consist of two basic layers: ________level and _______ level.

11.3 TYPE CHECKING

An important component of compilation process is finding the meaning of a program or what the program actually does. This is semantic analysis where a tree is constructed to perform efficient generation of codes. During semantic analysis phase, a considerable amount of time is spent in doing type checking. Different programming languages have different approaches of doing type checking. It is possible to make errors in a language with weaker type checking system. This in turn allows error detection at compile time.

Before going into type checking, we must determine the type of each identifier of an expression. Here, the role of symbol table comes into play. An identifier of an expression may refer to a local or global variable or a function parameter. The definition of identifiers can be found in symbol table and we refer to the table whenever we need to check the correctness of codes. Here, each variable is accessed with their corresponding type definitions. This process is termed as name resolution. Once name resolution completes, we are ready with all information required for doing type checking. As the correctness of codes is verified by the semantic analysis phase, the syntax tree must be referenced until this phase is over. We now have the type information of each identifier. We now combine them in order to compute the type information of complex expressions. Some standard conversion rules must be followed for finding the type information. If the expressions do not conform to the standard rules, there is a great chance of occurrence of errors.

Apart from gathering type information, semantic analysis also involves in checking other forms of correctness of codes. These may involve checking whether each operator has correct operands. For instance, the binary arithmetic operator between an integer and real is non-permissible. In that case, there is a need to convert integer into real. Similarly, the compiler reports an error every time a real number refers to an array index. Examining array limits, bad pointer traversal or examining flow of control also fall under the functionalities of semantic analysis phase.

11.3.1 Overview of Type Systems

Most programming languages allow codes to be written according to the specification defined for the language. They allow variables to be defined in terms of types. The type information describes whether the variable is an integer, floating point number, boolean, string or a pointer. The type information is associated with the storage space required to store a variable. These basic building blocks form the basis of creating more complex type variants such as structures and enumerations.

There are several purposes that the type system of a language must provide:

- **Correctness:** The type information of a compiler attempts to find errors and warnings occurring in a program. A good type system attempts to report errors during compile time rather than run time. As already mentioned, an array cannot be referenced using real numbers.
- **Performance:** Performance is an important component of a type system. It attempts to find the type information of a variable and tries to handle the code efficiently. Like for instance, if the type system finds that the value assigned to a variable is a constant, then it can be loaded on a register and used multiple times rather than to access it from memory every time it is required. This in turn increases the efficiency of the compiler.
- **Expressiveness:** A programming language is considered to be expressive if it can represent a variety of ideas to a greater extent. An expressive language can express solutions to problems defined in a particular domain. A piece of code is expressive if it does not take into account the facts derived during type checking. For example, in C-programming

language, the printf() function does not require to know whether it should print an integer, float, boolean or string. Automatically, it prints the value after inferring the type information from an expression.

11.3.2 Designing a Type System

The type system of a programming language corresponds to describing its primitive types, compound types and rules for assigning types to variables as well as converting between different types. The primitive or atomic data types refer to the simple types assigned to the variables. These simple structures may be integers, floating point numbers, boolean and so on. As mentioned above, these primitive data types also describe the range of values occupied by them.

The primitive data types may be implemented to form new data types termed as user defined data types like structure, union, enumeration and typedef in C – programming language.

Structure is the user defined data type that allows us to combine the primitive data types in order to generate more complex aggregations. It groups together the atomic data types as a single unit. Such compositions are useful for presenting the information of entities as records of related data items. Each data present in a structure is termed as member of the structure. Structures are always defined using the keyword ***struct***. For example, let us consider the **Student** structure, which represents the data related to a student entity.

```
struct Student
        {
        int roll_no;
        char name[40];
        char class[10];
```

```
                char section;
                }
        struct Student S;
```

We can create variables of Student structure by using the statement **struct Student S.** This creates a variable S of structure Student which possesses the same data members that Student has. The items are accessed by using a dot (".") operator between the structure variable and the member. For example, S.roll_no accesses the Roll no of a student. The storage space allocated to a structure is derived by adding the memory allotted to each member of the structure.

We can also write a structure declaration using the ***typedef*** keyword. Therefore, the above declaration can also be written as below:

```
typedef struct Student
        {
        int roll_no;
        char name[40];
        char class[10];
        char section;
        } S;
```

Union is similar to a structure. It also can possess elements of different data types. Just like structures begin with the keyword **struct**; unions also begin with the keyword **union**. But, the difference between the two can be understood when it comes to allocation of storage. In a union, only one member variable can be accessed at one time. Every time a new variable initialized, it overwrites the older ones. Therefore, the same previous declaration of structure can be used to declare using unions also.

```
typedef union Student
        {
        int roll_no;
        char name[40];
        char class[10];
```

```
                        char section;
                        } S;
```

At one given time, only one member can be accessed. This is because; the amount of memory allocated to a union is the amount of highest number of bytes required to store one member. Here, the string name requires the highest number of storage, i.e. 40 bytes.

The **enum** keyword is used to create an enumerated data type. It creates of named integral constants. It helps to assign constants to names so that the program becomes easier to understand and maintain. It is always declared using the keyword enum. If we do not assign values explicitly to the members of enum, the compiler automatically assigns values that start with 0. For example, let us consider the following enumeration code.

```
        enum week {Sunday=1, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday};
        int main()
          {
          enum  week today;
          today = Thursday;
          printf("Day    %d",today);
        }
```

The output of this code is 5. Here, the field Sunday is assigned with value 1. Automatically, Tuesday will be having value 2 and so on.

C has a special keyword *typedef* which specifies a new type of variable. It creates an alias of an already existing data type. This allows making assignments between types. Application of typedef has been seen in the previous examples of structures and unions, where we created variables of the same.

11.4 B-MINOR TYPE CHECKER

A language processor does semantic analysis by performing error checking. This error checking is typically based on type mismatching as well as misuse of reserved keywords. Once scanning and parsing is over, an Abstract Syntax Tree (AST) is created. AST is then traversed in order to find and report the type errors.

B- Minor is a static type checker for a simple language like C. It is strict type checker which tries to check two major classes of type errors: 1. When the type checker cannot determine the type of the expression and 2. When an expression's type does not match with the types of the assigned variables. It is a safe as well as explicit type checker. These properties have enabled the type checker to detect a large number of type errors. B-minor contains the primitive data types- integer, char, boolean, string and void. Two compound types array and function are also followed by B-Minor. There are some type rules that B-Minor follows:

- Value should be assigned to variables of the same type. Similarly, function parameters can only accept values of the same type.
- The return type of a function must be same with the return statement.
- While performing binary operation, the operands must be of same type.
- The equality operators == and != must return boolean. Apart from this they may operate on any data type except array, function or void.
- The comparison operators <, >, <= and >= also return boolean. But, they must be applied on integers.
- The operators !, && and | | operate on boolean and also return boolean.
- The arithmetic operators +, -, *, /, %, ++, -- always operate on integers and return integer.

B-Minor supports static type checking, functions, expressions and basic control flow.

Stop to Consider

Abstract Syntax Tree (AST) or simply Syntax Tree is a form of intermediate code generation. It is a hierarchical structure of the source code. During syntax analysis, AST is generated. In an abstract syntax tree of an expression, each interior node represents an operator and its child nodes represent the operands of the operator. Syntax trees are similar to parse trees except the fact that syntax trees contain operators as interior nodes; while parse trees contain non-terminals for the same nodes.

CHECK YOUR PROGRESS- II

9. _________ consumes a considerable amount of time during semantic analysis phase.
10. A language with weaker type checking system is prone to make ____________.
11. The _____________ information describes whether the variable is an integer, floating point number, boolean, string or a pointer.
12. The type information is associated with the ___________required to store a variable.
13. ______________ increases the efficiency of the compiler.
14. ______________data types describe the range of values occupied by them.
15. ________________is a user defined data type that allows combining the primitive data types and generates more complex aggregations.
16. Structures are always defined using the keyword_____________.
17. __________________ is a static type checker.

11.5 SUMMING UP

- It is very essential for a compiler to record the information about the attributes of the identifiers used in the source program. Information of such kind is essential for transforming a program written in source language construct into a target language equivalent.
- A symbol table is a large data structure which is indexed by a symbol name or lexeme. The lexical analyzer works in a manner similar to pattern recognition. It tries to find out the identifiers occuring in a code.
- The same identifier may be declared in different locations or procedures. All these instances must be recorded in the symbol table. By setting up different symbol tables for each block or by keeping pointers to blocks within a single symbol table these instances are maintained.
- A simple way to represent symbol table is in terms of array of structures where each structure represents a record of an identifier. Type definitions are found in them.
- The symbol table must consist of two basic layers. The outer layer is termed as the maintenance layer which deals with functions such as creating data structures for specific symbols, inserting such structures into the table and managing as well as the table. The inner layer handles the actual table maintenance tasks at the physical level; like inserting new entries in the table, searching them as well as deleting them.
- A symbol table can be implemented using array of structures.
- During semantic analysis phase, a considerable amount of time is spent in doing type checking. Different programming languages have different approaches of doing type checking. This in turn allows error detection at compile time.
- Before going into type checking, we must determine the type of each identifier of an expression. Each variable is accessed with their corresponding type definitions. This process is termed as name resolution. Once name resolution completes, the information required for doing type checking is ready.
- A programming language is considered to be expressive if it can represent a variety of ideas to a greater extent. An expressive language can express solutions to problems

defined in a particular domain. A piece of code is expressive if it does not take into account the facts derived during type checking.

- B- Minor is a static type checker for a simple language like C. It is strict type checker which tries to detect a large number of type errors. It supports static type checking, functions, expressions and basic control flow. This error checking is typically based on type mismatching as well as misuse of reserved keywords.

11.6 ANSWERS TO CHECK YOUR PROGRESS

1. name resolution
2. attributes, identifiers
3. symbol name
4. array of structures
5. True
6. Scale
7. declarations, expressions, statements.
8. Maintenance, physical
9. Type checking
10. errors
11. type
12. storage space
13. Performance
14. Primitive
15. Structure
16. Struct
17. dot (".")
18. union
19. enum
20. 0
21. Abstract Syntax Tree (AST)
22. B- Minor

11.7 POSSIBLE QUESTIONS

A. Short answer type questions.

1. Why is it necessary to store the records of the identifiers used in a program?

2. What is symbol table? Describe.
3. What is type checking? Describe.
4. What do you mean by name resolution?
5. How are the symbol table represented?
6. What are the two basic layers of a symbol table?
7. What is type checking? Describe in brief.
8. What are the functions performed by the semantic analysis phase?
9. What do you understand by expressiveness of a type system?
10. What is B-minor type checker? Explain in brief.
11. What are the two classes of type errors that the B-minor type checker can detect?

B. Long answer type questions.

1. What are the requirements of a good symbol table must contain?
2. Describe the structure of a symbol table.
3. Describe the organization of a symbol table.
4. How are the semantic analysis and type checking related with each other? Describe.
5. What is type system? Give an overview of the type system.
6. How does the primitive data types form new user defined data types?
7. Explain the B-minor type checking system.

11.8 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). *Introduction to compilers and language design*. Lulu. com.
- Holub, A. I. (1990). *Compiler design in C* (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT 12: SYNTAX DIRECTED TRANSLATION

Unit Structure

12.0 INTRODUCTION

The language system is guided by context-free grammars. This formal grammar is a means of implementing syntax analysis phase of the compiler. The grammar symbols are attached with attributes. We already know that the programming language constructs are always represented by the grammar symbols. The values of attributes are computed by some semantic rules associated with the production rules or grammar rules. The semantic rules are associated with productions in two ways: 1. Syntax directed definition 2. Syntax directed translation. The purpose of syntax analysis phase is mainly to produce syntax trees. A syntax tree is a data structure in which the interior nodes represent the operations and leaf nodes represent its operands. The parser produces syntax directed translations apart from finding syntax errors. Throughout this unit, we will learn how syntax level analysis is done on the programming language constructs. We will also learn the generation of syntax trees and association of semantic rules combine to derive values of attributes so that the syntax directed translation schemes can proceed.

12.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- learn the concepts behind syntax directed definitions
- understand different forms of syntax directed definitions
- know the differences between syntax and inherited attributes
- have knowledge on syntax directed translation scheme
- learn to construct the dependency graph
- have basic ideas on S-Attributed and L-Attributed Definitions
- define intermediate codes and its different forms
- gain knowledge of syntax trees
- learn to construct the Directed Acyclic Graph
- know how to form different types of three address codes

12.2 SYNTAX DIRECTED DEFINITIONS

The analysis phase of a compiler breaks the source code into constituent pieces and produces the intermediate code. Then the synthesis phase works on the intermediate code to generate the final code. The syntax of a language corresponds to the format in which the program has to be written. Syntax differs from semantics in which the meaning of the code is represented by the latter. Semantics correspond to the meaning of the program and also what the program produces when it executes. Syntax is primarily specified by the Context Free Grammar (CFG). Apart from specifying the syntax of a language, CFG also helps in grammar oriented compiling technique called Syntax directed translation. However, before going to translation, we must discuss the syntax directed definition.

Syntax directed definition is the generalized form of a context free grammar. Each grammar symbol has an associated set of attributes. These attributes are further classified into synthesized and inherited attributes. Each node of a parse tree represents a grammar symbol. An attribute of a parse tree node corresponds to a string, number, type or memory location etc. It is regarded as the any quantity associated with a programming language construct. Attributes might include data types of expressions, location of the first instruction in the generated code and the number of instructions in the generated code. Therefore, syntax directed definitions are the CFGs together with attributes and rules.

Apart from this, a set of semantic rules is associated with each production. This is because; the values of attributes are computed from the semantic rules associated with the symbols appearing in the productions. In other words, semantic rules associated with the production rules on a particular node deduce the attribute values associated with that node. The value of a synthesized attribute at a parse tree node is computed from the values of the attributes at the children of that node. Similarly, the value of an inherited attribute is computed from the values of the attributes of siblings as well as its parents of that node.

A very important element of syntax directed definition is finding the dependencies among attributes. This feature can be represented using a graph, known as the dependency graph. The dependency graph represents the evaluation order of the semantic rules. This evaluation derives from the values of the attributes at parse tree nodes of the input string.

A parse tree which contains the values of attributes at each node of the parse tree is termed as annotated parse tree. The process through which these attribute values are computed is called annotating or decorating the tree. For example, the parse tree of the postfix notation

12.3.1 Forms of Syntax Directed Definition

In syntax directed definition, the grammar production of the form $A \rightarrow \alpha$ has associated set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_n)$ where f is a function and

1. b is an inherited attribute of one of the grammar symbols belonging to the right side of the production and $c_1, c_2, \ldots, c_n$ are the attributes grammar symbols of the production.
2. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are the attributes of the grammar symbols of the production. This is extensively used in compilation.

The value of attribute b is derived from the attributes $c_1, c_2, \ldots, c_n$. The semantic rules may be written in terms of expressions. It is worth mentioning here that a semantic rule may sometimes create side-effects like printing a value or updating a global value. In such cases, semantic actions are written using procedures or program fragments.

12.3.2 Synthesized and Inherited Attributes

In this unit, we shall discuss two types of attributes associated with non-terminals:

i. Synthesized attribute for a non-terminal at a parse tree node can be defined using semantic rule. The semantic rule for the parse tree node N is associated with the production defined at N. The production must have the non-terminal at its head. The synthesized attribute S at a parse tree node N is defined in terms of the attribute values of the children of N and N itself. Terminals can have only synthesized attributes but not inherited attributes. The lexical analyzer supplies the lexical values of the attributes of the terminals. A syntax directed definition that involves synthesized attribute is called as S-attributed definition. In such definition, each rule computes the attribute of the node present at the head from the attribute values present at the body of the rule.

ii. Inherited attribute for a non-terminal at a parse tree node is also defined using semantic rules; but this time using the parent of the non-terminal. The semantic rule for the parse tree node N must be associated with the production at the parent of N. The production must have the non-terminal at its body. An inherited attribute I at a parse tree node N is defined in terms of the attribute values of the parent of N, N itself and N's siblings.

CHECK YOUR PROGRESS- I

1. The analysis phase of a compiler produces _______________________.
2. The synthesis phase of a compiler generates the _________________.
3. Attributes of a syntax directed defintion are further classified into _______________and ____________ attributes.
4. Finding the dependencies among attributes is represented using a __________________________.
5. What do you understand by annotating a parse tree?
6. Synthesized attributes of a parse tree node is computed from the _____________ of the node.
7. Inherited attributes of a parse tree node is computed from itself, its _____________ and the siblings of the node.

12.4 SYNTAX DIRECTED TRANSLATIONS

Syntax directed definition attaches a set of attributes to each grammar symbol. Apart from this a set of semantic rules are attached to each production in order to compute attributes associated with each grammar symbols.

We already know that code fragments are associated with grammar rules. One such production rule can be assumed as:

$E \rightarrow E + T$	; E and T are the non-terminal symbols. Also stand for Expression and Terminal

Now, using this production, we shall derive an annotated parse tree for the attribute value t as follows:

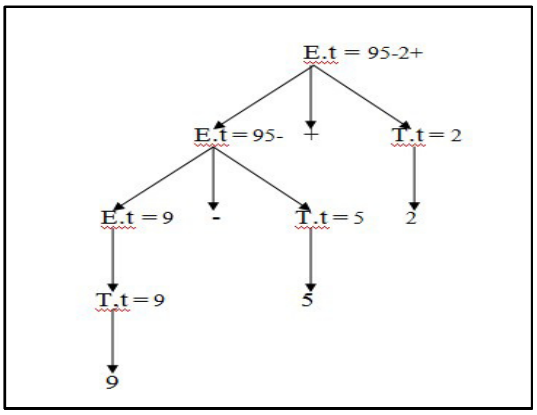


Fig 12.1: Annotated parse tree for the expression 9 - 5 + 2 (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The attribute value at the root of the parse tree derives the postfix notation 95-2+ for the expression 9 – 5 + 2.

A translation scheme is a concept that attaches programming constructs to the production rules. The programming constructs are executed only when they are assigned with productions. The execution order is decided by the syntax analysis phase and accordingly these constructs are executed to generate the final output. This output finally produces the translation of the program.

It is already mentioned that synthesized attributes are most commonly used during translation. An attribute at a parse tree node N is said to be synthesized if the value of the attribute at N is derived from the attributes of N and its children. A single bottom up traversal of the tree evaluates the synthesized attribute. An inherited attribute is the one whose value at a parse tree node is derived from the attribute values of itself, its parents and its siblings.

The following figure corresponds to the syntax directed definition of the parse tree created in Fig. 12.1

Production	Semantic Rule
$E \rightarrow E + T$	Expr.t = Expr.t || Term.t || '+'
$E \rightarrow E - T$	Expr.t = Expr.t || Term.t || '-'
$E \rightarrow T$	Expr.t = Term.t
$T \rightarrow 0$	Term.t = '0'
$T \rightarrow 1$	Term.t = '1'
…….	……..
$T \rightarrow 9$	Term.t = '9'

Fig 12.2: Syntax directed definition for the infix expression 9 – 5 + 2

The semantic rules correspond to the syntax directed definitions for translating expressions into postfix notation. Symbol || correspond to the concatenation operator.

Syntax directed definition must have the property of being simple. The string representing the translation scheme of the non-terminal at the head of each production rule is equal to the concatenation of translations of the non-terminals of the rule's body. The order of translation is same as the order in which they appear in the body. This may in turn require some additional strings to be optionally interleaved. The semantic rules defined in figure 12.2 can be considered as simple as they are concatenated in the same order as they appear in the production's body. By attaching attributes to the nodes of the parse tree in terms of strings, the translation scheme is built up.

We now consider another translation scheme that is similar to syntax directed definition except the fact that the order of evaluation of semantic rules is explicitly specified in translation. Sometimes, program fragments are needed to be embedded in a production's body. It is termed as semantic action. The position of semantic action is shown in between two curly brackets in the production's body. The actions are attached to the parse tree interior node using a dashed line. We may consider the action of printing during execution of codes.

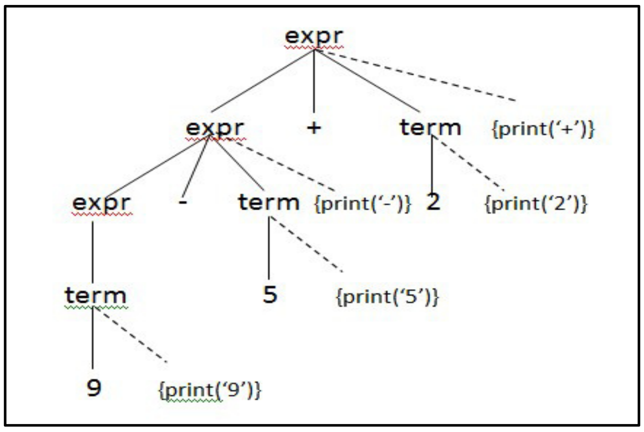


Fig. 12.3: Translation for the expression 9 – 5 + 2
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The grammar generates expression consisting of digits separated by plus and minus. A left-to-right depth first traversal of the tree executes the print whenever leaf node is visited. Finally, the actions translate the expression into postfix notation.

The root represents the first production to be considered. The next action would be to traverse the leftmost subtree of the root and perform all the actions. That is, the tree traversal will start from the left operand expr. This would produce the action '95-'. Next, the leaf node '+' will be encountered; but no action will take place. And finally, the right subtree traversal would lead to performing action '2+'.

Both the schemes presented in Fig. 12.1 and Fig. 12.3 produce the same translation; but they are constructed differently. Fig. 12.1 produces translation simply by attaching attribute values to the nodes. But, the other one does the same by printing the translation through semantic actions. The semantic actions are performed in the same way they appear during postorder traversal.

As specified, syntax directed definition attaches attributes to the grammar symbols simply by associating semantic rules to the productions. Let us refer to Table 12.2 and consider a semantic rule $Expr._{code}$ = $Expr_{1}._{code}$ || Term. $_{code}$ || '+' corresponding to the production $E \rightarrow E_1 + T$. The production has two non-terminals, E_1 and T; E distinguishes from E_1 in a fact that E occurs in the production head and E_1 occurs in production head. They have a string valued attribute code and $Expr._{code}$ is obtained by concatenating $Expr_{1}._{code}$, Term. $_{code}$ and +. We now again refer to figure 12.3 and reconsider the associated semantic action $E \rightarrow E_1 + T$ { print '+' }. Semantic actions are represented within curly brackets. Here, the action occurs after all grammar symbols. However, an action may occur anywhere within the production body.

As already mentioned, the lexical values of the terminals are gathered during lexical analysis; there is no semantic rule in the syntax directed definition for computing the values of the attributes of a terminal.

We consider another set of syntax directed definitions (SDD) for a simple desk calculator. It evaluates expressions terminated by n. The non-terminals have a single synthesized attribute *val*. Apart from this, the terminal **digit** has a single synthesized attribute *lexval* which is an integer value returned by the lexical analyzer.

Sl. No.	Production	Semantic Rule
1	$L \rightarrow E\ n$	$L._{val} = E._{val}$
2	$E \rightarrow E_1 + T$	$E._{val} = E_1._{val} + T._{val}$
3	$E \rightarrow T$	$E._{val} = T._{val}$
4	$T \rightarrow T_1 * F$	$T._{val} = T_1._{val} * F._{val}$
5	$T \rightarrow F$	$T._{val} = F._{val}$
6	$F \rightarrow (E)$	$F._{val} = E._{val}$
7	$F \rightarrow$ digit	$F._{val} = \text{digit}._{lexval}$

Fig. 12.4: Syntax directed definitions of a simple desk calculator (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The first production sets the value of L to the numerical value of E. The In the second production, the value attribute of E is derived by the sums of the value attributes of its child nodes E_1 and T. Similarly, the value of E is set to the value of T. The fourth production sets the value of symbol T as the multiplication of the numerical values of its child nodes T_1 and F. Productions 5 and 6 sets the values of T and F to the numerical values of their single child nodes F and E respectively. Finally, production 7 sets the val attribute of symbol F to the numerical value of the token digit returned by the lexical analyzer. The SDD of Fig. 12.4 is an S-attributed definition.

Rules are first applied and the parse tree is constructed. Then these rules of the SDD would evaluate the values of the attributes of the parse tree nodes. Thus, an annotated parse tree is built. However, evaluation of attributes of a parse tree nodes require to evaluate the attribute values of the nodes upon which it is dependent. For example, the synthesized attribute of a particular node in a parse tree is derived by evaluating the attributes of its child nodes. Synthesized attributes can derive attributes in any bottom-up order like postorder, preorder or inorder.

In both synthesized and inherited attributes, there is no guarantee about the order in which the attributes are evaluated.

We now construct an annotated parse tree for the input string (2 * 4) + (3 * 5) n using the grammars and rules of Fig. 12.4.

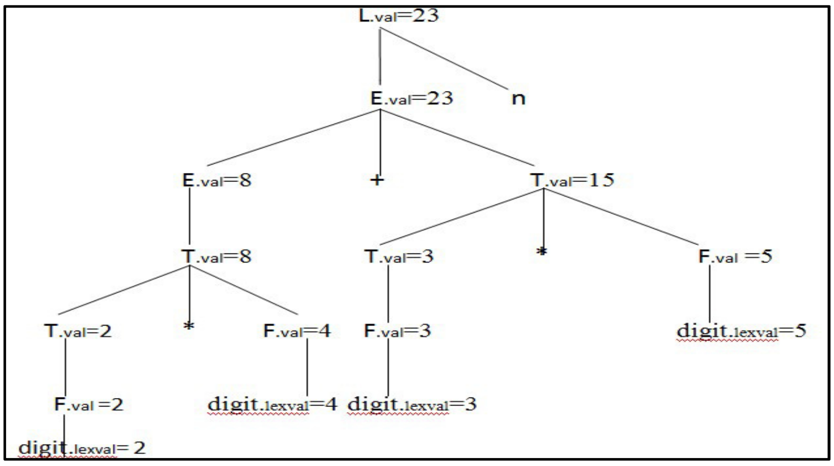


Fig. 12.5: Annotated parse tree for the expression (2 * 4) + (3 * 5) n

Each non-terminal has an associated attribute ***val*** which is computed in a bottom-up approach. Terminal ***digit*** has attribute ***lexval*** whose value is derived during lexical analysis. During bottom-up approach, this value propagates upwards into the parent nodes resulting in involving of S-attributed definition. Figure 12.5 represents translation process which involves synthesized attributes.

Let us consider a production rule of the form C → D representing a semantic rule D.val = C.val + 5. The attribute value of the child node D is computed form the attribute of its parent C. Therefore, ***val*** represents an inherited attribute. We may consider another inherited attribute ***type*** and the semantic rule might be like D.type = C.type. We again consider another parse tree for the purpose of representing inherited attributes. We consider the following set of productions and their corresponding semantic rules.

Sl. No.	Production	Semantic Rule
1	$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2	$T' \rightarrow * FT_1'$	$T_1'.inh = T'.inh \text{ X } F.val$ $T'.syn = T_1'.syn$
3	$T' \rightarrow €$	$T'.syn = T'.inh$
4	$F \rightarrow digit$	$F._{val} = digit._{lexval}$

Fig. 12.6: SDD involving both inherited and synthesized attribute (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

Non-terminals T and F have the synthesized attribute ***val***. Similarly, terminal digit also has a synthesized attribute ***lexval***. T' has two attributes: synthesized ***syn*** and inherited attribute ***inh***. To generate top down parsing which generates the expression 9 * 4 * 7 requires the left operand of the operator * is inherited. The following annotated parse tree derives the given string using both the attributes.

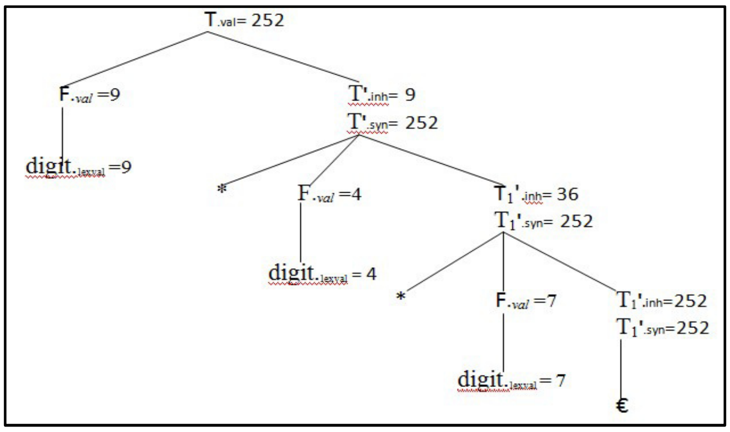


Fig. 12.7: Annotated parse tree for the expression 9 * 4 * 7

The leftmost leaf of the parse tree is labeled with the terminal digit and had the attribute name lexval= 9 which is supplied by the lexical analyzer. The semantic rue associated with this production is $F._{val} = digit._{lexval}$. This makes $F._{val} = 9$. In the second subtree, production $T' \rightarrow *FT_1'$ is applied. The corresponding semantic rules indicate that T'.inh inherits the value of its sibling $F._{val}$ which is equal to 9.

Now, production $T' \rightarrow * FT_1'$ is applied at T'. Here, two attributes have been introduced having semantic rules $T_1'.inh = T'.inh \text{ X } F.val$

and T'.syn = T_1'.syn. This makes T_1'.inh = 36. At T_1', again production T' → * FT_1' is applied with the same set of semantic rules as the previous one. Now, the same inherited attribute T_1'.inh derives its value equal to 252. Finally, at T_1', T' → € is used and the corresponding semantic rule T'.syn = T'.inh evaluates T'.syn = 252. This value propagates up to set the synthesized attribute value T_1'.syn and T'.syn equal to 252. Finally, at the root, the synthesized attribute value T.val is set to T'.syn and is also equal to 252. In this way, an expression is evaluated in an annotated parse tree.

It is important to understand the evaluation order of the attributes of a given annotated parse tree. Dependency graph is such a tool which determines how these attribute values at each parse tree node is evaluated. Once they are determined, they can be shown in the annotated tree.

12.5 DEPENDENCY GRAPH

Information flows among the attributes of parse tree nodes. A dependency graph depicts such flow of information from one attribute into another in a parse tree. An edge from one attribute into another means that the value of the first is needed to compute the second.

- Corresponding to each parse tree node labeled X, the dependency graph also has a node for each attribute associated with X.
- Let us consider that we have a semantic rule associated with a production p. The production defines the value of synthesized attribute A.b in terms of the value of X.c. Therefore, the dependency graph has an edge from X.c to A.b. At every node labeled A where p is applied, we create

an edge to attribute b from attribute c at the child node corresponding to the production body.

- Let us consider that we have another semantic rule associated with a production p. The rule defines the value of inherited attribute A.b in terms of the value of X.c. The dependency graph has an edge from X.c to A.b. At every node labeled A where p is applied, we create an edge to attribute b at A from attribute c at the node corresponding to X. X could be either the parent or sibling of A.

Considering the following production and the corresponding semantic rule declared in Fig. 12.5.

Production	**Semantic rule**
$E = E_1 + T$	$E._{val} = E_1._{val} + T._{val}$

Each node N labeled by E at the head is evaluated by adding the attribute value *val* of its children. Therefore, *val* is a synthesized attribute which derives the value of N from the *val* attributes of its body. It can be represented in terms of Fig. 12.7.

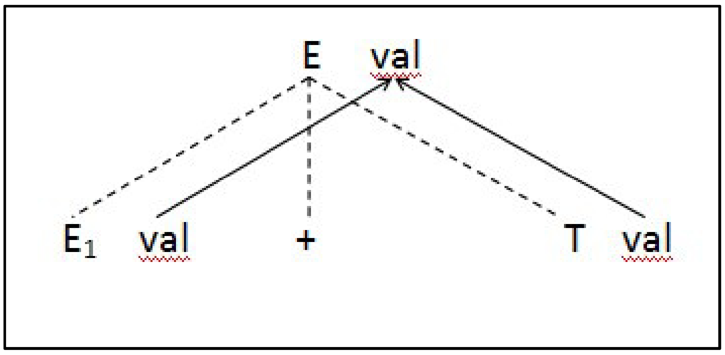


Fig. 12.8: Dependency graph representing synthesized attribute $E._{val}$ from its children

In this way, we can represent any annotated parse tree using dependency graph. The annotated parse tree evaluating the expression **9 * 4 * 7** is shown in Fig. 12.7. It involves both inherited and synthesized attributes. The same has been represented in terms of a dependency graph in the next figure.

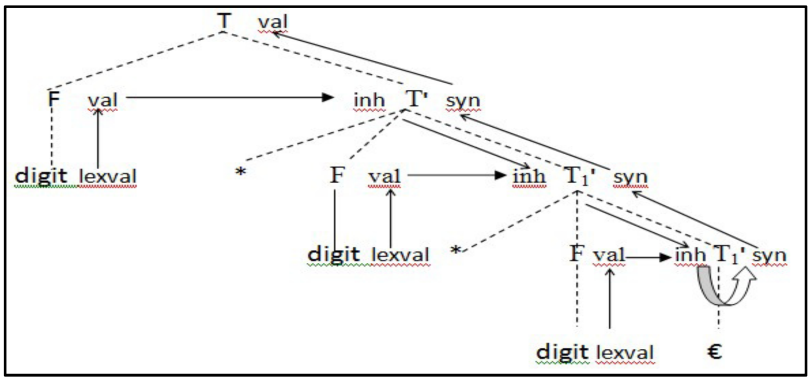


Fig. 12.9: Dependency graph representing annotated parse tree of Fig. 12.7

F.*val* is a synthesized attribute which gathers its value from the attribute value of its child node digit.*lexval*. The inherited attribute T'.inh inherits its value from F.*val* when production T → FT' is applied. The child node T_1' inherits values from its parent T' and also from F when production T'→ * FT_1' is applied. The semantic rule T_1'.inh = T'.inh X F.val is applied in order to derive the value of T_1'.inh. Same semantic rule is applied when same production T'→ * FT_1' is applied on T_1'. Finally, production T' → € is applied on T_1'. This in turn allows us to apply the semantic rule T'.syn = T'.inh. This assigns the inherited attribute of T' into the synthesized attribute of T'. Now, this synthesized attribute propagates upwards and finally gets assigned into the *val* attribute of the root node of the tree T. In this way, the order of evaluation proceeds and finally the result gets assigned in the head of the tree.

A dependency graph represents the possible order of evaluation of attributes at various nodes of the parse tree. If the graph has an edge from node M to N, then the attribute value of node M must be evaluated prior to that of N. It is worth to mention here that sometimes ordering embeds a directed graph into a linear order. This happens when we have nodes N_1, N_2, N_3, ……., N_k in which the graph has an edge of from N_i to N_j where i<j. Such ordering of

sequences of nodes is termed as topological sort of the graph. If the graph has cycles, then there is no topological sort and there is no way to evaluate SDDs.

CHECK YOUR PROGRESS- II

8. Syntax directed definition attaches a set of _____to each grammar symbol.

9. A single bottom up traversal of the tree evaluates the________.

10. Information flows among the ______ of parse tree nodes.

11. The edges of a depeendency graph are always drawn using _________.

12.6 S-ATTRIBUTED and L-ATTRIBUTED DEFINITIONS

Sometimes given an SDD, it is very difficult to whether there exists any dependency graph that has no cycle. SDDs implement translations and also guarantee the evaluation order. This is because; they do not permit dependency graphs with cycles. In an SDD, if every attribute is synthesized is termed as S-attributed definition. In an S-attributed SDD, the attributes of parse tree nodes are evaluated in a bottom-up order. By traversing the tree in a postorder format evaluates the attributes of the nodes. Bottom-up parse corresponds to postorder traversal of nodes. And postorder also corresponds exactly to the manner in which an LR parser reduces a production body to its head.

On the other hand, in an L-attributed SDD, the edges of a dependency graph can go from left t right but not from right to left. Each attribute in an L-attributed definition must be either

- Synthesized
- Inherited. Suppose, there is a production A→ X_1, X_2, X_3, ……., X_k and there is an inherited attribute $X_{i.a}$ computed by a semantic rule. The rule may use only:
  a. Inherited attribute associated with head A.
  b. Either synthesized or inherited attributes associated with the occurrence of the symbols X_1, X_2, X_3, ……., X_{i-1} towards the left of X_i.
  c. Inherited or synthesized attributes associated with the occurrence of X_i itself with condition that the dependency graph must not contain any cycles.

The SDDs in figure 12.6 are L-attributed. The inherited attributes gather values of attributes from above or from the left. However, some SDDs of the given definition are synthesized also. Let us again define two productions with their corresponding semantic definitions.

Production	**Semantic rule**
T → FT'	T'.inh = F.val
T' → * FT_1'	T_1'.inh = T'.inh

X F.val

The inherited attribute T'.inh uses the value of F.val where F appears towards the left of T'. Similarly, the value of the inherited attribute T_1'.inh is derived by multiplying the attribute values of its parent (T'.inh) and its sibling which appears at its left (F.val). Thus, both the SDDs can be regarded as L-attributed definitions.
We consider another production and also its corresponding SDDs and see that the definitions cannot be considered as L-attributed.

Production	**Semantic rule**
$A \rightarrow B\ C$	$A._a = B._b$ $B._i = A._a \times C._c$

The first semantic rule corresponds to assignment of the attribute $B._b$ into the value $A._a$. This is synthesized attribute definition where the attribute value of the chils node is assigned to the parent. Similarly, in the second definition, the attribute value of the child node is defined by multiplying the attribute values of both its parent and sibling. It is obviously an inherited attribute but node C appears at the right side of the production. This defies the basic principle of L-attributed definition. Therefore, the SDDs cannot be regarded as L-attributed definitions.

12.7 INTERMEDIATE CODES

In the analysis-synthesis model, the front end of the compiler analyzes the source program and produces an intermediate representation of the same. This representation is required to produce the target code of the source program. Therefore, intermediate codes play a crucial role during compilation. A compiler may create a sequence of intermediate representations while translating from source to target representation. Therefore, the representation must be easy to produce and also easy to translate into target code. High level representations are close to the source code and low level representations are close to the target code. One form of high level representation might be the syntax tree which depicts the hierarchical structure of the source program and also does tasks like static type checking. On the other hand, low level representations like three-address codes the closer to the machine dependent tasks like register allocation and selection of instructions. In this unit, we shall see some common forms of intermediate representations.

- Trees, which includes abstract syntax trees.
- Linear, which includes three-address code.

12.7.1 Abstract Syntax Tree

Programming constructs are represented by the abstract syntax trees or simply syntax trees. As already studied, during analysis phase of compiler, attributes are inserted to the nodes of the tree. Attributes are defined by the SDDs. An abstract syntax tree (AST) is a hierarchical structure that represents the source program construct. An AST for an expression consists of the followings:

- Each interior node represents an operator.
- Each child node represents the operand of the operator.

A syntax tree is a form of intermediate representation that resembles a parse tree; however, a syntax tree contains an operator as interior nodes, whereas, a parse tree contains a non-terminal for the same. We consider the following syntax tree which represents a simple arithmetic expression E_1 *op* E_2; where op represents the interior node and leaf nodes are represented by E_1 and E_2. Not only expressions, a syntax tree can represent any programming construct.

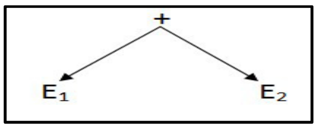


Fig. 12.10: An Abstract Syntax Tree

Apart from this, a syntax tree is also used by the semantic analyzer so that it can check whether the code is semantically correct or not. We now consider another expression 9-5+2 and try to draw the syntax tree for the same.

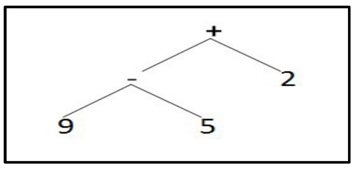


Fig. 12.11: An AST for the expression 9-5+2

The root '+' contains two sub expressions (9-5) and 2. Let us represent another C statement **while (expr) stmt** using a syntax tree. As the while statement consists of an operator, it becomes the root of the tree and the syntax trees for expr and stmt form the children of while.

For each statement construct that begin with a keyword, it forms the operator of the syntax tree. Thus, for **while** statement, the operator is 'while' and that of **do-while** statement, is 'do'. Conditional statement **ifelse** can have 'if' only as an operator in the syntax tree.

12.7.2 Directed Acyclic Graph

A Directed Acyclic Graph or DAG in short is a variant of syntax tree. Like syntax trees, DAG also contains operator as its interior node and operands as leaf nodes. Moreover, it is a tree contains common sub expression and it would be replicated as many time as the common sub expression appears in the construct. Thus, a DAG identifies the common sub expressions occurring in an expression. It gives an important clue regarding the generation of efficient code so that compilation can proceed firmly. DAGs and syntax trees are constructed using the same technique.

Let us draw a DAG for the expression: a + a * (b - c) + (b - c) * d.

The expression contains common sub expressions a, (b – c). The interior nodes would contain the operators. The leaf nodes would be the operands.

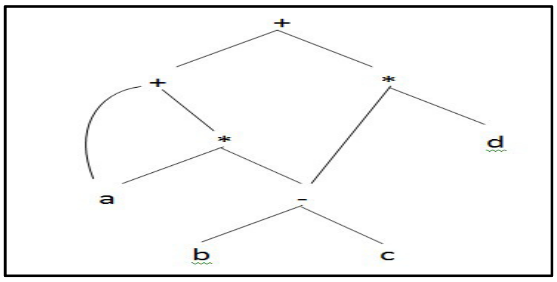


Fig. 12.12: DAG for the expression a + a * (b - c) + (b - c) * d

Often nodes of a syntax tree are stored as an array of records. Each record represents information of a node. The first field of the record indicates the operation code which designates the label of the node. Leaves have an additional field which holds its lexical value. Interior nodes have two fields- one for its left child and the other for right.

Let us consider the expression i = i + 1 and try to draw a DAG and also keep records of each node in an array.

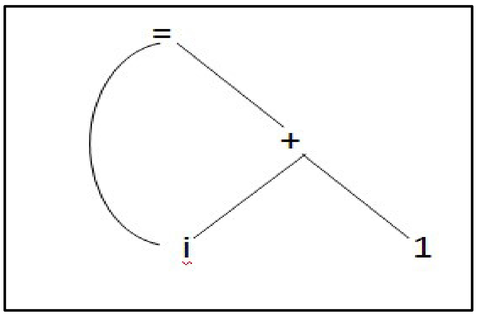


Table 12.1: Array of nodes for the expression i = i + 1

1	id	Entry for i	
2	num	10	
3	+	1	2
4	=	1	3

Fig 12.13: DAG for the expression i = i + 1

(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

Each node of the array is referred to by an integer index. In other words, the integer, also termed as the value number identifies each node and also the expression represented by the node. For instance, the node labeled '+' has the value number 3 and also its left and right children have value numbers 1 and 2 respectively. Similarly, node labeled '=' refer to its left and right children through value numbers 1 and 3 respectively. Thus, an interior node is represented by the 3-tuple (*op, l, r*) where op represents the label, l and r are the value numbers of the left and right children of the node. When r=0, it is assumed to be a unary operator.

12.7.3 Three-Address Code

It is already mentioned that once syntax trees are constructed, attribute values at the nodes are evaluated and code fragments are executed. The tree is traversed in order to generate the three address code for the programming construct. A three address code is an instruction sequence of the following forms:

1. An assignment instruction of the form ***z = x op y***; where z, x and y are the identifiers, constants or temporaries and op is a binary arithmetic or logical operator.
2. Another assignment instruction of the form ***z = op y;*** where op is a unary operator.
3. Copy instructions for copying the value of one operand to another is represented by the three address code: ***y = x***
4. The sequence of three address instructions are executed in a linear fashion until it encounters a conditional or unconditional **jump** statement. The following statements represent a control flow with a jump to the statements labeled L.

   goto L execute instruction with label L
   if x goto L if x is true, then execute the instruction labeled L.
   if False x goto L if x is flase, then execute the instruction labeled L
5. Another conditional jump statement such as if *x relop y goto L* applies relational operators such as <, <=,==, >, >=between x and y. If the operator evaluates to true, the statements with label L will be executed. If not, the instructions next to the statement *x relop y goto L* would be executed.

6. Procedure calls are implemented using statements such as **param x** and **call p, n** or **y = call p, n**. Similarly, returns are implemented using statement **return y**; y being the return value of a procedure. As part of the procedure call p(x_1, x_2, ……., x_n), the following three address codes are executed:

```
param x1
param x2
…..
param xn
call p, n
```

where n specifies the number of actual parameters in the procedure call.

7. Another kind of three address code can be the address and pointer assignments: x = &y, x = *y and *x = y.

In three address code, there is at most one operator at the right side of an instruction. An instruction of the form a+b*c can be represented in terms of the three address code as follows:

```
t1 = b * c
t2 = a + t1
```

where t1 and t2 are two compiler generated temporaries. A three address code is a linear representation of the syntax tree or a DAG. We again consider the DAG of Fig. 12.12 and transform it into its corresponding three address code.

```
t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

Three address codes are primarily based on two major components: addresses and instructions. This corresponds to the concept of class with the two components being its sub classes. Apart from this, a three address code can also be implemented using records with fields for addresses. Records are of various types: triples and quadruples.

An important design issue associated with generation of three address codes is the choice of allowable operators. The operator set must be rich enough to implement the source language construct. Richer the operator set, it is easier to generate the target code for a programming language construct. This is because; the optimizer and the code generator do not have work hard in order to generate a good code structure for these operations.

12.7.3.1 Quadruples

Three address instructions have different kinds of representations: triples, quadruples and indirect triples. As the name suggests, a quadruple consists of four fields: operator, arg1, arg2 and result. The op field indicates the internal representation of the operation. The two arguments arg1 and arg2 contains the two operands on which the binary operation is performed. Finally, the result field contains the result after computation.

However, unary operations or copy instructions do not follow such format as they do not use **arg2**. Similarly, other operators like **param** and **jump** neither use **arg2** nor **result**. Therefore, these operations can be treated as the exceptions to rules quadruples are formed.

Let us take a set of three address codes and try to convert it into an equivalent set of quadruples.

```
t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
res=t5
```

	op	arg1	arg2	result
0	minus	b	c	t1
1	*	a	t1	t2
2	+	a	t2	t3
3	*	t1	d	t4
4	+	t3	t4	t5
5	=	t5		res

Fig. 12.14: Three address code and corresponding quadruples

In case of unary operation, the arg2 field corresponding to the record will remain blank. Here, we have considered an additional statement res=t5 for the sake of understanding so that you get to know how quadruples are generated when only one argument is involved in a statement.

12.7.3.2 Triples

A three address code which consists of three fields: operation, arg1 and arg2. In quadruples, the result field contains temporary names of result of intermediate operations. In triples, result field is specified by the position or value numbers of the operation. Thus, rather than t1, the triple representation would refer to the position of the operation as 0.

	op	arg1	arg2
0	minus	B	C
1	*	A	(0)
2	+	A	(1)
3	*	(0)	D
4	+	(2)	(3)
5	=	(4)	Res

Fig 12.14: Triples equivalent to quadruples of Fig 12.13

Quadruples are beneficial over triples because they do not require instructions to be moved around. For instance, in an optimizing compiler, instructions are needed to be moved frequently. With quadruples, instructions do not need to be changed whenever they are moved. But with triples, the operations are referred to by positions. Therefore, moving an instruction may require changing all references to those operations. This problem of changing of references does not occur with indirect triples.

12.7.3.3 Indirect Triples

Indirect triple is an enhancement over triple representation. It consists of a list of pointers to triples rather than the triples themselves. It is basically implemented using an array which points to the list of pointers to triples. Therefore, instead of position, pointers are used to store results. It is very common for an optimizing compiler to move around the instructions. Indirect triples do it simply by rearranging the array and without affecting the triples themselves. The triples of figure 12.14 can be rewritten using indirect triples as follows.

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Fig. 12.15: Indirect triples equivalent to triples of Fig 12.14

CHECK YOUR PROGRESS- III

12. Bottom-up parse corresponds to ______________traversal of nodes.
13. In an S-attributed SDD, the attributes of parse tree nodes are evaluated in a _____order.
14. Intermediate representation must be easy to produce and also easy to translate into________.
15. High level representations are close to the ____________________.
16. Low level representations are close to the ____________________.
17. Syntax tree which depicts the ______________of the source program.
18. A Directed Acyclic Graph tries to eliminate _________ occurring in an expression.
19. A ________________consists of four fields.
20. __________consists of a list of pointers to triples rather than the triples themselves.

12.8 SUMMING UP

- A syntax tree is a data structure in which the interior nodes represent the operations and leaf nodes represent its operands. The parser produces syntax directed translations apart from finding syntax errors.
- The semantic rules are associated with productions in two ways: 1. Syntax directed definition 2. Syntax directed translation.
- The syntax of a language corresponds to the format in which the program has to be written.
- Syntax is primarily specified by the Context Free Grammar (CFG). Apart from specifying the syntax of a language, CFG also helps in grammar oriented compiling technique called Syntax directed translation. Syntax directed definition is the generalized form of a context free grammar.
- Each grammar symbol has an associated set of attributes. An attribute of a parse tree node corresponds to a string, number, type or memory location etc. Attributes might include data types of expressions, location of the first instruction in the generated code and the number of instructions in the generated code. Attributes are further classified into synthesized and inherited attributes.
- A parse tree which contains the values of attributes at each node of the parse tree is termed as annotated parse tree. The process through which these attribute values are computed is called annotating or decorating the tree.
- Synthesized attribute for a non-terminal at a parse tree node can be defined by the attribute values of the children of N and N itself. Inherited attribute for a non-terminal at a parse tree node is defined by the attribute values of itself, its parent and its siblings.

- Sometimes, program fragments are needed to be embedded in a production's body. It is termed as semantic action. The position of semantic action is shown in between two curly brackets in the production's body. The actions are attached to the parse tree interior node using a dashed line. Semantic actions are represented within curly brackets.
- A dependency graph depicts such flow of information from one attribute into another in a parse tree. An edge from one attribute into another means that the value of the first is needed to compute the second. A dependency graph represents the possible order of evaluation of attributes at various nodes of the parse tree.
- In an SDD, if every attribute is synthesized is termed as S-attributed definition. In an S-attributed SDD, the attributes of parse tree nodes are evaluated in a bottom-up order. On the other hand, in an L-attributed SDD, the edges of a dependency graph can go from left t right but not from right to left.
- A compiler may create a sequence of intermediate representations while translating from source to target representation. Therefore, the representation must be easy to produce and also easy to translate into target code.
- An abstract syntax tree (AST) is a hierarchical structure that represents the source program construct. It is a form of high level intermediate representation. Here, each interior node represents an operator and each child node represents the operand of the operator.
- A Directed Acyclic Graph or DAG in short is a low level intermediate representation. It identifies the common sub expressions occurring in an expression.
- A three address code is a linear representation of low level intermediate codes. Three address codes are primarily based on two major components: addresses and instructions. Three

address codes are of various types: indirect triples, triples and quadruples.

- A quadruple consists of four fields: operator, arg1, arg2 and result. The op field indicates the internal representation of the operation. The two arguments arg1 and arg2 contains the two operands on which the binary operation is performed. Finally, result field contains the result after computation.
- In triple a three address code which consists of three fields: operation, arg1 and arg2. The result field is specified by the position or value numbers of the operation.
- Indirect triple is an enhancement over triple representation. It consists of a list of pointers to triples rather than the triples themselves. It is basically implemented using an array which point to the list of pointers to triples.

12.9 ANSWERS TO CHECK YOUR PROGRESS

1. intermediate code
2. final code
3. synthesized, inherited
4. dependency graph
5. The process through which the attribute values of a parse tree node are computed is called annotating or decorating the tree.
6. Children
7. Parent
8. attributes
9. synthesized attribute
10. attributes
11. dotted lines
12. postorder
13. bottom-up
14. target code
15. source code
16. target code
17. hierarchical structure
18. common sub expression

19. quadruple
20. Indirect triples

12.10 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is Context Free Grammar?
2. What does the semantics of a language represent?
3. What is syntax directed definition? Explain in brief.
4. What does the attribute of a node represent?
5. Discuss the significance of using semantic rules in a syntax directed definition.
6. What do you understand by semantic action?
7. What is dependency graph? Answer in brief.
8. Discuss S-attributed and L-attributed definitions.
9. How does intermediate code play crucial role during compilation process?
10. What is syntax tree? Discuss.
11. Write down the functions performed by the high level and low level intermediate representation of codes.
12. What is abstract syntax tree? Explain with an example.
13. What is Directed Acyclic Graph? Explain with an example.
14. Describe the implementation strategy of quadruples?
15. Describe the implementation strategy of triples?
16. Describe the implementation strategy of indirect triples?
17. Describe why quaduples are beneficial over triples?

B. Long answer type questions.

1. What is syntax directed definition? Explain the different forms of syntax directed definitions.
2. Discuss the synthesized and inherited attributes in detail.
3. What do you understand by syntax directed translation? Explain elaborately with example.
4. Consider your own set of syntax directed definition and show how translation of an expression takes place using the definition.
5. What is dependency graph? Describe how edges are formed between a pair of attributes.

6. What are the prerequisites of deriving attribute values in an L-attributed definition?
7. What are the common forms of intermediate codes? Explain each of them.
8. Describe how Directed Acyclic Graphs are constructed.
9. What are three address codes? What are the different forms of three address codes? Describe.
10. What are the different types of three address codes? Explain each of them.

12.11 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 13
INTERMEDIATE CODE GENERATION

Unit Structure:

13.0 INTRODUCTION

The analysis-synthesis model of a compiler converts the source language program into an intermediate representation. Then this representation is fed into the code generator phase in order to generate the target code. The front end of a compiler basically consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation phases. Similarly, the back end consists of code optimization and target code generation phases. There are a variety of forms that an intermediate representation might be in. We already have got to study about syntax tree which is one of such forms. It is commonly used during syntax and semantic analysis. Compilers generally produce low-level machine level representation. In the previous unit, we had learnt about another low level form of intermediate representation- Three address code. It contains at most three operands. Later, the code optimization phase attempts to improve the intermediate codes for producing better target codes. These codes are shorter, consumes less resources as well as machine independent. In this unit, we shall try to construct the intermediate representations of different types of programming language constructs like conditional statements, loops, arrays, functions and different expressions.

13.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Know the concepts behind intermediate code representation and its different forms.
- Learn about arrays and its translation schemes
- Know about different flow of control statements
- Learn the translation schemes of boolean expressions
- Learn the translation schemes of switch-case statements
- Learn the translation schemes of procedures

13.2 INTERMEDIATE REPRESENTATION

The intermediate representation forms the final phase of the front end of the compiler. The source code passes through the lexical analysis, syntax analysis and semantic analysis phases respectively. The output of semantic analysis is entered into an intermediate code generation phase. The intermediate codes are basically of two types- high level and low level. High level codes are close to the source code. They are

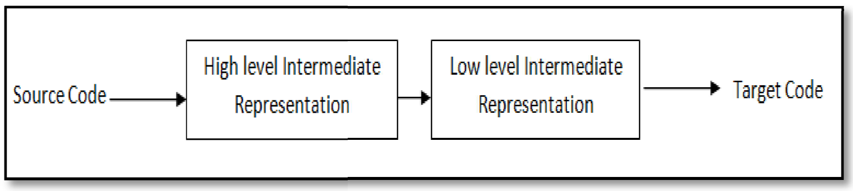

Source Code
High level Intermediate Representation
Low level Intermediate Representation
Target Code

Fig 13.1: Types of intermediate codes
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

13.3.1 Static checking And Forms of Intermediate Representation

During analysis phase of a compiler, the Static Checker phase takes place between the Parser and Intermediate Code Generator. This phase mainly does type checking and checks whether operators are compatible with operands. Static checking assures such errors must be detected and reported. Static checking checks the consistency of the codes. It tries to detect programming errors during compile time rather than run time. Static checking is conducted in terms of two ways: Syntactic Checking and Type Checking.

- **Syntactic checking** attempts to check whether the statements of a program follow the syntax of the language. This might include rules for declaration of variables, special characters allowed during the declaration, their scope, using of braces in different control statements as well as permissible usage of keywords etc.
- **Type checking** ensures that an operator or function is applied to the correct operand type as well as right number of operands. However, in some languages, type conversion is necessary; from integer to real.

In the previous unit, we learnt about two basic kinds of intermediate codes:

- Trees- which include parse trees or Abstract Syntax Trees or simply AST. Syntax trees are the hierarchical structures constructed during the parsing phase of a compiler. As the phases proceed, information is added to the nodes in the form of attributes. A Directed Acyclic Graph (DAG) is a variant of the syntax tree which identifies the common sub expressions.

- Linear representation- which includes the three-address codes. They contain codes having at most three operands. A long statement is broken into smaller sequences of three-address codes. These codes are executed one-after-another. However, for looping statements, the codes contain labels and jump instructions to represent the flow of control. Three address codes are efficient from the fact that they allow performing optimization, doing code generation and debugging, translation from one language into another. They do not have any hierarchical structure.

CHECK YOUR PROGRESS – I

1. ___________ contains at most three operands.
2. __________ attempts to improve the intermediate codes for producing better target codes..
3. What are the two types of intermediate codes?
4. The final phase of the front end compiler is the intermediate code generation. State whether true or false.
5. The ______ phase takes place between the Parser and Intermediate Code Generator.
6. Two ways of static checking: ___________ and ________.
7. Tree representation is linear representation. State True or False.

DAG and three-address codes have been discussed in the previous unit. They can be constructed parallely by generating three-address codes at each node of the syntax tree. The compiler stores nodes and their attributes as well as the data structure used for parsing. This is significant from the point that the compiler can retain those parts required for constructing syntax tree as well as three-address codes.

Now, let us try to construct the intermediate representations of some vital components of a programming language.

13.3 ARRAYS

Arrays are the data structures which store data in contiguous set of memory locations. They store the similar type of data structure. An array A of length n is stored in n-contiguous locations with indices starting from 0 to n-1. If the width of each element is w, then the i^{th} array element starts from location:

base + i × w   ; base being the relative address of A[0]

To calculate the relative address of the array element A[i][j], where j being element number of ith row; the formula would be:

$\text{base} + i \times w_1 + j \times w_2$

Here, w_1 is the width of the row and w_2 is the width of an element in the row.

Therefore, for an n-dimensional array the formula would be:

$\text{base} + i_1 \times w_1 + i_2 \times w_2 + i_2 \times w_3 + \ldots.. + i_n \times w_n$

Alternatively, the relative address of an array can be calculated in terms of the numbers of elements n_j along the j^th dimension of the array and w being the width of each element of the array. Therefore, the formula which calculates the relative address of element A[i][j] would be:

$\text{base} + (i \times n_j + j) \times w$

Similarly, for an n-dimensional array, the formula for calculating the array address would be:

$\text{base} + ((\ldots.((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots.) \times n_n + i_n) \times w$

Addresses of elements of a multidimensional array can be calculated during compile time. But for this to happen, the array must be static. However, when the array is dynamic, address calculation cannot be done during compile time.

13.3.1 Translation of Array References

Generation of codes for an array reference is a bit complicated as we need to associate an array name with a sequence of indices to a non-terminal L. Therefore, L can be of the following form:

$$L \rightarrow L\,[E] \mid id\,[E]$$

The translation scheme for various expressions using array references has been written below. This scheme generates three-address codes with productions and their corresponding semantic actions.

S → id = E ;   { gen(top.get (id.lexeme) ‘=’ E.addr); }

| L = E ; { gen(L.array.base ‘[’ L.addr ‘]’ ‘=’ E.addr); }

E → E_1 + E_2 { E.addr = new Temp ();

gen(E.addr ‘=’ E_1.addr ‘+’ E_2.addr); }

| id { E.addr = top.get(id.lexeme); }

| L { E.addr = new Temp ();

gen(E.addr ‘=’ L.array.base ‘[’ L.addr ‘]’); }

L → id [E]   { L.array = top.get(id.lexeme);

L.type = L.array.type.elem;

L.addr = new Temp ();

gen(L.addr ‘=’ E.addr ‘*’ L.type.width); }

| L_1 [E]   { L.array = L_1.array;

L.type = L_1.type.elem;

t = new Temp ();

L.addr = new Temp ();

gen(t ‘=’ E.addr ‘*’ L.type.width);

gen(L.addr ‘=’ L_1.addr ‘+’ t); }

Fig 13.2: Translation rules for Arrays
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

L.array denotes a pointer to the symbol table entry for the array name. The base address of the array is denoted by L.array.base. It determines the actual l-value of the array reference. L.type determines the type of the sub-array generated by L. Type is important and used as attribute since it is further used during type checking. Associated with each type, there is a width. Further, L.addr denotes a temporary that is used while computing the offset for the array reference during summation of terms $i_n \times w_n$.

The production $S \rightarrow id = E$ does assignment to a non-array variable. Production $S \rightarrow L = E$ creates an indexed copy instruction which assigns the value of expression E to the location specified by the array reference L. Similarly, the production $E \rightarrow L$ generates codes to copy the value from location L into a new temporary.

13.4 FLOW OF CONTROL STATEMENTS

Translation of boolean expressions means translation of statements such as if, if-else and while. These expressions are used as conditional statements that can alter the flow of control. An expression is evaluated first and if it evaluates to true, then some set of statements are executed. Otherwise, control flow passes to some other sequences of instructions. Since, a boolean expression results in either true or false, it is analogous to writing these expressions using three-address codes with logical operators. An expression might have the keyword **if** preceding it. Such expression transfers the flow-of-control to some other point of the program by evaluating the expression to some logical value.

13.4.1 Boolean Expressions

Boolean expressions are composed of boolean operators such as AND (&&), OR (| |) and NOT (!) employed between variables called boolean variables or relational expressions. However, NOT is considered as unary operator and AND and OR are binary. The relational expressions are of the form E_1 rel E_2 where E_1 and E_2 are arithmetic expressions and rel corresponds to relational operators such as <, <=, =, ! =, > or >=. The following grammar generates boolean expressions.

$$B \rightarrow B_1 \mid\mid B_2 \mid B_1 \&\& B_2 \mid !\, B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

It is already known to us that an '| |' operator in the expression B_1 | | B_2 evaluates to true if either B_1 or B_2 evaluates to true. Similarly, the expression B_1 && B_2 evaluates to false is either of B_1 or B_2 evaluates to false. Sometimes, it is possible to evaluate the truth value of the whole expression simply by computing the truth value of only a portion of the expression. This facility allows the compiler to optimize the evaluation of boolean expressions.

The following grammar generates three address codes for flow-control statements.

$$S \rightarrow \text{if (B) } S_1$$

$$S \rightarrow \text{if (B) } S_1 \text{ else } S_2$$

$$S \rightarrow \text{while (B) } S_1$$

B represents a boolean expression and S is a non-terminal.

There is a synthesized attribute ***code***, defined for both B and S. This is required for translation into three-address instructions. Therefore, in $S \rightarrow$ if (B) S_1, the translation rule consists of $B._{code}$ followed by $S._{code}$. If B is true, control passes to the first instruction of $S_1._{code}$. Otherwise, control flows to the instruction which immediately follows $S_1._{code}$. This is depicted as the following figure.

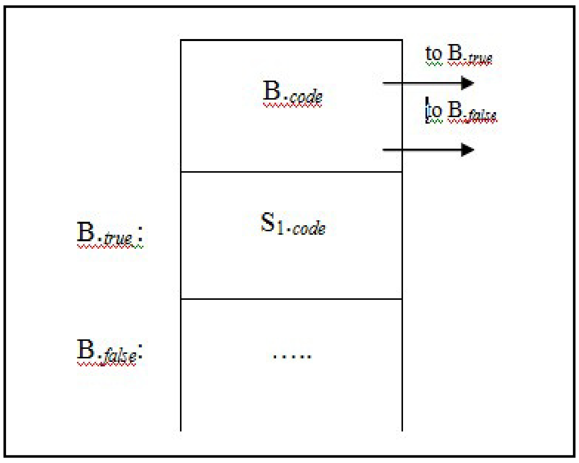


Fig. 13.3(a): Control flow using if condition
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

Fig. 13.3(b) illustrates flow of control of statements using if-else condition. There is a label for the jump instruction. Here, two labels: $B._{true}$ and $B._{false}$ are associated with the boolean expression B. An inherited attribute S.$_{next}$ denotes the label for the instruction which immediately follows the code for S.

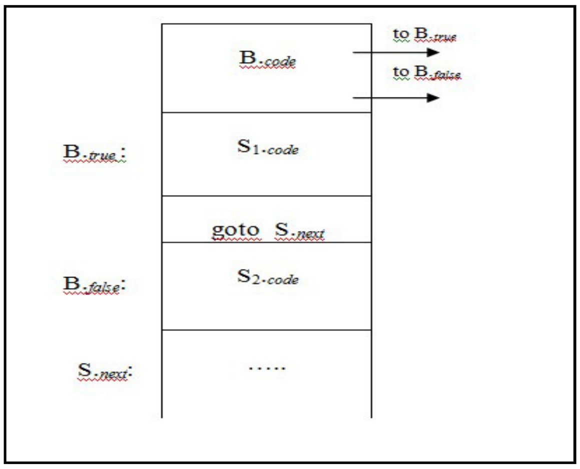


Fig. 13.3(b): Control flow using if-else condition (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

Similarly, figure 13.3(c) showcases the control flow within the while statement. There are two labels associated with the boolean expression B. If B is true, codes in B.$_{true}$ are executed. Otherwise, codes in B.$_{false}$ are executed. There is another label marking the beginning of the loop. The loop is executed repeatedly by jumping to the label begin.

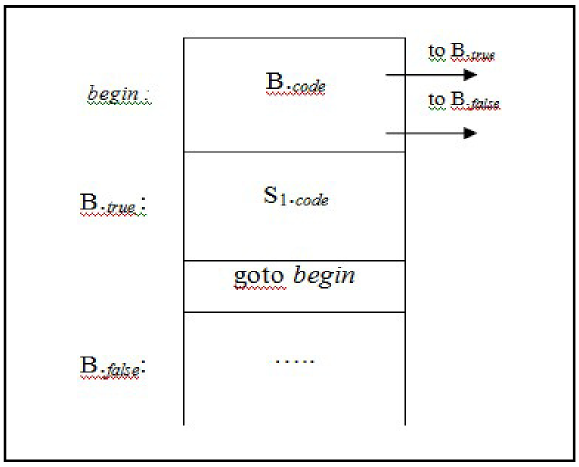


Fig. 13.3(c): Control flow using while (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The syntax directed definition for the above three statements has been mentioned below.

$P \rightarrow S$ $\quad S_{.next}$ = newlabel()

$P_{.code} = S_{.code}$ | | *label*($S_{.next}$)

$S \rightarrow$ assign $\quad S_{.code}$ = assign$_{.code}$

$S \rightarrow$ if (B) S_1 $\quad B_{.true}$ = newlabel()

$B_{.false} = S_{1.next} = S_{.next}$

$S_{.code} = B_{.code}$ | | *label*($B_{.true}$) | | $S_{1.code}$

$S \rightarrow$ if (B) S_1 else S_2 $\quad B_{.true}$ = newlabel()

$B_{.false}$ = newlabel()

$S_{1.next} = S_{2.next} = S_{.next}$

$S_{.code} = B_{.code}$

| *label*($B_{.true}$) | |

$S_{1.code}$

| | *gen* ('goto'

$S_{.next}$)

| | *label*($B_{.false}$) |

| $S_{2.code}$

$S \rightarrow$ while (B) S_1 $\quad$ begin = newlabel()

$B_{.true}$ = newlabel()

$B_{.false} = S_{.next}$

$S_{1.next}$ = begin

$S_{.code}$ = *label*(begin) | | $B_{.code}$ | | label($B_{.true}$)

| | $S_{1.code}$ | | *gen*(

'goto' begin)

$S \rightarrow S_1 S_2$ $\quad S_{1.next}$ = newlabel()

$S_{2.next} = S_{.next}$

$S_{.code} = S_{1.code}$ | | *label*($S_{1.next}$) | | $S_{2.code}$

Fig. 13.4: Syntax directed definition for flow-control statements (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

To create a new label, the newlabel() is called. A call to *label*(L) attaches the label L to the next three-address instruction. The semantic rule associated with production $P \rightarrow S$ initializes $S_{.next}$ to a new label. Similarly, $P_{.code}$ consists of $S_{.code}$ followed by label $S_{.next}$. The next production simply assigns the code attribute of token assign into $S_{.code}$.

The translation of production $S \rightarrow$ if (B) S_1 simply creates a label $B_{.true}$. The three-address code attaches it to the first instruction for statement S_1. The control passes to $S_{.next}$ if B evaluates to false.

During translation of the statement $S \rightarrow$ if (B) S_1 else S_2, control jumps to the first instruction of the code for S_1 if B results in true. Otherwise, control passes to the first instruction for code S_2. Further, control flows from both S_1 and S_2 to the three-address instruction which immediately follows the code for S.

The code for $S \rightarrow$ while (B) S_1 uses a variable begin which is actually a new label attached to the first instruction or the first instruction of B for while statement. The inherited label $S_{.next}$ marks the instruction to which the control flows when B is evaluated to false. Therefore, $B_{.false}$ is set to $S_{.next}$. Label $B_{.true}$ points to the first instruction of S_1. The statement goto begin causes a jump to the first instruction of boolean expression B.

Finally, the code corresponding to the production $S \rightarrow S_1 S_2$ consists of the code for S_1 followed by the code for S_2. The first instruction for code S_2 immediately follows the last instruction for S_2. The instruction after the code for S_2 is the instruction after the code for S.

Similarly, boolean expressions can be translated into three-address instructions. They are evaluated and accordingly jumps are made using two labels based on its truth value: $B_{.true}$ and $B_{.false}$.

$B \rightarrow B_1 \mid\mid B_2$	$B_{1.true} = B_{.true}$
	$B_{1.false} =$ newlabel()
	$B_{2.true} = B_{.true}$
	$B_{2.false} = B_{.false}$
	$B_{.code} = B_{1.code} \mid\mid$ *label*($B_{1.false}$) $\mid\mid B_{2.code}$
$B \rightarrow B_1$ && B_2	$B_{1.true} =$ newlabel()
	$B_{1.false} = B_{.false}$

	$B_{2.true} = B_{.true}$
	$B_{2.false} = B_{.false}$
	$B_{.code} = B_{1.code}$ | | *label*($B_{1.true}$) | | $B_{2.code}$
$B \rightarrow ! B_1$	$B_{1.true} = B_{.false}$
	$B_{1.false} = B_{.true}$
	$B_{.code} = B_{1.code}$
$B \rightarrow E_1$ rel E_2	$B_{.code} = E_{1.code}$ | | $E_{2.code}$
	| | *gen*('if' $E_{1.addr}$ rel.op $E_{2.addr}$ 'goto' $B_{.true}$)
	| | *gen*('goto' $B_{.false}$)
$B \rightarrow$ true	B.code = *gen*('goto' $B_{.true}$)
$B \rightarrow$ false	B.code = *gen*('goto' $B_{.false}$)

Fig. 13.5: Syntax directed definition for boolean statements (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The production $B \rightarrow E_1$ rel E_2 is directly translated into a three-address instruction with comparision operation. If B is of the form (a !=b), then it is translated into the following form.

if a !=b goto $B_{.true}$

goto $B_{.false}$

It is already mentioned that the production of the form $B \rightarrow B_1 \mid\mid B_2$ does not require both B_1 and B_2 to be evaluated every time in order to derive the boolean value of B. Rather, truth value of B_1 is sometimes sufficient to gather the output of B. If B_1 is true, B is true. But, if it is false, then a check on B_2 has to be made. If B_2 is true, B is true. Otherwise, B is false.

The translation of B_1 and B_2 are similar in the production $B \rightarrow B_1$ && B_2.

The truth value of B depends on B_1's truth value. If B_1 is true, B is false. Otherwise, B is true.

The code to translate the constants true and false is to jump to $B_{.true}$ and $B_{.false}$.

13.5 SWITCH STATEMENT

The switch-case statement is commonly available in many modern languages like C, C++ etc. There is a switch expression which needs to be evaluated. The output consists of some constant values or cases. However, switch statement includes a default value which always does something if no other value or case matches. The general structure of a program involving switch-case statement is given below.

```
switch( expr )
        {
        case v1: S1
        case v2: S2
        ......
        case vn-1: Sn-1
        default: Sn
        }
```

Fig. 13.6: Structure of a switch-case program

Expression **expr** is evaluated first. Then the list of cases with their respective values for the expression is evaluated. The statement against case matching the expression is executed. However, if none of the case values match with the expression, the default value is chosen for output.

The program mentioned above can be translated into an intermediate form. When the compiler finds the keyword **switch**, it generates two labels test and next as well as a temporary t. Expression **expr** is evaluated and code is generated into t. Then, there is a jump to label test. With each case keyword, a new label L_i is created and made to enter into the symbol table.

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
```

```
          .....
$L_{n-1}$:   code for $S_{n-1}$
          goto next
$L_n$:       code for $S_n$
          goto next
test:     if t = $V_1$ goto $L_1$
          if t = $V_2$ goto $L_2$
  if t = $V_{n-1}$ goto $L_{n-1}$
          goto $L_n$
next:
```

Fig. 13.7: Translation of switch statements
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

Such translation does n-way branching of code. There is a jump using goto next statement to which the control passes upon execution of statement(s) S_i again label L_i.

13.6 FUNCTIONS

Functions are the set of instructions that are executed at any point of call in a program. During translation of procedures, prior to calling the function or procedure, its parameters are evaluated. A function definition DF consists of keyword define, a return type, name of the function, formal parameters in parenthesis and function body written within brackets. Non-terminal F generates zero or more formal parameters. The non-terminals S and E represent statements and expressions respectively. The production for S returns an expression. The production for E refers to function calls, with A as actual parameters. The actual parameter A returns an expression.

$$
\begin{aligned}
&DF \rightarrow \text{define } T \text{ id } (F) \{ S \} \\
&F \rightarrow € \mid T \text{ id} , F \\
&S \rightarrow \text{return } E ; \\
&E \rightarrow \text{id} (A) \\
&A \rightarrow € \mid E , A
\end{aligned}
$$

Fig 13.8: Functions in source language
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

The function name is entered into the symbol table. The formal parameters of a function are analogous to the field names in a record of a symbol table. Function type resembles the return type and type of its formal parameters. Function type is implemented by using the constructor *fun* applied to the return type and ordered list of types for parameters. During function calls, the generation of three-address codes requires evaluation or reduction of its parameters. Like during the call to function f (Expr$_1$, Expr$_2$, Expr$_3$, ..., Expr$_n$), expression Expr must be evaluated during generation of three-address codes. Then, it must be followed by a param instruction for each parameter.

CHECK YOUR PROGRESS – II

8. __________ are the data structures which store data in contiguous set of memory locations.
9. Jumps are associated with the flow of control instructions. State whether True or false.
10. What are the boolean operators?
11. What are relational operators?
12. The output of a switch expression consists of some___________.

13.7 SUMMING UP

- The analysis-synthesis model of a compiler converts the source language program into an intermediate representation. Then this representation is fed into the code generator phase in order to generate the target code.
- The front end of a compiler basically consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation phases. Similarly, the back end consists of code optimization and target code generation phases.
- The Static Checker phase takes place between the Parser and Intermediate Code Generator. This phase mainly does type checking and checks whether operators are compatible with operands. It attempts to check whether the statements of a program follow the syntax of the language.

- Type checking ensures that an operator or function is applied to the correct operand type as well as right number of operands. However, in some languages, type conversion is necessary; from integer to real.
- Trees- which include parse trees or Abstract Syntax Trees or simply AST. A Directed Acyclic Graph (DAG) is a variant of the syntax tree which identifies the common sub expressions. In three-address codes, a long statement is broken into smaller sequences of three-address codes.
- An array A of length n is stored in n-contiguous locations with indices starting from 0 to n-1. If the width of each element is w, then the i^{th} array element starts from location:

  base + i × w ; base being the relative address of A[0]
- Boolean expressions are used as conditional statements that can alter the flow of control. They are composed of boolean operators such as AND (&&), OR (| |) and NOT (!) employed between variables called boolean variables or relational expressions.
- The switch-case statement consists of a switch expression which needs to be evaluated. The output consists of some constant values or cases. However, switch statement includes a default value which always does something if no other value or case matches.
- During translation of procedures, prior to calling the function or procedure, its parameters are evaluated. A function definition consists of keyword define, a return type, name of the function, formal parameters in parenthesis and function body written within brackets.

13.8 ANSWERS TO CHECK YOUR PROGRESS

1. Three address code
2. Code optimization phase
3. High level and low level
4. True
5. Static Checker
6. Syntactic Checking and Type Checking
7. False
8. Arrays

9. True
10. AND (&&), OR (| |) and NOT (!)
11. <, <=, =, ! =, > or >=.
12. Constant cases

13.9 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is intermediate representation of codes? Explain in brief.
2. What is static checking? What does it do?
3. Discuss syntactic checking and type checking in brief.
4. What is an array? How do you calculate the relative address of a particular element of an array?
5. What are flow of control statements? Discuss with examples.
6. What are boolean expressions? Discuss in brief.
7. Explain the general program structure of the switch-case statement. Also explain briefly the workflow of the program.
8. Explain functions and its associated production rules.

B. Long answer type questions.

1. Describe the different forms of intermediate representation of codes with some suitable examples.
2. What is array? How do you calculate the relative address of array elements? Explain.
3. Describe the translation of array references in detail.
4. What do you mean by the flow of control statements? Show the control flow of these statements with respect to some suitable figures.
5. How are flow of control statements and boolean expressions related?
6. Describe the translations of flow of control statements into syntax directed definition.
7. Describe the translations of boolean expressions into syntax directed definition.

13.10 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 14
REPRESENTING INTERMEDIATE CODE GENERATOR FOR A PARSER

Unit Structure

14.0 INTRODUCTION

Intermediate code generation is an important phase in compiler design.This phage blurs the gap between high-level source code and low-level machine code by producing an intermediate representation (IR) of the source program.IR is machine-independent. The use of machine-independent intermediate code enhances portability by providing a standard format for code that is not tied to any specific hardware or architecture. By translating source code into an intermediate representation, a compiler can then convert this intermediate code to run on different target machines without re-compiling the high-level source code.The IR is typically a sequence of instructions resembling assembly language but simplified to aid optimization. Some common forms of IR include three-address code, abstract syntax trees, and control flow graphs. By using anintermediate code forms, the compiler can optimize program logic and structure before translating it into the low-level machine code. In a compiler ideally, the front end handles all details of the source language, while the back end is responsible for managing specifics of the target machine.

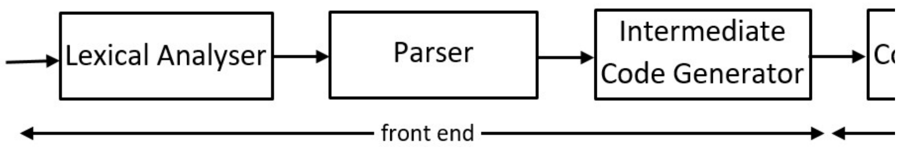


Fig. 14.1: Logical structure of front end and back end of a compiler

14.1 UNIT OBJECTIVES

- Understand Intermediate Code Representation.
- Apply Code Optimization Techniques
- Construct DAGs
- Implement Intermediate Code Generation
- Explain Three-Address Code (TAC) Generation

14.2. THREE-ADDRESS CODE

Three-Address Code (TAC) is an intermediate representation used by compilers to simplify code generation. It breaks down complex expressions into a sequence of simpler steps, each involving a maximum of three addresses: two operands and one result. Results in TAC are stored in compiler-generated temporary variables, providing a clear order of operations. Due to its simplicity, TAC is well-suited for optimization and efficient translation to machine code. Representing control flow and data dependencies, TAC serves as a bridge between high-level source code and low-level machine instructions, making it a critical phase in compiler design.

a + b * c + d	T1 = b * c
	T2 = a + T1
	T3 = T2 + d

A standard expression	*An equivalent TAC*

Fig. 14.2: Example of a Three Address Code

x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)	t1 := b * b
	t2 := 4 * a
	t3 := t2 * c
	t4 := t1 - t3
	t5 := sqrt(t4)
	t6 := 0 - b
	t7 := t5 + t6
	t8 := 2 * a
	t9 := t7 / t8
	x := t9

The standard expressionfor roots of a quadratic equation	*An equivalent TAC*

Fig. 14.3: Example of the Three Address Code for roots of a quadratic equation

for (i = 0; i < 10; ++i)
{
        b[i] = i*2;
}
	t1 := 0
	L1: if t1 >= 10 goto L2
	        t2 := t1 * 2
	        t3 := t1 * 4
	        t4 := b + t3
	        *t4 := t2
	        t1 := t1 + 1
	        goto L1
	L2:

AC code initializing an array using for loop	*An equivalent TAC*

Fig. 14.4: Example of the Three Address Code for loop

There are 3 ways to represent a Three-Address Code:

i) Quadruples

ii) Triples

iii) Indirect Triples

14.2.1 QUADRUPLES REPRESENTATION

Quadruples are a data structure consisting of four fields: op, arg1, arg2, and result. Here, op represents the operator, while arg1 and arg2 denote the two operands, and result holds the outcome of the expression.It facilitates the rearrangement of code, which is useful for global optimization efforts, allowing compilers to reorganize and optimize code more efficiently at a larger scope. Additionally, the symbol table provides quick access to the values of temporary variables, streamlining the retrieval process during compilation. However, this representation often generates many temporary variables, especially when dealing with complex expressions. This results in increased time and space complexity as the compiler must manage and process a larger number of temporary variables.

x = y * – z + y * – z	t1 = uminus z
	t2 = y * t1
	t3 = uminus z
	t4 = y * t3
	t5 = t2 + t4
	x = t5

An expression in high level language	*Equivalent TAC*

Fig. 14.5: Example ofthe Three Address Code in Quadruple Representation

14.2.2 TRIPLES REPRESENTATION

Triples Representation is an alternative intermediate code structure that avoids creating additional temporary variables for single operations. Instead of creating temporary variables, this representation uses pointers to reference other triples in the code when their values are needed. This structure consists of only three fields: op (operator), arg1, and arg2 (operands).The lack of temporary variables makes it more efficient in certain cases like lesser time and space complexity as the compiler need to manage and process a small number of temporary variables.

One drawback of this representation is that temporary variables are implicit, and therefore code rearrangement is challenging. Optimization of code becomes complex because if a triple moves from one place to another in the code it requires modification of all other triples that refers it. However, pointers enable direct access to the corresponding symbol table entries, somewhat easing the process of variable management.

x = y * – z + y * – z	(0) uminus z (1) * (0) y (2) uminus z (3) * (2) y (4) + (1) (3) (5) = x (4)
An expression in high level language	*Equivalent TAC*

Fig. 14.6: Example of the Three Address Code in Triple Representation

14.2.3 INDIRECT TRIPLES REPRESENTATION

Indirect Triples utilize pointers to reference a separate listing of all computations, allowing for efficient organization.This representation employs pointers to a separate listing of computations rather than directly storing the results of operations in temporary variables. Each operation consists of a pointer referencing an entry in the computation list that includes the operator and its operands. This approach keeps temporary variables implicit, making code rearrangement easier and more manageable.

		List of pointers to Table
x = y * – z + y * – z	(a) uminus z (b) * (0) y (c) uminus z (d) * (2) y (e) + (1) (3) (f) = x (4)	(0) (a) (1) (b) (2) (c) (3) (d) (4) (e) (5) (f)

An expression in high level language	*Equivalent TAC*

Fig. 14.7: Example of the Three Address Code in Indirect Triple Representation

14.3 DIRECTED ACYCLIC GRAPH

In compiler design, a Directed Acyclic Graph (DAG) is essential for representing expressions and optimizing code. A DAG consists of directed edges without any cycles, ensuring that no path returns to the starting node. This structure is particularly effective in eliminating redundant computations and identifying common sub-expressions, thereby enhancing the efficiency of program execution. The DAG representationof intermediate code helps in optimizing the Intermediate Code of a high level language in compiler.

14.3.1 WHAT IS DAG?

DAGs are useful in compiler design because they don't contain cycles. A DAG is a directed graph with vertices and edges, where each edge is directed from one vertex to another. In DAG following the directions of the edges will never form a closed loop. The Directed Acyclic Graph serves multiple purposes in representing the structure of basic blocks, visualizing the flow of values between them, and enabling optimization techniques within these blocks, where a basic block is a set of statements that execute one after another in sequence. When applying optimization techniques to a basic block, a DAG represents the three-address code generated during the intermediate code generation phase.

14.3.2 CHARACTERSTICS OF DIRECTED ACYCLIC GRAPH

- DAG is a Data Structure used to represent the structure of basic blocks.

- The leaf nodes of the directed acyclic graph represent a unique identifier. An identifier can be a variable or a constant.
- The non-leaf nodes represent an operator symbol.
- Eachnodeis also given a string of identifiers to use as labels for the computed value.

14.3.3 ALGORITHM FOR CONSTRUCTION OF DAG

For constructing a DAG, basic one basic block is given as input. The output will be a DAG where each node of the graph represents a label. The label of each leaf node represents an identifier. Label of all non-leaf node represents an operator.Each node contains a list of attached identifiers to hold the computed value. For any three address code there are three possible scenarios on which we can construct a DAG.

Case 1: x = y op z where x, y, and z are operands and op is an operator.

Case 2: x = op ywhere x and y are operands and op is an operator.

Case 3: x = ywhere x and y are operands.

Now, we will discuss the following algorithm to draw a DAG which can handle the above three cases.

Step 1	
	If, in any of the three cases, the y operand is not defined, then create a node(y).
	If, in case 1, the z operand is not defined, then create a node(z).
Step 2	
	For case 1, create a node(op) with node(y) as its left child and node(z) as its right child. Let the name of this node be n.
	For case 2, check whether there is a node(op) with one child node as node(y). If there is no such node, then create a node.
	For case 3, node n will be node(y).
Step	

3	
	For a node(x), delete x from the list of identifiers. Add x to the list of attached identifiers list found in step 2. At last, set node(x) to n.

Example:

Let us consider the expression "d=a+b+c+a+b" and it basic block in TAC as follows.

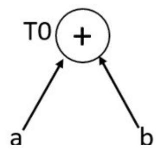
```
T0 = a + b
T1 = T0 + c
d = T0 + T1
```

Fig. 14.8: Basic block in TAC of "d=a+b+c+a+b"

Now let us try to apply the algorithm to generate DAG of Figure 8.

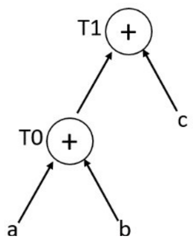
Expression 1:

If we notice the expression T0=a+b, it can be map to case 1 and can apply the Step 2 of the algorithm. According to Setp2 we can generate the following of the required DAG-



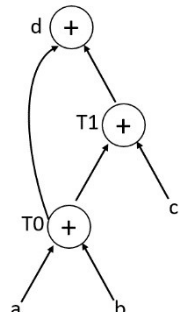
Expression 2:

If we notice the expression T1=T0+C, it can be map to case 1 and can apply the Step 2 of the algorithm. Here since T0 is already generated we shall use T0 and apply Setp2.



Expression 3:

If we notice the expression d=T0+T1, it can be map to case 1 and can apply the Step 2 of the algorithm. Here since T0 and T1 are already generated we shall use T0 and T1 apply Setp2 to get the following final DAG.



14.4 CONCLUSION

Among various techniques employed in compiler design, the Directed Acyclic Graph is considered as efficient for its ability to ensure optimized code generation. By observing the computations represented in the DAG, a compiler can identify and eliminate redundant operations. This elimination of redundant operation not only smooth the code but also significantly reduces both memory usage and computational overhead. As a result, DAGs play a critical role in the overall optimization of code, contributing to more efficient program execution. A solid understanding of DAGs and their application in compiler design serves as a foundation for developing effective optimization techniques. By using the properties of DAGs, compiler developers can implement strategies that enhance performance and efficiency of a compiler.

14.5 CHECK YOUR PROGRESS

1. Which of the following is a common form of intermediate code used in compilers?
   A) Assembly code
   B) Abstract Syntax Tree (AST)
   C) Machine code
   D) Three-Address Code (TAC)

2. In Three-Address Code (TAC), how many operands are typically involved in each instruction?
   A) One

B) Two
C) Three
D) Four

3. What does TAC help in achieving within the compiler process?
   A) Direct machine code generation
   B) Syntax verification
   C) Simplification of complex expressions
   D) Linking external libraries

4. Which representation uses fields such as op, arg1, and arg2?
   A) Quadruples
   B) Triples
   C) Syntax Trees
   D) DAG

5. Which technique is used in DAGs to optimize code?
   A) Identifying redundant computations
   B) Parsing syntax rules
   C) Simplifying tokenization
   D) Specifying grammar rules

6. Which statement is true about the intermediate code generation phase?
   A) It translates machine code directly to high-level code.
   B) It allows code optimization before machine-specific translation.
   C) It translates source code directly to machine code.
   D) It optimizes only the syntax tree structure.

7. What kind of variables are often created in Three-Address Code to store results?
   A) Global variables
   B) Constant variables
   C) Static variables
   D) Temporary variables

8. Which of the following is NOT an intermediate representation format?
   A) Abstract Syntax Tree (AST)
   B) Control Flow Graph (CFG)
   C) Machine Code
   D) Directed Acyclic Graph (DAG)

9. What is the primary purpose of using an intermediate code in compiler design?
   A) To ensure platform independence
   B) To enhance code readability
   C) To increase program execution speed
   D) To simplify lexical analysis

10. Which representation is known for its efficiency in handling the control flow of programs by using pointers to references rather than creating temporaries?
    A) Indirect triples
    B) Quadruples
    C) Triples
    D) Three-address code

14.6 ANSWERS TO CHECK YOUR PROGRESS

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
D	C	C	B	A	B	D	C	A	A

14.7 MODEL QUESTIONS

Short answer type questions

1. What is intermediate code in compiler design, and why is it important?
2. Explain the role of three-address code (TAC) in intermediate code generation.
3. What are the main differences between triples and indirect triples?
4. How does a Directed Acyclic Graph (DAG) help in code optimization?
5. Why is machine-independent intermediate code beneficial for compiler portability?

Long answer type questions

1. How does the intermediate code phase bridge the gap between the front end and back end of a compiler?

2. Discuss Directed Acyclic Graphs (DAGs) and their role in representing expressions within a basic block. Explain how DAGs help in code optimization by identifying common sub-expressions and eliminating redundant operations.
3. What is the purpose of code optimization at the intermediate level? Describe with example.
4. Explain the structure and purpose of three-address code (TAC) in intermediate code generation. How does it facilitate optimization? Illustrate with examples.

14.8 REFERENCES AND SUGGESTED READINGS

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools., Pearson

×××

UNIT: 15
TARGET CODE GENERATION

Unit Structure

15.0 INTRODUCTION

The final phase of the compiler is the code generator. It takes input from the intermediate representation produced by the front end and other information from symbol table. The output produced by the compiler is the target program equivalent to the source program. The target program must preserve the condition of being semantically equivalent to the source program. This means that the target code must possess the same meaning with that of the source. Apart from this, the code produced must be of high quality so that it can make efficient use of the available resources of the destination machine. But the target code must be optimal. In other words, the code must be efficient in terms of register allocation technique. This requires inserting a phase between the intermediate code generator and code generator phases of a compiler. The optimizer takes intermediate codes as its input. It then maps these intermediate codes into some efficient intermediate codes. Finally, code generator accepts these codes in order to produce the target code they occupy less space as well as run in less time.

15.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Describe the concepts behind intermediate code generation.
- Understand code generator in detail.
- Know about the issues associated with target code generation.
- Learn how the target machine model is implemented.
- Have the basic concepts on program and instruction costs.
- Know the various memory allocation techniques.
- Learn the working of run-time environment

- Describe activation records and activation trees in detail.
- Understand how heap management takes place.
- Have knowledge and understanding on garbage collection.

15.2 CODE GENERATOR

The code generator phase represents the back end of a compiler. The position of the code generator is shown in the Fig. 15.1

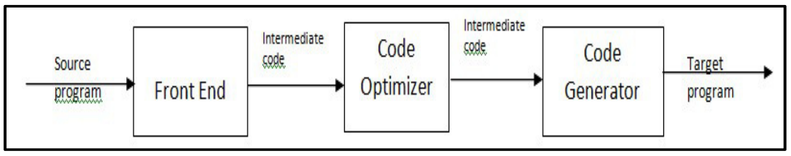


Fig 15.1 Position of the target code generator in a compiler (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

We already have mentioned above that the front end produces the intermediate representations of the source program codes**.** Commonly they are in three-address format. Then these representations are fed to the code optimizer phase to regenerate the same set of codes; but this time in terms of optimized codes. These codes are efficient from the perspective of consumption of resources of the target machine. Finally, the code generator accepts the optimized codes and produces the target code of the same.

The code generator does three major tasks: instruction selection, register allocation and assignment and instruction ordering. The intermediate representations are needed to be implemented in the target machine. Instruction selection involves choosing the appropriate instructions to implement the representations. Now, the next task is to decide which registers to keep as storage. Finally, the instructions are executed in a sequence. Instruction ordering involves deciding the schedule as well as the manner in which those instructions are executed. Sometimes a sequence of instructions is executed together which are called as “basic block”. The intermediate representation has to be partitioned into basic blocks. Later, some local transformations convert the basic blocks into modified basic blocks.

15.3 TARGET CODE GENERATION ISSUES

The ultimate goal of the code generator phase is to produce the correct code. We already have mentioned that the target code must represent the same meaning that source code does. A design goal of an important target code generator involves some important issues to be incorporated into it.

15.3.1 Input to Code Generator

The code generator phase accepts the intermediate representation of the source program produced by the front end of the compiler and also the information of the symbol table. The symbol table is necessary for determining the run-time address of data of intermediate representation.

The intermediate representation includes three-address codes such as quadruple, triple or indirect triple; graphical representation such as syntax tree and DAGs. Apart from this, virtual machine representations such as byte codes and stack-machine code as well as postfix notations also form the input. The codes are low-level with values of names represented by the quantities suitable for manipulation by the target machine especially in terms of integers or real numbers. All the necessary type checking along with type conversion takes place prior to generation of such representation.

15.3.2 Target Program

The most common instruction set architecture of the target machine RISC (Reduced Instruction Set Computer), CISC (Complex

Instruction Set Computer) and stack based. RISC has relatively simple instruction set architecture, many registers, three-address instructions as well as simple addressing modes. CISC machine has variable-length instruction set, few registers with different classes, two-address instructions, a variety of addressing modes and instructions with side effects. In a stack-based machine, operands are pushed onto a stack and operations are performed on the operands at the top of the stack. In order to achieve high performance, the top of the stack is kept as registers. Due to too many swaps, the stack organization has become obsolete.

When the target program is produced in terms of absolute machine language program, they are stored in a fixed location in memory and can be executed immediately. Compilation is quicker in this technique. And when the target program is produced in terms of relocatable machine code, allows subprograms to be compiled separately. The object codes are linked and loaded for execution by a linking loader. The flexibility of being able to compile the subroutines separately provides added advantage to the system though there is an additional expense of linking and loading involved in producing the relocatable object code.

15.3.3 Instruction Selection

The intermediate representation must be mapped into a sequence of target codes. The factors affecting this mapping are:

- The level of intermediate representation
- The instruction set architecture
- The quality of generated code

The level of representation may be high level- the code generator translates each statement into a sequence of machine instructions. Code generation occurs statement by statement. Then optimization takes place on those codes. Instruction selection is strongly dependent on the nature of instruction set architecture of the target machine. An important factor affecting this is- sometimes the target machine does not support each data type in a uniform manner, then the exception must be specially handled. One such exception might be the handling of floating point numbers, for which special registers are used to perform operations on the same. For each three-address statement, we can design some codes equivalent to the target codes. For example, consider the following statements:

```
a = b + c
x = a – y
```

These statements are converted into equivalent target codes as follows:

```
LD     R0, b
ADD    R0, R0, c
ST     a, R0
LD     R0, a
SUB    R0, R0, y
ST     x, R0
```

This scheme produces redundant load and store instructions. For instance, the third statement loads a into R0 and the fourth statement again reloads a into R0. This in turn leads to inefficient code by compromising with the quality of the code.

15.3.4 Allocation of Registers

In 15.4.3, we discussed that during target code generation, registers are used for allocation and storage. Registers are the fastest memory unit of a computer which makes the execution faster. The most challenging task in code generation is deciding which register to hold what value. Values not held by registers are assigned in memory. Execution using registers are more aster than memory execution. Register utilization is associated with two concepts:

- **Register allocation** is associated with selection of set of variables which will be held in registers at each point in the program.
- **Register assignment** is the problem of picking a specific register into which a variable will be assigned. This assignment is crucial and challenging with a single-register system.

15.3.5 Evaluation Order

Evaluation order plays an important role during generation of target code. It is the order in which computations are performed. Evaluation order affects the performance of a system and picking the best order is quite challenging.

CHECK YOUR PROGRESS- I

6. The optimizer takes _______________ as its input.
7. The linear intermediate representation includes ___________.
8. The graphical intermediate representation includes ___________ and ________.
9. ________________ is defined as the selection of set of variables which will be held in registers at each point in the program.
10. _______________ is the problem of picking a specific register into which a variable will be assigned.

15.4 TARGET MACHINE MODEL

A good code generator needs to have understanding of the target machine and its instruction set. As mentioned previously, machines having three-address codes with load and store instructions, conditional and unconditional jump instructions as well as computational instructions form the basis of target machines. A simple target machine models a byte addressable machine with n numbers of general purpose registers. The machine would have a simple set of instruction sets with its operands being integers. The assembly code consists of an optional label, an operator, followed by the target variable and finally followed by a set of source operands.

- The **load instruction** is of the form ***LD des, addr*** loads the contents of location addr into dest. The variable dest is also referred to as location. In other words, it is an assignment operation of the form **dest = addr.** Another form of using this instruction is **LD R, x** which loads the contents of x into register R. Then the instruction **LD R0, R1** copies the contents of register R1 into register R0. This is one kind of register-to-register transfer instruction.
- The **store instruction** is of the form **ST x, R** stores the contents of register R into the variable x. This is an assignment instruction of the form **x = R**.
- The **computation instruction** of the form **OP dest, src1, src2**. The operation is identified by OP which might be ADD or SUB etc. The operands are specified by scr1 and scr2. Finally, the result is stored in location dest. For example, the instruction ADD R0, R0, a adds the contents of register R0 and value of location a and stores the result back to register R0. Similarly, the same operation can be implemented using registers only. Therefore, the instruction ADD R0, R0, R1 does the addition of registers R0 and R1 and stores the result of addition back to register R0.
- The **unconditional jump** instruction jumps to another section of the program with a particular label. Jumping may occur without any condition. Like the instruction **BR L** branches control to the instruction with label L.

- Finally, the **conditional jump** instruction of the form **Bcond R, L** jumps to the instruction labeled L upon fulfillment of condition cond on register R. An example of such kind may be BGTZ R, L causes the control to jump to instructions with label L if the value of register R is greater than zero. Conditional jump occurs when the condition is true. But, if it is false, control passes to the instruction following the condition.

We already have mentioned that the target machine contains a variety of addressing modes. Some of them are:

- A location for a variable name x is designated by the memory location reserved for x itself.
- The content of memory location denoted by a(r) is computed by adding the l-value of variable a and the value of register r. The location is an indexed address and therefore, the load instruction **LD R1, a(R0)** is computed by performing the operation **R1 = contents(a + contents(R0))**, where **contents(R0)** denotes the contents of register R0. Array access is efficient using this mode where a represents the base address of the array and r denotes the number of bytes to move in order to reach a particular position of the array.
- A memory location can also be indexed by a register. The instruction **LD R1, 100(R2)** derives the value of R1 by setting **R1 = contents(100 + contents(R2))**. That is, the value of R1 is set by adding 100 to the contents of register R2.
- Another kind of indirect addressing mode that a target machine model allows: *r means the memory location found in the location which is represented by the contents of a register. Likewise, the value of *100(r) is obtained by deriving the contents found the location obtained by adding

100 to the contents of register r. In this way, we can take the example of an instruction **LD R1, *100(R2)** which derives the contents of R1 by setting **R1 = contents(contents(100 + contents(R2)))**.

- Finally, another form of immediate instruction mode may also be allowed; but this time it is constant. Like for example, **LD R1, #100** is an instruction which loads integer into register R1. Similarly, the instruction **LD R0, R0, #100** adds integer 100 into register R0. The constant is always preceded by the symbol #.

Example 1: The three-address code x = x + 1 can be implemented using the following machine instructions:

```
LD     R0, a

ADD    R0, R0, #1

ST     a, R0
```

However, if the target machine has the "increment" instruction (INC), then the three-address code can be implemented in a single instruction **INC a** more efficiently rather than using load and store instructions.

Example 2: The three-address code x = y – z can be implemented using the following machine instructions:

```
LD     R1, y
LD     R2, z
SUB    R1, R1, R2
ST     x, R1
```

Example 3: Suppose, A is an array with elements of 8-byte values. The indexing begins with 0. The three-address code x = A[i] can be executed using the following machine instructions:

```
LD   R1, i
MUL  R1, R1, 8
LD   R2, A(R1)
ST   b, R2
```

The first step loads the value of I into register R1. The second step computes 8i and the third step places the value of the i th element of A into register R2. Finally, the result is placed in variable b.

Example 4: Suppose, A is an array with elements of 8-byte values. The three-address code A[i] = x can be executed using the following machine instructions:

```
LD   R1, x
LD   R2, j
MUL  R2, R2, 8
ST   A(R2), R1
```

Example 5: The three-address code for the pointer statement x = *y can be implemented using the following machine instructions:

```
LD   R1, y
LD   R2, 0(R1)
ST   x, R2
```

Example 6: The three-address code for the pointer statement *y = x can be implemented using the following machine instructions:

```
LD   R1, y
LD   R2, x
ST   0(R1), R2
```

Example 7: The three-address code for the conditional jump statement if x < y goto L can be implemented using the following machine instructions:

```
LD   R1, x
LD   R2, y
SUB  R1, R1, R2
BLTZ R1, M
```

M is the label that represents the first machine instruction of the three address code which has label L.

The target language is also dependent on the cost associated with compilation of programs and instructions.

CHECK YOUR PROGRESS- II

6. The assembly code consists of an optional ___________.
7. The load instruction loads the contents of location address into __________.
8. The store instruction stores the contents of ______ into a variable.
9. The unconditional jump may occur without any_______.
10. Conditional jump occurs when the condition is _________.

15.5 PROGRAM AND INSTRUCTION COSTS

A very important aspect associated with target code generation is the cost of compiling and running a program. It is a complex problem and some of the common cost measures are the length of compilation time and size, running time taken by the target program as well as its power consumption. The cost of an instruction is assumed to be one associated with the costs occurring with addressing modes of the operands. Addressing modes involving registers cost zero. On the other hand, addressing modes involving memory location or constant have an additional cost one as such operands have to be stored in the words that follow the instruction.

- The instruction LD R0, R1 copies the contents of register R1 into register R0. As the instruction does not involve additional memory, the instruction incurs a cost of one.
- The instruction LD R0, a loads the contents of memory location a into register R0. The cost of this instruction is two since the address of memory location a is in the word that follows the instruction.
- Similarly, the instruction LD R0, *100(R1) loads the value contents(contents(100 + contents(R1))) into register R0. The cost of this instruction is three because constant 100 is stored in the word that follows the instruction.

15.6 ADDRESSES IN THE TARGET CODE

The names in the intermediate representation can be converted into corresponding addresses in the target code by means of two allocation techniques. The static and stack allocation techniques do this generation for simple procedure calls and returns. Each program run in its own logical address space and the space is partitioned into four code and data areas.

- The size of the target code is determined during compile time. Therefore, the size of this area called Code is fixed and it holds the executable target code.
- The sizes of the global constants and other data used in a program are generated by the compiler. It is a statically determined area called Static which holds these data.
- During execution of programs, objects are allocated and freed. This makes the area to be expanded or shrinked. This dynamically managed area is termed as heap memory allocation which holds the data objects.
- Apart from these, there is a Stack which holds the activation records which are created with each procedure call and destroyed when the call returns. Stacks are also created during run time and hence it is dynamically managed.

15.6.1 Static Allocation Technique

The following three-address codes must be considered in order to generate codes for simple procedure calls and returns.

- Call *callee*
- Return
- Halt
- Action

Symbol table plays an important role during code generation as it stores the information of each identifier used in a program. The generator determines the size and layout of the activation record using this information. The activation record holds the return address on a procedure call and how control returns once the call is over.

The intermediate statement **call** ***callee*** can be implemented using the following two target machine instructions.

```
ST      callee.staticArea, #here + 50
BR      callee.codeArea
```

The store instruction saves the return address at the beginning of the activation record for callee. The branch instruction transfers control to the target code destined for the procedure callee. The attribute *callee.codeArea* is a constant that refers to the address of the first instruction of callee and it resides in the Code area of the run-time memory. Operand #here + 50 contains the return address with #here referring to the address of the current instruction. The procedure call ends with a return to the calling procedure. A simple jump instruction transfers control to the return address saved at the beginning of the activation record for callee.

```
BR      *callee.staticArea
```

Static allocation is possible when the compiler knows the size of data objects during compile time.

15.6.2 Stack Allocation Technique

Creation of data objects is done during run-time. These are pushed onto the stack using Last-In-First_out (LIFO) technique. The position of an activation record for a procedure is not known until run-time. The position is usually stored in a register. This technique uses relative addresses for storage of activation records. Relative addresses can be taken from any known position in the activation record. A special register called Stack Pointer (SP) is used to point to the beginning of the activation record on the top of the stack. When call to a procedure occurs, the caller increments SP and transfers control to the called procedure. After execution is over and control returns to the called procedure or caller, SP is again decremented. This results in deallocation of activation record of the called procedure. The corresponding codes for the same are as follows.

```
LD      SP, #stackStart
HALT
```

The first code sets SP to the start of the stack area in memory. A procedure first increments SP, saves the return address and transfers control to the target code area of the called procedure.

```
ADD     SP, SP, #caller.recordSize
ST      *SP, #here + 20
BR      callee.codeArea
```

The operand *#caller.recordSize* is a constant that represents the size of the activation record. Therefore, ADD instruction makes SP to point to the next activation record. The operand #here + 20 in ST instruction stores the return address of callee, that is, the address of the instruction that follows BR and is saved in the address pointed by SP. The called procedure transfers control to the return address using the code

```
BR      *0(SP)
```

The return sequence consists of two parts. Parallely, there are two indirections involved: 0(SP) contains the address of the first word in the activation record and *0(SP) represents the return address. Finally, the value of must be restored to its previous value of SP.

For this purpose, a subtraction operation has to be performed so that SP points to the beginning of the activation record of the caller.

```
SUB    SP, SP, #caller.recordSize
```

15.6.3 Run-time Addresses for Names

Names of intermediate representations are determined by storage allocation strategy and layout of local data in an activation record for a procedure. We already have mentioned that names of three-address codes are contained as entries in a symbol table. This technique has the advantage that the front end need not be changed even when the compiler is ported into a different machine requiring different run-time organization. Names are accessed by codes to access storage locations. A three-address code statement like y = 5 copies 5 into the location specified by y. Now, assume that the symbol table entry for y contains the relative address 20. Further, if y resides in a statically allocated area that begins with address base, the actual run-time address for y would be **base + 20**. The compiler can eventually determine the value of **base + 20** during compile time. But, the position of static area may not be known when intermediate code to access name is generated. In such situation, the three-address code must compute **base + 20** during code generation phase or during loading phase by the loader or before the program runs. Therefore, the statement y = 5 would be translated into

```
static [20] = 5
```

If the static area starts with address 200, the target code for the statement would be

```
LD      220, #5
```

CHECK YOUR PROGRESS- III

11. The cost of an instruction is assumed to be __________ associated with the costs occurring with addressing modes of the operands.
12. Addressing modes involving registers cost __________.
13. The instruction LD R0, R1 incurs a cost of ___________.
14. What are the two allocation techniques through which names of intermediate representation converted into addresses in the target code?
15. The store instruction saves the return address at the beginning of the activation record for ___________.
16. The ____________instruction transfers control to the target code destined for the procedure callee.
17. Creation of data objects is done during ____________.
18. A special register called ___________ is used to point to the top of the stack.
19. Names of three-address codes are contained as entries in a ____________.

15.7 RUN-TIME ENVIRONMENT

The compiler must accurately implement the abstractions prevailing in the source language like names, data types, scopes, scopes, operators, procedures, parameters and flow-of-control constructs. These implementations require using both the operating system and other system softwares. For this purpose, the compiler creates and manages a run-time environment in which the target programs are supposed to be executed. This environment deals with a variety of issues such as mechanisms to access variables, allocation and layout of storage locations for the source program objects, linkages between procedures, mechanisms for passing parameters, interfaces to the operating system, input/output devices and other programs.

It is already mentioned that the target program runs in its own logical address space. The operating system maps the logical addresses into physical addresses of memory. The management and organization of logical address space is shared among the compiler, operating system and target machine. The object code is spread across the space in terms of data and program areas. These areas are assumed to be represented in terms of blocks of continuous bytes. An organization of the same for a C++ compiler has been shown below:

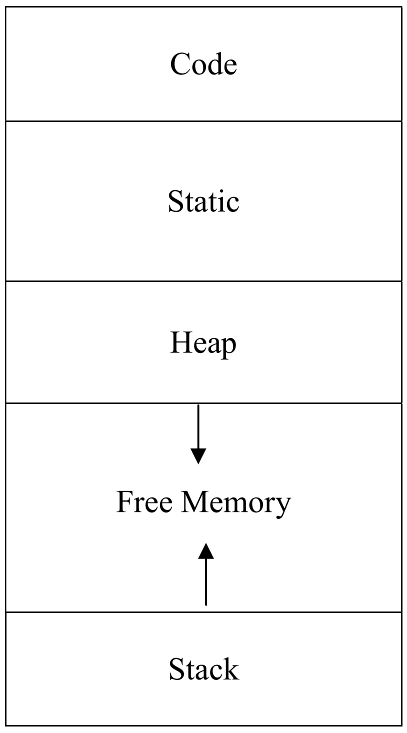


Fig. 15.2 Subdivision of run-time memory into code and data areas (Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

The amount of storage required for a name is determined by its type. The basic data types, integer, character or float are stored in terms of bytes. Composite data types like arrays or structures are allocated with aggregate storage of all its component types.

The size of the generated target code is determined during compile time and is placed in a statically determined area called Code which we have discussed in the last section. The program may contain some data objects such as global constants or compiler generated data may also be placed in a statically determined area called as

Static. Data objects must be statically allocated as much as possible since the addresses data objects can be converted into target code during compile time. The other two Stack and Heap are dynamic. They maximize the utilization of space during run-time and change when the program executes. With each procedure call, an activation record is generated. These activation records are stored inside the stack. It stores information such as the values of program counter and other machine registers when a call to the procedure is made. When control returns back from the called procedure to the activation of the calling procedure, the values of machine registers are reloaded and also setting the program counter to the point immediately after the call. We shall now study various memory management techniques for allocation and deallocation of storage.

15.7.1 Static vs. Dynamic Storage Allocation

The allocation of data to memory locations is a major issue in run-time environment. One such issue includes when the program contains variables with the same name in different parts of the program. Therefore, they refer to multiple locations at run time. When the topic of storing program text comes, the allocation strategy is static, that is, the decision takes place during compile time. But, when the issue of execution of programs happens, the decision has to be dynamic, that is, during run-time, the allocation of memory takes place. Stack allocation is a dynamic storage strategy which holds the activation records of each procedure occurring in a program. The local data declared in a procedure are assigned storage in a stack. Another virtual memory area used to allocate storage to data objects and other data elements when they are created is called as heap storage. When the data are invalidated, the allocated storage is returned. Garbage collection is technique through which the run-time system detects the useless data elements and reuses their storage. Though complicated, automatic garbage collection is much essential today for proper management of heap.

15.8 STACK ALLOCATION OF SPACE

We already have mentioned that every time a procedure is called, space for local variables is pushed onto the stack and when the procedure terminates, the space is released from the stack. Almost all programming languages using user-defined procedures, functions or methods use a segment of their run-time memory as stack. Stack allocation allows space to be shared by procedures called at different timestamps.

15.8.1 Activation Trees

In any programming language, the concept of nested procedures exists. When a procedure calls another, nested procedure comes into play at different time. For example, a procedure p may call another procedure q. Correspondingly; activations are also nested in time. Therefore, activation of q must end before activation of q ends. But, termination of activations takes place with three common cases.

1. Activation q completes its normal execution and terminates. In this case, control resumes to the instruction just after the point of p at which the call to q was made.
2. Sometimes, the activation of q or some procedure q called is unable to continue its normal execution. In such situation, activation q aborts and with q, p also aborts.
3. Sometimes, activation q terminates because it encounters an exception which q cannot handle. Procedure p may handle the exception, which results in termination of activation q and continuation of activation p. If p is unable to handle, then both p and q terminates at the same time. In such

situation, the exception would be handled by some other activation of the procedure.

The activations of procedures during execution of a program can be represented using a tree, called an activation tree. Each node represents an activation and the root designates the activation of the main procedure that initiates the execution. Nested activations correspond to the children of a node. For example, if procedure p calls procedure q, then q becomes the child of p. Activations are shown in the order in which they are called from left to right. An activation at the child node must finish before the activation to its right begins.

15.8.2 Activation Records

A control stack is a run-time stack whose primary task is to manage the procedure calls and returns. It contains the activation record of the procedures being called. The root of the activation tree resides at the bottom while the most recently called activation at the top. The entire sequence of activation records corresponds to the path of the activation tree. The contents of an activation record vary with kinds of languages.

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Fig 15.3 A common activation record
Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman

During execution of expressions temporaries are generated. Local data refers to those data which belong to the procedure being called. The status of the machine just before call to a procedure must be saved somewhere. This information typically includes the return address and the contents of the registers used by the calling procedure. These values must be restored when return from the procedure occurs. Activations may be nested within each other. Access links represent a chain from the activation record from the top of stack to the one at the lowest. Like for instance, if q is a procedure which resides within procedure p, then the access link in any activation of q points to the most recent activation of p. Nesting depth of p is exactly one less than that of q. A control link always points to the activation record of the caller. Procedures may return value(s). This value is placed in the returned value section. The calling function uses the actual parameters. These values are not placed in the activation record but rather in registers.

15.8.3 Calling Sequences

A calling sequence is a set of codes that allocates an activation record onto a stack and accordingly enters information into its fields. In other words, procedure calls are implemented using a calling sequence. Similarly, a return sequence reloads the machine to its previous state from which the procedure was called. The calling procedure can continue its normal execution after the return sequence is called. If the caller procedure calls another from n different positions, then the calling sequence of the caller is generated n times. And the calling sequence for the callee is generated once only. Calling sequence and layout of activation record are different. While designing the both, the following principles must be followed.

- Values that pass between caller and callee are placed at the beginning of the callee's activation record. This is makes them closer to the caller's activation record. The basic idea behind this principle is that the caller can compute the values of the actual parameters and place them at the top of its activation record. This debars creation of entire activation record of the callee.
- Control link, access link and machine status fields are generally fixed-length items. They are mainly placed in the middle of the activation record. For each call, if same set of components represent the machine status, then the same set of code can do saving and restoring of these components. This concept is efficient from the perspective of debugging of programs because once the machine status information is standardized; it is easier for the debugger to check the contents of the stack if error occurs.
- Most local variable's size is derived simply by looking the type of the variable. They are fixed length data and can be examined during compile time. However, some local variable's size cannot be derived until it executes. One such example includes the dynamic array whose size is determined by one of the parameters of the callee procedure. Such kind of items whose sizes are not known early are placed at the end of the activation record.
- A general approach to find the top-of-stack is to point to the end of fixed length fields of the activation record. Fixed-length data has already been mentioned previously. They are accessed by fixed offsets. The variable length fields of the activation record are generally placed above the top-of-stack.

The caller and callee must cooperate with each other in managing the stack. The top-sp is a register that points to the end of the machine status field of the current activation record. The caller knows this position of callee's activation record and caller sets top-sp before control passes to the callee. The calling sequence between the caller and callee are as follows:

1. Caller evaluates the actual parameters.
2. Caller stores the return address and the previous value of top-sp into the callee's activation record. Then the caller increments top-sp to move it past the caller's local data and temporaries as well as callee's parameters and status fields.
3. Register values and other status information of the callee are saved.
4. Local data of the callee are initialized and execution of the same begins.

The return sequence works in the following manner:

1. The callee places the return value next to the actual parameters.
2. The callee restores top-sp and other registers and branches to the return address that the caller had placed.
3. The caller knows the return address relative to the current value of top-sp and uses it.

15.8.4 Variable length Data on Stack

A procedure may contain local data objects whose sizes are known during run-time and thus may be allocated onto the stack. In modern compilers, such objects are allocated space in the heap. To avoid garbage collection, it is always preferable to place objects, arrays and other structures onto the stack.

A procedure can have local arrays whose sizes cannot be determined at compile time. The activation record stores only a pointer to the beginning of each array rather than storing the arrays. The target code can access the elements of the array through these pointers. If q is a nested procedure of the parent procedure p, then the activation record for q begins after the arrays of p and the arrays of q are placed beyond that.

Data accessing on stack can be done through two pointers: top and top-sp. The actual top of the stack is denoted by top and the next activation record begins at this position. Similarly, top-sp is used to locate the local, fixed-length fields of the top activation record. It is already mentioned that top-sp points to the end of the machine status field of q's activation record. The control link field for q leads us to the position of p's activation record where top-sp pointed when p was on the top. When q returns, top-sp is restored from the control link of q's activation record. And the new value of top is set as top-sp minus the length of machine, control and access link, return value and parameter fields of q's activation record. This length can be derived during compile time.

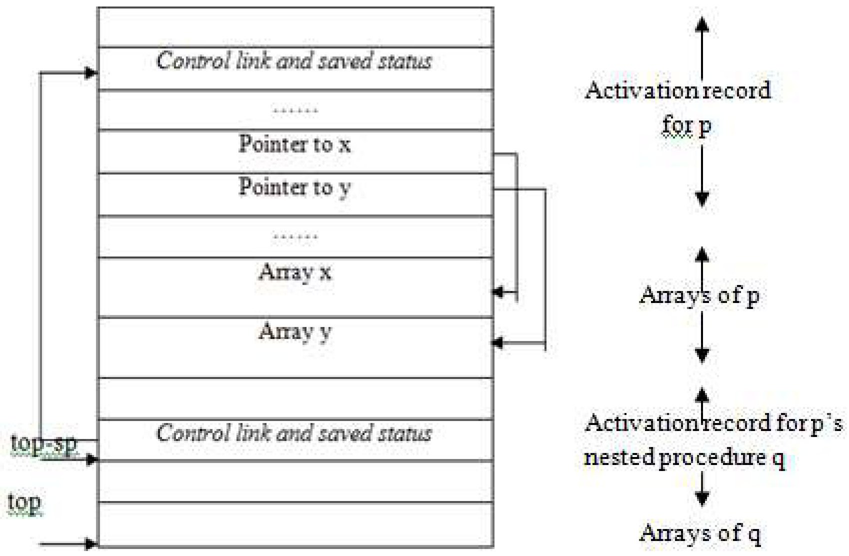


Fig 15.4 Allocation of dynamically allocated arrays
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

15.9 HEAP MANAGEMENT

Unlike stack, heap is that portion of memory that stores data indefinitely. Data gets erased only when the program explicitly deletes it. Sometimes, languages like C++ or Java enable us to create data objects through operators like new and also pass from one procedure into another. They continue to exist once the procedure to which they were local. Such objects are stored in a heap.

A very important aspect of heap management is the allocation and deallocation of space within the heap. The memory manager is a part of the operating system that allocates and deallocates space within heap. It serves as an interface between the application program and the operating system. There are some keywords in C or C++ which do the removal of chunks from the heap.

Apart from keeping track of all free spaces of heap storage, memory manager performs two basic functions.

- **Allocation:** During execution of programs, there is a need for storing the variables and objects residing in the program. The memory manager finds the continuous chunk of free space enough to holds the program in the heap and allocates the same to the program. If no such space is available, the heap size is increased by getting contiguous bytes of virtual memory from the operating system.
- **Deallocation:** The allocated space is to be returned to the available pool of free spaces when the execution of the program is over. The idea behind this is to reuse the space so that other programs can be allocated the same.

Memory management is complicated as the allocation requests for programs are not of same size. Apart from this, the storage allocated first is not released first. So, allocation and deallocation is not foreseeable. Thus, the memory manager's major task includes allocation of memory of any size to the programs with no prediction on when the memory would be released. Memory managers must possess some properties as mentioned below:

- **Space Efficient**: One purpose of memory manager is to minimize the total heap size required by a program. Space efficiency is a must in heap memory to avoid fragmentation. Space efficiency is concerned in heap as larger programs are run in a fixed virtual address space.
- **Program Efficient**: The time to execute an instruction is widely dependent on where the objects are placed in memory. With the better use of space, the program can run faster. The time taken to execute an instruction varies with respect to the time taken to access various parts of the memory.

15.9.1 Memory Hierarchy of a Computer

All modern computers arrange their memory system in terms of a hierarchy. The hierarchy of memory consists of cascade of storage elements. The smaller and faster storage elements remain closer to the processor and the larger and slower ones are further away. Typically, the smaller and faster memory includes registers. In fact, they are the fastest memory. Next, there exist one or more levels of cache, usually made of static RAM. The next memory module is the physical or main memory, whose size is more than that of cache memory. Then the physical memory is followed by virtual memory, implemented on disks most commonly. The size of this memory is the largest, typically in terabytes. During memory access operation, the memory at the lowest level looks for data. If not available there, the machine looks for the same in the next higher level. Large blocks of data are assigned to the slower level of the hierarchy.

Between main memory and cache, data is transferred in blocks. And between virtual memory and main memory data is transferred in terms of blocks of pages.

A program may contain hundreds of instructions. But many of them are seldom executed. With change of requirements, programs evolve and some of the codes are never used. Apart from this, the program spends most of its time executing innermost loops and recursive procedures of the program. Therefore, only a fraction of the code is actually executed. To lower the average memory access time of a program, the most commonly used instructions and data are placed

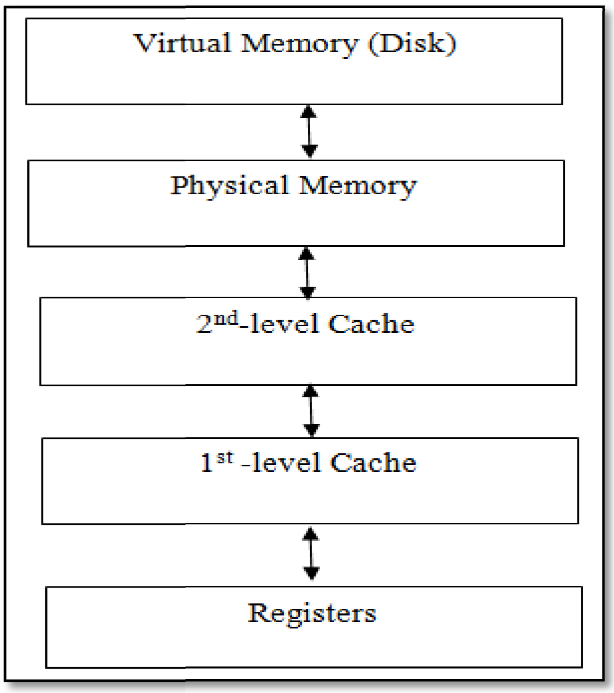


Fig 15.5: Memory hierarchy of a typical computer
(Source: Compilers- Principles, Techniques & Tools by Aho, Lam, Sethi, Ullman)

inside the fast and small storage. While the slow and large storage keeps the less commonly used instructions and data. This happens when programs spend most of their time in executing a small fraction of code as well as data. That is, the same locations may be accessed repeatedly. If locations accessed are likely to be accessed within a short period of time. This is termed as temporal locality of the program. In some cases, memory locations closer to the locations accessed are likely to be accessed within a short period. Locality of such kind is termed as spatial locality. Programs may showcase both temporal and spatial locality. We must keep on adjusting the contents of the fastest cache dynamically as they might be of size bigger than the size of the available storage. However, it is not that possible to say which chunk of code will be executed heavily simply by looking at the code.

The locality principle works well when the most recently used instructions are placed inside the cache. When an instruction is executed, it is highly probable that its immediate instruction will also be executed next. This is spatial locality which can further be improved by keeping the contiguous instructions in a basic block and then loading them on the same page or same cache line if possible. Further, we can improve the temporal and spatial locality by changing the data layout or the order of computation.

15.9.2 Reducing Fragmentation

A heap is basically a continuous memory of free space when no program is executing. When allocation and deallocation of storage takes place in heap, the free chunks are not contiguous and they leave holes in the heap. When an allocation request comes, the memory manager looks for the largest hole enough to hold that program. Once, the perfect hole of exactly the same size is not found, the hole has to be split. This splitting leaves a small hole in the heap. Similarly, during each deallocation of memory, the free spaces are added to the pool of free space. This process leaves the memory fragmented. This consists of large number of smaller, non contiguous holes. The implication of this technique is that it involves creating holes not large enough to contain any program in future.

This problem of fragmentation can be reduced effectively by taking some good strategies. One strategy might be allocating the requested memory to the smallest available hole large enough to hold. This is best-fit algorithm which assigns that amount of space required by a program. Alternatively, the first-fit strategy places the program to the first available storage in which it fits. The advantage of this technique is that it takes less amount of time for allocation.

The free space chunks are separated into bins according to their sizes. Each bin is a multiple of 8-byte chunk; that is from 16 bytes to 512 bytes. The chunks within these bins are arranged in terms of their size. The bin that can hold the desired page is found out. Then either the best-fit or first-fit strategy is used to find the sufficiently large first chunk or the smallest chunk respectively. Sometimes, the fit may leave some extra space in the chunk which needs to be placed in a bin with smaller sizes. Best-fit strategy tries to improve space utilization.

Management of free space is a very important of memory manager. When memory is deallocated manually, it is the task of the memory manager must make the free chunk allocated again. It should be possible to combine the chunk with the adjacent chunk of the heap and form a larger chunk. It is possible to include a larger size program into a larger chunk or small chunks of equal size can be run in a large chunk The process of combining these small chunks are termed as coalesce.

If chunks of one fixed size is kept in a bin, then it is better not to coalesce the adjacent blocks into one big size. Rather, a simple mechanism can be followed by keeping all the chunks of fixed size and keep a bitmap of allocation/deallocation scheme. The bit value 1 indicates that the chunk is occupied; whereas 0 indicate it is free. A deallocation of chunk changes bit 1 to 0. Similarly, allocation requires finding a chunk with 0 bit and changes it to 1.

There are two data structures that support coalescing of adjacent free blocks.

- **Boundary Tags:** Each chunk consists of two ends- low and high. The chunk may be free or allocated. This information is kept by using free/used bit for each chunk. The total chunk size is also kept adjacent to each free/used bit.
- **Doubly Linked, Embedded Free List:** Free chunks are linked with a doubly linked list. Pointers are maintained to keep track of the next adjacent block in the list. The list must accommodate two boundary tags and two pointers; though the object is a single byte. The list must be ordered so that it can implement best-fit placement strategy.

15.9.3 Manual Deallocation and Associated Problems

Manual deallocation involves the programmer to explicitly deallocate data. Languages such as C or C++ provide such facility. The data that is no longer be used should be deleted and accordingly the memory must be released. Conversely, any storage that may be required to be referenced should not be deleted. However, there are some difficulties to enforce these properties.

There are two common problems that occur during manual deallocation of storage: 1. Referencing data which are already deleted. This is called dangling-pointer-dereference error. 2. Failing to delete data that cannot be referenced. This is called as memory-leak error.

Let us discuss the second one first. It is unpredictable that a program will never refer to some storage in the future. Therefore, such storage should be deleted. The execution speed of the program is highly affected by the memory leaks. However, the long-running programs such as the operating system or the server may not tolerate such leakage.

There is a way to resolve this problem. Automatic garbage collection is a technique through which garbage can be deallocated in order to get rid of memory leaks. If an object will never be referenced in the future, then those references must be deleted. Consequently, the objects can also be deallocated automatically.

Now, the second common mistake of dangling pointer dereference may occur. While deleting storage, it may happen that the corresponding data may get referenced. Dangling pointers are the pointers which point to the deallocated storage. Once data item is reallocated to the storage, any kind of operation can be performed through the pointers. Referencing operations such as read, write or deallocate via pointer and tries to use the object it points to is termed as dereferencing the pointer. Reading through dangling pointer return an arbitrary value whereas writing arbitrarily changes the value of a variable. Deallocation of dangling pointer's storage means that storage of the new variable can be assigned to another one leading to conflict of actions on the old and new variables. After reallocation, dereferencing a dangling pointer may create a program error. An error may be referencing an illegal address. Errors of such kind may include dereferencing null pointers and accessing an out-of-bounds array element. There are conventions and tools that handle such difficulties in managing of memory.

CHECK YOUR PROGRESS- IV

20. The operating system maps the logical addresses into ________of memory.
21. The amount of storage required for a name is determined by its __________.
22. Stack allocation strategy holds the ____________of each procedure occurring in a program.
23. A _________ is a set of codes that allocates an activation record onto a stack and accordingly enters information into its fields.
24. The portion of memory that stores data indefinitely is ___________.
25. The fastest memory is _____________________.
26. Between main memory and cache, data is transferred in________________.
27. Between virtual memory and main memory data is transferred in terms of blocks of _________________
28. Referencing data which are already deleted is called _________________.
29. Failing to delete data that cannot be referenced is called as ________________ ___________.

15.10 GARBAGE COLLECTION

Most programming languages provide the facility of automatic garbage collection. It is a technique through which the chunks of memory holding objects that are no longer be accessed by a program can be reclaimed. Objects used in a program have a type and it determines the size of the objects. The type information also tells which components of an object refer to other objects. References include objects are always pointed to the address at the beginning of the object. Thus, all references to an object have the same value and this makes the object unique.

A mutator is a user defined program which creates objects by acquiring space from the memory manager. References to existing objects are created and dropped using this program. When mutator cannot reach any object, then the object becomes garbage. The garbage collector finds such unreachable objects and hands over their occupied space to the memory manager.

Type safety is a basic component needed by an automatic garbage collector. This indicates whether a given data element or component of a data element could be used as a pointer to a chunk of allocated memory space. A type safe language is the one for which the type of its data components can be determined during compile time. However, there are programming languages like Java whose types cannot be determined during run time rather than compile time. Such kind of languages is called dynamically typed language. An unsafe language is neither statically nor dynamically type safe. Unsafe languages are bad candidates for automatic garbage collection. A program can refer to any memory location at any time. This in turn results in considering each memory location manipulated arbitrarily and finally storage can never be safely reclaimed. Similarly, arbitrary arithmetic operations on pointers create new pointers and arbitrary integers can be casted as pointers. Examples of such kind include languages like C or C++.

There are some performance matrices that must be considered while designing a garbage collector.

- Garbage collection is considered to be slow. Eventually, it increases the total execution time of a program. This causes a drawback to the system. Therefore, the overall execution time is greatly affected the garbage collector takes its leverages on the memory subsystem.
- Space usage plays a vital role in designing a garbage collector. One important thing that must be considered while

designing a garbage collector is how it can avoid fragmentation and makes the best possible use of memory.

- Since, allocation and deallocation of data objects happens through mutators, the garbage collectors cause mutators pause suddenly for an extremely long time. It is desirable to minimize such pause time. Apart from that, there are some sensitive applications which are required to be executed during real time must suppress garbage collection so that the application or task completes smoothly. Garbage collection is less practiced in real-time operations. Therefore, pause time is important in the design of garbage collector.
- The running time cannot solely determine the speed of a garbage collector. Data locality, which we had discussed section 5.10.1 pertaining to the mutator program, is controlled by the garbage collector. The temporal locality of the mutator program is improved by space free up and reusing the space. Similarly, the mutator's spatial locality is enhanced by data relocation in the same cache or page.

This is worth mentioning here that allocation of smaller objects must not incur large overhead. Moreover, relocation costs more when it deals with large objects rather than the smaller ones.

A vital point that should be understood is referred to as the root set. Root set consists of set of all the data that can be accessed directly by a program without having to dereference any pointer. By traversing the root set, the program may reach any member at any time. However, reachability is complex when the program is optimized by the compiler. Compilers may keep references variables in registers. And sometimes, the compiler may manipulate the memory addresses in order to speed up the code. When a program executes, the set of its reachable objects changes. As a new object is created, the set increases and shrinks when it becomes unreachable. Once an object is unreachable, it never becomes reachable again. To change the set of reachable objects, the mutator performs some specific task.

- Object allocation is something that is returned by the memory manager. It returns the reference to a newly allocated chunk of memory. This operation in turn adds members to the set of reachable objects.
- References to objects are passed from the actual parameters and from the returned result. Such references to objects are reachable.
- Reference assignments involving two references u and v of the form u = v have two effects. First, u refers to the object referred to by v. Until the time u is reachable, the object is refers is also reachable. Second, if the original reference to u is lost, the object is unreachable. As long as an object is reachable, all objects that are reachable only through references contained in that object are also reachable. However, the converse is not possible.
- When a procedure exists, the local variables of the procedure also exist in the stack. Whenever, the procedure end, the variables are also popped off. If any reachable reference was there, the corresponding object would become unreachable. Consequently, more object references may get lost.

There are two basic ways to find the unreachable objects. Transitions happen to take place when reachable objects become unreachable. These transitions are caught in order find such unreachable objects. Another way is to periodically keep track of all the objects reachable and then find all the objects unreachable. It is termed as Reference counting which keeps a count of the references to an object. As mutators change the set of reachable objects, the count may go to zero and the object becomes unreachable. The reference count is 1when a new object is created. The reference transitivity includes the task of labeling the objects that are reachable. Initially, all objects in the root are marked as reachable. Periodically, the reachable set is computed which gives many unreachable objects. This eventually leads to locating some free storage.

15.11 SUMMING UP

- The code generator takes input from the intermediate representation produced by the front end and other information from symbol table. A good code generator needs to have understanding of the target machine and its instruction set.
- The output produced by the compiler is the target program equivalent to the source program. The target program must preserve the condition of being semantically equivalent to the source program.
- The front end of a compiler produces the intermediate representation such as three-address format of the source program codes**.** These representations are fed to the code optimizer phase to regenerate the same set of codes; but this time in terms of optimized codes.
- The code generator does three major tasks: instruction selection, register allocation and assignment and instruction ordering.

- The intermediate representation includes three-address codes such as quadruple, triple or indirect triple; graphical representation such as syntax tree and DAGs.
- The code generator translates each statement into a sequence of machine instructions. Code generation occurs statement by statement.
- Register allocation is associated with selection of set of variables which will be held in registers at each point in the program. Register assignment is the problem of picking a specific register into which a variable will be assigned.
- The cost of compiling and running a program is a complex problem and some of the common cost measures are the length of compilation time and size, running time taken by the target program as well as its power consumption.
- Addressing modes involving registers cost zero. On the other hand, addressing modes involving memory location or constant have an additional cost one as such operands have to be stored in the words that follow the instruction.
- The names in the intermediate representation can be converted into corresponding addresses in the target code by means of two allocation techniques. The static and stack allocation techniques do this generation for simple procedure calls and returns.
- During execution of programs, objects are allocated and freed. This makes the area to be expanded or shrinked. This dynamically managed area is termed as heap memory allocation which holds the data objects.
- Apart from these, there is a Stack which holds the activation records which are created with each procedure call and destroyed when the call returns. Stacks are also created during run time and hence it is dynamically manged. A

special register called Stack Pointer (SP) is used to point to the top of the stack.

- The compiler must accurately implement the abstractions prevailing in the source language like names, data types, scopes, scopes, operators, procedures, parameters and flow-of-control constructs.
- The compiler creates and manages a run-time environment in which the target programs are supposed to be executed. This environment deals with a variety of issues such as mechanisms to access variables, allocation and layout of storage locations for the source program objects, linkages between procedures, mechanisms for passing parameters, interfaces to the operating system, input/output devices and other programs.
- The activations of procedures during execution of a program can be represented using a tree, called an activation tree. Each node represents an activation and the root designates the activation of the main procedure that initiates the execution.
- A control stack is a run-time stack whose primary task is to manage the procedure calls and returns. It contains the activation record of the procedures being called. The root of the activation tree resides at the bottom while the most recently called activation at the top. The entire sequence of activation records corresponds to the path of the activation tree.
- A calling sequence is a set of codes that allocates an activation record onto a stack and accordingly enters information into its fields. A calling sequence is a set of codes that allocates an activation record onto a stack and accordingly enters information into its fields. A return

sequence reloads the machine to its previous state from which the procedure was called.

- Heap is that portion of memory that stores data indefinitely. Data gets erased only when the program explicitly deletes it. The memory manager is a part of the operating system that allocates and deallocates space within heap. It serves as an interface between the application program and the operating system.
- When allocation and deallocation of storage takes place in heap, the free chunks are not contiguous and they leave holes in the heap. When an allocation request comes, the memory manager looks for the largest hole enough to hold that program.
- The technique through which the chunks of memory holding objects that are no longer be accessed by a program can be reclaimed is called as garbage collector.
- A mutator is a user defined program which creates objects by acquiring space from the memory manager. References to existing objects are created and dropped using this program. When mutator cannot reach any object, then the object becomes garbage.

15.12 ANSWERS TO CHECK YOUR PROGRESS

1. intermediate codes
2. three-address codes
3. syntax tree and DAGs
4. Register allocation
5. Register assignment
6. Label
7. Destination
8. Registers
9. Condition
10. True
11. one

12. zero
13. one
14. static and stack allocation techniques
15. callee
16. Branch
17. Run-time
18. Stack Pointer (SP)
19. symbol table
20. physical addresses
21. type
22. activation records
23. calling sequence
24. heap
25. register
26. blocks
27. pages
28. dangling-pointer-dereference error.
29. memory-leak error

15.13 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is code generator? Explain in brief.
2. What are the representations that form inputs to the code generator phase?
3. What do you mean by program and instruction cost? Discuss in brief.
4. Describe how the addresses in the target code are generated.
5. What do you understand by run-time addresses for names?
6. What is run-time environment? Discuss.
7. Differentiate between the Static vs. Dynamic Storage Allocation techniques.
8. What is activation tree? Discuss in brief.
9. What is activation record? Discuss in brief.
10. What is heap? Explain in brief.
11. What is fragmentation? Explain.
12. What is garbage collection? Describe.
13. What is mutator? What does it perform?
14. What are the ways of locating an unreachable object?
15. What are the functions that a mutator perform to change the set of reachable objects?

B. Long answer type questions.

1. Explain the functions performed by the code generator phase.
2. Write down the issues that occur in generation of target codes?
3. Explain the target machine model in detail.
4. Write down the significance of program and instruction costs asociated with target code generation.
5. Describe the static and stack allocation techniques in detail.
6. Describe the run-time environment in detail.
7. Describe the technique of allocation of space using a stack.
8. What is calling sequence? Describe the design principles of calling sequences.
9. How is it possible to allocate variable length data on a stack? Describe.
10. What is heap management?
11. Give a detailed discussion on the memory hierarchy of a computer.
12. Explain how is fragmentation reduced in a heap?
13. Explain the technique of garbage collection in a heap.
14. What are the matrices considered during the design of a garbage collector?

15.14 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: Prentice Hall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques, and tools Second Edition.

×××

UNIT: 16
TRANSFORMATION OF BASICBLOCKS

Unit Structure

16.0 INTRODUCTION

In the previous unit, we have learnt that graphical representations are constructed from the intermediate codes for efficient code generation. We also have learnt that register allocation is necessary for better throughput. Such representations are constructed by partitioning the intermediate codes into basic blocks which are the sequences of consecutive three-address statements. The control flow enters the basic block simply through the first instruction of the block. Control passes in a sequence within the block without branching or halting and finally leaves the block. Once we are ready with the basic blocks, we can move towards the construction of flow graph. A flow graph is a graphical representation of the basic blocks. Each node of the graph is represented by a basic block. In this unit, we shall try to get some idea regarding the generation of basic blocks. We shall also discuss how flow graphs are constructed using these blocks.

16.1 UNIT OBJECTIVE

After going through this unit you will be able to:

- Learn how basic blocks are constructed.
- Explain the liveness and next-use information of basic blocks
- Know how to construct a flow graph
- Understand the optimization techniques of basic blocks
- Learn the DAG representation of basic blocks

16.2 BASIC BLOCKS

In the previous unit, we have known that the three-address instructions are partitioned into sequence of simple codes called the basic blocks. To begin with, the first instruction forms the beginning of a new basic block. Further, instructions are added to the basic block until an unconditional jump, conditional jump or a label following an instruction is attained. Control proceeds sequentially when there is no jump statement. Let us see the following algorithm which partitions the three-address codes into corresponding basic blocks.

Algorithm 16.1: Partitioning three-address codes into basic blocks

Input: Sequence of three-address instructions.

Output: List of basic blocks for the sequence in which each instruction is assigned exactly to one basic block.

Method: Leaders are the first instructions of the basic blocks. At first, the leaders occurring in the intermediate codes are determined. Leader does not include the instruction just past the end of the intermediate program. Here are some rules for finding leaders.

1. The first three-address instruction in the intermediate code is a leader.
2. An instruction that is the target of a conditional or unconditional jump is a leader.
3. An instruction that immediately follows a conditional or unconditional jump instruction is a leader.

For a leader, its basic block consists of itself and all instructions upto but not including the next leader or the end of the intermediate codes. To get a clear picture of how it forms, let's consider the following set of intermediate codes.

```
1.  i = 1
2.  j = 0
3.  t1 = 20 * i
4.  t2 = t1 + j
5.  t3 = t2 * 2 - 40
6.  A [t3] = 5
7.  j = j + 1
8.  if j <= 10 goto 3
9.  i = i + 1
10. if i <= 10 goto 2
11. t4 = i
12. a[t4] = 10
13. i = i + 1
14. if i <= 10 goto12
```

As per rule (1) of the algorithm, instruction 1 forms the leader. Our next task is to find the jumps. There are three such instructions; which contain conditional jumps; instructions (8), (10) and (14). Now, according to rule (2) of the algorithm, the targets of these instructions also form leaders. Therefore, instructions (3), (2) and

(12) are the targets and they form leaders. Then, rule (3) says that the instruction immediately following a jump is also a leader. Instructions (9) and (11) are such instructions. The last instruction (14) is not followed by any instruction. Therefore, there is no leader following this instruction. Finally, we conclude that instructions (1), (2), (3), (9), (11) and (12) form the leaders. Each basic block begins with a leader and it continues until the instruction just before the next leader. The basic block of (1) contains only (1). Like this, leader (2) contains basic block only consisting of (2) only. However, for leader (3), the corresponding basic block consists of (3) through (8). Similarly, instruction (9) has a basic block consisting of (9) and (10). The basic block of leader (11) contains of 11 only. Then, finally for leader (12), the basic block contains instructions (12) through (14).

16.2.1 Next-Use Information

The next-use information is essential for keeping track of when the value of a variable will be used next so that efficient codes can be generated. The subsequent use of a name in a three-address code has to be determined. If the value of a variable which is currently in a register will never be subsequently referenced, that register can be assigned to another variable. The use of a name in a three-address statement can be conceptualized by the following set of codes.

```
1. k = 10
2. x = k + k
3. m = 1
4. S = 2 * m * x
```

The statement 1 of the above set of codes initializes the value of variable k. Now, statement 2 considers the value of k and computes x. Then, there is no intervening change to the value of x and finally statement 4 computes the value of S simply by using the value of x derived by statement 2. In other words, statement 4 uses the value of x computed by statement 2. It can be further concluded that x is live at statement 2. Our next algorithm determines the liveness and next-use information of a basic block.

Algorithm 16.2: Determining the liveness and next-use information for each statement in a basic block.

Input: A basic block B of three-address statements. We assume that the symbol table initially shows all non-temporary variables in B as being live on exit.

Output: At each statement ***i: x = y + z*** in B, we attach to i the liveness and next-use information of x, y and z.

Method: We begin at the last statement of B and scan backwards to the first statement of B. At each statement ***i: x = y + z*** in B, we do the following:

1. Attach to each statement *i*, the information currently found in the symbol table regarding the next use and liveness of *x, y* and *z*.
2. In the symbol table, set *x* to "not live" and "no next use".
3. In the symbol table, set *y* and z to "live" and the next uses of *y* and *z* to i.

If the three-address statement does not include + as an operator between operands, statement i may contain codes of the form $x = +y$ or $x = y$. This ignores variable *z*.

CHECK YOUR PROGRESS – I

1. Intermediate codes are partitioned into __________.
2. A _____________ is a graphical representation of the basic blocks.
3. Each node of the flow graph is represented by a ________.
4. Control proceeds sequentially when there is no ______ statement.
5. The first instructions of the basic blocks are ________.

16.3 FLOW GRAPH

We have already discussed that the basic blocks generated from intermediate codes can be represented in terms of a graphical representation called as flow graph. Therefore, the nodes of a flow graph represent the basic blocks. We consider two basic blocks A and B. An edge can be formed between A and B if and only if the last instruction of block A is immediately followed by the first instruction of block B. An edge can be formed in two ways:

1. There is a conditional or unconditional jump from the last instruction of A to the beginning of B.
2. B immediately follows A in the original order of the three-address instructions and A is not an end of an unconditional jump.

A is termed as the predecessor of B and conversely, B is the successor of A.

Often, two nodes termed as the entry and exit, are added to the graph. They do not correspond to executable intermediate instructions. Rather, there is an edge coming from the entry node to the first executable node or basic block of the graph. There is an edge to the exit node from that basic block of the flow graph that contains the last instruction of the program. Therefore, the set of instructions used in the example for finding basic blocks can be demonstrated in terms of the following flow graph.

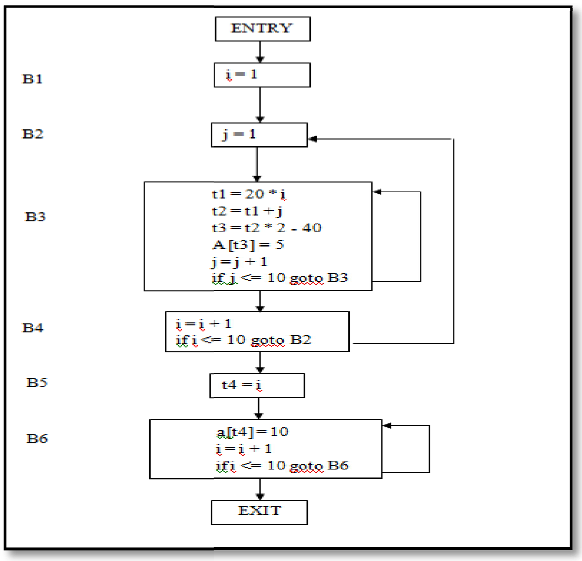


Fig 16.1: Flow graph representing control flow between nodes

Flow graphs can be thought of as an ordinary graph consisting of both nodes and edges. In basic blocks, a jump to instructions or labels took place when conditional or unconditional jump instructions were encountered. Accordingly, such jumps can occur to basic blocks also. This happens with the flow graph where jumps to leaders of basic blocks are replaced by jumps to basic blocks. However, the contents of each node or a basic block of a flow graph can be represented by some data structures. One way is to keep a pointer to point to the leader of an array of three-address instructions along with a count of the total number of instructions. A second pointer may also be inserted to the last instruction instead of keeping the count. However, it would be considered as a good practice to create a linked list of instructions for each basic block since, the number of instructions in a basic block changes frequently.

16.3.1 Loops in Flow Graphs

Loops in programming languages are the constructs for-loop, while-loop and do-while loop statements. They occur frequently in a program and it is the task of the compiler to generate a good code for loops. A set of nodes L in a flow graph is a loop if L contains a node e called the loop entry such that

1. e is not ENTRY of the entire flow graph
2. Except e, no node in L has a predecessor outside L. In other words, a path from ENTRY to any node in L goes through e.
3. Within L, every node in L has a non-empty path to e.

The above flow graph consists of three loops:

1. B3 by itself
2. B6 by itself
3. {B2, B3, B4}

B3 and B6 are single nodes with an edge to the nodes itself. They form loops with themselves being entries to some non-empty paths. But, node B2 cannot be considered as a loop as it does not have an edge to itself. In the third case, B2 is the entry to the loop. Also it has a predecessor B1, which is not a part of L. Furthermore, there is a non-empty path which leads to B2 within L.

16.4 OPTIMIZATION OF BASIC BLOCKS

As already mentioned, it is important to transform codes into some form so that efficient codes can be generated. The running time required to execute such codes would be less. In such situation, code optimization comes into play. Optimization technique can be performed on basic blocks also. This can happen in two ways: local and global. Local optimization takes place within the block on the instructions. And global optimization takes place within the blocks and it sees how flow of information occurs among the basic blocks of a program.

16.4.1 DAG Representation of Basic Blocks

Just like the optimization techniques we studied in a previous unit, local optimization technique can be applied on basic blocks too. It begins by transforming a basic block into a DAG (Directed Acyclic Graph). In a previous unit, we learnt about representation of DAG for a single expression. Now, here, the idea has got extended into a set of expressions making up a basic block. DAG for basic blocks has the advantage of eliminating local common sub-expressions. It is necessary to eliminate such instructions which compute values which are already computed. Moreover, some instructions compute a value that is never used during execution. Such codes, termed as dead code must be eliminated from the basic block. Separate and independent statements which do not depend on each other for data inputs can be reordered. Such reordering reduces the required time to store temporary values to be preserved in a register. Such reordering also happens on operands of three-address codes.

Reordering is performed by applying algebraic laws on the instructions.

1. Finding local common sub-expressions in a basic block is important for generating an efficient code. This concept is similar to the concept that we introduced for DAG in unit 12. Considering a case when a new node N is about to be added. Such addition requires checking whether an existing node M has the same children in the same order and with the same operator. Therefore, M computes the same value as N and may be used in its place.Now, let us consider the following block of codes.

    1. A = B * C
    2. B = A + E
    3. C = B * C
    4. E = A + E

    This can be represented using the hierarchical structure called DAG. Statement 1 computes the value of A using operands B and C. the second statement derives the value of B simply by adding A and E. Now, this is B's most recent value. The third statement uses this current value of B instead of the one in the first statement. Therefore, the value of B in the third statement refers to the node with label "+" of the second statement. The corresponding DAG structure for the above statements would be as follows:

    However, statements 2 and 4 have the same operands with same operator and no change in values of the operands have taken place in between, therefore no extra node need to be created. Simply adding an extra label will be sufficient. So, B and E will point to the same node.

    Labels B_0, C_0 and E_0 are used to distinguish between statements.

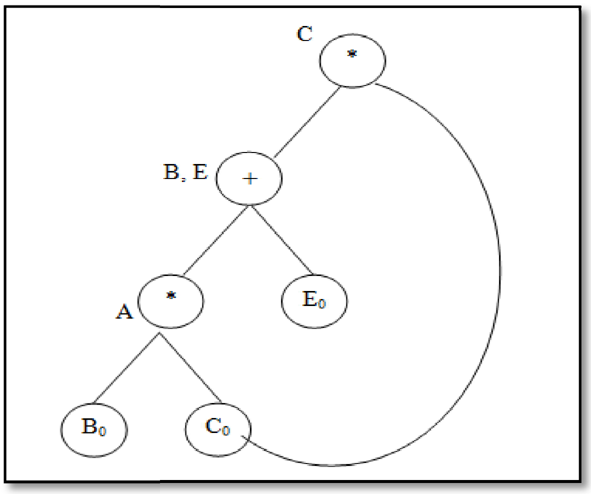


Fig 16.2: DAG representation of basic blocks

2. Dead code elimination plays a vital role in maintaining the efficiency of compilation. A dead code is the one which has no live variables attached to it. A root having no live variables attached to it can be deleted. Such repeated application will remove all the nodes from a DAG corresponding to dead code.

   Figure 16.2 contains three non-leaf nodes; whereas our given basic block of three-address codes contains four statements. Therefore, the basic block can be replaced by three statements. Here, if B is not live on exit from the block; we do not need to compute B and replace it with E in order to receive the same value that B computes at the node labeled "+". Therefore, the basic block becomes

   1. A = B * C
   2. D = A + E
   3. C = D * C

3. Another optimization technique employed in basic blocks is the representation of algebraic identity. The basic arithmetic identities can be applied on basic blocks such as the following to eliminate computations from a basic block.

$$X + 0 = 0 + X = X$$
$$X * 1 = 1 * X = X$$

Another optimization technique includes replacing more expensive operators with a cheaper one. This technique is termed as reduction in strength. Examples of such kind includes

X^2	=	X * X
2 * X	=	X + X
X/2	=	X * 0.5

$X^2$is somewhat more expensive than its equivalent statement X * X. Similarly, 2 * X can be replaced by X + X, which is considered to be comparatively cheaper.

Constant folding is a class of optimization technique. Constant expressions are evaluated during compile time and these expressions are replaced by their corresponding values. Therefore, 3^2 can have its value replaced by 9.

The DAG construction may apply other algebraic transformations such as commutativity and associativity. Therefore, before creating a new node, a check has to be made whether the node already exists. Here, the concept associated with commutative property comes into play. Like for example, before creating a node labeled * with x as its left child and y as its right child, we have to check if the same labeled node exists with y as its left child and x as its right child. Because, both evaluates the same output since they satisfy commutative property.

4. Array indexing can be represented using a DAG. Considering the following set of three-address codes accessing array references.

```
x     =   A[i]
A[j]  =   y
z     =   A[i]
```

The assignment statement as the first one is represented by creating a node with operator =[]. Its children are represented by A and index i. The label of this node is given by x.

The second statement A[j] = y is represented by a new node []= having three child nodes A, j and y. This creation of nodes would eventually kill all the nodes which are dependent on A for their values. Thus, node labeled x would be killed. Now, when node z is created it cannot be referred or associated with x. A new node with operands A and i would be created.

5. The pointer assignment statement x = *p contains operator =* that must take all the nodes that are currently associated with identifiers as arguments which is pertinent to dead code elimination. On the other hand, the assignment statement *p = x has the operator *= which kills all the other nodes constructed in the DAG so far.

Once we are finished with whatever optimizations required for manipulating DAG, reassembling may be done on the basic block by reconstituting the three-address codes. A three-address statement is reconstituted by considering the nodes having one or more variables attached to them. The result of computation must be done on a variable that is live on exit from the block. If there are more than one live on exit variables, their actual values must be deduced. Finally, global optimization may eliminate the duplicate copies.

The reconstruction of DAG basic block requires some rules to be followed. The order in which the instructions are sequenced for evaluation is a primary concern. Apart from that, the variables whose values must be assigned to the DAG's nodes must also be computed. Some of such rules may be as follows:

1. The values of the child nodes of a node must be computed prior to computing its parent node's value.
2. Assignments may take place in the same order it is there in the original basic block. Array assignments are preceded by all previous assignments to or evaluated from the same array.
3. As per point number 2. array elements are evaluated after the previous assignments to the same array. Two evaluations from the same array may be done in either order, as long as neither of them crosses over the assignment to that array.
4. All previous procedure calls or indirect assignments through a pointer must precede any use of a variable.
5. All previous evaluations to any variable must precede all previous procedure calls or indirect assignments.

CHECK YOUR PROGRESS – II

6. If control jumps from basic block A to basic block B, A is called the ______________ B.
7. The nodes ______________ and _______________do not correspond to executable codes
8. The running time required to execute the _________ ________would be less.
9. ___________ optimization takes place within the block on the instructions.
10. DAG for basic blocks has the advantage of eliminating local ___________ _______________.
11. A _______is the one which has no live variables attached to it.

16.5 SUMMING UP

- Basic blocks are the sequences of simple and consecutive three-address statements. Control flow enters the basic block simply through its first instruction. Leaders are the first instructions of the basic blocks.
- Instructions are added to the basic block until an unconditional jump, conditional jump or a label following an instruction is attained. Control passes in a sequence within the basic block without branching or halting and finally leaves the block.
- The next-use information is essential for keeping track of when the value of a variable will be used next so that efficient codes can be generated. If the value of a variable which is currently in a register will never be subsequently referenced, that register can be assigned to another variable.
- The basic blocks generated from intermediate codes can be represented in terms of a graphical representation called as flow graph. Therefore, the nodes of a flow graph represent the basic blocks.
- Two nodes termed as the entry and exit, are added to the flow graph. They do not correspond to executable intermediate instructions.
- Optimization technique can be performed on basic blocks also. This can happen in two ways: local and global. Local optimization takes place within the block on the instructions. And global optimization takes place within the blocks and it sees how flow of information occurs among the basic blocks of a program.
- Local optimization technique can be applied on basic blocks too. It begins by transforming a basic block into a DAG

(Directed Acyclic Graph). DAG for basic blocks has the advantage of eliminating local common sub-expressions.

- Some instructions compute a value that is never used during execution. Such codes, termed as dead code must be eliminated from the basic block. Optimization is necessary for efficient generation of target codes.

16.6 ANSWERS TO CHECK YOUR PROGRESS

1. basic blocks
2. flow graph
3. basic block
4. jump
5. leaders
6. predecessor
7. entry and exit
8. optimized code
9. Local
10. common sub-expressions
11. dead code

16.7 POSSIBLE QUESTIONS

A. Short answer type questions.

1. What is basic block? Explain in brief.
2. What do you mean by next-use information? Discuss in brief.
3. What is flow graph? What are the two ways that an edge between two nodes of a flow graph can be formed?
4. What do you mean by optimization of basic blocks?
5. What is dead code elimination of basic blocks?
6. What do you mean by common sub-expression elimination of DAG representation of basic blocks?

B. Long answer type questions.

1. Explain how the three-address codes are partitioned into basic blocks.

2. Explain how the liveness and next-use information for each three-address statement of a basic block can be determined.
3. Describe how flow graphs are constructed for a set of three-address codes.
4. What do you understand by loops in a flow graph? Describe.
5. What is DAG representation of basic blocks? Explain elaborately.
6. What are the advantages of DAG representation of basic blocks?
7. Explain the rules for reconstructing DAGs for basic blocks.

16.8 REFERENCES AND SUGGESTED READINGS

- Bergmann, S. D. (2017). Compiler design: theory, tools, and examples.
- Thain, D. (2016). Introduction to compilers and language design. Lulu. com.
- Holub, A. I. (1990). Compiler design in C (pp. I-XVIII). Englewood Cliffs, NJ: PrenticeHall.
- Aho, A. V., Lam, M. S., Sethi, R., &amp; Ullman, J. D. (2007). Compilers: principles, techniques and tools Second Edition.

×××

UNIT: 17
STRATEGIES OF CODE OPTIMIZATION

Unit Structure

17.0 INTRODUCTION

Code optimization is a crucial phase in the compilation process where the compiler transforms the code to improve its efficiency, performance, and resource utilization while preserving its functionality. The primary goal of code optimization is to produce optimized machine code that executes faster, consumes fewer resources such as memory and energy, and exhibits better runtime behavior.

Optimization is essential because it directly impacts the performance and quality of software systems. Efficiently optimized code can lead to faster execution times, reduced memory footprint, and improved energy efficiency, resulting in better user experience and cost savings.

The process of code optimization involves analyzing the source code, identifying inefficiencies, and applying transformations to eliminate or reduce them. This process requires a deep understanding of the target hardware architecture, programming language semantics, and optimization techniques.

Various strategies and techniques are employed during code optimization, include Variable Propagation, Code Motion, Strength Reduction, Elimination of Dead Code, Loop Optimization

However, there are trade-offs associated with optimization. Aggressive optimization approaches may result in lengthier development cycles and possible maintenance issues by increasing compilation time and code complexity. To make sure the advantages of code optimization outweigh the disadvantages, it is crucial to measure its efficacy. Runtime metrics and performance benchmarks are frequently used to assess how optimization strategies affect program performance and resource usage.

In summary, code optimization plays a vital role in compiler design by improving the efficiency and quality of generated code. By understanding the goals, principles, and techniques of code optimization, developers can create software systems that deliver better performance, scalability, and reliability.

17.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand the importance of code optimization in compiler design.
- Learn the goals and principles of code optimization.
- Explore various strategies and techniques used for code optimization.
- Learn how to measure the effectiveness of code optimization.

17.2 VARIABLE PROPAGATION

Variable propagation, also known as constant propagation or constant folding, is a compiler optimization technique aimed at replacing variables with their constant values where possible, thereby potentially reducing the runtime overhead and improving the efficiency of the generated code. This optimization is particularly effective in cases where variables hold constant values throughout the execution of the program.Benefits of Variable Propagation include first reduced memory usage i.e.constants don't need to be stored in memory as variables, leading to potential memory savings. Secondly performance is improved, as direct use of constants instead of variables can reduce the number of memory accesses and calculations, improving runtime performance and finally simplifies thecode thus eliminating unnecessary variables to cleaner and more readable code.

Consider the following Variable Propagation example:

```
float calculate_area(float radius) {
   float pi = 3.14159;
   float area = pi * radius * radius;
   return area;
}
int main() {
printf("%f\n", calculate_area(5));
   return 0;
}
```

In this example, pi is a constant value that doesn't change throughout the execution of the program. Variable propagation optimization can replace references to pi with its constant value 3.14159, resulting in more efficient code.

After variable propagation optimization, the code will look like this:

```
float calculate_area(float radius) {
   float area = 3.14159 * radius * radius;
   return area;
}

int main() {
printf("%f\n", calculate_area(5));
   return 0;
}
```

17.3 CODE MOTION

Code motion, also known as loop invariant code motion (LICM), is a compiler optimization technique used to improve the efficiency of code by moving calculations or expressions out of loops when those calculations produce the same result for every iteration of the loop. This helps reduce redundant computations and can lead to significant performance improvements, especially in loops that are executed frequently.

Consider the following Code Motion example:

```
i=0;
total=500;
while (sum<= total – 5) // without code motion
{
sum=sum + i;
i++;
}
printf("%d", sum);
```

The while() loop in the example above determines whether the variable "i" is less than "total-5" and whether it should stay in the loop or not. Since the while loop's body does not change the variable "total," the statement "total-5" will evaluate to the same value, indicating that it is a non-motion variable. Since this expression must be calculated each time the while() loop is triggered, it is possible to avoid using it. Below are the specifics following the implementation of code motion:

```
i=0;
total=500;
test=total – 5; // with code motion
while (sum<= test)
{
sum=sum + i;
i++;
}
printf("%d", sum);
```

17.4 STRENGTH REDUCTION

Strength reduction is a technique used in code optimization to replace expensive operations with cheaper ones. This typically involves replacing costly operations like multiplication or division with less expensive ones like addition or bit manipulation. The goal is to improve the efficiency of the code while maintaining its correctness.

Examples of Strength Reduction are:

- Replacing Multiplication with Addition i.e. instead of performing repeated multiplication, we can use addition, which is generally faster.

```
// Original code with multiplication
int result = 3 * salary;
```

```
// After strength reduction
int result = salary + salary + salary;
```

- Replacing division with multiplication i.e. division operations are generally slower than multiplication. If we need to divide by a constant, we can replace it with a multiplication by the reciprocal of that constant.

```
// Original code with division
int result = y / 4;
```

```
// After strength reduction

int result = y * 0.25; // or x >> 2 (for division by powers of 2)
```

- Using Bitwise operations for division/multiplication by Powers of 2 i.e.division or multiplication by powers of 2 can be replaced with bitwise operations (right shift for division, left shift for multiplication), which are usually more efficient.

```
// Original code with division by 2
int result = x / 2;
```

```
// After strength reduction

int result = x >> 1;
```

- Replacing expensive operations with cheaper ones i.e. in some cases, there are alternate ways to achieve the same result with cheaper operations. For example, replacing exponentiation with repeated multiplication for integer powers, or using lookup tables for expensive computations.

```
// Original code with exponentiation
int result = pow(y, 2);
```

```
// After strength reduction
int result = y * y;
```

17.5 ELIMINATION OF DEAD CODE

Dead code elimination is the process of removing code from a program that is never executed, thus reducing the size of the executable and potentially improving performance. In C, dead code can arise due to conditional statements, unreachable code blocks, or unused variables or functions. Below are some examples demonstrating dead code elimination in C:

- Unreachable code:

```
#include <stdio.h>
intmain() {
int y = 10;
   if (y>20)
     {
printf("y is greater than 20\n");
     }
     else
     {
printf("y is not greater than 20\n");
     }
   return 0;
}
```

In the above example, the condition *y > 20* will always evaluate to false because *y* is initialized to 10. Therefore, the code block within the *if* statement will never execute. Dead code elimination would remove the *printf* statement inside the *if* block.

- Unused variables:

```
#include <stdio.h>
```

```
intmain() {
int x = 15;
int y = 11; // Unused variable
printf("x = %d\n", x);
   return 0;
}
```

In this example, the variable *y* is declared but never used. Dead code elimination would remove the declaration of y.

- Unused functions:

```
#include <stdio.h>
void test_function() {
printf("This function is never called\n");
}
intmain() {
printf("Main function\n");
   return 0;
}
```

The test_function is defined but never called from main(). Dead code elimination would remove the test_function from the final executable.

- Constant folding:

```
#include <stdio.h>
#define FLAG 0
intmain() {
if FLAG
printf("FLAG is true\n");
else
printf("FLAG is false\n");
   return 0;
}
```

In this example, the macro FLAG is defined as 0. During compilation, the preprocessor will evaluate the conditional expression *if FLAG*. Since *FLAG* is 0, the *else* block will always be executed. Dead code elimination would remove the *if FLAG* block entirely and only keep the else block.

17.6 LOOP OPTIMIZATION

Loop optimization is a crucial aspect of code optimization, especially in performance-critical applications where loops constitute a significant portion of execution time. The goal of loop optimization is to improve the efficiency of loops by minimizing redundant computations, reducing memory access overhead, and maximizing parallelism. Here are some common techniques for loop optimization along with examples:

- Loop unrolling that involves replicating the loop body multiple times to reduce loop overhead. This can be particularly beneficial when the loop body contains simple arithmetic operations.

```
// Original loop
for (inti = 0; i< 5; i++) {
   result += x;
}
```

```
// After loop unrolling

result += x;

result += x;

result += x;

result += x;
```

- Loop Fusion combines multiple loops that operate on the same data into a single loop, reducing memory access overhead.

```
// Separate loops
for (inti = 0; i< N; i++) {
   array1[i] = array2[i] + 1;
}
for (inti = 0; i< N; i++) {
   array2[i] = array1[i] * 2;
}
```

```
// Fused loop
for (inti = 0; i< N; i++) {
   array1[i] = (array2[i] + 1) * 2;
   array2[i] = array1[i];
}
```

- Loop Interchange is the process of switching the two iteration variables that a nested loop uses in order is known as loop interchange. The outer loop uses the variable that was used in the inner loop, and vice versa. In order to improve locality of reference, it is frequently done to make sure that the elements of a multi-dimensional array are accessed in the order that they are stored in memory.

```
for (int j = 0; j < 50; j++) {
   for (inti = 0; i< 20; i++) {
      a[i][j] = i + j + 1;
   }
}
```

After loop interchange

```
for (inti = 0; i< 20; i++) {
  for (int j = 0; j < 50; j++) {
      a[i][j] = i + j + 1;
   }
}
```

- Loop blocking, also known as loop tiling, divides a loop into smaller blocks that fit into the cache more efficiently, reducing cache misses.

```
// Original loop
for (inti = 0; i< N; i++) {
   array[i] = array[i] * 2;
}
```

After loop Blocking

```
// Blocked loop
for (int block = 0; block < N; block += BLOCK_SIZE) {
   for (inti = block; i<min(block + BLOCK_SIZE, N); i++) {
      array[i] = array[i] * 2;
   }
}
```

17.7 CHECK YOUR PROGRESS

i. What does variable propagation aim to achieve in code optimization?

a) Removal of unused variables
b) Propagation of constant values through variables
c) Dynamically allocating variables
d) Minimizing variable scope

ii. Which of the following describes code motion in optimization?
a) Moving frequently accessed code into cache memory
b) Reorganizing code to enhance readability
c) Moving code segments to reduce redundant calculations
d) Rewriting code in a different programming language

iii. What is the primary goal of strength reduction in optimization?
a) Decreasing the complexity of algorithms
b) Transforming expensive operations into cheaper equivalents
c) Enhancing code readability
d) Eliminating code duplication

iv. What does dead code elimination involve in code optimization?
a) Removing code that is unreachable or never executed
b) Minimizing variable declarations
c) Reusing code across different modules
d) Enhancing code documentation

v. Which of the following is not a common technique used in loop optimization?
a) Loop unrolling
b) Loop blocking
c) Loop inversion
d) Loop fusion

vi. Which of the following statements is true regarding variable propagation?
a) It is primarily concerned with renaming variables
b) It focuses on identifying and removing redundant variables
c) It aims to determine the data type of variables
d) It ensures variables have global scope

vii. In code motion, which type of code segment is typically targeted for movement?

a) Code with minimal complexity
b) Code with high computational cost
c) Code with frequent branching
d) Code with extensive comments

viii. Which of the following is an example of strength reduction?

a) Replacing division by multiplication with shift operations
b) Adding more variables to improve code readability
c) Converting iterative loops into recursive functions
d) Removing unnecessary type casting

ix. What is dead code?

a) Code that contains errors and crashes the program
b) Code that is executed repeatedly in a loop
c) Code that is no longer reachable or useful
d) Code that performs mathematical calculations

x. Which loop optimization technique aims to minimize cache misses by dividing loops into smaller blocks?

a) Loop unrolling
b) Loop fusion
c) Loop blocking
d) Loop inversion

xi. What is the purpose of constant propagation in variable propagation optimization?

a) To eliminate unused variables
b) To identify constant values assigned to variables
c) To dynamically allocate memory for variables
d) To redefine the scope of variables

xii. Which of the following is a potential drawback of code motion?

a) Improved cache locality
b) Increased register pressure
c) Reduced execution time
d) Enhanced code modularity

xiii. Which arithmetic operation is often replaced in strength reduction optimization?
  a) Subtraction with addition
  b) Multiplication with division
  c) Addition with subtraction
  d) Division with multiplication

xiv. In dead code elimination, what does unreachable code refer to?
  a) Code segments that are difficult to understand
  b) Code segments that are executed in every loop iteration
  c) Code segments that are commented out
  d) Code segments that cannot be reached during program execution

xv. Which loop optimization technique focuses on reducing the number of loop iterations by combining multiple iterations into one?
  a) Loop unrolling
  b) Loop fusion
  c) Loop blocking
  d) Loop inversion

17.8 ANSWERS TO CHECK YOUR PROGRESS

i, b	ii, c	iii, b	iv, a	v, c
vi, b	vii, b	viii, a	ix, c	x, c
xi, b	xii, b	xiii, b	xiv, d	xv, a

17.9 LET US SUM UP

- Variable propagation involves replacing occurrences of variables with their values whenever possible, reducing the need for memory accesses and potentially improving performance.
- Code motion involves moving computations or assignments out of loops when their results do not depend on loop iterations. This reduces redundant computations and improves efficiency.

- Strength reduction is a technique used to replace expensive operations with cheaper ones. For example, replacing multiplication with addition or shift operations.
- Dead code elimination involves removing code that has no effect on program output or behavior. This can include unreachable code or code whose results are never used.
- Loop optimization refers to techniques used to improve the efficiency of loops in computer programs. This includes loop unrolling, loop fusion, loop blocking, loop interchange, loop invariant code motion, and other techniques aimed at reducing loop overhead and improving cache locality.

17.10 FURTHER READING

- Modern Compiler Implementation in C by Andrew W. Appel, Publisher: Cambridge University Press,Edition: 2nd Edition
- Engineering a Compiler by Keith D. Cooper and Linda Torczon, Publisher: Morgan Kaufmann,Edition: 2nd Edition
- Compiler Construction: Principles and Practice by Kenneth C. Louden, Publisher: Cengage Learning,Edition: 1st Edition
- Principles of Compiler Design" by Aho, Ullman, and Sethi, Publisher: Addison-Wesley,Edition: 1st Edition

17.11 MODEL QUESTIONS

1. Explain variable propagation and its significance in optimizing code.Provide an example where variable propagation can eliminate redundant computations.

2. What is code motion, and why is it important in optimizing code? Give an example where moving code outside a loop improves performance.

3. Define strength reduction and its role in optimizing arithmetic operations. Compare and contrast strength reduction with loop unrolling. Illustrate strength reduction with an example code snippet.

4. What is dead code, and why is it undesirable in programs? Explain the difference between static and dynamic dead code elimination. Provide an example demonstrating dead code elimination.

5. Describe loop optimization techniques commonly used to enhance performance. Explain the significance of loop fusion and loop unrolling in optimizing loops.

6. Define loop unrolling and its purpose in loop optimization. Discuss the trade-offs associated with loop unrolling. Provide an example where loop unrolling improves performance.

7. Explain loop fusion and its benefits in optimizing code. Compare and contrast loop fusion with loop tiling. Give an example demonstrating loop fusion.

8. Define loop blocking and its relevance in optimizing memory access patterns. Discuss the impact of loop blocking on cache performance. Provide an example where loop blocking enhances cache utilization.

9. Describe loop interchange and its purpose in optimizing nested loops. Explain how loop interchange can improve data locality.

10. Define loop-invariant code motion and its role in reducing redundant computations. Discuss the benefits and limitations of loop-invariant code motion.

17.12 REFERENCES AND SUGGESTED READINGS

- Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Publisher: Pearson, Edition: 2nd Edition

×××

UNIT: 18
STRATEGIES OF CODE OPTIMIZATION

Unit Structure

18.0 INTRODUCTION

Code optimization strategies play a pivotal role in enhancing the performance and efficiency of software systems. Leveraging data flow analysis, compilers scrutinize the flow of data within programs to identify optimization opportunities. Object-oriented principles are seamlessly integrated into compiler design, enabling modularization, reusability, and flexibility. The choice between static and dynamic Remote Method Invocation (RMI) in distributed computing impacts the system's performance and dynamism. Various parameter passing examples, such as pass by value and pass by reference, influence compiler optimizations and runtime behavior. A case study exemplifies how a compiler employs a multitude of optimization techniques to boost code efficiency, from constant folding to target-specific optimizations, yielding significant performance gains across diverse platforms.

18.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand the basics of data flow analysis and its importance in program analysis.
- Understand the concepts of static and dynamic RMI and their differences.
- Explore the advantages and disadvantages of static and dynamic RMI.
- Understand different parameter passing mechanisms such as pass-by-value, pass-by-reference, and pass-by-pointer.
- Analyze real-world code examples to identify performance bottlenecks.

18.2 DATAFLOW ANALYSIS

In compiler design, data flow analysis is a technique used to examine the movement of data within a program. It entails monitoring the values of expressions and variables as they are utilized and computed throughout the program in order to spot possible problems and scope for optimization.

Modeling the program as a graph, with program statements as nodes and data flow relationships between the statements as edges, is the fundamental principle underlying data flow analysis. The data flow information is then sent across the graph by computing the values of variables and expressions at each stage of the program through the use of a set of guidelines and formulas.

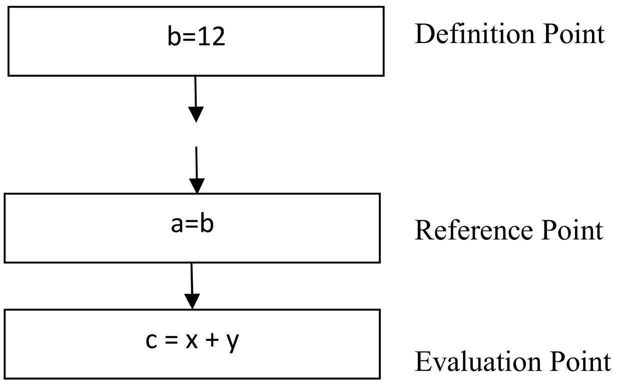
Compilers often do the following kinds of data flow analysis:

- Reaching definitions monitors the definition of a variable or expression throughout the program and identifies the instances where the definition influences a specific usage of the variable or expression. This data aids in the identification of variables that can be safely optimized or eliminated.
- Live Variable Analysis identifies the segments in the program where a variable or expression remains "live," indicating that its value is necessary for future computations. This insight helps in pinpointing variables that can be safely removed or optimized.
- Available Expressions Analysis identifies points in the program where a particular expression is "available," indicating that its computed value can be reused. This information enables the detection of opportunities for common subexpression elimination and other optimization techniques.
- Constant Propagation Analysis monitors constant values and determines where they are used within the program. This knowledge helps in identifying opportunities for constant folding and other optimization techniques.

Data flow analysis uses the terminologies

- Definition Point: A definition point within a program is where a data item is defined.
- Reference Point: A reference point within a program is where a reference to a data item is found.
- Evaluation Point: An evaluation point within a program is where an expression is present for evaluation.

The diagram below illustrates an example of a definition point, a reference point, and an evaluation point within a program.



Data Flow Analysis Equation: The data flow analysis equation serves to gather information about a program block. The following represents the data flow analysis equation for a statement, denoted as 's':

Out[s] = gen[s] ∪ In[s] - Kill[s]

Where:

Out[s] represents the information at the end of statement 's'.

gen[s] signifies the information generated by statement 's'.

In[s] stands for the information at the beginning of statement 's'.

Kill[s] denotes the information eliminated or replaced by statement 's'.

The primary objective of data flow analysis is to derive a set of constraints on the In[s]'s and Out[s]'s for statement 's'. These constraints encompass two types:

- **Transfer Function:** The transfer function encapsulates the semantics of the statement, defining the constraints for data flow values before and after the statement's execution. For instance, consider the statements 'x = y' and 'z = x'. After execution, both 'x' and 'z' hold the same value, i.e., 'y'.

Therefore, a transfer function illustrates the relationship between data flow values pre and post the statement. There are two types of transfer functions:

- **Control-Flow Constraints:** The second set of constraints arises from the control flow. If block 'B' contains statements S_1, S_2, ..., S_n, then the control flow value of S_i will be equal to the control flow values into S_{i+1}:

  $IN[S_{i+1}] = OUT[S_i]$, for all $i = 1, 2, ..., n - 1$.

18.3 OBJECTS

In compiler design, particularly in the context of code optimization, several objects play crucial roles in analyzing and transforming code to improve its efficiency, size, and speed. Some of these objects include:

- **Intermediate Representation (IR):** IR is an abstract representation of the source code that facilitates analysis and transformation during various stages of compilation. Different levels of IR may exist, such as High-Level IR (HIR), Middle-Level IR (MIR), and Low-Level IR (LIR), each serving specific optimization purposes.
- **Control Flow Graph (CFG):** CFG is a graphical representation of the flow of control within a program, where nodes represent basic blocks and edges denote control flow between them.CFG helps in analyzing and optimizing control structures, loop optimizations, and identifying code paths for transformations.
- **Data Flow Graph (DFG):** DFG represents data dependencies between operations in a program, helping in analyzing and optimizing data-related aspects such as

register allocation, data reuse, and parallelization opportunities.

- **Symbol Table:** Symbol table stores information about identifiers (variables, functions, constants) present in the program, including their names, types, scope, and memory locations. Symbol table is essential for performing scope analysis, type checking, and generating optimized code.
- **Optimization Primitives:** These are fundamental operations applied during optimization, such as constant folding, common subexpression elimination, loop unrolling, dead code elimination, and code motion. Optimization primitives target specific patterns or inefficiencies in the code to improve its performance or size.
- **Dependency Analysis:** Dependency analysis identifies dependencies between different parts of the program, such as data dependencies, control dependencies, and memory dependencies. Dependency analysis helps in understanding the relationships between program elements and guiding optimizations like parallelization and pipelining.
- **Transformation Rules:** Transformation rules define the conditions under which specific optimizations can be applied and the actions to be taken to perform those optimizations. These rules guide the compiler in selecting appropriate optimization techniques based on the characteristics of the code being compiled.
- **Register Allocation Table:** Register allocation table maps variables to processor registers to minimize memory accesses an optimize performance. Register allocation strategies aim to maximize register usage efficiency while satisfying constraints like register availability and register interference.

18.4 INTEGRATING CLIENTS AND OBJECTS

Integrating clients and objects in compiler design involves understanding how object-oriented principles can be applied to various components of a compiler, such as its architecture, data structures, and optimization techniques. Clients and objects are typically integrated in compiler design by:

Architecture Design:

Object-oriented design principles, such as encapsulation, inheritance, and polymorphism, can be applied to the architecture of the compiler. Different compiler phases (lexical analysis, syntax analysis, semantic analysis, code generation, optimization, etc.) can be represented as objects with well-defined interfaces. Each phase can encapsulate its specific functionality and interact with other phases through well-defined interfaces, promoting modularity and maintainability.

Data Structures:

Data structures used within the compiler can be designed using object-oriented principles. For example, abstract syntax trees (ASTs), symbol tables, and intermediate representations (IR) can be represented as objects. Objects representing AST nodes can encapsulate information about program constructs and provide methods for traversal and manipulation. Symbol table objects can encapsulate information about program symbols (variables, functions, types, etc.) and provide methods for lookup and update operations.

Optimization Techniques:

Object-oriented programming (OOP) concepts can be leveraged in implementing various optimization techniques. For example,

optimization algorithms can be implemented using classes and methods that encapsulate specific optimization strategies.
Object-oriented design patterns, such as the Visitor pattern, can be used to implement optimizations that traverse and manipulate ASTs.
Optimization passes can be implemented as objects that can be composed and configured flexibly to apply a sequence of optimization transformations.

Compiler Frontend and Backend:
The compiler frontend (lexical analysis, syntax analysis, semantic analysis) and backend (code generation, optimization) can be modularized using object-oriented principles. Each phase can be implemented as a separate object with well-defined interfaces, allowing for easy replacement or extension of individual components. Object-oriented design facilitates the development of reusable compiler components and promotes code reusability across different compiler projects.

Overall, integrating clients and objects in compiler design involves applying object-oriented principles to the architecture, data structures, optimization techniques, and various components of the compiler. This approach promotes modularity, maintainability, and flexibility, enabling the development of robust and efficient compilers.

18.5 STATIC VS. DYNAMIC RMI

Remote Method Invocation (RMI) is a mechanism used in distributed computing to enable communication between remote objects. In the context of compiler design and code optimization, both static and dynamic RMI can play significant roles, each offering distinct advantages and use cases.

Static RMI involves compiling the method calls at compile time, meaning that the method calls are bound to specific implementations before runtime. This approach offers several benefits in the realm of compiler design and code optimization:

- **Early Binding:** Static RMI allows for early binding of method calls, which can lead to improved performance as the method calls are resolved at compile time rather than runtime.
- **Compile-Time Optimization:** Since the method calls are resolved at compile time, compilers can perform various optimizations, such as inlining, constant propagation, and dead code elimination, to improve the efficiency of the generated code.
- **Reduced Overhead:** Static RMI typically incurs less overhead compared to dynamic RMI because the method calls are resolved once at compile time, avoiding the need for runtime lookups.
- **Improved Predictability:** Since the method calls are bound at compile time, developers have more predictability regarding the behavior of the program, which can aid in debugging and performance tuning.

Dynamic RMI, on the other hand, involves resolving method calls at runtime, allowing for greater flexibility and dynamism. While dynamic RMI may not offer the same level of performance optimizations as static RMI, it has its own set of advantages:

- **Late Binding:** Dynamic RMI allows for late binding of method calls, meaning that the method implementations can be changed or updated at runtime without requiring recompilation of the code.

- **Dynamism:** Dynamic RMI enables dynamic loading and unloading of classes and objects at runtime, which can be beneficial in scenarios where the code base is expected to evolve frequently or where dynamic behavior is required.
- **Remote Method Invocation:** Dynamic RMI is well-suited for scenarios where method calls need to be invoked across network boundaries, as it allows for the discovery and invocation of remote methods at runtime.
- **Reflection and Introspection:** Dynamic RMI often leverages reflection and introspection mechanisms, which provide powerful ways to inspect and manipulate objects at runtime.

In compiler design and code optimization, both static and dynamic RMI can be valuable tools, each offering distinct advantages depending on the requirements of the system. Static RMI excels in scenarios where performance and predictability are paramount, while dynamic RMI shines in situations requiring flexibility, dynamism, and remote method invocation across network boundaries. The choice between static and dynamic RMI ultimately depends on factors such as performance constraints, system requirements, and the desired level of flexibility and dynamism.

18.6 PARAMETER PASSING EXAMPLES

Parameter passing refers to the mechanism by which arguments are passed to functions or methods in a programming language. Different parameter passing methods have implications for code optimization in compiler design. Here are some examples of parameter passing methods and their impact on optimization:

Pass by Value:

- In pass by value, a copy of the argument's value is passed to the function.

- This method ensures that the original value is not modified within the function.
- Pass by value can simplify optimization because the compiler doesn't need to consider potential side effects on the original variables.
- Additionally, if the argument is a constant or immutable, the compiler may perform constant folding optimizations.

Pass by Reference:

- In pass by reference, a reference to the original variable is passed to the function.
- Any modification to the parameter within the function affects the original variable.
- Pass by reference can complicate optimization because the compiler must consider potential side effects on the original variables.
- However, optimizations such as copy propagation and common subexpression elimination can still be applied.

Pass by Pointer:

- Pass by pointer is similar to pass by reference but more explicit, as it passes the memory address of the argument.
- Like pass by reference, modifications to the parameter within the function affect the original variable.
- Pass by pointer can complicate optimization similarly to pass by reference, but it may also enable more fine-grained optimizations because of explicit memory address manipulation.

Pass by Value-Result:

- Pass by value-result is a hybrid approach where the function receives a copy of the argument's value but writes back the modified value to the original variable upon return.
- This method can complicate optimization because the original variable's value may change, but optimizations such as copy propagation can still be applied.

Understanding the semantics and implications of different parameter passing methods is crucial for optimizing code effectively.

Depending on the language and context, compilers may apply various optimizations to improve code performance while preserving the intended behavior of parameter passing.

18.7 CASE STUDY

Code optimization is a crucial phase in compiler design aimed at improving the efficiency, performance, and quality of generated code. In this case study, we'll explore a real-world scenario where various strategies of code optimization were applied to enhance the performance of a compiler.

Scenario: A software company is developing a new compiler for a high-level programming language. The company aims to optimize the generated code to improve the execution speed and reduce memory consumption. The compiler will target a range of platforms, including desktop computers, servers, and embedded systems.

Strategies Employed:

- **Constant Folding:** The compiler analyzes constant expressions during compilation and evaluates them at compile time. For example, if the expression 5 + 3 is encountered, the compiler computes the result 8 during compilation rather than at runtime. This strategy reduces the number of runtime computations and improves code efficiency.
- **Dead Code Elimination:** The compiler identifies and removes unreachable or redundant code segments. Unused variables, unreachable branches, and redundant assignments are eliminated to streamline the code. Dead code elimination reduces the size of the generated code and improves runtime performance.

- **Loop Optimization:** The compiler applies various techniques to optimize loops, such as loop unrolling, loop fusion, and loop interchange. Loop unrolling involves replicating loop bodies to reduce loop overhead and improve instruction-level parallelism. Loop fusion combines adjacent loops to reduce loop overhead and memory accesses. Loop interchange reorders loop nests to improve data locality and cache utilization.
- **Inline Expansion:** The compiler selectively replaces function calls with the actual function code at the call site. Small, frequently called functions are inlined to eliminate the overhead of function calls. Inlining reduces the call overhead, enables further optimizations across function boundaries, and improves overall code performance.
- **Register Allocation:** The compiler optimizes register usage to minimize memory accesses and maximize CPU register utilization. It allocates variables to CPU registers wherever possible, reducing memory traffic and improving execution speed. Techniques such as register coloring, graph coloring, and spilling are employed to efficiently allocate registers.
- **Data Flow Analysis:** The compiler performs data flow analysis to analyze the flow of data within the program. It identifies opportunities for common sub expression elimination, constant propagation, and copy propagation. Data flow analysis optimizes the use of variables and expressions, reducing redundant computations and memory accesses.
- **Target-Specific Optimization:** The compiler generates platform-specific code optimizations tailored to the target architecture. Architecture-specific optimizations, such as instruction scheduling, vectorization, and CPU-specific code

generation, are applied. Target-specific optimization maximizes the performance of the generated code on each platform.

By implementing the aforementioned strategies of code optimization, the compiler achieved significant improvements in code efficiency, performance, and memory usage across various target platforms. Benchmarks conducted on a range of applications demonstrated substantial speedups and reduced resource consumption compared to unoptimized code. The optimized compiler contributed to the development of faster, more efficient software solutions, enhancing the company's competitiveness in the market.

Code optimization is a critical aspect of compiler design, enabling compilers to generate high-performance, efficient code for diverse computing platforms. By employing a combination of optimization techniques tailored to the characteristics of the programming language and target architecture, compilers can unlock the full potential of software applications, delivering superior performance and user experience. In today's rapidly evolving technology landscape, continuous advancements in code optimization techniques are essential to meet the growing demands for faster, more efficient software solutions.

18.8 CHECK YOUR PROGRESS

i. In compiler design, what is data flow analysis primarily used for?

   a) Identifying syntactical errors in the code
   b) Analyzing the flow of data within a program
   c) Generating optimized machine code
   d) Managing memory allocation

ii. Which of the following represents the fundamental principle underlying data flow analysis?

a) Abstract Syntax Trees (AST)
b) Program statements as nodes and data flow relationships as edges
c) Control Flow Graphs (CFG)
d) Intermediate Representation (IR)

iii. Which type of data flow analysis identifies points in the program where a particular expression is "available" for reuse?

a) Reaching Definitions Analysis
b) Live Variable Analysis
c) Available Expressions Analysis
d) Constant Propagation Analysis

iv. What is a definition point in data flow analysis?

a) A point where a program is defined
b) A point where a data item is declared
c) A point where a data item is defined
d) A point where a function is defined

v. What does the data flow analysis equation 'Out[s] = gen[s] ∪ In[s] - Kill[s]' represent?

a) Information at the end of a statement
b) Information generated by a statement
c) Information at the beginning of a statement
d) Information eliminated or replaced by a statement

vi. In compiler design, what do objects such as Intermediate Representation (IR) and Control Flow Graphs (CFG) represent?

a) Data structures used for runtime memory management
b) Strategies for data flow analysis
c) Components of the compiler architecture
d) Optimized machine code

vii. What distinguishes static Remote Method Invocation (RMI) from dynamic RMI?

a) Static RMI involves resolving method calls at runtime.
b) Dynamic RMI allows for early binding of method calls.

c) Static RMI incurs less overhead compared to dynamic RMI.
d) Dynamic RMI involves compiling method calls at compile time.

viii. Which parameter passing method involves passing a copy of the argument's value to the function?
a) Pass by Reference
b) Pass by Pointer
c) Pass by Value
d) Pass by Value

ix. What is the primary objective of the data flow analysis equation?
a) To derive a set of constraints on the In[s] and Out[s] for a statement
b) To calculate the runtime complexity of a program
c) To identify syntactical errors in the code
d) To determine the optimal memory allocation

x. Which strategy of code optimization involves selectively replacing function calls with the actual function code at the call site?
a) Constant Folding
b) Loop Optimization
c) Inline Expansion
d) Dead Code Elimination

18.9 ANSWERS TO CHECK YOUR PROGRESS

i, b	ii, b	iii, c	iv, c	v, a
vi, c	vii, c	viii, c	ix, a	x, c

18.10 LET US SUM UP

- Data flow analysis is a vital technique in compiler design, examining data movement within programs to identify optimization opportunities and potential issues.
- It models programs as graphs, with statements as nodes and data flow relationships as edges, enabling the analysis of variable and expression values across the program.

- Data flow analysis includes techniques like reaching definitions, live variable analysis, available expressions analysis, and constant propagation, aiding in optimization and problem detection.
- The data flow analysis equation captures information flow within program blocks, considering generated and killed information at each statement.
- Transfer functions define how data flow values change before and after a statement's execution, aiding in understanding and optimizing code semantics.
- Control-flow constraints ensure consistency in data flow values across program statements, essential for accurate analysis and optimization.
- In compiler design, objects like intermediate representations, control flow graphs, and symbol tables play crucial roles in code optimization, facilitating analysis and transformation.
- Integrating clients and objects in compiler design leverages object-oriented principles to enhance modularity, maintainability, and flexibility across compiler components.
- Static and dynamic Remote Method Invocation (RMI) in distributed computing offer distinct advantages, impacting performance and flexibility in compiler design and optimization.
- Parameter passing examples, such as pass by value and pass by reference, influence compiler optimizations and runtime behavior, affecting code efficiency and performance.
- A case study demonstrates how various strategies of code optimization, from constant folding to target-specific optimizations, yield significant performance gains in a compiler, improving efficiency and competitiveness in the market.

18.11 FURTHER READING

- Modern Compiler Implementation in C by Andrew W. Appel, Publisher: Cambridge University Press, Edition: 2nd Edition
- Engineering a Compiler by Keith D. Cooper and Linda Torczon, Publisher: Morgan Kaufmann, Edition: 2nd Edition
- Compiler Construction: Principles and Practice by Kenneth C. Louden, Publisher: Cengage Learning, Edition: 1st Edition
- Principles of Compiler Design" by Aho, Ullman, and Sethi, Publisher: Addison-Wesley, Edition: 1st Edition

18.12 MODEL QUESTIONS

1. How does data flow analysis contribute to code optimization in compiler design?
2. What fundamental principle underlies data flow analysis in compiler design?
3. What are the main types of data flow analysis commonly performed by compilers?
4. Define and explain the terminologies used in data flow analysis, such as definition point, reference point, and evaluation point.
5. What is the data flow analysis equation, and what information does it provide about a program block?
6. How do transfer functions and control-flow constraints contribute to data flow analysis?
7. Describe the role of objects in compiler design, particularly in the context of code optimization.

8. How are different objects, such as Intermediate Representation (IR) and Symbol Table, utilized in compiler optimization?
9. What are the key principles involved in integrating clients and objects in compiler design?
10. Compare and contrast static and dynamic Remote Method Invocation (RMI) in the context of compiler design and code optimization.
11. Provide examples of parameter passing methods and discuss their implications for code optimization.
12. Describe a case study where various strategies of code optimization were applied in compiler design, and discuss the outcomes and benefits of these optimizations.

18.13 REFERENCES AND SUGGESTED READINGS

- Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Publisher: Pearson, Edition: 2nd Edition

×××