COMPILER DESIGN CONCEPTS, WORKED OUT EXAMPLES AND MCQS FOR NET/SET

Book · March 2017

2 authors:

Annal Ezhil Selvi S
Bishop Heber College
12 PUBLICATIONS **8** CITATIONS
SEE PROFILE

J . Persis Jessintha
Bishop Heber College
6 PUBLICATIONS **1** CITATION
SEE PROFILE

INTRODUCTION TO COMPILER

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct the machine, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes that is why we have compilers to write such codes.

1.1 Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

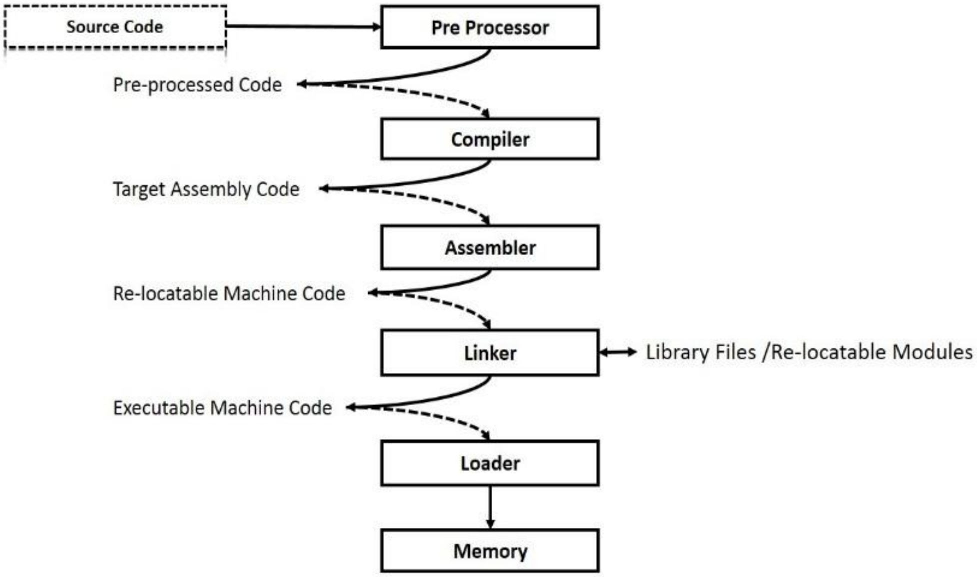


Figure 1.1 Language Processing System

The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object code).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).

- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

1.1.1. Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

1.1.2. Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

1.1.3. Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

1.1.4 Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

1.1.5 Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

1.2. Cross-Compiler

A compiler that runs on platform and is capable of generating executable code for platform is called a cross-compiler.

1.3. Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

1.3.1. Compiler – writing – tools

Number of tools has been developed in helping to construct compilers. Tools range from scanner and parser generators to complex systems, called compiler-compilers, compiler-generators or translator-writing systems.

The input specification for these systems may contain:

1. A description of the lexical and syntactic structure of the source languages.
2. A description of what output is to be generated for each source language construct.
3. A description of the target machine.

The principle aids provided by the compiler-compilers are:

1. For Scanner Generator the Regular Expression is being used.
2. For Parser Generator the Context Free Grammars are used.

1.4 Bootstrapping

A compiler is characterized by three languages:

1. source language
2. object language
3. The language in which it is written.

1.4.1 How the First Compiler compiled?

1. We have new language L, needed for several machines A, B.
   First small compiler is written for machine A, C_A^{SA}. That small compiler, C_A^{SA} translates a subset of language L into machine or assembler code for A.
2. Write compiler C_S^{LA}in the simple language S. When this program run throughC_A^{SA}becomes C_A^{LA}.

$$C_S^{LA} \longrightarrow \boxed{C_A^{SA}} \longrightarrow C_A^{LA}$$

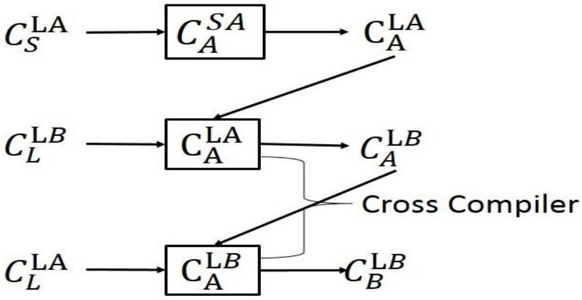
ie. Complete language L on machine A a produce object code for A.

Similarlyfor B

ie. C_S^{LA} into C_L^{LB} using C_L^{LB} to produce C_B^{LB} .....

For machine A a small compiler C_A^{SA} that translates a subset S of language L into the machine or assemble code of A.

For Machine B:



1.5 Compiler Architecture

A compiler can broadly be divided into two phases based on the way they compile.

1.5.1 Analysis Phase

Analysis phase is known as the front-end of the compiler, this phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

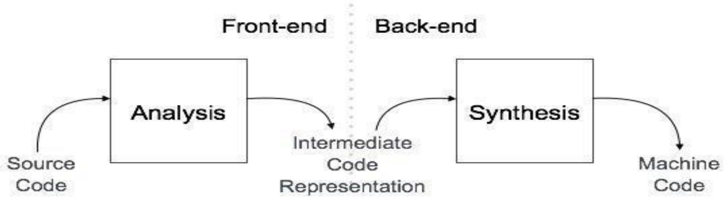


Figure 1.2 Working Principle of Compiler

1.5.2 Synthesis Phase

Synthesis phase is known as the back-end of the compiler, this phase generates the target program with the help of intermediate source code representation and symbol table. A compiler can have many phases and passes.

- Pass: A pass refers to the traversal of a compiler through the entire program.
- Phase: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

Phases v/s Passes:

- Phases of a compiler are sub tasks that must be performed to complete the compilation process. Passes refers to the number of times the compiler has to traverse through the entire program.

1.6 Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

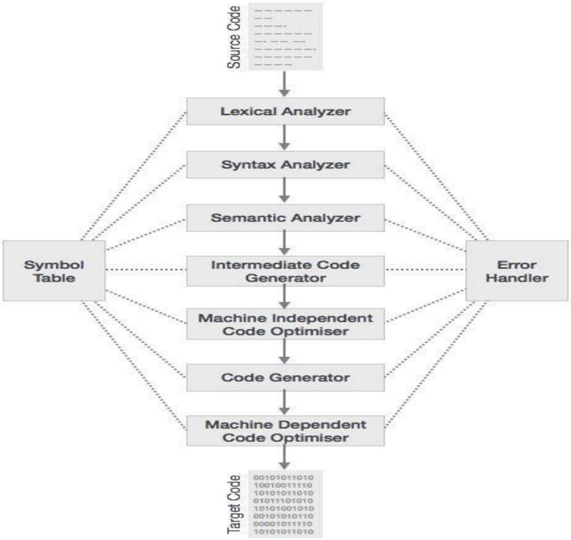


Figure 1.3 Architecture of the Compiler

1.6.1 Lexical Analysis

The first phase of compiler is also known as Scanner. The scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as: <Token-name, attribute-value>

1.6.2 Syntax Analysis

The next phase is called the Syntax Analysis or Parser. It takes the token produced by lexical analysis, as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct or not.

1.6.3 Semantic Analysis

Semantic analysis checks whether the parse tree constructed thus follows the rules of language. For example, it checks type casting, type conversions issues and so on. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

1.6.4 Intermediate Code Generation

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. The intermediate code may be a Three Address code or Assembly code.

1.6.5 Code Optimization

The next phase does code optimization, it is an optional phase. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources like CPU, memory. The output of this phase is an optimized intermediate code.

1.6.6 Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

1.6.7 Symbol Table

Symbol Table is also known as Book Keeping. It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their information like type, size, etc., are stored here. The symbol table makes it easier for the compiler to quickly search and retrieve the identifier's record. The symbol table is also used for scope management.

1.6.8 Error Hander

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and do parsing with the rest of the inputs. Mostly it is expected from the parser to check for errors. But errors may be encountered at various stages of the compilation process.

Summary

- A compiler is a program that converts high-level language to assembly language.
- A linker tool is used to link all the parts of the program together for execution.
- A loader loads all of them into memory and then the program is executed.
- A compiler that runs on machine and produces executable code for another machine is called a cross-compiler.
- A Compiler divided into two parts namely Analysis and Synthesis.
- The compilation process is done in various phases.
- Two or more phases can be combined to form a pass.
- A parser should be able to detect and report any error in the program.

Questions

1. Write a short note on Compiler Writing tools.
2. Differentiate Linker and Loader
3. Explain Bootstrapping.
4. Differentiate Analysis phase and Synthesis phase.
5. Describe the phases of the Compiler.

LEXICAL ANALYSIS

2.1. Introduction

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these sentences into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token as invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

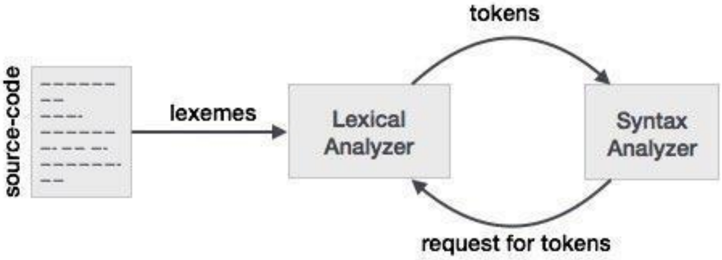


Figure 2.1 Working principle of Lexical Analyser

2.2 Tokens

Lexemes are said to be a sequence of characters (alphanumeric) which is also called as tokens. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators, and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

Contains the tokens:

1) int (keyword)
2) value (identifier)
3) = (operator)
4) 100 (constant)
5) ; (symbol)

2.2.1 Specifications of Tokens

Let us understand how the language theory considers the following terms:

2.2.1.1 Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

2.2.1.2 Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of alphabets in the string, e.g., the string S is “INDIA”, the length of the string, S is 5 and is denoted by $|S| = 5$. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

2.2.1.3 Special Symbols

A typical high-level language contains the following symbols:-

Symbols	**Purpose**
Addition(+), Subtraction(-), Modulo(%), Multiplication(*) and Division(/)	Arithmetic Operator
Comma(,), Semicolon(;), Dot(.), Arrow(->)	Punctuation
= , +=, /=, *=, -=	Assignment
==, !=, <, <=, >, >=	Comparison
#	Preprocessor
&	Location Specifier
&, &&, |, ||, !	Logical
>>, >>>, <<, <<<	Shift Operator

2.3 Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

2.4 The role of Lexical Analysis

1. It could be a separate pass, placing its output on an intermediate file from which the parser would then take its input.
2. The lexical analyzer and parser are together in the same pass; the lexical analyzer acts as a subroutine or co routine, which is called by the parser whenever it needs new token.
3. Eliminates the need for the intermediate file.
4. Returns a representation for the token it has found to the parser.

Example:

a (op) b

i. Treats the op as the token.

ii. Checks whether the operator found is +, -, *, &,etc...

2.4.1 The need for lexical analysis

The purpose of splitting the analysis into lexical analysis and syntactic analysis are,

- To simplify the overall design.
- To specify the structure of tokens that is the syntactic structure of the program easily.
- To construct more efficient recognizer for tokens than for syntactic structure.

2.4.2 Input Buffering

The lexical analyzer scans the characters of the source program one at a time to discover token.

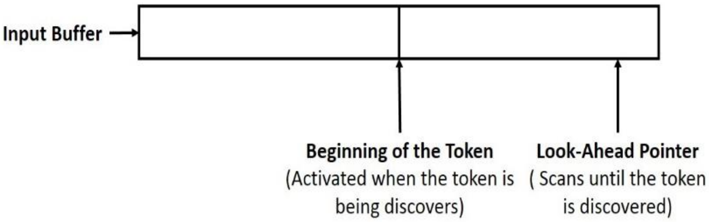

Input Buffer
Beginning of the Token
(Activated when the token is being discovers)
Look-Ahead Pointer
(Scans until the token is discovered)

Figure 2.2 Input Buffering

Eg:

```
{
  int a, b, c;
  c = a+b;
 }
```

2.4.3 Preliminary Scanning

There are certain process that are best performed as characters are moved from the source file to the buffer. For example delete comments, ignore blanks, etc… All these processes may be carried out with an extra buffer.

2.5 Regular Expressions

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules. Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as Regular Grammar. The language defined by regular grammar is known as Regular Language.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

2.5.1 Operations

The various operations on languages are:

1. Union of two languages L and M is written as L U M = {s | s is in L or s is in M}
2. Concatenation of two languages L and M is written as LM = {st | s is in L and t is in M}
3. The Kleene Closure of a language L is written as L* = Zero or more occurrence of language L.

2.5.2 Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- Union : (r)|(s) is a regular expression denoting L(r) U L(s)
- Concatenation : (r)(s) is a regular expression denoting L(r)L(s)
- Kleene closure : (r)* is a regular expression denoting (L(r))*

Note: (r) is a regular expression denoting L(r)

2.5.3 *Precedence and Associativity*

a) (Closure), concatenation (.), and | (pipe sign) are left associative
b) has the highest precedence
c) Concatenation (.) has the second highest precedence.
d) | (pipe sign) has the lowest precedence of all.

For any regular expressions R, S & T the following axioms hold:

i. R|S = S|R (| is commutative)
ii. R | (S|T) = (R|S) | T (| is associative)
iii. R(ST) = (RS)T (· is associative)
iv. R(S | T) = RS | RT & (S | T)R = SR | TR (· is distributive over |).
v. ∈R = R∈ = R (∈ is the identity for concentration).

2.5.4 *Representing valid tokens of a language in regular expression*

If X is a regular expression, then:

1) X* means zero or more occurrence of x. i.e., it can generate { e, x, xx, xxx, xxxx, ... }
2) X^+ means one or more occurrence of X. i.e., it can generate { x, xx, xxx, xxxx ... } or x.x*
3) x? Means at most one occurrence of x.
4) [a-z] is all lower-case alphabets of English language. [A-Z] is all upper-case alphabets of English language. [0-9] is all natural digits used in mathematics.

2.5.4.1 *Representing occurrence of symbols using regular expressions*

a) Letter = [a – z] or [A – Z].
b) Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9] sign = [+ | -].

2.5.4.2 *Representing language tokens using regular expressions*

1) Decimal = $(\text{sign})^?(\text{digit})^+$
2) Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

2.6 Finite Automata

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognized for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

A recognizer/finite automata for a language is a program that takes as input a string x and answers 'yes' if x is a sentence of the language L 'no' otherwise.

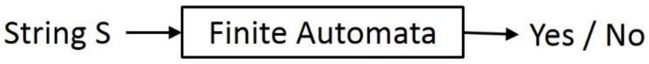


Figure 2.3 Finite Automaton

Types:

1. Non Deterministic Finite Automata (NFA)
2. Deterministic Finite Automata (DFA)

The mathematical model of finite automata consists of:

1. Finite set of states (Q)
2. Finite set of input symbols (Σ)
3. One Start state (q0)
4. Set of final states (qf)
5. Transition function (δ)

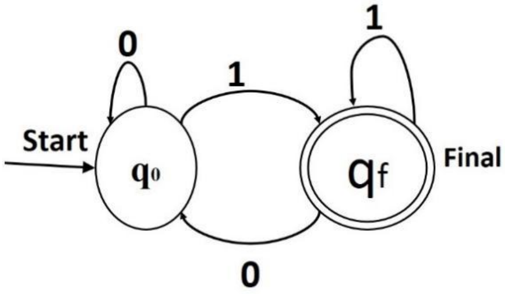
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔Q

2.6.1 NFA Construction

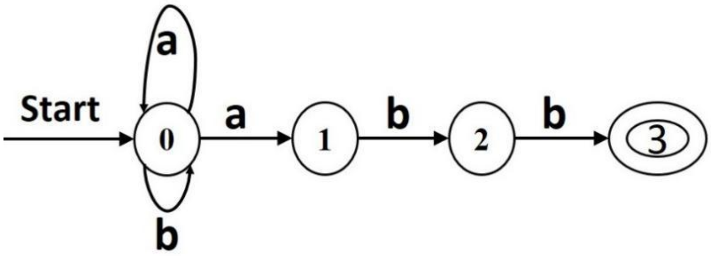
Let L(r) be a regular language recognized by some finite automata (FA).

1) States: States of FA are represented by circles. State names are written inside circles.
2) Start states: The state from where the automata starts is known as the start state. Start state has an arrow pointed towards it.
3) Intermediate states: All intermediate states have at least two arrows; one pointing to and another pointing out from them.
4) Final state: If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. odd = even+1.
5) Transition: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows point to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example 1: We assume FA accepts any three digit binary value ending in digit 1.
FA = {Q(q0, qf), Σ(0,1), q0, qf, δ}



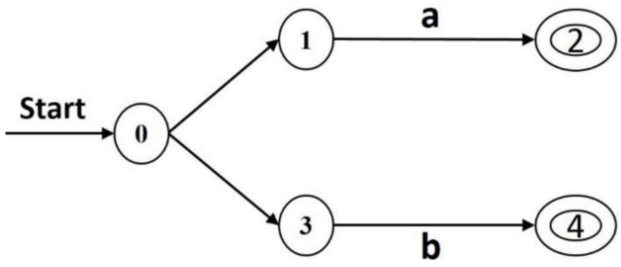
Example 2:
Regular Expression R= (a|b)*abb (FA accepts the string which is ending with abb)



The transitions of an NFA can be conveniently represented in tabular form by means of a transition table.

State	Input(a)	Symbol(b)
0	{0, 1}	{0}
1	-	{2}
2	-	{3}

Example 3: R= aa* | bb*



2.6.2 *DFA Construction*

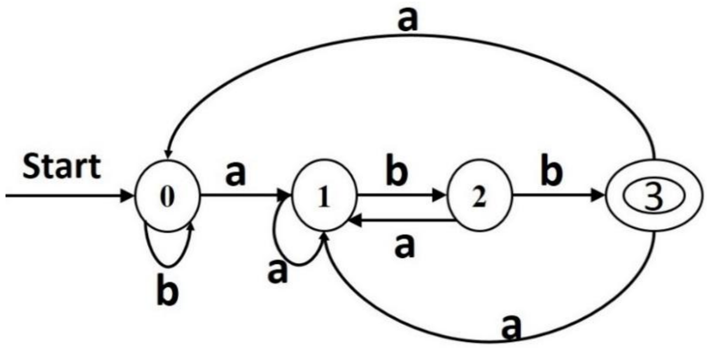
A DFA is a special case of a NFA in which,

1) No state has an ε based {ε} transition on input ε
2) For each state S and input symbol 'a', there is at most one edge labeled 'a' leaving S.

For an Example:

Given Regular Expression: R= (a|b)*abb

NFA for given regular expression R is,



2.6.2.1 Constructing DFA from NFA:

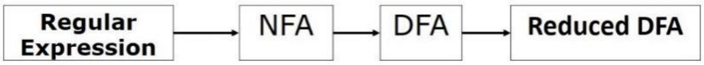


Figure 2.4 Regular Expression to Reduced DFA

An Algorithm for converting a DFA from a NFA:

Input : An NFA N.

Output : A DFA D accepting the same language.

Method :

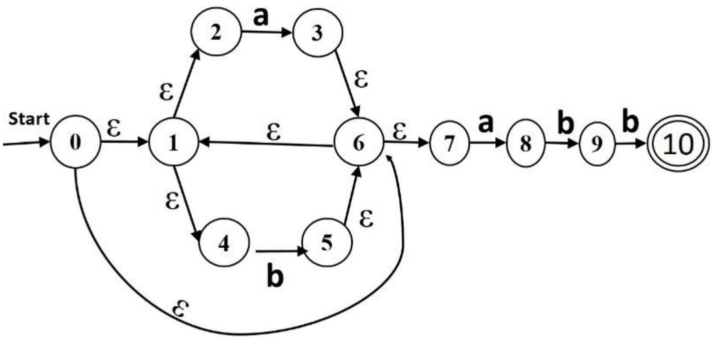
Each state D is a set of state which N could be in after reading some sequence of input symbols. Thus D is able to simulate in parallel all possible moves N can make on a given input string.

Let us define the function ∈- closure(s) to be the set of states of N built by applying the following rules:

1) S is added to ∈-closure(s).

2) If t is in ∈-closure(s) and there is an edge labeled ∈ from t to u, repeated until no more states can be added to ∈-closure(s).

Example: 1



Regular Expression R = (a|b)*abb

Solution:

E-closure(0) = {0, 1, 2, 4, 7} = A

Move(A,a) = {3, 8} →2 reads ‘a’ goes to 3 & 8

Move(A, b) = {5}

E-closure(Move(A,a)) = {3, 6, 1, 2, 4, 7, 8}

Ie. {3, 8} = {1, 2, 3, 4, 6, 7, 8} = B

E-closure(Move(A, b)) = {5, 6, 1, 2, 4, 5, 7}

Ie. {5} = {1, 2, 4, 5, 6, 7} = C

Move (B, a) = {3, 8}

Move (B, b) = {5, 9}

Move (C, a) = {3, 8}

Move (C, b) = {5}

E-closure (Move (B, a)) = {1, 2, 3, 4, 6, 7, 8} = B

{3, 8}

E-closure (Move (B, b)) = {1, 2, 4, 5, 6, 7, 9} = D

{5, 9}

E-closure (Move (C, a)) = {1, 2, 3, 4, 6, 7, 8} = B

{3, 8}

E-closure (Move (C, b)) = {1, 2, 4, 5, 6, 7} = C

{5}

Move (D, a) = {3, 8}

Move (D, b) = {5, 10}

E-closure(Move(D, a)) = {1, 2, 3, 4, 6, 7, 8} = B

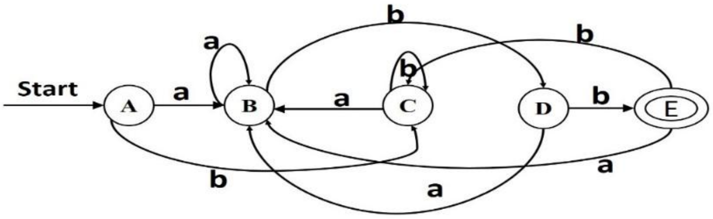
{3, 8}

E-closure(Move(D, b)) = {1, 2, 4, 5, 6, 7, 10} = E
{5, 10}
Move (E, a) = {3, 8}
Move (E, b) = {5}
E-closure(Move(E, a)) = B
{3, 8}
E-closure(Move(E, b)) = C
{5}

Transition table:

States	Input System	
	a	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA:



Minimizing DFA:

π = {A, B, C, D, E}

π_{new} = {A, B, C, D} {E}

$\pi = \pi_{new}$

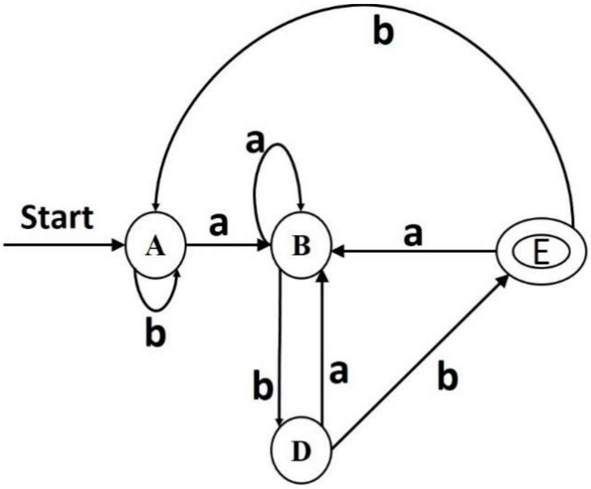
π = {A, B, C, D} {E}

π_{new} = {E} {A, B, C} {D} = π

π = {E} {D} {A, C} {B}

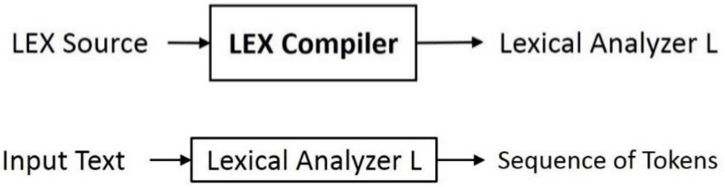
Transition table:

States	Input System	
	a	**b**
A	B	A
B	B	D
D	B	E
E	B	A



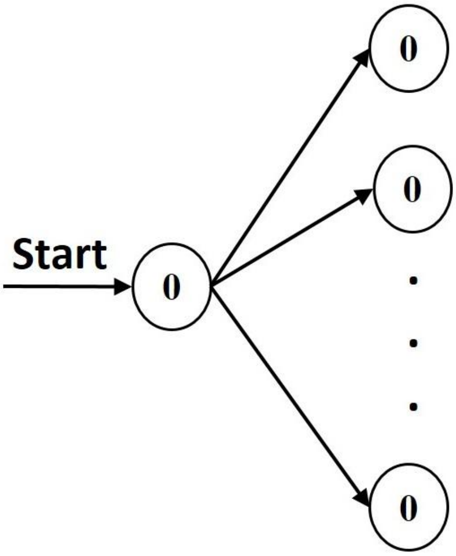
2.7 Implementation of a lexical analyzer

LEX is a tool which automatically generating lexical analyzer for a language L.



2.7.1 Implementation

- LEX can build from its input, a lexical analyzer that behaves roughly like a finite automaton
- The idea is to construct a NFA 'N' for each token pattern P in the translation rules.
- Constructing an NFA from a regular expression and then link these NFA's together with a new start state.

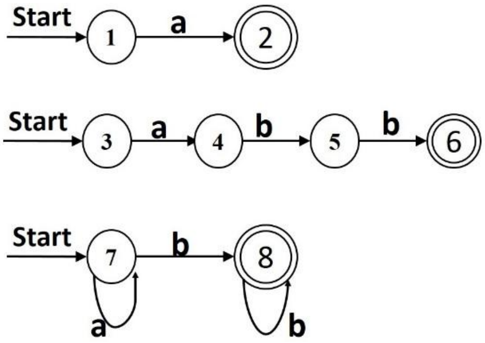


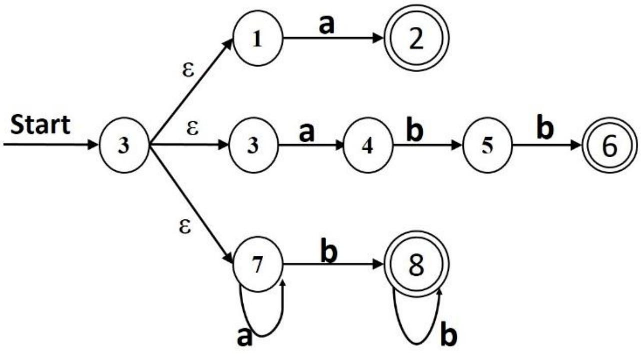
2.7.2 Translation Rules

- Empty strings are omitted.
- String divided based on delimiters such as |, (and) operators.
  Example:
  Regular expression r = a | abb | a* b+

The above regular expression divided into 3 parts for an implementation of FA then they are connected with a single start state.

a {} //are omitted here
abb {}
a*b+ {}





Summary

- Lexical Analyzer is also known as Scanner.
- Lexical Analyzer and parser can be in same pass.
- Lexical Analysis simplifies the overall design.
- Preliminary scanning is also done in the first phase.
- Extra Buffering is needed for preliminary scanning.
- LEX is the tool to construct lexical analyzer.
- The output of lexical analyzer is a stream of tokens.
- Finite Automata is also called recognizer.
- Regular expressions is needed to define the tokens.

Questions

1. Explain the need and role of the lexical analyzer.
2. Describe regular expressions.
3. Construct an NFA for (i) R=(a/b)* a (a/b)
   (ii) R= (a/b)* (a/b)*
4. Convert it into minimized DFA
   (i) aa*/ bb*
   (ii) (a/b)(a/b)(a/b)
   (iii) (a/b)*abb (a/b)*
   (iv) 001*(1|0)*11
   (v) (00)*|(11)*

SYNTAX ANALYSIS

3.1 Introduction

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. Due to the limitations of regular expressions the lexical analyzer cannot check the syntax of a given sentence. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

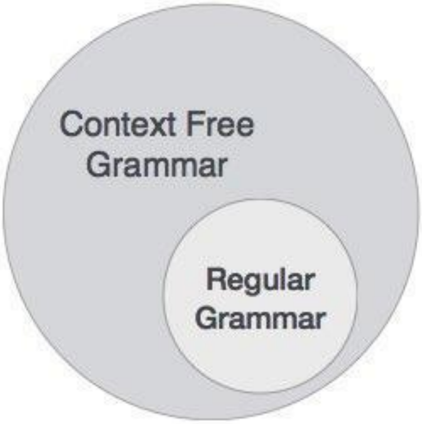


Figure 3.1 Grammar Hierarchy

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

3.2 Context-Free Grammar

In this section, we will first see the definition of context-free grammar and terminologies used in parsing technology.

A context-free grammar has four components:

- A set of non-terminals (N). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as terminal symbols (T). Terminals are the basic symbols from which strings are formed.
- A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, L = { w | w = w^R } is not a regular language. But it can be described by means of CFG, as illustrated below:

G = (N, T, P, S) or G = (V_n, V_t, P, S)

Where,

N or V_n = { Q, Z, N }

T or V_t= { 0, 1 }

P = { Q → Z | Q → N | Q → Ɛ | Z → 0Q0 | N → 1Q1 }

S = { Q }

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

3.3 Syntax Analyzer

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

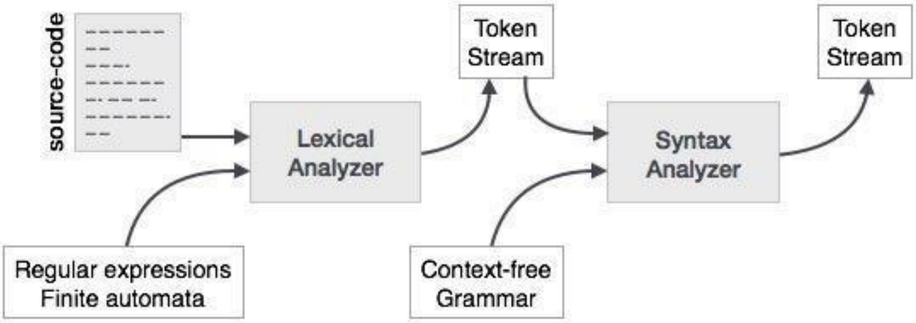


Figure 3.2 Working principle Syntax Analyzer

In this way, the parser accomplishes two tasks, i.e., parsing the code and looking for errors. Finally a parse tree is generated as the output of this phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in Error Handling Chapter.

3.4 Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

1) Deciding the non-terminal which is to be replaced.
2) Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

3.4.1 Left-Most Derivation(LMD)

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

3.4.2 Right-Most Derivation(RMD)

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

E → E + E
E → E * E
E → id

Input string: id + id * id

The left-most derivation is:	The right-most derivation is:
E → E * E E → E + E * E E → id + E * E E → id + id * E E → id + id * id	E → E + E E → E + E * E E → E + E * id E → E + id * id E → id + id * id
Note: In this the left-most side non-terminal is always processed first.	Note: Here the Right-most side non-terminal is always processed first.

3.5 Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

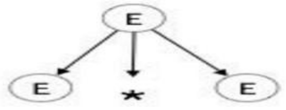
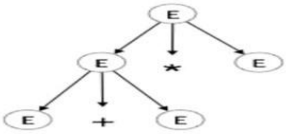
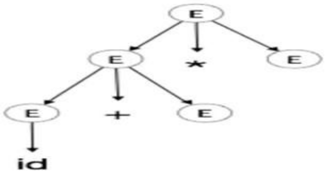
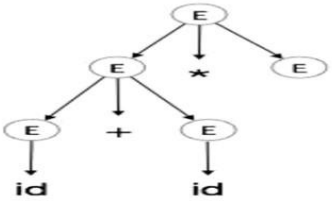
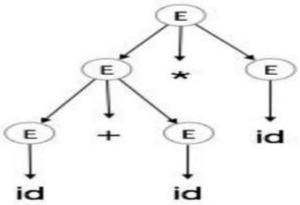
Let us see this by an example,

The given string is, a + b * c

The given Grammar is E→E*E / E+E / id.

The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

Steps	Parse Tree
Step 1 : E → E * E	
Step 2: E → E + E * E	
Step 3: E → id + E * E	
Step 4: E → id + id * E	
Step 5: E → id + id * id	

In a parse tree:

1. All leaf nodes are terminals.
2. All interior nodes are non-terminals.
3. In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Exercise 1:

Consider the grammar

S → iCtS
S → iCtSeS
S → a
S → b

The following example explains bottom up approach of the leftmost derivations for the sentence W=ibtibtaea

S→ aAcBe; A→Ab; A→ b; B→d and the input string is abbcde, we have to reduce it to S.

Abbcde → abbcBe
→ aAbcBe
→ aAcBe
→S

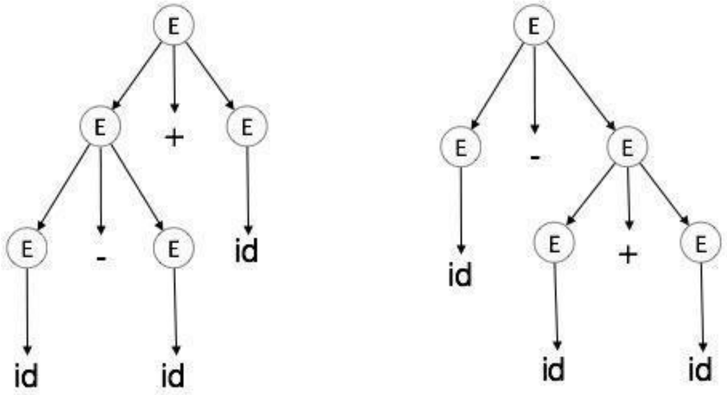
3.6 Ambiguity

A grammar G is said to be an ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example

E → E + E
E → E – E
E → id

For the string id + id – id, the above grammar generates two parse trees:



When the non-terminal on the right side of given production depends on the non-terminal on the left side of the same production, the grammar thus formed is called inherently Ambiguous. From the above example, the language generated by an ambiguous grammar is said to be inherently ambiguous. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

3.6.1 Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator; or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

id op id op id

it will be evaluated as:

(id op id) op id

For example, (id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

id op (id op id)

For example, id ^ (id ^ id)

3.6.2 Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4). By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as:

2 + (3 * 4)

These methods decrease the chances of ambiguity in a language or its grammar.

3.6.3 Using Ambiguous Grammars

Let the natural ambiguous grammar for arithmetic expressions with operator + and * be,

E→E+E\E*E\(E)\id

Assuming that the precedence and associativity of the operators + and * has been specified elsewhere

There are 2 reasons for using this grammar instead of using

E→E+T,E→T,T→T*F,T→F,F→(E),F→id

1) We can easily change the associative and precedence levels of the operator + and * without disturbing the production in 1 or the number of states in the resulting parse.
2) The parser for the unambiguous grammar will spend a substantial of its time reducing by the single productions E→T and T→F ,where role function if to enforce associativity and precedence information .the parser for 1 will not waste time reducing by single productions

3.7 Left Recursion

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

Example:

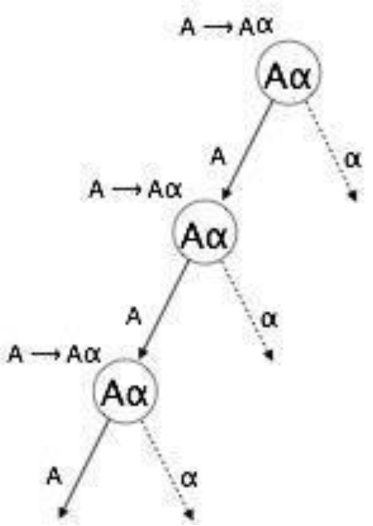
(1) A => Aα | β

(2) S => Aα | β

A => Sd

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

3.7.1 Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

A => Aα | β

is converted into following productions A=> βA'

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

Example

The production set

S => Aα | β

A => Sd

after applying the above algorithm, should become

S => Aα | β

A => Aαd | βd

and then, remove immediate left recursion using the first technique.

A => βdA'

A' => αdA' | ε

Now none of the production has either direct or indirect left recursion.

Example

Production Rule No	**With Left Recursion** **If the production is A→ Aα | β**	**Without Left Recursion** **A→βA'** **A' => αA' | ε**
1	E→E+T | T	E→ TE' E'→ +TE' | ε
2	T→T*F | F	T→FT' T'→ *FT' | ε
3	F→(E) | id	No need for Left Recursion.

3.7.2 Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

A ⇒ $\alpha\beta$ | α | …

Determine which production to follow to parse the string, as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

Example

The above productions can be written as

A => αA'

A'=> β | | …

Now the parser has only one production per prefix which makes it easier to take decisions.

3.8 Limitations of Syntax Analyzers

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks:

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- It cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.

3.9 Types of Parsing

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.

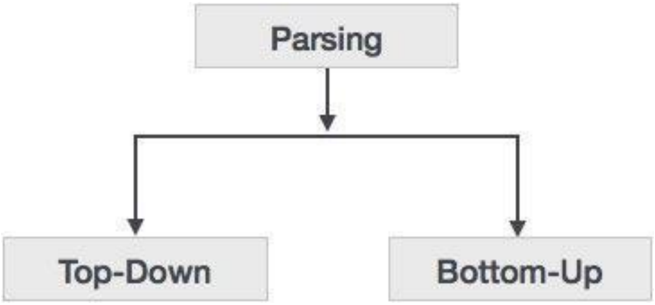


Figure 3.3 Types of Parsing

Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- Recursive descent parsing: It is a common form of top-down parsing. It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Backtracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Note:

In both the cases the input to the parser is being scanned from left to right, one symbol at a time.

The bottom-up parsing method is called “Shift Reduce” parsing. The top-down parsing is called “Recursive Decent” parsing.

An operator-precedence parser is one kind of shift reduce parser and predictive parser is one kind of recursive descent parser.

Example:

Input string : a + b * c

Production rules:

S → E
E → E + T
E → E *T
E → T
T → id

Let us start bottom-up parsing.

a + b * c

Read the input and check if any production matches with the input:

a + b * c
T + b * c
E + b * c
E + T * c
E * c
E * T
E
S

3.9.1 TOP-DOWN PARSING

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:

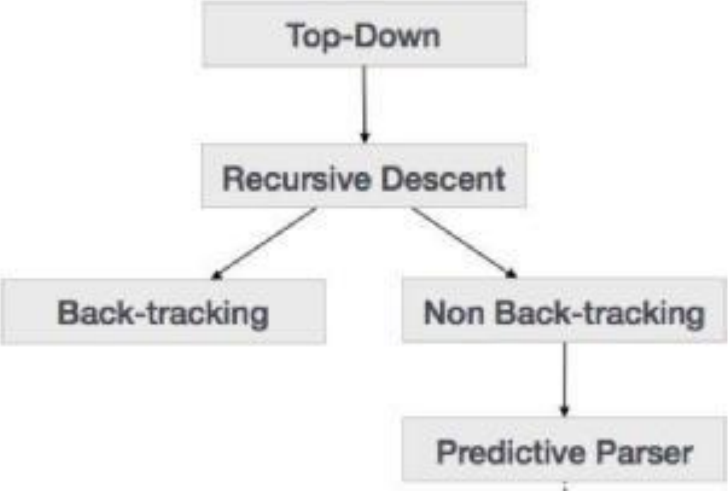


Figure 3.4 Top down Parser

3.9.1.1 Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

This parsing technique is regarded recursive, as it uses context-free grammar which is recursive in nature.

Back-tracking

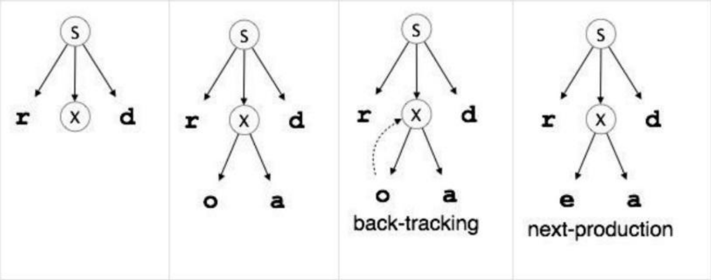
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

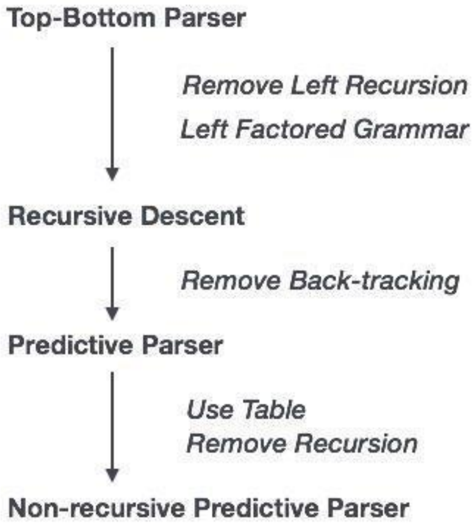
S → rXd | rZd
X → oa | ea
Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

Now the parser matches all the input letters in an ordered manner. The string is accepted.





In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input; whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

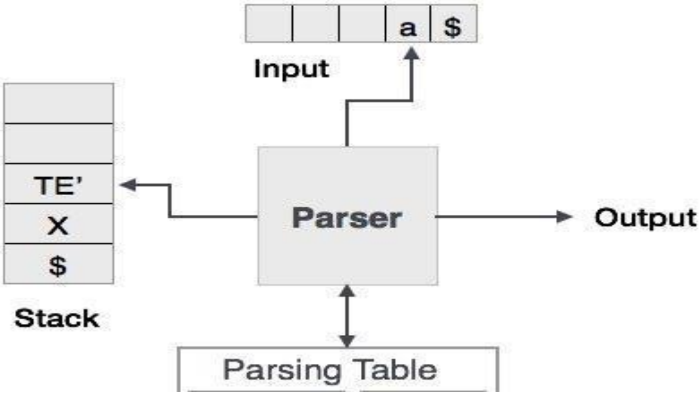


Figure 3.5 Principle of Predictive Parser

Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

Steps to be involved in Parsing Method:

- ✓ Stack is pushed with $.
- ✓ Construction of parsing table T.
- ▪ Computation of FIRST set.
- ▪ Computation of FOLLOW set.
- ▪ Making entries into the parsing table.
- ✓ Parsing by parsing routine.

Construction of parsing table

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

For example,

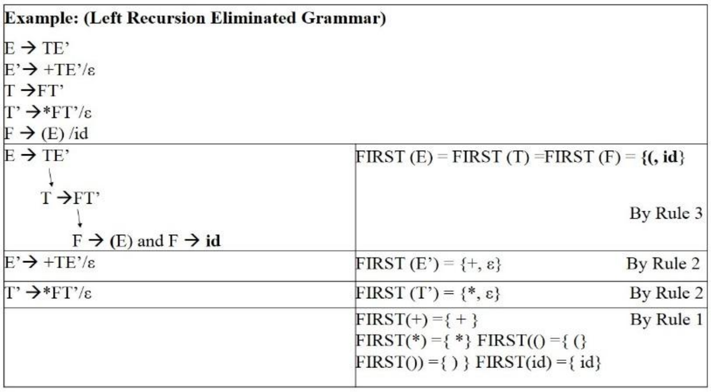
A→ t β

That is, A derives t (terminal) in the very first position. So, t ∈ FIRST(A).

Algorithm for Calculating First Set

Look at the definition of FIRST (X) set:

1) If X is terminal, then FIRST{X}
2) If X is non terminal and X→aα is a production then add a to FIRST(X).if x→ε is a production, then add ε to FIRST(X)
3) If X→Y_1 ,Y_2...Y_k is a production, then ---- I ------- all of Y_1 ,Y_2...Y_{i-1} are non terminals X FIRST(Y_j) contains ε for j=1,2...j_{-1} (i.e. Y_1 ,Y_2...Y_{i-1} *=>ε add every non ε symbol in first (Y_i) for all j=1,2...k then add ε to FIRST (X)



Example: (Left Recursion Eliminated Grammar) E → TE' E'→ +TE'/ε T →FT' T' →*FT'/ε F → (E) /id	
E → TE' ↓ T →FT' ↓ F → (E) and F → **id**	FIRST (E) = FIRST (T) =FIRST (F) = **{(, id}** By Rule 3
E'→ +TE'/ε	FIRST (E') = {+, ε} By Rule 2
T' →*FT'/ε	FIRST (T') = {*, ε} By Rule 2

	FIRST(+) ={ + } By Rule 1
	FIRST(*) ={ *} FIRST(() ={ (}
	FIRST()) ={) } FIRST(id) ={ id}

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal A in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

To compute follow (A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

Algorithm for Calculating Follow Set

1. $ is in follow (S), where S is the start symbol.
2. If there is production $A \rightarrow \alpha B\beta$, $\beta \neq \varepsilon$ then everything in FIRST (β) but ε is 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 ε (i.e. $\beta^{*}=>\varepsilon$) then everything in FOLLOW (A) is in FOLLOW (B).

<table>
<tr><td colspan="2">Example: (Left Recursion Eliminated Grammar)
E → TE’
E’ → +TE’/ε
T → FT’
T’ → *FT’/ε
F → (E) /id</td></tr>
<tr><td>Productions Without ε
Rule 2 Follows
A→ αBβ, β≠ε
FOLLOW (B)= FIRST (β) Except ε</td><td>Productions With ε
Rule 3 Follows
A→αB
Or
A→ αBβ, β=ε
FOLLOW(B)=FOLLOW(A)</td></tr>
<tr><td></td><td>E → TE’
FOLLOW(E’)=FOLLOW(E)= {),$}</td></tr>
<tr><td>E’ → +TE’
E’≠ε
FOLLOW(T)=FIRST(E’)={ + }</td><td>E’ → +TE’
E’=ε
FOLLOW(T)=FOLLOW(E’) = {), +,$ }</td></tr>
<tr><td></td><td>T →FT’
FOLLOW(T’)=FOLLOW(T)= {),+,$ }</td></tr>
<tr><td>T’ →*FT’
T’≠ε
FOLLOW(F)=FIRST(T’)={ * }</td><td>T’ →*FT’
T’=ε
FOLLOW(F)=FOLLOW(T’)= {),+, *, $}</td></tr>
<tr><td>F → (E)
FOLLOW(E)=FIRST(‘)’)={), $}
E is the Start symbol so $ included in FIRST set according to the rule 1.</td><td></td></tr>
<tr><td colspan="2">Result:
FOLLOW (E) =FOLLOW (E’) = {), $}
FOLLOW (T) =FOLLOW (T’) = {), +, $}
FOLLOW (F) = {), *, +, $}</td></tr>
</table>

Then

FIRST (E) = FIRST (T) =FIRST (F) = {(, id}
FIRST (E') = {+, ε}
FIRST (T') = {*, ε}
FOLLOW (E) =FOLLOW (E') = {), $}
FOLLOW (T) =FOLLOW (T') = {), +, $}
FOLLOW (F) = {), *, +, $}

Parsing table for Grammar:
Rules for making entries in to the table:

1. If there is a ε transition means, make entries on FIRST (α) and FOLLOW(α) by Production rule.
   a. On FIRST (α) with corresponding production rule.
   b. On FOLLOW (α) with ε production rule.
2. If there is no ε transition means, make entries on FIRST (α) set only by production rule.

	Id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

3.9.2 Bottom-Up Parsing

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.

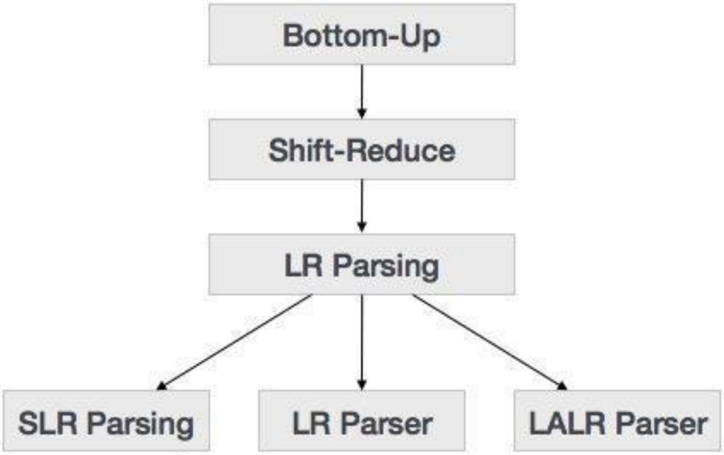


Figure 3.6 Shift Reduce Parsing Methods

3.9.2.1 Shift-Reduce Parsing

It is called as bottom up style of parsing. Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

 Shift step

The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

 Reduce step

When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Reducing a string W to the start symbol S of a grammar.

At each step a string matching the right side of a production is replaced by the symbol on the left.

Example:

S→ aAcBe; A→Ab; A→ b; B→d and the string is abbcde, we have to reduce it to S.

Abbcde → abbcBe
→ aAbcBe
→ aAcBe
→S

Each replacement of the right side of the production the left side in the process above is called reduction .by reverse of a right most derivation is called Handle

$S^{*}\rightarrow \alpha Aw \rightarrow \alpha\beta w$,then $A\rightarrow\beta$ in partition following is a handle of $\alpha\beta w$. The string w to the right of the handle contains only terminal symbol.

A rightmost derivation in reverse often called a canonical reduction sequence, is obtained by "Handle Pruning".

Example:

E→ E+E
E→ E*E
E→(E)
E→id

Input: $id_1+id_2*id_3$→E

Right Sentential Form	Handle	Reducing production
$id_1+id_2*id_3$	id_1	E→id
$E+id_2*id_3$	id_2	E→id
$E+E*id_3$	id_3	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

Ie. This example is the reverse of the sequential in the rightmost derivations.

Stack Implementation of Shift Reduce Parsing

There are two problems that must be solved if we are to automate parsing by handle parsing.

1) To locate a handle in a right sentential form.
2) What production to choose.

There are 4 possible actions a shift reduce parser can make

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 the reduce action the parser knows
- Right end of the handle.
- To locate left end of the handle within the stack.
- Decide what non-terminal to replace the handle.

3) In an accept action, the parser announces successful completion of parsing.
4) In an error action, parser looks for syntax error and calls an error recovery routine.

Example:

E→ E+E
E→ E*E
E→(E)
E→id

Input: $id_1+id_2*id_3$

Stack	Input	Action
$	$Id_1+id_2*id_3\$$	Shift
$\$id_1$	$+id_2*id_3\$$	Reduce by E→id
$E	$+id_2*id_3\$$	Shift
$E+	$id_2*id_3\$$	Shift
$\$E+id_2$	$*id_3\$$	Reduce by E→id
$E+E	$*id_3\$$	Shift
$E+E*	$id_3\$$	Shift
$\$E+E*\ id_3$	$	Reduce by E→id
$E+E*E	$	Reduce by E→E*E
$E+E	$	Reduce by E→E+E
$E	$	Accept

3.9.2.1.2 Operator Precedence Parsing (one kind of Shift Reduce Parsing)

Definition:

A grammar is said to be operator grammar if it has no 2 adjacent non terminals, if it has no production right side is ε_1

Example:

Consider the grammar
E→EAE/(E)/-E/id
A→+/-/*/..... / ↑→is not an operator grammar

Because EAE→α adjacent non terminals
However it can be easily converted into operator grammar as follows.
E→E+E/E*E/E-E/E^E/(E)/-E/id

Precedence relation between pair of terminal:

Relation	Meaning
A< .b	'a' yields precedence
A= .b	'a' has the same precedence as 'b'
A.>b	'a' takes precedence over 'b'

Two ways to determine precedence relations:
1) Intuitive: based on precedence and associative rules of operators
2) Constructing unambiguous grammar.

3) Operator precedence relation from associativity and precedence.

If operator θ_1 has higher precedence than operator θ_2, then

$\theta_{1.\,>}\theta_2$

$\theta_{2\,<\,.}\,\theta_1$

Example:

1) * has higher precedence over +, then the relations are

*. > +

+ <. *

2) If θ_1 and θ_2 are operator of equal precedence then,

$\theta_{1.\,>}\,\theta_2$. & $\theta_{2.>}\,\theta_1$ → left associative

$\theta_{1\,<}\,.\theta_2$. & $\theta_{2<.}\,\theta_1$ → right associative

Example:

+, - are left associative

+ . > + , - .> -

+ .> -, - . > +

↑ has right associative

↑ <.↑

3) Make

Θ<.id; id.> Θ; Θ<.c; (<.Θ

). Θ; Θ.>); Θ.>$;$<. Θ ..... θ and

(=·) ; $<. (;$< .id

(<. (; id.>$;)<. $

(<.id; id.>) ;.>)

Example:

Relation Table

	Id	**+**	*****	**$**
Id		. >	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Definition:

An operator precedence is an € free operator grammar in which the precedence relations <. ,=· &.> constructed are disjoint

Note:

LEADING(A)={a/A$=^{+}>$ γaδ, where γ is € or a single non terminal}

TRAILING(A)={a/A$=^{+}>$ γaδ, where δ is € or a single non terminal}

Precedence Function

Precedence table can be encoded by a precedence functions f & g.

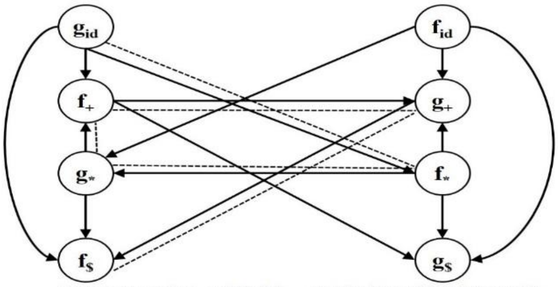
We select f & g for symbol a & b

- f(a)<g(b) whenever a<.b
- f(a)=g(b) whenever a=·b
- f(a)>g(b) whenever a.>b

Example

	Id	+	*	$
Id		. >	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Graph Representation:



Longest root is g_{id} to $f_\$$ is 5 (----The dotted line denotes it)

F$=0 F$=0 Fid=4 Fid=4

F+=2 F+=2

F*=4 F*=4

	+	*	Id	$
f	2	4	4	0
g	1	3	5	0

3.10 LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions.

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms, namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

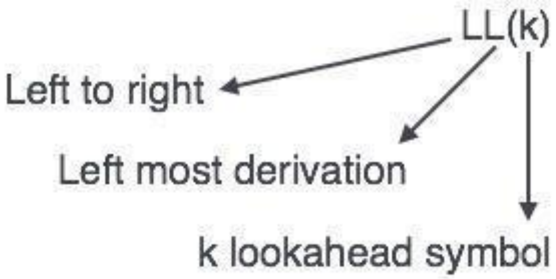


Figure 3.7 LL Parser

3.10.1 LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

There are three widely used algorithms available for constructing an LR parser:

I. SLR(1) – Simple LR Parser:

1. Works on smallest class of grammar
2. Few number of states, hence very small table
3. Simple and fast construction

II. LR(1) – LR Parser:

1. Works on complete set of LR(1) Grammar
2. Generates large table and large number of states
3. Slow construction

III. LALR(1) →Look-Ahead LR Parser:

i. Works on intermediate size of grammar
ii. Number of states are same as in SLR(1)

3.10.2 Simple LR Parser

Simple LR parser has 2 components:

1) Constructing of LR(0) items
2) Constructing of parsing table.

Construction of LR (0) Items

The collection of sets of LR (0) item is called SLR

The collection of sets of LR (0) items can be constructed with the help of functions called CLOSURE and GOTO functions. This collection is called canonical collection of LR (0) items.

Step 1: Creating augmented grammar

Consider the grammar G & S is the start symbol. The augmented grammar of G is with a new start 'S' and having a production S'→S

Example:

Grammar G	The augmented grammar of G'
E→E+T E →T T→T*F T→F F →(E) F→id	E'→E E→E+T E→T T→T*F T→F F →(E) F→id

Step 2: CLOSURE ()

Let us say I is a set of items for a grammar G, then CLOSURE of I can be computed by using the following steps.

1. Initially, every item in I is added to CLOSURE(I)
2. Consider the following,

If A→X.BY an item in I, and B→Z production then add this production in I in the following form B→ .Z ,but if it is not already there.

Example:

Closure (E'→.E) = E'→.E E→.E+T E→.T T→.T*F T→.F F→. (E) F→.id	I_0

Step 3: perform GOTO (I, X) (I is CLOURE set and X is all Grammar symbol)

This is computed for a set I on a grammar symbol X. GOTO (I, X) is defined to be the closure of the set of all items [A→αX.β] such that [A→α.Xβ] is in I.

Consider an item in I's production is maybe like this means,

A→.XBY then

GOTO (I, X) will be performed based on the following Rules:

1. If A→X.BY where B is a Terminal, including this item only in the CLOSURE (X) Item.
2. If A→X.BY where B is a Non-Terminal including this item along with B's CLOSURE (B).

Example:

I_0 : E'→.E E→.E+T E→.T T→.T*F T→.F F→. (E) F→.id

GOTO (I_0, E) = E'→E. E→E. +T	I_1
GOTO (I_0, T) = E→T. T→T.*F	I_2
GOTO (I_0, F) = T→F.	I_3
GOTO (I_0,+) =GOTO(I_0,*)= GOTO (I_0.))= null	
GOTO(I_0,() = F→(.E) E→.E+T E→.T T→.T*F T→.F F→. (E) F→.id	// 2 rule in step 3 is applied here I_4
GOTO (I_0, id) =F→id	I_5

I_1: E'→E. E→E. +T is GOTO (I_1, X)

GOTO (I_1, E) =GOTO (I_1, T) = GOTO (I_1, F) =GOTO (I_1,*) = GOTO (I_1,	

() =GOTO (I_1,))=GOTO (I_1, id) =Null	
GOTO (I_1, +) =E→E+.T T→.T*F T→.F F→. (E) F→.id	// Rule 2 in step 3 I_6

I_2: E→T.
T→T.*F GOTO(I_2,X)

GOTO(I_2,E)=GOTO(I_2,T)=GOTO(I_2,F)=GOTO(I_2,+)=GOTO (I_2,() = GOTO (I_2,)) =GOTO (I_2,id)=Null	
GOTO(I_2,*)=T→T*.F F→.(E) F→.id	// Rule 2 in step 3 I_7

I_3: T→F. I.e. GOTO(I_3,X)

GOTO(I_3,E)=GOTO(I_3,T)=GOTO(I_3,F)=GOTO(I_3,*)= GOTO(I_3,+)= GOTO(I_3,()=GOTO(I_3,))=GOTO(I_3,id)=null ;	

I_4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→. (E)
F→.id GOTO(I_4,X) is,

GOTO(I_4,E)= F→(E.) E→E.+T	I_8
GOTO(I_4, T)= T→T.*F E→ T .	I_2
GOTO(I_4,F)= T→F.	I_3
GOTO(I_4,+)=Null ; GOTO(I_4,*)=null ; GOTO(I_4,))=Null .	
GOTO(I_4,()= F→(.E) E→.E+T E→.T T→.T*F T→.F F→(.E) F→.id	I_4
GOTO(I_4,id)=F→id.	I_5

I_5: F→id.

GOTO(I_5,E)= GOTO(I_5,T)= GOTO(I_5,F)= GOTO(I_5,+)= GOTO(I_5,*)=	

GOTO(I_5,()=GOTO(I_5,))= GOTO(I_5,ID)=Null	
I_6: E→E+.T T→.T*F T→.F F→. (E) F→.id	
GOTO (I_6,E)=GOTO(I_6,+)=GOTO(I_6,*)= GOTO(I_6,))= Null	
GOTO (I_6,T)=E→E+T. T→T.*F	I_9
GOTO (I_6,F)=T→F.	I_3
GOTO (I_6,()=F→(.E) E→.E+T E→.T T→.T*F T→.F F→. (E) F→.id	I_4
GOTO(I_6,id)=F→id	I_5
I_7 : T→T*.F F→.(E) F→.id	
GOTO(I_7,E)=GOTO(I_7,T)=GOTO(I_7,+)= GOTO(I_7,*)= GOTO(I_7,))=Null	
GOTO(I_7,E)=T→T*F.	I_{10}
GOTO(I_7,() =F→(.E) E→.E+T E→.T T→.T*F T→.F F→.(E) F→.id	I_4
GOTO(I_7,id)=F→id.	I_5
I_8: F→(E.) E→E.+T	
GOTO(I_8,E)=GOTO(I_8,T)=GOTO(I_8,F)=GOTO(I_8,*)= GOTO(I_8,()= GOTO(I_8,id)=Null	
GOTO(I_8,+)= E→E+.T T→.T*F T→.F F→.(E) F→.id	I_6
GOTO (I_8,)) =F→(E).	I_{11}
I_9: E→E+T.	

T→T.*F	
GOTO(I_9,E)= GOTO(I_9,T)= GOTO(I_9,F)= GOTO(I_9,+)= GOTO(I_9,()= GOTO(I_9,))= GOTO(I_9,id)=Null	
GOTO(I_9,*)= T→T*.F F→.(E) F→id	I_7
I_{10}: T→T*F.	
GOTO(I_{10},E)=GOTO(I_{10},T)=GOTO(I_{10},F)=GOTO(I_{10},+)= GOTO(I_{10},*)= GOTO(I_{10},()= GOTO(I_{10},))= GOTO(I_{10},id)=Null	
I_{11}: F →(E).	
GOTO(I_{11},E)=GOTO(I_{11},T)=GOTO(I_{11},F)=GOTO(I_{11},+)= GOTO(I_{11},*)= GOTO(I_{11},()= GOTO(I_{11},))= GOTO(I_{11},id)=Null	

LR (0) items:

I_0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→id.

I_1: E'→E.
E→E. +T

I_2: E→T.
T→T.*F

I_3: T→F.

I_4: F →(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→. (E)
F→.id

I_5: F→id.

I_6: E→E+ .T
T→.T*F
T→.F
F→. (E)

F→.id

I_7: T→T* .F
F→. (E)
F→.id

I_8: F →(E.)
T→E. +T

I_9: E→E+T.
T→T. *F

I_{10}: T→T*F.

I_{11}: F →(E).

Construction of SLR parsing Table:

This is also a 2 dimensional array in which the rows are states & columns are terminals & non terminals. This table has 2 parts,

1) Action
2) Go to entries

The action may one of the following:

1) Shift
2) Reduce
3) Accept
4) Errors

Steps for Constructing SLR parsing table:

1. Let C={I_0,I_1…I_n} is the collection of sets of LR (0) items.
2. Consider I_j as a set in C, then
   If GOTO (I_j, a) =I_k then set action [j,a] to shift k, 'a' is always terminal.
   If A→x. (x can be either terminal /non terminal) is in I_j then set action [j,a] to reduce A→x …. …… 'a' in FOLLOW (A) if x is a terminal. If x is a non-terminal set action [j, a] to reduce A→x …. …… 'a' in FOLLOW (X).
3. If S'→S is in I_j then set action [j,$]=k Accept.
4. If GOTO [I_j, A] =$I_{K,\text{ then}}$ set GOTO (j, a) =k.
5. All the under defined entries are errors.

Example:

1) E→E+T. is in I_9 ,FOLLOW (T) ={+,$,)}
2) E→T. is in I_2 ,FOLLOW (T) ={+,$,)}
3) T→T*F. is in I_{10} ,FOLLOW (F) ={+,*,$,)}
4) T→F. is in I_3 ,FOLLOW (F) ={+,*,$,)}
5) F→(E). is in I_{11} ,FOLLOW ( )) =FOLLOW(F)={+,*,$,)}
6) F→id. is in I_5 ,FOLLOW (id) =FOLLOW(F)={+,*,$,)}

States	**ACTION entries in Terminals**	**GO TO entries in Non-Terminals**

	Id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	S_5			S_4			1	2	3
1		S_6				A			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

SLR parsing Algorithm

This algorithm works in conjunction with the parsing table, for parsing an input string .The possible action are as follows:

1) Shift
2) Reduce
3) Accept
4) Errors

The input string is in I/p buffer followed by the right end marker $. The stack keeps the states of the parsing table

The I/p string has 'n' symbols & are marked by a_1, a2…a_n and the stack has states S0, S1, S2…S_m

Steps involved in SLR parsing

1) If action [S_0,a_1]=S_j, the parser has to make a Shift of the current i/p symbol & a new state will be 'j' on the stack.
2) If action [S_m,a_j]=Reduce by A→x,then the parser has to reduce by A→x .find out the no. of symbols available on the right hand side of A after " →" .let us say it r ,then POP of (2,r) symbols from the stack.

(2.1) if GOTO [S_{m-r},A]=J(states) then PUSH A onto the stack.

3) If action [S_m,a_j]=Accept ,then announce that the parsing is completed successfully and then halt.
4) If action [S_m,a_j]=Error ,then the parser encounter error and calls error recovery routine or generates error message.

Example: Input: id +id

Stack	Input	Event
0	Id+id$	Initial State
0id$_5$	+id$	Shift
0F$_3$	+id$	Reduce by F→id
0T$_2$	+id$	Reduce by T→F.
0E$_1$	+id$	Reduce by E→T.
0E$_1$+6	id$	Shift
0E$_1$+6 id$_5$	$	shift
0E$_1$+6F$_3$	$	Reduce by F→id

$0E_1+6\ T_9$	$	Reduce by T→F
$0E_1$	$	Reduce by F→E+T

LL vs. LR	
LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Summary

- Syntax analyzer is also called parser.
- Context free grammar is used as recognizer.
- Parse tree is an output of Parser
- Precedence rules are used to operate the operators.
- Two types of parsing, Bottom up parsing and Top Down parsing.
- Recursive descent parsing is a kind of Top Down parsing.
- Shift reduce parsing is a kind of Bottom Up parsing.
- Predictive Parser is a kind of Recursive Decent parser.

Questions

1. Explain the types of Parser.
2. Construct a Parsing Table for the Grammar
   E→E+T/T
   T→T*F/F
   F→(E)/id

3. Differentiate left recursion and left factoring.
4. Explain the stack implementation of shift reduce parsing.
5. Describe different kinds of LR parser with an example.

ERROR HANDLING

4.1 Introduction

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- Lexical Error: name of some identifier typed incorrectly
- Syntactical Error : missing semicolon or unbalanced parenthesis
- Semantical Error: incompatible value assignment
- Logical Error: code not reachable, infinite loop

When an error is detected the reaction of compiler different.
(a) A system crash
(b) To emit invalid output
(c) To merely quit on the first detected error.

4.1.1 Reporting Errors

- Good error diagnostics should possess a number of properties,
- The massage should pinpoint the errors in terms of the original source program rather than in terms of some internal representation
- The error message should be understandable by the user
- The message should be specific and should localize the properties.
- The message should not be redundant.

4.1.2 Sources of Errors

- The insertion of an extraneous character or token.
- The deletion of a required character or token.
- The replacement of correct character or token by an incorrect character or token.
- The transpiration of two adjacent characters or tokens.

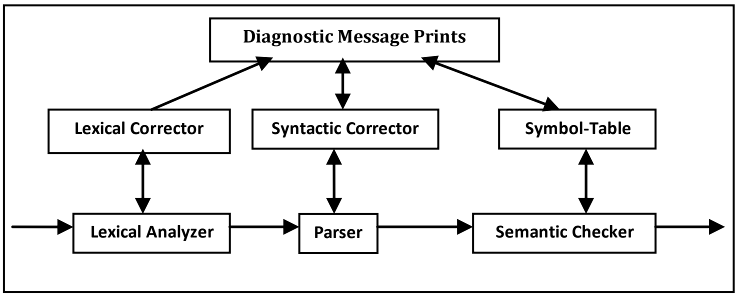


Figure 4.1 Error Handling

4.1.2.1 Lexical Phase Errors

- Minimum Distance Matching -> Spelling

4.1.2.2 Syntactic – Phase Errors:

- Minimum distance correction of syntactic errors.
- Black (e.g. If-else) (spelling of token)
- Time of detection LL (1) and LR (1). Two or more production E-> E+E/E*E
- Panic Mode:
- Crude but effective systematic method of error recovery in any kind of parsing.
- Here parser discards input symbol until a "Synchronizing" token (;) is encountered.

Error recovery in operator – precedence parsing
Parser can discover syntactic errors:

1. If no precedence relation holds between the terminals a top of the stack and the current input symbol.
2. If a handle has been found but there is no production with this handle as a right side.
    - Handling errors during reduction (No production rule to reduce)
    - Handling shift reduce errors.
    - Error recovery in LR parsing (place where there is no entry)
    - Mid-Hoc error recovery for LR parsers (loop)

4.1.2.3 Semantic Errors

Undeclared name and type incompatibilities.

4.2 Error Recovery

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

4.2.1 Panic Mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

4.2.2 Statement Mode

When a parser encounters an error, it tries to take corrective measures so that the rest of the inputs of the statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon, etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

4.2.3 Error Productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

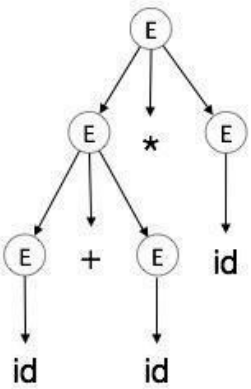
4.2.4 Global Correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.

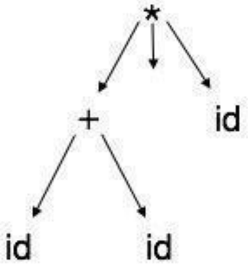
When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

4.3 Abstract Syntax Trees(AST)

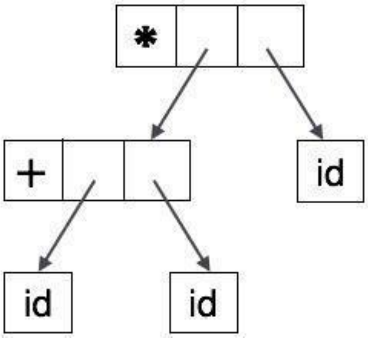
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

Summary

- Errors may be encountered at various stages of the compilation process.
- The parser may discover Syntactic errors.
- Syntactic occurs when there no precedence errors.
- AST is a data structure more compact than the parse tree.
- Augmented grammar is used to generate erroneous constructs.
- Global error correction may done.

Questions

1. Explain various types of errors.
2. Describe error handling with neat diagram.
3. Write note on panic mode.
4. Differentiate parse tree and abstract syntax tree.

SEMANTIC ANALYSIS

5.1 Introduction

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example:

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

Should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

5.2 Semantic Actions

- Syntax directed translation schema is merely a context-free grammar in which a program fragment called an output action is associated with each production.
- A value associated with a grammar symbol is called a translation of that symbol.

5.2.1 The syntax directed translation:

- Schema allows subroutines or semantic actions to be attached to the productions of a context free grammar.
- These subroutines generate intermediate code when called at appropriate time by a parser for that grammar.
- It enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.

Example :

Production	Semantic Action
E→E +E	{E.Val :=E .val+E .val}

- The semantic action is enclosed in braces, and it appears after the production.
- This translation is not suitable for a compiler, but for a “DESK CALCULATOR” program that actually evaluates expressions rather than generating code for them.

Consider, A → XYZ { Y.Val := 2* A.Val }

Here, the translation of a non-terminal on the Right side of the production is defined in terms of non-terminal on the Left side. Such a translation is called inherited translation.

5.2.2 Translation on the parse tree

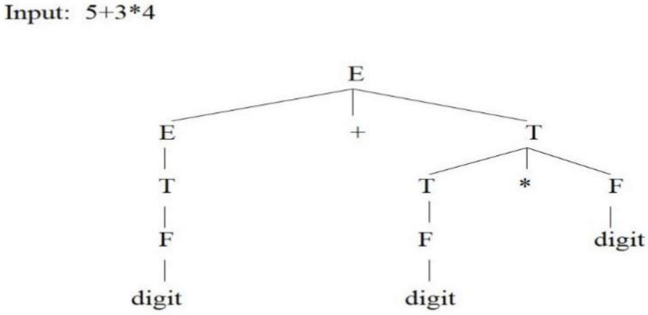
Consider the following syntax-directed translation schema suitable for a "DESK CALCULATOR" program in which E.Val is an integer-valued translation.

PRODUCTION	SEMANTIC ACTION
E→ E + E	{E.VAL :=E .VAL+E .VAL}
E →Digit	{E.VAL := digit }

Here, digit stands for any digit between 0 & 9.

Example:

Let the input string be 5+3*4, and then the parse tree is



5.2.3 Implementation Of Syntax-Directed Translators

- The syntax directed translation scheme is a convenient description used to construct the parse tree, mechanism used to compute the translation.
- Describes an input-output mapping.
- One way to implement a syntax-directed translator is to use extra fields in the parser stack entries corresponding to the grammar symbol.

There are 2 steps to implement:

- Decide what intermediate code to generate for each programming language construct.
- Implement an algorithm for generating this code.

Example :

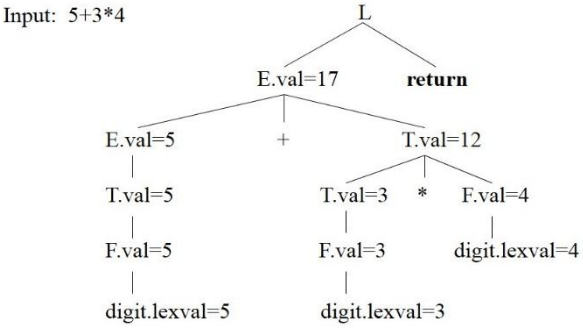
S→ E $
E→ E+E
E→ E*E
E→ (E)
E→ (I)
I→ I
I→ I Digit

[Note I – for integer]

- To implement this syntax – directed translation scheme, we need to construct a lexical analyzer and a bottom-up parser.
-

PRODUCTION	SEMANTIC ACTION
1. S→E$	{print E.Val}
2. E→E + E	{E.Val : =E.Val+E.Val}
3. E→E * E	{ E.Val : =E.Val*E.Val }
4. E→(E)	{ E.Val : =E.Val}
5. E→I	{ E.Val : =I.Val }
6. I→I digit	{I.VAL: = 10*I.VAL+LEXVAL}
7. I→digit	{ I.VAL: = LEXVAL }

5.3 Parse Tree (Annotated parse tree / Dependency Graph):



- A compiler – compiler would tie the parser and the semantic action program fragments together, producing one module.

PRODUCTION	PROGRAM FRAGMENT
1. S→E$	Print VAL [TOP]
2. E→E+E	VAL[TOP]:=VAL[TOP]+VAL[TOP-2]
3. E→E*E	VAL[TOP]:=VAL[TOP]*VAL[TOP-2]
4. E→(E)	VAL[TOP]:=VAL[TOP-1]
5. E→I	none
6. I→I digit	VAL[TOP]:=10*VAL[TOP]+LEXVAL
7. I→digit	VAL[TOP]:=LEXVAL

Input : 23*5+4$

NO	INPUT	STATE	VAL	PRODUCTION USED
1.	23*5+4$	--	--	
2.	3*5+4$	2	--	
3.	3*5+4$	I	2	I → digit
4.	*5+4$	I3	2-	
5.	*5+4$	I	(23)	I → I digit
6.	*5+4$	E	(23)	E→I
7.	5+4$	E*	(23)-	
8.	+4$	E*5	(23)- -	
9.	+4$	E*I	(23)-5	I → digit
10.	+4$	E*E	(23)-5	E→I
11.	+4$	E	(115)	E→E*E
12.	4$	E+	(115)-	
13.	$	E+4	(115)- -	
14.	$	E+I	(115)-4	I → digit

15.	$	E+E	(115)-4	E→I
16.	$	E	(119)	E→E+E
17.	--	E$	(119)-	
18.	--	S	--	S→ E$

[Sequence of moves]

5.4 Semantic Errors

We have mentioned some of the semantic errors that the semantic analyzer is expected to recognize:

1. Type mismatch
2. Undeclared variable
3. Reserved identifier misuse
4. Multiple declaration of variable in a scope
5. Accessing an out of scope variable
6. Actual and formal parameter mismatch

5.5 Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

5.6 Synthesized Attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node.

Synthesized attributes never take values from their parent nodes or any sibling nodes.

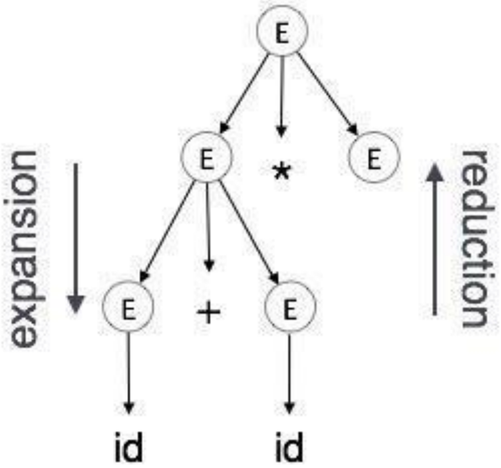
5.7 Inherited Attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B, and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion: When a non-terminal is expanded to terminals as per a grammatical rule.



Reduction

When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

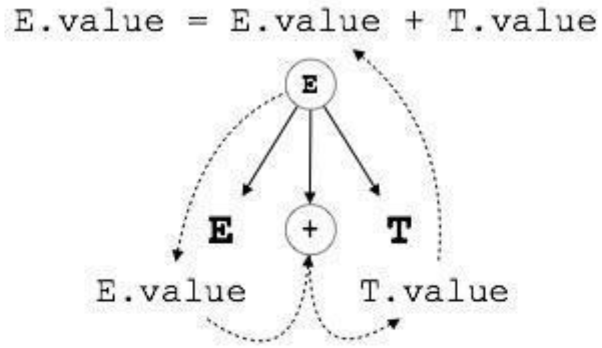
<type, “integer”>

<presentvalue, “5”>

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production,

S → ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

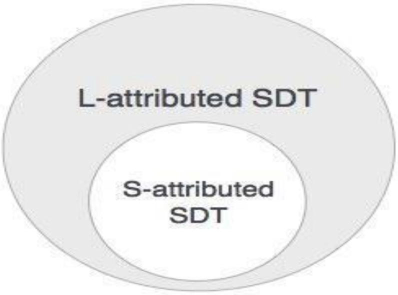


Figure 5.1 Hierarchy of SDT

We may conclude that if a definition is S-attributed, then it is also L-attributed, as L-attributed definition encloses S-attributed definitions.

5.8 Runtime Environment

A program as a source code is merely a collection of text (code, statements, etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers, etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

5.9 Activation Trees

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries		Stores temporary and intermediate values of an expression.
Local Data		Stores local data of the called procedure.
Machine Status		Stores machine status such as Registers, Program Counter, etc.,
		before the procedure is called.
Control Link		Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.	
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send	
	input to the called procedure.	
Return Value	Stores return values.	

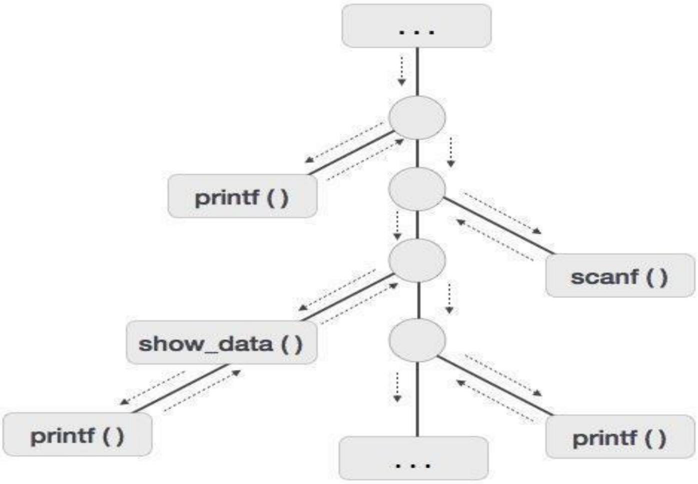
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.

To understand this concept, we take a piece of code as an example:

```
printf(“Enter Your Name: “); scanf(“%s”, username); show_data(username);
printf(“Press any key to continue…”);
. . .
int show_data(char *user)
{
printf(“Your name is %s”, username); return 0;
}
```

Given below is the activation tree of the code:



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

SUMMARY

- Syntax directed translation is a context free grammar.
- Syntax directed translation is helps to generates Intermediate code.
- Sematic actions is enclosed in curly braces.
- The translation of a non-terminal on the Right side of the production is defined in terms of non-terminal on the Left side is called inherited translation.
- Input and output mapping is described using Syntax directed translator.
- The parse tree having values in the node is called Annotated Parse tree.

Questions

1. Explain syntax directed translation.
2. Define sematic actions.
3. Write note on decency graph.
4. Generate a sematic action for production S→ E$

E→E + E/ E*E/(E)/(I)
I→I/Idigit (Where I is a Integer)

6. SYMBOL TABLE

8.1 Introduction

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

8.2 Symbol Tables

- A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a Symbol Table.
- Thus, the information collected about the name includes the string of characters by which it is denoted, its type (e.g. Integer, real, string), its form (e.g. simple variable a structure), its location in memory and other attributes depending on the language
- Each entry in the symbol table is a pair of the form (name and information)
- Each time a name is encountered, the symbol table is searched to see whether that name has been seen previously. If it is new, then it is entered into the table.
- Information about the name is entered into the symbol table during lexical and syntactic analysis.
- Symbol table is used in the several stages of the compiler.

8.2.1 The Contents of a Symbol Table

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

- Using a Symbol Table, we can able to
  (i) determine whether a given name is in the table
  (ii) add a new name to the table
  (iii) access the information associated with a given name, and
  (iv) add new information for a given name
  (v) delete a name or group of names from the table
- There may be separate tables for variable names, labels, procedure names, constants, field names and other types of names depending on the language.
- Depending on how lexical analysis is performed, it may be useful to enter keywords (reserved keywords) into the symbol table initially. If not a warning may occur.
- Let us consider the data can be associated with a name in the symbol table, This information includes,

1. The string of characters denoting the name.
2. Attributes of the name and information identifying what use is being made of the name.
3. Parameters, (Dimensions of arrays etc.)
4. An offset describing the partition in storage to be allocated for the name.
5. The syntax of the language may also implicitly declared variables to play certain role.

8.2.1.1 Names and Symbol – Table Records

- The simplest way to implement a symbol table is as a linear array of records, one record per name.
- A record consists of number of consecutive words of memory, identifier.

8.2.1.2 Reusing Symbol – Table space

- The identifier used by the programmer to denote a particular name must be preserved in the symbol table until no further references to that identifier can possibly denote the same name.
- This is essential so that all users of the identifier can be associated with the same symbol table entry, and hence the same name.
- A compiler can be designed to run in less space if the space used to store identifiers can be reused in subsequent passes.

8.2.1.3 Array Names

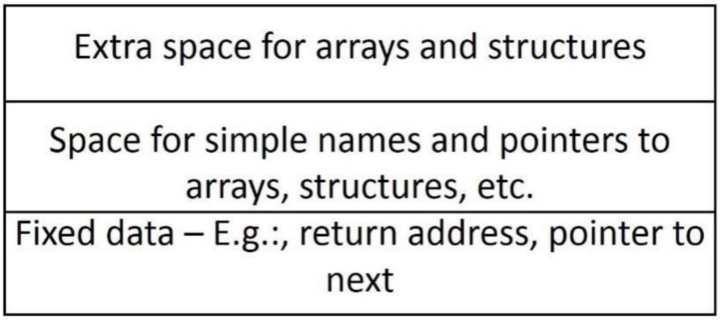
- If the language places a limit on the number of dimensions, then all subscript information can in principle, be paced in the symbol table record itself
- The upper limit and lower limit of a dynamically allocated array can be any expression evaluate at run time, when the storage is allocated for the array.
- If an expression is a constant, its value can be stored in the symbol table.
- If a limit is declared to be an expression, the compiler must generate code to evaluate that expression and assign the result to a temporary variable T.

8.2.1.4 **Indirection In Symbol-Table Entries**

- Designing Symbol-Table formats that have pointers to information that is of variable length.
- Save space i.e. allocating in each symbol-table entry the maximum possible amount of space.
- The most significant advantage in using indirection comes when we, have a type of information that is applicable to only a minority of the entries.

8.2.1.5 Storage Allocation Information

- To denote the locations in the storage belonging to objects at run time.
- Static storage → if the object code is assembly language → generating assembly code →scan the symbol table → generates definition → appended to the executable portion.
- Machine code → generated by compiler – stored with a fixed origin.
- The same remark applies to blocks of data loaded as a module separate from the executable program.
- In the case of names where storage is allocated on a stack, the compiler need not allocate storage at all.
- The compiler must plan out the activation record for each procedure.



Extra space for arrays and structures

Space for simple names and pointers to arrays, structures, etc.

Fixed data – E.g.:, return address, pointer to next

Figure 6.1 Activation Record

8.3 Implementation of Symbol Table

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only.

8.3.1 Data Structure for Symbol Tables

- In designing a Symbol-Table mechanism, there should be a scheme that allows, adding new entries and finding existing entries in a table efficiently.
- A symbol table can be implemented in one of the following ways:
  1. Linear (sorted or unsorted) list
  2. Binary Search Tree
  3. Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

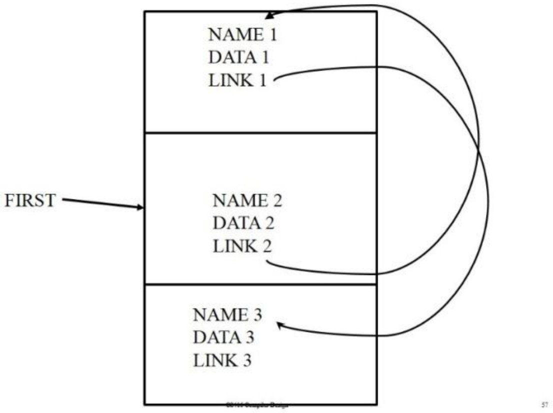
- Each scheme is evaluated on the basis of the time required to add n entries and make in enquiries

8.3.2 Lists

- Simple and easy to implement.
- Use a single array or equivalent several arrays to store names and associated information.
- To retrieve information about a name, we have to search from the beginning of the array up to the partition marked by pointer AVAILABLE, which indicates the beginning of the empty portion of the array.
- To add a new name, stores it immediately following AVAILABLE and increase the pointer by the width of a symbol-table record.

Self-Organizing Lists

- Needs little extra space, save a little bit time.
- Three fields, NAME1, DATA1, and LINK1 are there.

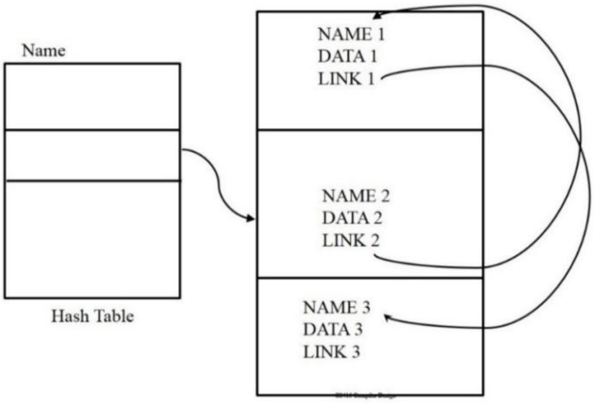


8.3.3 Search Trees

- To add two link fields, LEFT, RIGHT, to each record.
- Use these fields to link the records into a binary search tree.

```
While p ≠ null do
    If NAME =NAME (p) then ……. /* Name found, take action on success
    Else if NAME<NAME (p) then p: =LEFT (p) /*visit left child*/
Else if NAME (p) <NAME then p: =RIGHT (p) /*visit right child*/
```

8.3.4 Hash Tables



- Two tables, a hash table and a storage table are used.
- Hashing mean variation of searching techniques.

Open hashing → no limit on the number of entries.

The Average time to insert ‘n’ Name and to make ‘e’ enquires is n (n+e)/m

If m is large, average time will be reduced.

If m is smaller, average time will be high.

Hashing Method

- Consist of a fixed away of m pointers to table entries.
- Table entries organized into ‘m’ separate linked list called buckets
- The hash table consists of K words, numbered a, 1…. K-1. → There are pointers to the storage table.
- Hash function h such that h (NAME) is an integer value between 0 and k-1, is used to find whether NAME is in the symbol table.

Characteristic

- Uniform Distribution

8.4 Operations

A symbol table, either linear or hash, should provide the following operations.

8.4.1 insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:
int a;
should be processed by the compiler as:
insert(a, int);

8.4.2 lookup()

lookup () operation is used to search a name in the symbol table to determine:

1. If the symbol exists in the table.
2. If it is declared before it is being used.
3. If the name is used in the scope.
4. If the symbol is initialized.
5. If the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:
lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

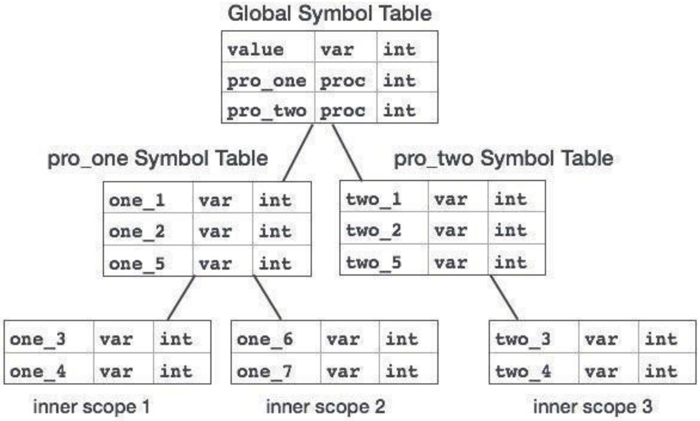
8.4.3 Scope Management

A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;
void pro_one()
{

int one_1; int one_2;

{
int one_3;           inner scope 1
int one_4;
}

int one_5;
{
int one_6;           inner scope 2
int one_7;
}
}

void pro_two()
{
int two_1;
int two_2;
{
int two_3;           inner scope 3
int two_4;
}

int two_5;
}
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



Global Symbol Table

value	var	int
pro_one	proc	int
pro_two	proc	int

pro_one Symbol Table

one_1	var	int
one_2	var	int
one_5	var	int

pro_two Symbol Table

two_1	var	int
two_2	var	int
two_5	var	int

one_3	var	int
one_4	var	int

inner scope 1

one_6	var	int
one_7	var	int

inner scope 2

two_3	var	int
two_4	var	int

inner scope 3

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

1. first a symbol will be searched in the current scope, i.e., current symbol table,
2. if a name is found, then search is completed, else it will be searched in the parent symbol table until,
3. either the name is found or the global symbol table has been searched for the name.

8.5 Implementation of A simple stack – Allocation Scheme

- Consider an implementation of the UNIX programming language C.

- Data in C can be global, meaning it is allocated as static storage and available to any procedure or local, meaning it can be accessed only by the procedure in which it is declared.

6.6 Implementation of Block-Structure Languages

- Here activation records must be reserved for blocks.
- Permit array of adjustable length.
- The data-referencing environment of a procedure or block includes all procedures and blocks surrounding it in the program.
- Display, parameter passing, creation of space for arrays.

6.7 Storage Allocation in FORTRAN

- Permits static storage allocation
- Compiler can create a number of data areas, in which the values of names can be stored.

Two Types of data areas:

- common
- equivalence

6.7.1 Data in Common Area

- For each block, a record giving the first and last names of the current routine that are declared to be in that common block.

COMMON / BLOCK1 NAME1 /NAME2

- Creates a common Block Block1.
- NAME1 AND NAME2 set a pointer to the symbol-table entry for BLOCK 1.

6.7.2 A Simple Equivalence Algorithm

Equivalence statements all of the form

EQUIVALENCE A, B+OFFSET

Where A and B are the names of locations.

- The effect of the above statement is to make A denote the location which is OFFSET memory units beyond the location for B.
- The sequence of EQUIVALENCE statements groups names into equivalence sets whose partitions relative to one another are all defined by the EQUIVALENCE statements.

E.g.:

```
EQUIVALENCE A, B+100
EQUIVALENCE C, D-4
EQUIVALENCE A, C+30
EQUIVALENCE E, F
```

- Last in First out use of Temporaries, if so many temporary variables.

6.8 Storage Allocation in Block-Structured Languages

- When an array is declared, the count is incremented by the size of a pointer rather than by the size of the array itself.
- Follows LIFO – when handling temporaries.

6.9 Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- Code : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

- Procedures : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- Variables : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

6.9.1 Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

6.9.2 Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

6.9.3 Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required. Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.

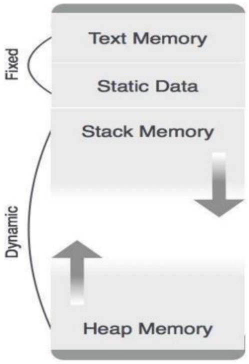


Figure 6.2 Memory Allocation

As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month, and year, all have r-values. Only variables have l-values, as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

Formal Parameters

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
int actual_parameter = 10;
call fun_two(int actual_parameter);
}
fun_two(int formal_parameter)
{
print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter, as they should also point to the same value.

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters, if manipulated, have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y; calling_procedure()
{
y = 10;
copy_restore(y); //l-value of y is passed printf y; //prints 99
}
copy_restore(int x)
{
x = 99; // y still has value 10 (unaffected) y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y, making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

Summary

- Symbol table also called Book Keeping.
- Each entry in the Symbol table is a pair of the form Name and Information.
- Information are entered into the Symbol table during Lexical and Syntactic phases.
- AVAILABLE is a pointer which indicates the beginning of the empty portion of the Array.
- Stack pointer is used to point a particular position an activation record.
- LIFO mechanism is using in handling temporaries.
- Symbol tables are mostly implemented as hash tables.
- The source code symbol itself is treated as a key for the hash function.
- The return value from hash table is the information about the symbol.

Question

1. Explain the contents of Symbol Table.
2. Describe the data structures for Symbol table.
3. Explain the Stack allocation scheme.
4. Write note on Storage Allocation in Block structure Language.
5. Briefly explain

a. Common Data area.

b. Equivalence Data area.

INTERMEDIATE CODE GENERATION

8.6 Introduction

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.

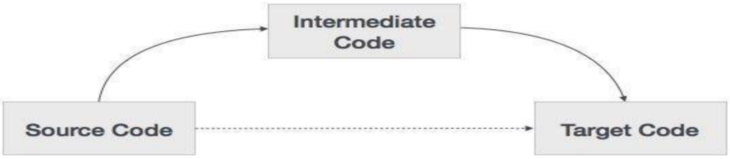


Figure 7.1 Role of Intermediate code

If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. The second part of compiler, synthesis, is changed according to the target machine.

It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

8.7 Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

8.7.1 *High Level IR* - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

8.7.2 *Low Level IR* - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

There are 3 types of intermediate representations discussed below,

(i) Post Fix
(ii) Syntax tree
(iii) 3-Address code, Quadruples and Triples.

8.8 Implementation of Intermediate code generator

- Intermediate codes are machine independent codes, but they are close to machine instruction.
- The given program in source language is converted to an equivalent program in an intermediate language, by the intermediate code generator.
- Intermediate languages can many different languages, and designer of compiler decides this intermediate language.
  - ✓ Postfix notation can be used as an intermediate language
  - ✓ Syntax tree can be used as an intermediate language.
  - ✓ Three address code (Quadruples)

- Quadruples are close to machine instruction. But they are not machine instructions.
  - ✓ Some programming languages have well defined intermediate languages.
- Java → java virtual machine.
- Prolog → warren obstruct machine.

In fact, there are byte code emulators to execute instructions in these intermediate languages.

8.8.1 *Postfix:* (Reverse Polish or Postfix Polish)

- Places the operator at the right end.

Examples:

S.No	Infix	Postfix
1	a+b	ab+
2	(a+b)*c	ab+c*
3	a*(b+c)	abc+*
4	(a+b)*(c+d)	ab+cd+*
5	If a then if c-d then a+c else a*c else a+b	acd-ac+ac*? ab+?

8.8.1.1 Evaluation of postfix expressions

- To evaluate the postfix expression, a stack is used.
- The general strategy is to scan the postfix code left to right.

E.g.:

Consider ab+c*, to evaluate 13+5*

The actions are,

1. stack1
2. stack 3
3. Add the two topmost elements, pop them off the stack and then stack the result 4.
4. stack 5
5. Multiply the two topmost elements pop them off the stack and then stack the result 20.

The syntax directed translation scheme for a simple grammar is given by,

Production	Semantic action
1. E -> $E^{(1)}0pE^{(2)}$	E.code := $E^{(1)}$.code || $E^{(2)}$.code|| 'op'
2. E -> ($E^{(1)}$)	E.code := $E^{(1)}$.code
3. E -> id	E.code := id

Note: Parenthesized expression is the same as the unparenthesized expression.

The program fragment corresponding to the above semantic actions are

Production	Program Fragment
E -> $E^{(1)}$ OP $E^{(2)}$	{ Print op }
E -> $E^{(1)}$	{}
E -> id	{Print id}

The sequence of moves for a+b*c

1. shift a
2. reduce by E -> id and print a
3. shift +
4. shift b

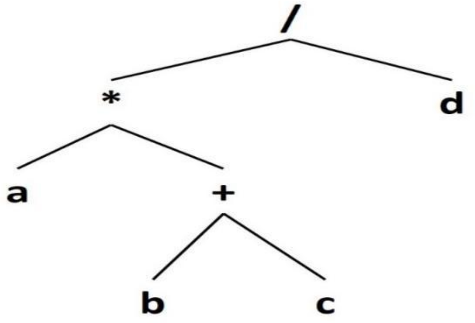
5. reduce by E -> id and print b
6. shift *
7. shift c
8. reduce by E-> id and print c
9. reduce by E -> E op E and print +
10. reduce by E -> E op E and print *

8.8.2 Parse tree and Syntax Tree:

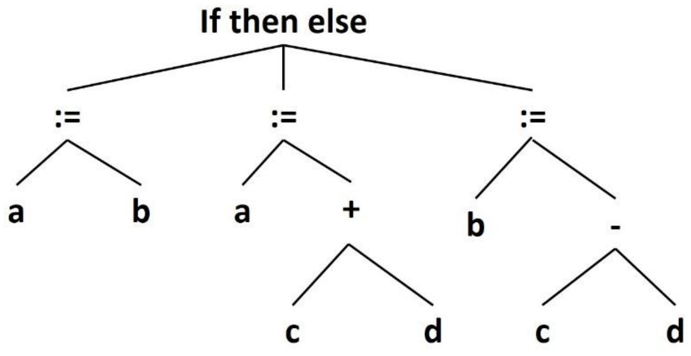
- If a tree in which each leaf represents an operand and each interior node an operator.

E.g.:

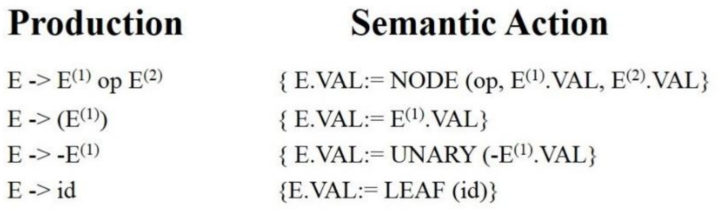
(i) a* (b+c)/d [abc+*d/]



(ii) if a=b then a:=c+d else b:=c-d



Syntax directed translation scheme for syntax trees:



Production	Semantic Action
E -> $E^{(1)}$ op $E^{(2)}$	{ E.VAL:= NODE (op, $E^{(1)}$.VAL, $E^{(2)}$.VAL}
E -> ($E^{(1)}$)	{ E.VAL:= $E^{(1)}$.VAL}
E -> -$E^{(1)}$	{ E.VAL:= UNARY (-$E^{(1)}$.VAL}
E -> id	{E.VAL:= LEAF (id)}

8.8.3 Three address code:

The general form is, A: =B op c

Where A,B, C are either programmer defined names, constructor or compiler generated temporary names. Op stands for any operator.

- usually the “Three address code” contains address two for the operand and one for the result

Additional 3-address statements:

1. Assignment statements of the form A:=B op C (binary arithmetic logical operator)
2. Assignment instruction, A:=op B (Unary operator) (A:=B→ B is assigned to A)
3. Unconditional: goto L.
4. Conditional : if A reop B goto L (relational op)
5. param A and call p,n

Eg: Procedure call:
Param A1
-
-
Param An
Call p,n

6. Indexed assignment : A:=B[I] (Location (array))
7. Pointer and address assignments,

A: = addr B(Address of B), A= *B(Value at B) and *A = B

The 3-address statement is an obstruct form of intermediate code. These statements can be implemented by either of the following way,

- Quadruples
- Triples
- Indirect Triples

8.8.3.1 Quadruples

- record structure has 4 fields,

Op, arg1, arg2, result

Op -> contains an internal code for the operator

Eg: -> A: = B op C puts B in ARG1, C IN ARG2 And A in RESULT.

- Conditional and unconditional jump put the target label in RESULT.

E.g.:

A: = -B * (C+D)

The 3-address code will be

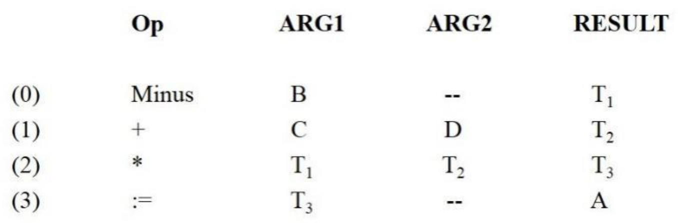
T_1:= -B

T_2:= C+D

T_3:= T_1 * T_2

A: = T_3

The Quadruples representation be



	Op	ARG1	ARG2	RESULT
(0)	Minus	B	--	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3	--	A

8.8.3.2 Triples

- Used to avoid temporary names into the symbol table.
- Here only 3 Fields are used

- Parenthesized numbers are used to represent pointers into the triple structure

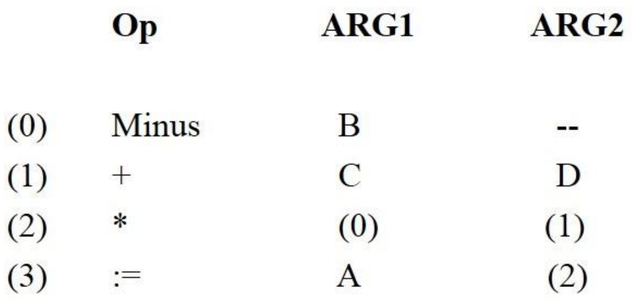
E.g.:

A: = -B * (C+D)

3-address code: $T_{1:}$= -B

T_2:=C+D

T_3:=T_1 * T_2



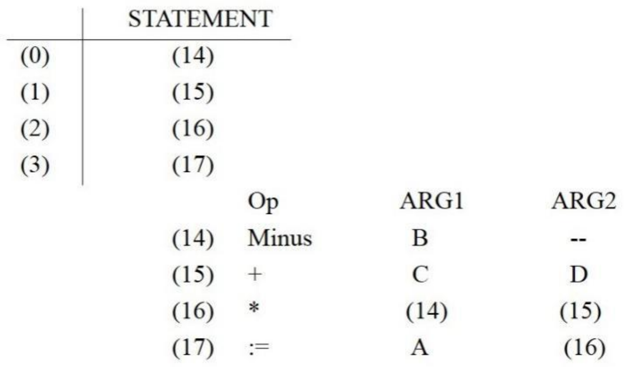
	Op	ARG1	ARG2
(0)	Minus	B	--
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

A: = T_3

8.8.3.3 Indirect Triples

- Listing pointers to triples, rather than listing the triples themselves.

E.g.:



	STATEMENT
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)

	Op	ARG1	ARG2
(14)	Minus	B	--
(15)	+	C	D
(16)	*	(14)	(15)
(17)	:=	A	(16)

A: = -B * (C+D)

8.9 Translation of Assignment Statements

- Translation of basic programming – language constructs into code of this form.

8.9.1 Assignment statements with Integer Types

→ Statements involving only integer values

Example:

A -> id: = E Consists of code to evaluate E into sometimes operators

E -> E+E | E*E | -E | (E)| id

A means assignment statement.

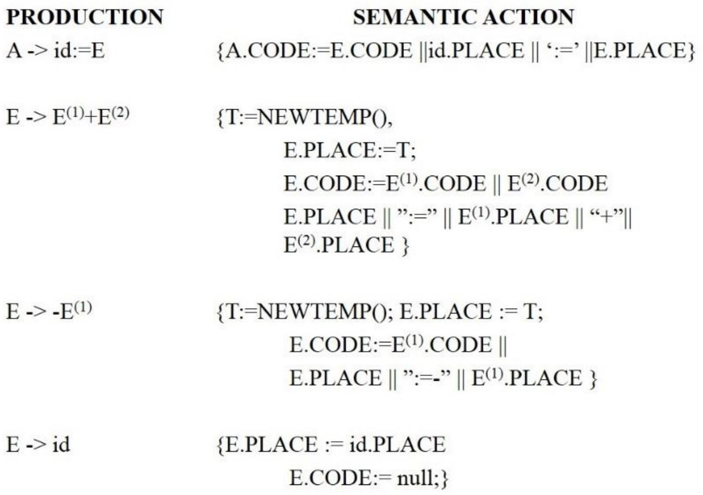
8.9.2 The abstract Translation Scheme

Abstractly, the translation of E to be structure with two fields:

1. E.PALCE, hold the value of the expression.
2. E.CODE, sequence of 3-address statements evaluating the expressions.

- A .CODE, which is a 3-address code to execute the assignment.
- Id . PLACE, to denote the name corresponding to this instance of taken id.
- To create new temporary name, NEWTEMP () is used to return an appropriate name.

Example:



PRODUCTION	SEMANTIC ACTION
A -> id:=E	{A.CODE:=E.CODE ||id.PLACE || ‘:=’ ||E.PLACE}

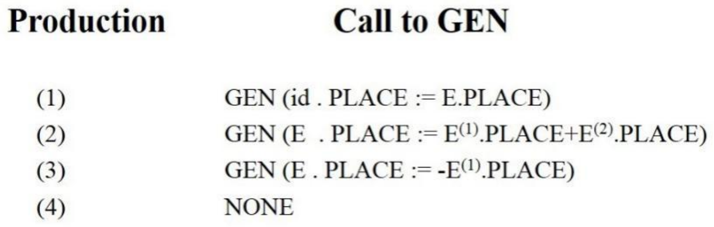
E -> E$^{(1)}$+E$^{(2)}$	{T:=NEWTEMP(),
	E.PLACE:=T;
	E.CODE:=E$^{(1)}$.CODE || E$^{(2)}$.CODE
	E.PLACE || ”:=” || E$^{(1)}$.PLACE || “+”||
	E$^{(2)}$.PLACE }

E -> -E$^{(1)}$	{T:=NEWTEMP(); E.PLACE := T;
	E.CODE:=E$^{(1)}$.CODE ||
	E.PLACE || ”:=-” || E$^{(1)}$.PLACE }

E -> id	{E.PLACE := id.PLACE E.CODE:= null;}

8.9.3 More Specified Form Of The Translation Scheme

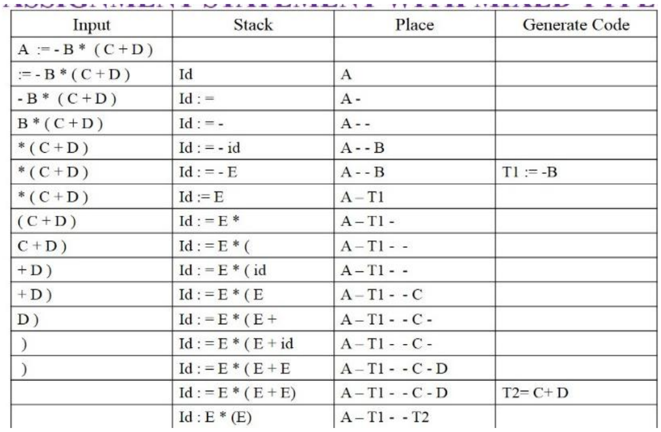
- We shall use a procedure GEN (A:=B+C) to emit the three address statement A:=B+C with actual values substituted for A,B and C. We can modify the above scheme in the following way



Production	Call to GEN
(1)	GEN (id . PLACE := E.PLACE)
(2)	GEN (E . PLACE := E$^{(1)}$.PLACE+E$^{(2)}$.PLACE)
(3)	GEN (E . PLACE := -E$^{(1)}$.PLACE)
(4)	NONE

8.9.4 Assignment Statement With Mixed Type

- Bottom up parse.



Input	Stack	Place	Generate Code
A := - B * (C + D)			
:= - B * (C + D)	Id	A	
- B * (C + D)	Id : =	A -	
B * (C + D)	Id : = -	A - -	
* (C + D)	Id : = - id	A - - B	
* (C + D)	Id : = - E	A - - B	T1 := -B
* (C + D)	Id := E	A – T1	
(C + D)	Id : = E *	A – T1 -	
C + D)	Id : = E * (	A – T1 - -	
+ D)	Id : = E * (id	A – T1 - -	
+ D)	Id : = E * (E	A – T1 - - C	
D)	Id : = E * (E +	A – T1 - - C -	
)	Id : = E * (E + id	A – T1 - - C -	
)	Id : = E * (E + E	A – T1 - - C - D	
	Id : = E * (E + E)	A – T1 - - C - D	T2= C+ D
	Id : E * (E)	A – T1 - - T2	

8.10 Boolean Expression

8.10.1 3-Address statements for Boolean Expressions:

→ The branching statements of the form:

goto L
if A goto L
if A relop B goto L

Here A and B is simple variables or constants, L is a quadruple Label and relop is any of <, ≤, = ≠, or ≥….

- Consider the implementation of Boolean expression using 1 to denote TRUE and 0 to denote FALSE.
- Expression will be evaluated from left to right.

E.g.:

A relational expression if A<B then 1 else 0, its 3-address code will be,

(1) if A<B goto (4)
(2) T : = 0
(3) Goto (5)
(4) T:=1

8.10.2 3-address code for array reference:

- We assume static allocation of arrays, where subscripts range from 1 to some limit known at compile time.
- Array elements are taken to require one word each.
- Let A is A[1], A[2]…., if addr(A) denotes the 1st word of block, A[1], then A[i] is in location addr (A)+i-1.
- If there were 4 bytes per word then the 3-address statement for A[i] is

$T_{1:}$=4*i
$T_{2:}$=addr (a)-4
T_3:= T_2 [T_1]

Summary

- The complexity of code generated by the intermediate code generator is in between the source code and the machine code.
- Intermediate is in form of 3-Address code.
- Triples is to avoid temporary name into the Symbol table.
- Intermediate codes are machine independent codes, but they are close to machine instruction.

- The source language is converted to an equivalent program in an intermediate language, by the intermediate code generator.

Questions:

1. Explain intermediate code representation.
2. Draw the Syntax tree for the following,
   I.a+b*(c*a)-(c/a)
   II.if a<b then c=b else c=a
   III.a-b*a+c
3. Write the postfix notation for the following,
   I.a+b*(c*a)-(c/a)
   II.if a<b then c=b else c=a
   III.a-b*a+c
4. Describe 3-Address code.
5. Explain Abstract Translation Scheme.

CODE OPTIMIZATION

8.11 Introduction

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

1. The term "code optimization" refers to techniques, a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program.
2. The quality of the object program is generally measured by its size (for small computation) or its running time (for large computation).
3. It is theoretically impassible for a compiler to produce the best possible object program for every source program under any reasonable cast function.
4. The accurate term for "code optimization" is "code improvement".
5. There are many aspects to code optimization.
   (i) Cast
   (ii) Quick & straight forward translation (time).

8.12 The Principal Sources Of Optimization

- The code optimization techniques consist of detecting patterns in the program and replacing these patterns by equivalent but more efficient construct.
- Patterns may be local or global and replacement strategy may be machine dependent or machine dependent.

8.12.1 Inner Loops

- "90-10" rule states that 90% of the time is spent in 10% of the code. Thus the most heavily traveled parts of a program, the inner loops, are an obvious target for optimization.

8.12.2 Language Implementation Details Inaccessible To the User:

The optimization can be done by

1) Programmer- Write source program (user can write)

2) Compiler -e.g.: array references are made by indexing, rather than by pointer or address calculation prevents the programmer from dealing with offset calculations in arrays.

8.13 Further Optimizations:

- The important sources of optimization are the identification of common sub expression and replacement of run time computation by compile time computation.
- The term constant folding is used for the latter optimization.

Example:

A [i+1]:=B [i+1] is easier.
J: =i+1
A[j]:=B[j]

- There are three types of code optimization

I. Local optimization-performed within a straight line and no jump.
II. Loop optimization
III. Data flow analysis-the transmission of useful information from one part of the program to another.

Note:

Optimization is mainly depending on the algorithm

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

1. The output code must not, in any way, change the meaning of the program.
2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
3. Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

1. At the beginning, users can change/rearrange the code or use better algorithms to write the code.
2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
3. While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

8.13.1 Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
item =10;
value = value + item;}while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item =10;do
{
value = value + item;} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

8.13.2 Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

8.13.3 Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

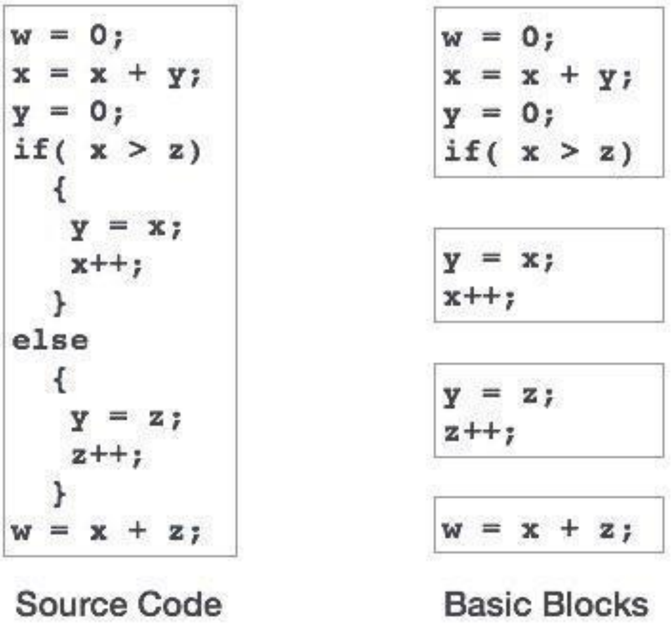
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

8.13.3.1 Basic Block Identification

We may use the following algorithm to find the basic blocks in a program:

1. Search header statements of all the basic blocks from where a basic block starts:
   i. First statement of a program.
   ii. Statements that are target of any branch (conditional/unconditional).
   iii. Statements that follow any branch statement.
2. Header statements and the statements following them form a basic block.
3. A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

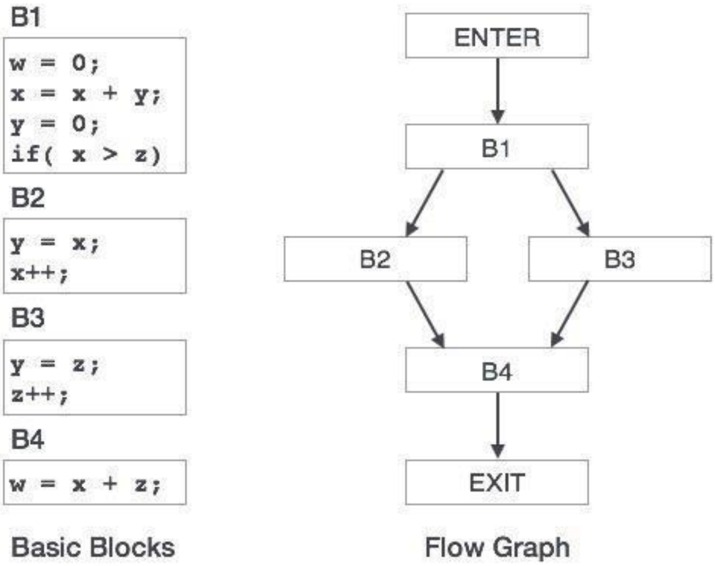
```
w = x + z;
```

Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by locating any unwanted loops in the program.



```
B1
w = 0;
x = x + y;
y = 0;
if( x > z)
B2
y = x;
x++;
B3
y = z;
z++;
B4
w = x + z;
```

Basic Blocks

Flow Graph

8.14 Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- Invariant code : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- Induction analysis : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- Strength reduction : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

8.15 Dead-code Elimination

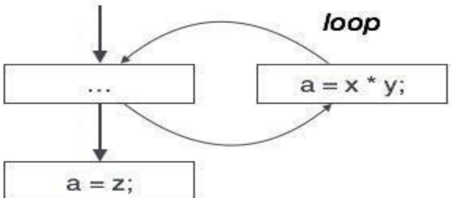
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore, it can simply be eliminated.

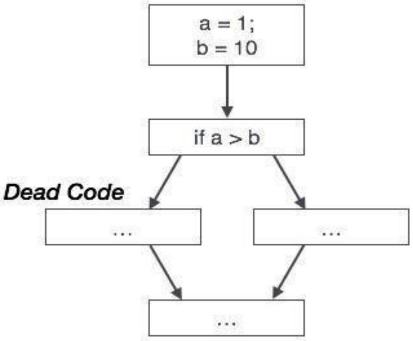
8.15.1 Partially Dead Code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



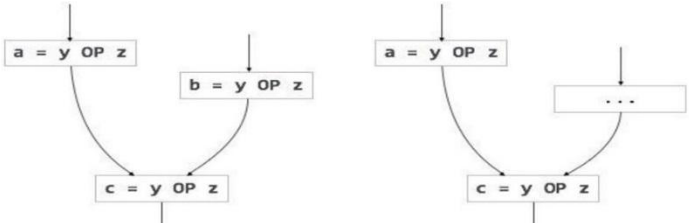
The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop. Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.

Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.



8.15.2 Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands; whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



[redundant expression] [partially redundant expression]

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
a = y OP z;
}
else
{
...
}
c = y OP z;
```

We assume that the values of operands (y and z) are not changed from assignment of variable a to variable c. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
...
tmp = y OP z;
a = tmp;
...
}
else
{
...
tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

8.16 Code Optimization

8.16.1 Code Motion:

- An important source of modifications of the code is called code motion, where we take a computation that yields the same result independent of the number of times through the loop and place if before the loop.

8.16.2 Induction Variable:

- If some sequence of statements from arithmetic progressions, we say such identifiers as induction variables.
- When two or more induction variable is a loop we an opportunity to get rid of all but one, and we call this process, induction variable elimination.

E.g.:

```
    While (I <= 20)          {for i->1->20}
    T1=4*I                    {t1->4, 8,. . . .}    AP
```

8.16.3 Reduction In Strength:

- The replacement of an expensive operation by a cheaper one is called reduction in strength.

    (T1:=4*I)= (T1=T1+4)

8.17 The Dag Representation Of Basic Blocks

- A useful data structure for automatically analyzing basic blocks is a directed acyclic graph (DAG).
- DAG is a directed graph with no cycle.
- Constructing a DAG from 3 address statement is a good way of determining common sub expressions.

8.17.1 DAG with following labels nodes:

- Leaves are labeled by unique identifiers.
- Interior nodes are labeled by an operator symbol.
- Nodes are also optionally given an extra set of identifiers for labels.(to store value)

8.17.2 Advantages

- We can detect common sub expression.
- We can determine value used in the block.
- Compute values used outside the block.
- To reconstruct a simplified list of quadruples.

Summary

- Code optimization are called code improvement.
- The aspect to Code is cost and Time
- Constant timing is used for later optimization.
- Optimization is mainly depends on the algorithm
- Data structure to analyze basic blocks DAG.
- DAG is the Direct Graph with no cycles.
- DAG is useful to determine common sub expressions.

Questions

1. Explain the principle sources of Optimization.
2. Describe Loop Optimization.
3. Write note on reduction and Strength.
4. Describe DAG representation.
5. State the properties and uses of DAG.

9. CODE GENERATION

9.1 Introduction

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

1. It should carry the exact meaning of the source code.
2. It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

9.2 Problems in Code Generation

- What instruction should we generate?
  There are variety of ways –which way is select.
- In what order should we perform computations?
  Picking best is difficult.
- What Register should we use?
  E.g.: - Certain machine requires register-pairs.

9.3 A simple code generator

- For each operator in a quadruple there is a corresponding machine code operator.
- Computed result can be left in registers as long as possible, storing them only
- If their registers is needed for another computation.
- Just before a procedure call, jump (or) labeled statement.
- Everything must be stored just before the end of a basic block.

Next_Use Information:

- To make more informed decisions concerning register allocation we compute the next uses of each name in a quadruple.

Use is defined as,

```
E.g.:  int n=5;
int i, j;
For (i=0; i<5; i++)
        For (j=i+1; j<5; j++)
            Computation.
The final result is stored.
```

9.4 Descriptor

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used

Register descriptor: Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

Address descriptor: Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

Register Allocation & Assignment:

There are various strategies for deciding what names in a program should reside in registers, i.e. register allocation & in which register each should reside i.e. register assignment.

Advantage: Simplifies the design of a compiler.

Disadvantage: When strictly handled, it uses registers inefficiently.

Global Register Allocation

- To assign registers to frequently used variables and keep their registers consistent across block boundaries (globally).

  Usage Counts: Reducing the usage of variables repeatedly in a loop and save cost and time.

Register Assignments For Outloops:

- The above said same procedure is also followed here.
- When the outer and the inner loop have to access the same register, then the register is declare outside to both the loops.

9.5 The Basics of Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

Note: If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise, cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses getReg function to determine the status of available registers and the location of name values. getReg works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location (register or memory) of y by consulting the Address Descriptor of y. If y is not presently in register L, then generate the following instruction to copy the value of y to L:

MOV y', L

where y' represents the copied value of y.

- Determine the present location of z using the same method used in step 2 for y and generate the following instruction:

OP z', L

where z' represents the copied value of z.

- Now L contains the value of y OP z, that is intended to be assigned to x. So, if L is a register, update its descriptor to indicate that it contains the value of x. Update the descriptor of x to indicate that it is stored at location L.
- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

9.5.1 The Code-Generation Algorithm:

1. Invoke a function GETREG ().
2. Consult the address descriptor.
3. Generate the instruction & update the address descriptor.
4. If there is no next use then the result is obtained. Then those registers no longer will contain the variables.

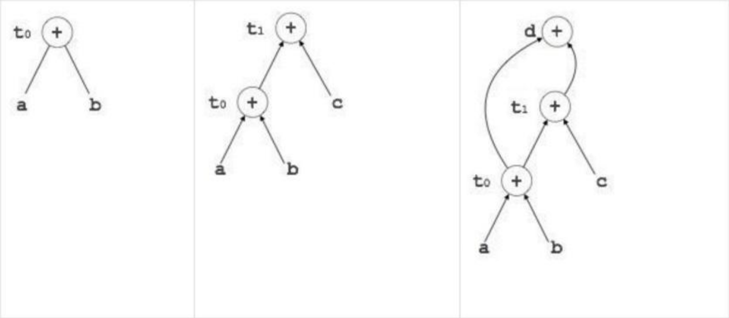
9.6 Code Generation from DAG's

9.6.1 Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

1. Leaf nodes represent identifiers, names, or constants.
2. Interior nodes represent operators.
3. Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:
t0 = a + b
t1 = t0 + c
d = t0 + t1



9.6.2 Steps involved in constructing machine using DAG

1. Rearranging the order:

```
                    T1:=A+B;
                    T2:=C+D;
            T3:=E-T2;
            T4:=T1-T3;
```

Called a heuristic ordering for DAG'S.

2. Optimal ordering of trees:

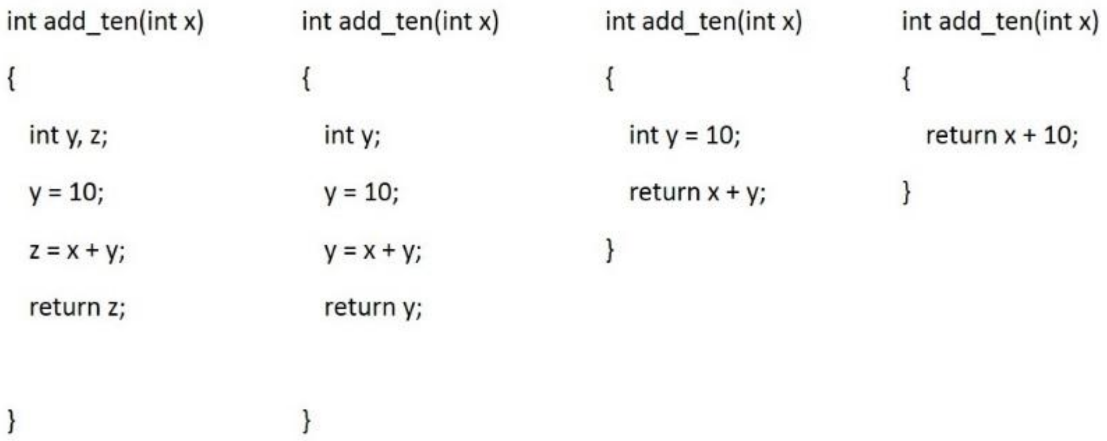
- The DAG representation of a quadruple is a tree and if should be in a designed register.
- Using labeling algorithm- we can find the node and the leaf.
- Multi register and algebraic properties are made.
- Sub expressions are made i.e. partition of trees into sub trees.

9.7 Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization

9.7.1 Redundant Instruction Elimination

At source code level, the following can be done by the user:



```
int add_ten(int x)      int add_ten(int x)      int add_ten(int x)      int add_ten(int x)
{                       {                       {                       {
  int y, z;               int y;                  int y = 10;             return x + 10;
  y = 10;                 y = 10;                 return x + y;         }
  z = x + y;              y = x + y;            }
  return z;               return y;


}                       }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

□ MOV x, R0

□ MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

9.7.2 Unreachable Code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

void add_ten(int x)

```
{
return x + 10;
printf("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed, as the program control returns back before it can execute, hence printf can be removed.

9.7.3 Flow of Control Optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :        GOTO L2
L2 :        INC R1
```

In this code, label L1 can be removed, as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
   MOV R1, R2
GOTO L2
... L2 : INC R1
```

9.7.4 Algebraic Expression Simplification

There are occasions where algebraic expressions can be made simple. For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

Strength Reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, x * 2 can be replaced by x << 1, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Summary

- Code generation can be considered as the final phase of compilation.
- The output of the code generator is machine code.
- The address descriptor is used to track the location in the memory.
- DAG representation made the code generation simple.
- If the value of a name is found at more than one place, the register's value will be preferred over the cache and main memory.
- Redundancy in loads and stores are reduced in peephole optimization.

Question

1. Explain register descriptor and address descriptor.
2. Write note on GETREG ().

3. Explain heuristic ordering with an example.
4. Explain Peephole optimization.

MULTIPLE CHOICE QUESTIONS

1. A ----------- is a program that takes as input a program written in one language (source language) and produces as output a program in another language (object language).
   a)translator b)assembler c)compiler d)interpreter **Ans:a**
2. If the source language is high-level language and the object language is a low-level language(assembly or machine), then such a translator is called as a -------------.
   a)translator b)assembler c)compiler d)interpreter **Ans:c**
3. An interpreter is a program that directly executes an---------- code.
   a)source b)object c)intermediate d)subject **Ans:c**
4. If the source language is an assembly language and the target language is a machine language, then the translator is called an -----------------.
   a)translator b)assembler c)compiler d)interpreter **Ans:b**
5. ------------ is used for translators that take programs in one high-level language into equivalent programs in another high-level language.
   a)Preprocessor b)Compiler c)Assembler d)Translator **Ans:a**
6. A macro is a ----------- replacement capability.
   a)text b)image c)language d)none **Ans:a**
7. The two aspects of macros are ----------- and -------------.
   a)description, definition b)description, use
   c) definition, use d) definition, function **Ans:c**
8. A compiler takes as input a source program and produces as output an equivalent sequence of ----------------.
   a) user program b)object language
   c)machine instructions d)call **Ans:c**
9. The compilation process is partitioned into a series of sub processes called -------------
   a)phases b)sub program c)module d)subsets **Ans:a**
10. The first phase of the compiler is also called as ------------.
    a)scanner b)parser c)tokens d)macro **Ans:a**
11. The output of the lexical analyzer are a stream of ------------.
    a)instructions b)tokens c)values d)inputs **Ans:b**
12. Tokens are grouped together into syntactic structure called as an -------------.
    a)expression b)tokens c)instructions d)syntax **Ans:a**
13. Syntactic structure can be regarded as a tree whose leaves are the ------------.
    a)scanner b)parser c)tokens d)macro **Ans:c**
14. ------------ phase designed to improve the intermediate code.
    a)Code optimization b) Code Generation
    c) Intermediate code generator d) Syntax Analyzer **Ans:a**
15. Data structure used to record the information is called a --------- table.
    a)syntactic b)symbol c)value d)tokens **Ans:b**
16. In an implementation of a compiler, portions of one or more phases are combined into amodule called a --------.
    a)pass b) parser c)scanner d)set **Ans:a**

17. -------------- is a special kind of notation used to define the language.
a) Expression b) Proper Expression
c) Irregular Expression d) Regular Expression **Ans:d**

18. The ------------ phase receives optimized intermediate codes and generates the code for execution.
a)lexical analyzer b)syntax analyzer
c)code optimizer d)code generator **Ans:d**

19. A compiler may run on one machine and produce object code for another machine, such a compiler is called a ------------.
a) cross compiler b)medium compiler
c) back compiler d)mixed compiler **Ans:a**

20. The main function of lexical analyzer is to read a --------------.
a) source program b)object program
c)intermediate code d)sub **Ans:a**

21. One character is read at a time and translated into a sequence of primitive units called -----
a)instructions b)tokens c)values d)numbers **Ans:b**

22. Which is not a token?
a)operator b)instructions c)keywords d)identifier **Ans:b**

23. To recognize the tokens in the input stream-------------------- and ----------- are convenient ways of designing recognizers.
a) transition diagrams, finite automata
b) transaction diagram, finite automata
c) transition diagram, NFA
d) transaction diagram, NFA **Ans:a**

24. When the lexical analyzer and parser are in the same pass, the lexical analyzeracts as a ------.
a)subroutine b)stack c)analyzer d)parser **Ans:a**

25. It is easy to specify the structure of tokens than the ----------- structure of the program.
a)syntactic b)syntax c)both (a) and (b) d)main **Ans:a**

26. ----------- is used to define a language.
a) Lexical Analyzer b)Parser
c)Regular Expression d)Identifier **Ans:c**

27. A string is a finite sequence of -----------.
a)symbols b)tokens c)instructions d) passes **Ans:a**

28. The concatenation of any string with an empty string is the -----------.
a)string itself b)null c)symbol d)alphabet **Ans:a**

29. ---------------is used to describe tokens and identifiers.
a) Lexical Analyzer b)Parser
c)Regular Expression d)Random **Ans:c**

30. The symbol table keeps account of the attributes of the ------------------.
a) identifiers b)values c)numbers d)text **Ans:a**

31. A ----------------- or finite automata for a language is a program that takes as input a string x and answers 'yes' if x is a sentence of the language L 'no' otherwise.
a)recognizer b)parser c)lexical analyzer d)identifier **Ans:a**

32. DFA stands for ------------------
a) Deterministic Finite set Automata b) Deterministic Finite Automata
c) Non Deterministic Finite Automata d) Non Deterministic Finite set Automata
Ans:b

33. NFA stands for ------------------
a) Deterministic Finite set Automata
b) Deterministic Finite Automata
c) Non Deterministic Finite Automata
d) Non Deterministic Finite set Automata **Ans:c**

34. A NFA should have ------------------ start state.
a)1 b)0 c)finite d)infinite **Ans:a**

35. The generalized transition diagram for a regular expression is called ------------- .
a) finite automaton b)infinite automaton
c)regular automaton c)irregular automaton **Ans:a**

36. -------------- is a tool that automatically generating lexical analyzer.
a)LEX b)HEX c)SLR d)CLR **Ans:a**

37. LEX can build from its input, a lexical analyzer that behaves roughly like a ----------.
a) Finite Automaton b)Deterministic Finite Automata
c)Non-Deterministic Finite Automata d)Finite Set **Ans:a**

38. -------------- are used by lexical analyzers to recognize tokens.
a) Line Graphs b)Bar Charts
c)Transition Diagrams d)Circle Charts **Ans:c**

39. In CFG ,the basic symbols of the language are called -------------.
a)terminals b)non-terminals c)symbols d)digits **Ans:a**

40. Tokens are ------------.
a)terminals b)non-terminals c)symbols d)digits **Ans:a**

41. Special symbols and syntactic variables are --------------.
a)terminals b)non-terminals c)symbols d)lines **Ans:b**

42. The symbol ==> means ----------------.
a)derives in one step b)derives in zero or more steps
c) derives in one or more steps d)does not derive **Ans:a**

43. The symbol =*=> means ----------------.
a)derives in one step b)derives in zero or more steps
c) derives in one or more steps d)does not derive **Ans:b**

44. The symbol =$^{+}$=> means ----------------.
a)derives in one step b)derives in zero or more steps
c) derives in one or more steps d) does not derive **Ans:c**

45. A graphical representation for derivations that filter out the choice regarding replacement order is called the -----------------.
a) parse tree b) graph tree c)syntax tree d) symbol tree **Ans:a**

46. A parse tree consists of a finite set of labeled ---------- connected by -----------.
a) nodes, edges b)edges, nodes
c)terminals, lines d)lines, terminals **Ans:a**

47. A parser for Grammar G is a program that takes as input string W and produces as output is --------------- for W.
a) parse tree b) slr c) error message d) string **Ans:a**

48. If W is a sentence of G, or an ------------ indicating that W is not a sentence of G.
a) parse tree b) slr c) error message d) string **Ans:c**

49. Syntax Analyzer is also called as a -------------.
    a) parser b) lexer c) converter d) inverter **Ans:a**
50. Bottom-up parserbuild the parse trees from the bottom ---------- to the top ---------.
    a) leaves, root b) root, leaves
    c) none d) combination of leaves and root **Ans:b**
51. In both parsing type ,the cases the input to the parser is being scanned from -----------, one symbol at a time.
    a) left to right b) right to left c) middle of a string d) end **Ans:a**
52. In a top-down parser, the starting --------------- is expanded to derive the given input string.
    a) terminal b) 3letter c) digit d) non-terminal **Ans:d**
53. The bottom-up parsing method is called --------------- parsing.
    a)shift reduce b) recursive decent c) bottom-up d) top-down **Ans:a**
54. The top-down parsing is called ------------ parsing.
    a)shift reduce b) recursive decent c) bottom-up d) top-down **Ans:b**
55. An operator-precedence parser is one kind of ------------- parser.
    a) shift reduce b) descent c)bottom-up d)top-down **Ans:a**
56. Predictive parser is one kind of --------------- parser.
    a)shift reduce b)recursive descent c)bottom-up d)top-down **Ans:b**
57. The output of a parser is the representative of a ---------------.
    a) parser tree b) slr c) error message d) tree **Ans:a**
58. ------------ is a program that produces valid parse trees.
    a)Reader b)Parser c)Writter d)Producer **Ans:b**
59. A rightmost derivation in reverse is called as ----------- .
    a)reduction b)sequence
    c)reduction sequence d)canonical reduction sequence **Ans:a**
60. Rightmost derivation is sometimes called ------------- derivations.
    a)canonical b)RMD c)LMD d)low **Ans:b**
61. ---------- makes grammar suitable for parsing.
    a)Factoring b)Right Factoring c) Left Factoring d) Reverse Factoring **Ans:c**
62. Left Factoring is a transformation for factoring out the ---- prefixes.
    a)odd b)common c)positive d)negative **Ans:b**
63. Reverse of a right most derivation is called ------------.
    a)reduction b)handle c)production d)base **Ans:b**
64. The canonical reduction sequence is obtained by ------.
    a)reduction b)handle c)production d)handle pruning **Ans:d**
65. Which is not a shift reduce parser action
    a)Shift b)Reduce c)Accept d)go **Ans:d**
66. If a grammar has no two adjacent non-terminals ,then it is called as an --------------- grammar.
    a)precedence b)operator c)regular d)irregular **Ans:b**
67. The parsing table is generally a --------------- dimensional array.
    a) one b) two c) three d)four **Ans:b**
68. Precedence table can be encoded by ------------ functions.
    a) reduce b) shift c) precedence d) various **Ans:c**
69. Stack is pushed with --------------- symbol.
    a)$ b)% c)* d)& **Ans:a**

70. LR Parser is a -------------- parser.
    a)Bottom-Up b)Top-Down c)reverse d)forward **Ans:a**
71. LR parser construct a ---------------- type of derivation.
    a)RMD b)MMD c)LMD d)CLR **Ans:a**
72. LR parser has ------------- components.
    a)2 b)3 c)5 d)1
73. What are the components of LR Parser?
    a) Parsing algorithm b) Parsing table construction
    c) both a and b d)Parsing note **Ans:c**
74. --------------- function is a collection, called canonical collection of LR (0) items.
    a) GOTO b) FIRST c) FOLLOW d) COMPUTE **Ans:a**
75. The collection of sets of LR (0) item is called ----------------.
    a)SLR b)CLR c)LALR d)DMR **Ans:b**
76. The SLR table has 2 parts they are ------- and -------------.
    a) action, goto entries b)action, error
    c)error, shift d)action, shift **Ans:a**
77. The input string is in I/p buffer followed by the right end marker ----------------.
    a)$ b)% c)* d)& **Ans:a**
78. If Left Recursion is available------------ occurs.
    a) stack b) cycle c) queue d) symbols **Ans:b**
79. ----------------- keeps the grammar symbols.
    a)Top b) Stack c)Queue d)Bottom **Ans:b**
80. The ---------- keeps the input string.
    a)input buffer b)output buffer c) stack d)queue **Ans:a**
81. ------------ directed translation allows subroutines or semantic actions to be attached to the productions of a context free grammar.
    a)syntax b)semantic c)both d)error **Ans:a**
82. Syntax directed translation subroutines generate --------------- code.
    a)intermediate b)source c)object d)error **Ans:a**
83. A syntax directed translation scheme is merely a -------------- grammar.
    a)regular b)context-sensitive c)context-free d)single **Ans:c**
84. The ------------ action is enclosed in braces.
    a)syntax b)semantic c)both d)error **Ans:b**
85. Implementation of syntax-directed translators describes an ------------ mapping.
    a)input b)output c)input-output d)parse table **Ans:c**
86. A compiler – compiler would tie the parser and the semantic action program fragments together, producing ----------- module.
    a)one b)two c)three d)more than one **Ans:a**
87. ------------ polish places the operator at the right end.
    a) Postfix b) Prefix c) Both d) Polish **Ans:a**
88. To evaluate the -------------- expression, a stack is used.
    a) postfix b) prefix c) both d) polish **Ans:a**
89. The general strategy is to scan the postfix code -------------.
    a)left-right b)right-left c)middle d)end **Ans:a**
90. If the attributes of the parent depend on the attributes of the children ,then they are called as ------------- attributes.
    a)made b)discovered c)new d) inherited **Ans:d**

91. ------------- is a tree in which each leaf represents an operand and each interior node an operator.
a) Parser Tree b)Semantic Tree c)Syntax Tree d)Structured Tree **Ans:c**

92. The properties of an entity are called as --------------.
a)values b)attributes c)numbers d)digits **Ans:b**

93. Usually the "Three address code" contains address two for the ---------- and one for the result.
a) operand b)operator c)result d) statement **Ans:a**

94. The --------------- statement is an abstract form of intermediate code.
a)2-address b)3-address c)Intermediatecode d)address **Ans:b**

95. Which is not the way of implement the 3-address statement.
a) Quadruples b) Triples c) Indirect Triples d) Parse Tree **Ans:d**

96. -------------- record structure has 4 fields.
a) Quadruples b) Triples c) Indirect Triples d) Parse Tree **Ans:a**

97. Parenthesized numbers are used to represent ----------- into the triple structure.
a)pointer b)stack c)queue d)value **Ans:a**

98. ---------------- Triples are listing pointers to triples, rather than listing the triples themselves.
a)Direct b)Indirect c)Multiple d)New **Ans:b**

99. ---------------- refers to the location to store the value for a symbol.
a) value b)place c)code d)number **Ans:b**

100. ----------- refers to the expression or expressions in the form of three address codes.
a) value b)place c)code d)number **Ans:c**

101. ----------- is associating the attributes with the grammar symbols.
a)rotation b)translation c)transformation d)evolving **Ans:b**

102. In 3-address code for array reference we assume static allocation of arrays, where subscripts range from 1 to some limit known at ------------------ time.
a) compile b) run c) execution d) process **Ans:a**

103. In Triples uses only 3 ----------------.
a) fields b) operator c) operand d) instruction **Ans:a**

104. _____________ is used in the several stages of the compiler.
a)Table b) Symbol Table c) Records d) Program. **Ans:b**

105. Information about the name is entered into the symbol table during ____________ and ___________.
a) lexical and syntactic analysis b) lexical and code generation
c) lexical and error handler d) lexical and code optimization. **Ans:a**

106. Each entry in the symbol table is a pair of the form ______ and _______.
a) Name and information. b) Name and function.
c) Name and Data. d) Name and procedures. **Ans:a**

107. A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a ______________.
a)Symbol Table b) Lexical analysis
c) Syntactic analysis d) Records. **Ans:a**

108. Undeclared name and type incompatibilities in _____________.
a)Syntactic errors b) Semantic errors
c) Lexical Phase errors d) Reporting errors. **Ans:b**

109. Minimum distance matching in __________.
a)Syntactic errors b) Semantic errors
c) Lexical Phase errors d) Reporting errors **Ans:a**

110. Minimum distance correction is___________ errors.
a)Syntactic Phase errors b) Semantic errors
c) Lexical Phase errors d) Reporting errors. **Ans:a**

111. Parser discards input symbol until a __________________ token is encountered.
a)synchronizing b) Synchronizing
c) Group d) none. **Ans:b**

112. The message should not be redundant in ______________.
a) Syntactic Phase errors b) Semantic errors
c) Lexical Phase errors d) Reporting errors. **Ans:d**

113. When an error is detected the reaction of compiler is different,
a)A system crash
b)To emit invalid output
c)To merely quit on the first detected error.
d)All of the above. **Ans:d**

114. Two types of data areas _____________.
a)Common and stack b)Common and equivalence.
c)Register and stack d)Code and equivalence. **Ans:b**

115. Hashing meaning_____
a)Variation of searching techniques b)Variation of inserting techniques
c)Variation of updating techniques. d)Variation of Deleting Techniques. **Ans:a**

116. An ___________ describing the partition in storage to be allocated for the name.
a)Pointer b) AVAILABLE c) Offset d) Attributes. **Ans:b**

117. The simplest way to implement a symbol table is as a _____________ of records, one record per name.
a)Linear array b) Multidimensional array
c) Rectangular array d) Jagged Array. **Ans:a**

118. What is the length of identifier for DIMPLE?
a)5 b) 6 c) 4 d) 3 **Ans:b**

119. The accurate term for "Code Optimization" is __________________.
a)Intermediate Code b) Code Improvement
c) Latter Optimization d) Local Optimization. **Ans:b**

120. The quality of the object program is generally measured by its _________.
a)Cost b) Time
c) Size or Its running time d) Code Optimization. **Ans:C**

121. The code optimization techniques consist of detecting _________ in the program and _________ these patterns.
a)Errors and replacing b) Patterns and replacing
c) Errors and editing d) Patterns and editing. **Ans:b**

122. __________ may be local or global.
a)Code Optimization b) Variable
c) Sub expression d) Patterns. **Ans:a**

123. "90-10" rule states that _________ of the time is spent in __________ of the code.
a)90%, 20% b) 80%, 10% c) 90%, 10% d) 90%, 90%. **Ans:a**

124. The important sources of optimization are the identification of common ____________.
a)Regular expression b) Sub expression
c) expression d) time. **Ans:b**

125. The term constant folding is used for the __________________.
a)Local optimization b) Code optimization
c) Latter optimization d) Loop optimization. **Ans:c**

126. ___________ performed within a straight line and no jump.
a)Local optimization b) Code optimization
c) Latter optimization d) Loop optimization. **Ans:a**

127. From anyone in the loop to any other, there is a path of length one or more is _________.
a)Weakly Connected b) Unique Entity
c) Multi Connected d) Strongly Connected. **Ans:d**

128. If some sequences of statements from arithmetic progressions, we say such identifiers as _____________.
a)Reduction b) Induction Variables
c) Code motion d) Inner Loops. **Ans:b**

129. The replacement of an expensive operation by a cheaper one is called ___________ in strength
a) Reduction b) Induction Variables
c) Code motion d) Inner Loops. **Ans:a**

130. Full form of DAG
a)Dynamic acyclic graph b)Data acyclic graph
c)Directed acyclic graph d)Detecting acyclic graph. **Ans:c**

131. A useful data structure for automatically analyzing basic block is a ____________.
a)Dynamic acyclic grap b)Data acyclic graph
c)Directed acyclic graph d)Detecting acyclic graph. **Ans:c**

132. Constructing a DAG from ________________ is a good way of determining common sub expression.
a)2 address statement b) 4 address statement
c) 3 address statement d) 5 address statement. **Ans:c**

133. _____________ are labeled by operator symbol.
a) Nodes b) Leaves c) Interior Nodes d) Root **Ans:c**

134. Computed results can be left in _____________ as long as possible.
a)Registers b) Triples c) Indirect Triples d) Quadruples. **Ans:a**

135. Initially the register descriptor shows that all registers as ___________.
a)Full b) empty c) Half-filled d) None **Ans:b**

136. To keep track of the location _________is used.
a)Flag register b) Address descriptor
c) Allocation descriptor d) register. **Ans:b**

137. ____________ invoke a function GETREG ().
a)Code optimization b) Code motion
c) the code generation algorithm d) intermediate code. **Ans:c**

138. The DAG representation of a Quadruples is a _________.
a)Nodes b) Leaves c) Tree d) Pattern. **Ans:c**

139. Multiple jumps are reduced accordingly to _____________.
a)Local optimization. b) Code optimization
c) Peephole optimization d) Latter optimization **Ans:c**

140. Loads and stores are reduced in ______.
a) optimization b)peephole optimization
c)latter optimization d)none **Ans:b**

Exercise

1. If W is a string of terminals and A, b are two non-terminals, then which of the following are right-linear grammars?
(a) A→ Bw
(b) A→ Bw|w
(c) A→wb|w
(d) None of the above

2. If a is a terminal and S, A, B are three non-terminals, then which of the following are regular grammars?
(a) S → E
A → aS|b
(b) A → aB|a
B → bA|b
(c) A → Ba|Bab
(d) A → abB|aB

3. Representing the syntax by a grammar is advantageous because
(a) It is concise
(b) It is accurate
(c) Automation becomes easy
(d) Intermediate code can generated easily and efficiently

4. CFG can be recognized by a
(a) Push-down automata
(b) 2-way linear bounded automata
(c) Finite state automata
(d) None of the above

5. CSG can be recognized by
(a) Push-down automata
(b) 2-way linear bounded automata
(c) Finite state automata
(d) None of the above.

6. Choose the correct statements.
(a) Sentence of a grammar is a sentential from without any terminals.
(b) Sentence of a grammar should be derivable from the start state.
(c) Sentence of a grammar should be frontier of a derivation tree, in which the node has the start state as the label
(d) All of the above.

7. A grammar can have
(a) A non-terminal A that can't derive any string of terminals.
(b) A non-terminal A that can be present in any sentential from
(c) E as the only symbol on the left hand side of a production
(d) None of the above.

8. A top-down parser generates
(a) Left-most derivation
(b) Right-most derivation
(c) Right-most derivation in reverse
(d) Left-most derivation in reverse

9. A bottom-up parser generates
(a) Left-most derivation
(b) Right-most derivation
(c) Right-most derivation in reverse
(d) Left-most derivation in reverse

10. A given grammar is said to be ambiguous if
(a) Two or more productions have the same non-terminal on the left hand side.
(b) A derivation tree has more than one associated sentence.
(c) There is a sentence with more than one derivation tree corresponding to it
(d) Parenthesis are not present in the grammar

11. The grammar E → E+E | E*E | a, is
(a) Ambiguous
(b) Unambiguous
(c) Ambiguous or not depends on the given sentence
(d) None of the above

12. Choose the correct statement
(a) Language corresponding to a given grammar, is the set of all strings that can be generated by the given grammar.
(b) A given language is ambiguous if no unambiguous grammar exist for it.
(c) Two different grammars may generate the same language.
(d) None of the above.

13. Consider the grammar
S → ABSc | Abc
BA → AB
Bb → bb
Ab → ab
Aa → aa

Which of the following sentences can be derived by this grammar?
(a) abc
(b) aab
(c) abcc
(d) abbc

14. The language generated by the above grammar is the set of all strings made uo of a, b, c, such that
(a) The number of a's, b's, and c's will be equal
(b) a's always precede b's
(c) b's always precede c's
(d) The number of a's b's and c's are same and the a's precede b's. Which precede c's.

15. In an incompletely specified automata
(a) No edge should be labelled E
(b) From any given state, there can't be token leading to two different states
(c) Some states have no transition on some tokens
(d) Start state may not be there

16. The main difference between a DFSA and an NDFSA is
(a) In DFSA, E transition may be present
(b) In NDFSA, E transition may be present
(c) In DFSA, from any given, there can't be alphabet leading to two different states.
(d) In NDFSA, from any given state, there can't be any alphabet leading to two different states.

17. Two finite state machines are said to be equivalent if they
(a) Have the same number of stages
(b) Have same number of edges
(c) Have the same number states and edges
(d) Recognize the same set of tokens

18. Choose the correct answer.
FORTRAN is a
(a) Regular language
(b) Context-free language
(c) Context-sensitive language
(d) Turing language

19. If two finite states machine M and N isomorphic, then M can be transformed to N by relabeling
(a) The states alone
(b) The edges alone
(c) Both the states and edges
(d) None of the above.

20. In a syntax directed translation scheme, if the value of an attribute of a node is a function of the values of the attributes of its children, then it is called a
(a) Synthesized attribute
(b) Inherited attribute
(c) Canonical attribute
(d) None of the above.

21. Synthesized attribute can easily be simulated by an
(a) LL grammar
(b) Ambiguous grammar
(c) LR grammar
(d) None of the above

22. For which of the following situations, inherited attribute is a natural choice?
(a) Evaluation of arithmetic expressions
(b) Keeping track of variable declaration
(c) Checking for the correct use of L-values and R-values
(d) All of the above.

23. The graph depicting the inter-dependencies of the attributes of different nodes in a parse tree is called a
(a) Flow graph
(b) Dependency graph
(c) Karnaugh's graph
(d) Steffi graph

24. Choose the correct statements.
(a) Topological sort can be used to obtain an evaluation order of a dependency graph.
(b) Evaluation order for a dependency graph dictates the order in which the semantic rules are done.
(c) Code generation depends on the order in which semantic actions are performed.
(d) Only(a) and (c) correct.

25. A syntax tree
(a) Is another name for a parser tree
(b) Is a condensed form of parse tree
(c) Should not have keywords as leaves
(d) None of the above.

26. Syntax directed translation scheme desirable because
(a) It is based on the syntax
(b) Its description is independent of any implementation
(c) It is easy to modify
(d) Only (a) and (c) are correct.

27. Which of the following is not an intermediate code form?
(a) Postfix notation
(b) Syntax trees
(c) Three address codes
(d) Quadruples.

28. Three address codes can be implemented by
(a) Indirect triples
(b) Direct triples
(c) Quadruples
(d) None of the above.

29. Three address code involves
(a) Exactly 3 addresses
(b) At the most 3 addresses
(c) No unary operator
(d) None of the above.

30. Symbol table can be used for
(a) Checking type compatibility
(b) Suppressing duplicate error messages
(c) Storage allocation
(d) None of the above.

31. The best way to compare the different implementation of symbol table is to compare the time required to
(a) Add a new name
(b) Make an inquiry
(c) Add a new name and make an inquiry
(d) None of the above.

32. Which of the following symbol table implementation is based on the property of locality of reference?
(a) Linear list
(b) Search tree
(c) Hash table
(d) self-organization list

33. which of the following symbol table implementation is best suited if access time to be minimum?
(a) Linear list
(b) Search tree
(c) Hash table
(d) self-organization list

34. which of the following symbol table implementation, makes efficient use of memory?
(a) List
(b) Search tree
(c) Hash table
(d) Self-organizing list.

35. Access time of the symbol table will be logarithmic, it is implemented by a
(a) Linear list
(b) Search tree
(c) Hash table
(d) Self-organizing list.

36. An ideal compiler should
(a) Detect error
(b) Detect and report error
(c) Detect, report and correct error
(d) None of the above.

51. Which of the following is not a source error?
(a) Faulty design specification
(b) Faulty algorithm
(c) Compiler themselves
(d) None of the above

52. Any transcription error can be repaired by
(a) Insertion alone
(b) Deletion alone
(c) Insertion and deletion alone
(d) Replacement alone.

53. Hamming distance is a
(a) Theoretical way of measuring errors
(b) Technique for assigning codes to a set of items known to occur with a given probability
(c) Technique for optimizing the intermediate code
(d) None of the above

54. Error repair may
(a) Increase the number of errors
(b) Generate spurious error messages
(c) Mask subsequent error
(d) None of the above

55. A parser with the valid prefix property is advantageous because
(a) It detects error as soon as possible
(b) It detects errors as and when they occur
(c) It limits the amount of erroneous output passed to the next phase
(d) All of the above.

56. The advantages of panic mode of error recovery is that
(a) It is simple to implement
(b) It is very effective
(c) It never gets into an infinite loop
(d) None of the above.

57. To recover from an error, the operator precedence parser may
(a) Insert symbols onto the stack
(b) Insert symbols onto the input
(c) Delete symbols from the stack
(d) Delete symbols from the input.

58. Which of the following optimization techniques are typically applied on loops?
(a) Removal of invariant computation
(b) Elimination of induction variables
(c) Peephole optimization
(d) Invariant computation

59. The technique of replacing run time computation by compile time computations is called
(a) Constant folding
(b) Code hoisting
(c) Peephole optimization
(d) Invariant computation

60. The graph that shows the basic blocks and their successor relationship is called.
(a) Control graph
(b) Flow graph
(c) DAG
(d) Hamiltonian graph

61. Reduction in strength means
(a) Replacing run computation by compile
(b) Removing loop invariant computation
(c) Removing common sub-expression
(d) Replacing a costly operation by a relatively cheaper one

62. A basic block can be analysed by a
(a) DAG
(b) Graph which may involve cycles
(c) Flow-graph
(d) None of the above

63. ud-chaining is useful for
(a) Determining whether a particular definition is used anywhere or not
(b) Constant folding
(c) Checking whether a variable is used, without prior assignment
(d) None of the above

64. Which of the following concepts can be used to identify loops?
(a) Dominators
(b) Reducible graphs
(c) Depth first ordering
(d) None of the above

65. Which of the following concepts are not loop optimization tecniques?
(a) Jamming
(b) Unrolling
(c) Induction variable elimination
(d) None of the above

66. Running time of a program depends on the
(a) Way the registers are used
(b) Order in which computations are performed
(c) Way the addressing modes are used
(d) Usage of machine idioms

67. du-chaining
(a) Stands for use definition chaining
(b) Is useful for copy propagation removal
(c) Is useful for induction variable removal
(d) None of the above

68. Which of the following comments about peep-hole optimization are true?
(a) It is applied to a small part of the code.
(b) It can be used to optimize intermediate code
(c) To get the best out of this technique, it has to be applied repeatedly.
(d) It can be applied to a portion of the code that is not contiguous.

69. Shift-reduce parsers are
(a) Top-down parsers
(b) Bottom-up parsers
(c) May be top-down or bottom-up parsers
(d) None of the above

REFERENCES

[1] Alfred V. Aho and Jeffrey D. Ullman, “Principles of Compiler Design”, 1989.

[2] Y.N. Srikant and Priti Shankar, “The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition”, Dec 7, 2007.

[3] Adesh K.Pandey,” Concepts of compiler Design”, S.K.Kataria and ons publisher of India Books, India

[4] Reinhard Wilhelm & Dieter Maurer,” Compiler Design”, Addion-Wesley, I edition

[5] Gajendra Sharma,” Compiler Design”, S.K.Kataria and ons publisher of India Books, India

[6] K.Muneewaran,” Compiler Design”, Oxford University Press, 2012

[7] K.Krishnakumari,”Compiler Design”, ARS Publications,2013

[8] A.A.Puntambekar, “Compiler Design(Principles of Compiler Design), Technical Publications, 2013

[9] Steven S.Muchnick,” Advanced Compiler Design Implementation”, Morgan Kaufman Publisher,2012

[10] Alexander Meduna,”Elements of Compiler Design”, Auerbach Publication, 2009.

[11] G.Sudha Sadasivam,”Compiler Design”, Scitech Publication, 2009.

[12] P.Kalaiselvi, AAR Senthilkumaar,”Principles of Compiler Deign”,Charulatha Publications,2013.

[13] https://www.tutorialspoint.com/compiler_design/

[14] http://www.dreamincode.net/forums/topic/260592-an-introduction-to-compiler-design-part-i-lexical-analysis/