unit - 1.

Introduction to Compiler.

The todays world depends up on programming languages, because all The softwares running on computers was written in some programming languages. But before a program can run, it first must be Translator to a suitable form in which it can execute on computer.

The software that does this Translation is called as compiler.

Language Pre processors.

Compiler is a program That can read a program in One language i.e., source language, and translate it into another language, known as target language.

An important role of the compiler is to report any errors in source program That it detects during the Translation process.

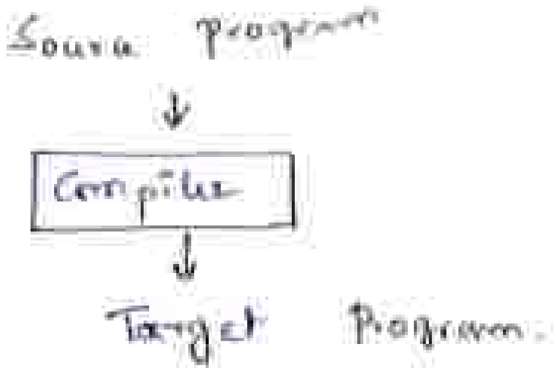
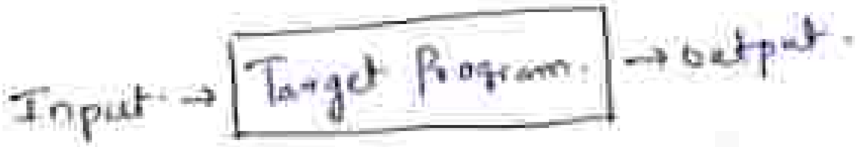
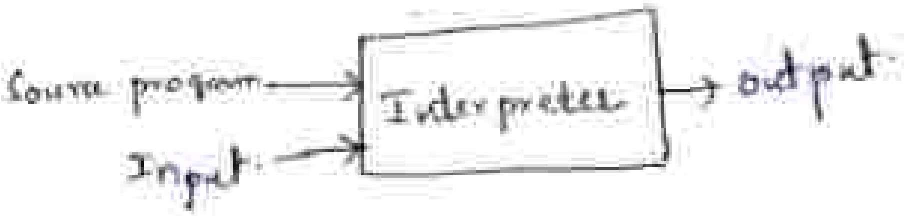


fig. Compiler.

If target program is executable machine program, It can take the inputs from user and produce output.



Interpreter is another common kind of language pre processor, instead of producing target program, interpreter directly executes the operations specified in the source program, and on inputs provided by user.



Machine language Target program produced by a compiler is much faster an interpreter But interpreter can give better error diagnostics

Than Compiler because it executes source program statement by statement.

In Addition to a compiler several Other programs may be needed to Create an executable target program.

1. A source program may be devided into several modules and stored in separate files. Preprocessor collects these files into one unit. Preprocessor may also can expand short cuts, called as macros, into source language statements

2. The modified Source program is Then fed to a Compiler. Compiler program as produces an Assembly-language program as output. Because Assembly language program is easier to debug and to produce Output.

3. The Assembly language is Then processed by Assembler, That produces machine code as output.

— to Synthesis Part.

4. Large programs are often compiled as pieces, The relocatable machine code has to be linked together. The linker links these files.

5. The loader puts together all executable object file into main memory for execution

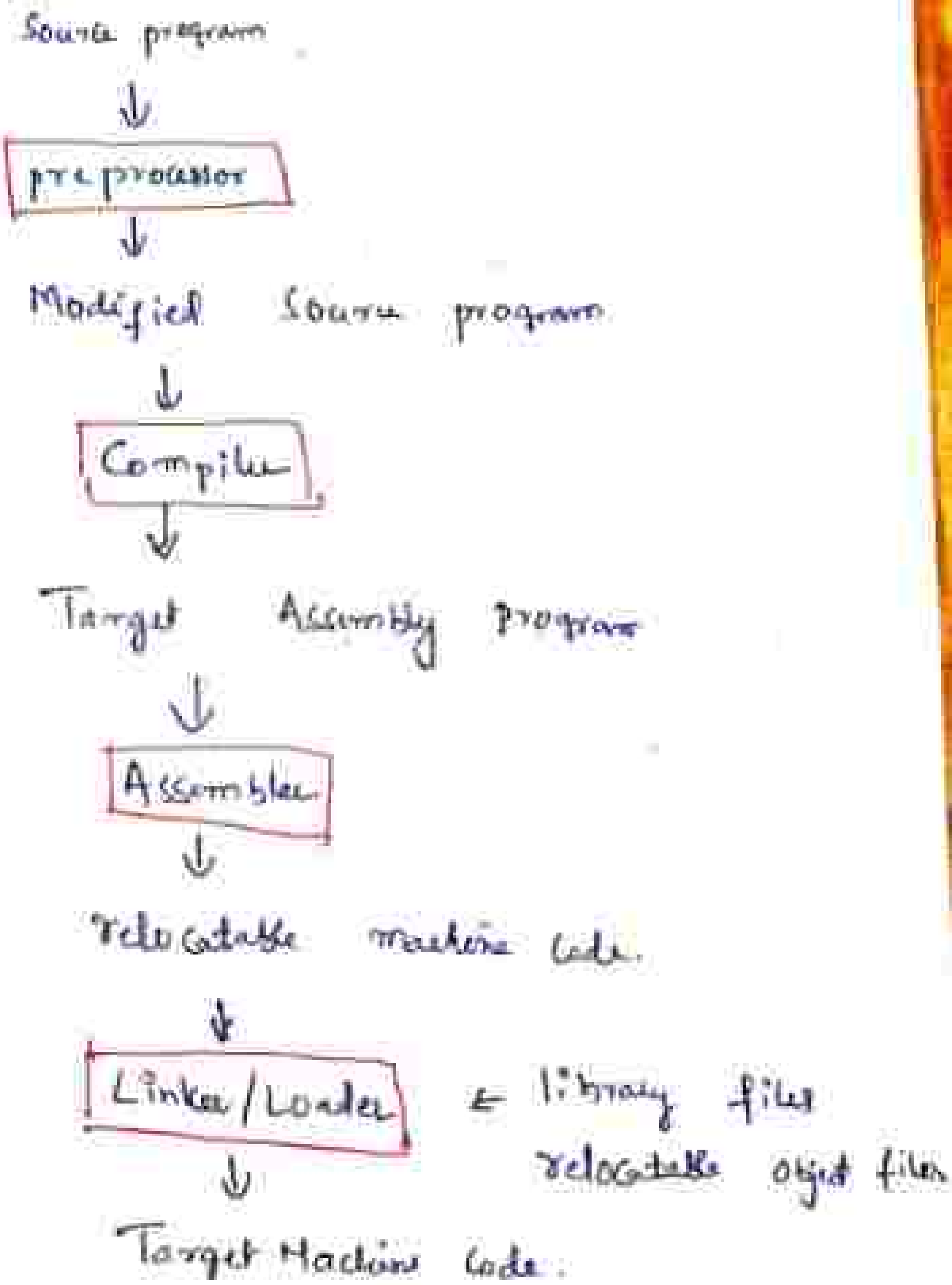


Fig: language Processing System.

The Structure Of a Compiler

Compiler Consists of Two parts.

I. Analysis.
II. Synthesis.

Analysis part breaks up source program into pieces and imposes a grammatical structure on them. It then uses this structure to create intermediate ~~code~~ representation of source program.

If the Analysis part detects that source program is syntactically or semantically weak, i.e; not following the rules of Grammar, Compiler provides informative messages to user in the format of errors, so that user can take corrective action.

This Analysis part collects the info^n about source program, and stores in a Symbol table, which is passed along with Intermediate code to synthesis part.

The Synthesis part construct desired target program from intermediate code and info^n in Symbol Table.

The Analysis part is often called as front end of the compiler, The Synthesis part is called as backend.

Compilation process, operates as a sequence of phases, each of which transforms one representation to other. A typical decomposition of compiler into phases is shown in below figure.

Several phases may be Grouped together.

The Symbol table which stores the info^n about the entire source program is used by all phases of the compiler.

Some compiler have Optimization phase between the front end and backend. The purpose of code Optimization is to do some transformations in intermediate code so that backend can produce a better target program.

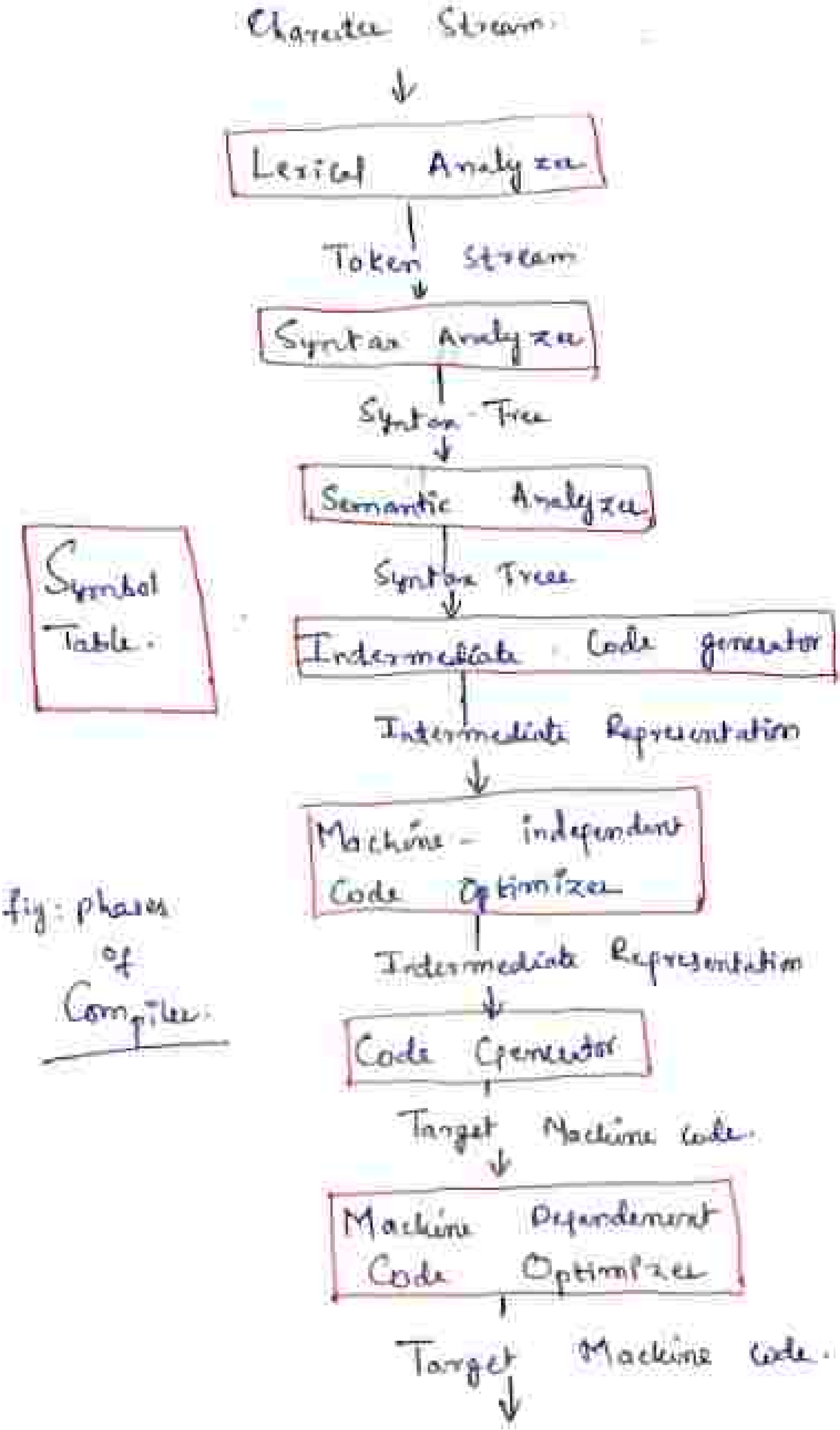


fig: phases of Compiler.

Lexical Analysis

The first phase of Compiler is called as Lexical Analysis or scanning. The Lexical Analyzer reads the stream of characters in a source program, and groups them into meaning full sequences called as lexemes.

For each lexeme, The lexical analyzer produces as o/p a token of the form

<token-name, Attribute value>

which is passed to next phase Syntax Analysis.

In token first component <token-name> is a Symbol that is used during Syntax analysis, Attribute value points to an entry in the Symbol table for this token. Infoⁿ in Symbol table is needed for Semantic analysis and code generation.

For exe consider a source program contains assignment statement

position = initial + rate * 60.

So then ———

1. position is a lexeme, that will be mapped to a token <id,1>, where id stands for identifier, 1 point to symbol table entry. This symbol table entry holds info^n about identifier such as its name and type.

2. The assignment symbol '=' is a lexeme which is mapped to token <=>, it will not have attribute value, so second component is omitted.

3. initial is a lexeme that is mapped to a token <id,2>, where 2 points to symbol table entry for initial.

4. + is a lexeme, mapped to token <+>

5. rate is a lexeme, mapped to token <id, 3>, where 3 points to symbol table entry for rate

6 * is a lexeme, mapped to token <*>

7. 60 is a lexeme, mapped to token <60>

After lexical Analysis The assignment statement can be represented as sequence of tokens as below

position = initial + rate * 60

<id,1><=><id,2><+><id,3><*><60>

Syntax Analysis

The second phase of compiler is syntax analysis or parsing. Parser uses first component of token produced by lexical analyzer to create a Tree like intermediate representation that depicts grammatical structure of token stream. This Tree structure is known as syntax Tree.

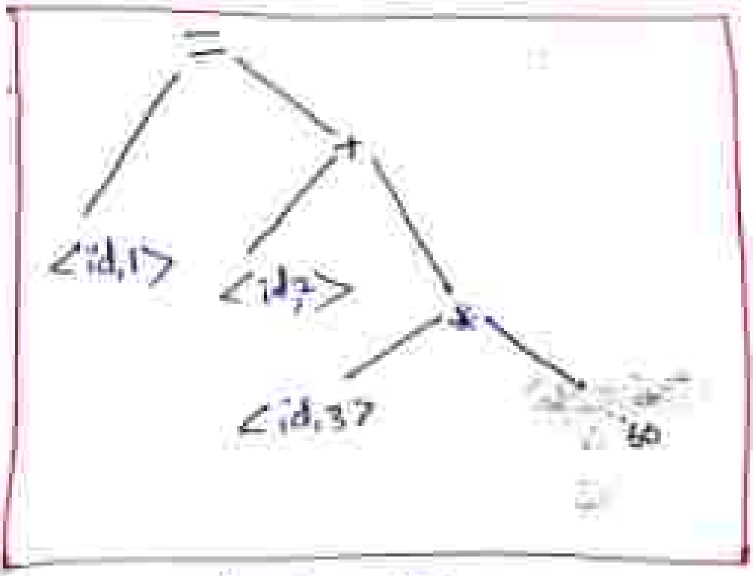
The Tree shows the order in which assignment is performed. For ex in the statement position = initial + rate * 60,

The Tree has a node *, with left child <id,3> as its left child, and 60 as its right child. The node * tells that

We must first multiply rate by 60.

→ The node + represent, we should add the result of multiplication, to the values of initial.

→ The root "=" denotes the right child result should be stored in identifier position.



Syntax Tree.

Semantic Analysis.

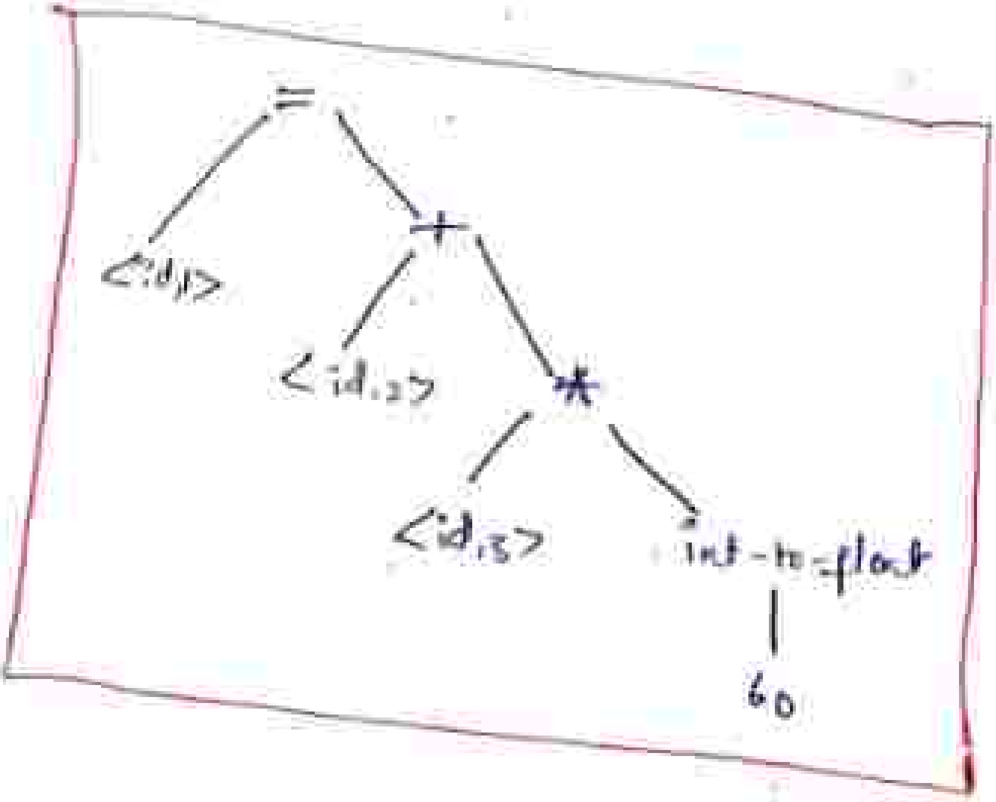
Semantic Analyzer uses The Syntax Tree and the info° in Symbol table to Check the Source program for Semantic Consistency with language definition.

An important part of semantic analysis is type checking, compiler checks each Operator has matching Operands.

The language specification may permit type conversions known as coercions.

For ex, binary arithmetic operator may be applied to either on two integers, or on Two floating point numbers. If operator is applied to floating point number and integer, The compiler will convert integer into a floating point number.

Such coercion appears in following syntax tree, we consider that position, initial and rate have been declared to floating point numbers, and 60 is an integer. The Type checker in semantic analysis converts 60 to float.



Intermediate Code Generation

While converting source program into machine code, a compiler may convert to one or more intermediate forms. Syntax trees are a form of intermediate representation which are used in syntax and semantic phases.

After syntax and semantic analysis, compiler will generate an explicit low-level (or) machine like a intermediate representation. It will have two important properties.

i. it should be easy to produce
ii. it should be easy to translate into the target machine code.

One format of intermediate code used is Three address code, which consist of assembly like instructions with three operands per instruction. Each operand act like a register.

t_1 = inttofloat (60).
t_2 = $id_3 * t_1$
t_3 = $id_2 + t_2$
id_1 = t_3.

In Three address code.

1. At most one Operator should be present at right side of instⁿ.
2. To store the result temporary names are used at left side.
3. Some instructions may have fewer than 3. Operands.

Code Optimization.

Machine independent code optimization phase attempts to improve the improve intermediate code so that better target code will result. Better means faster, shorter, and which consume less power.

In above intermediate code representation The Optimizer can deduce the conversion of 60 from int to float, by replacing 60 by 60.0

t_3 is used only once, to transmit its value to id1, So optimizer can transform previous code as below.

$$t_1 = id_3 * 60.0$$

$$id_1 = id_2 + t_1$$

Code generation

→ Code generator takes intermediate representation of source program and maps it into the target language.

→ If target language code is machine code & code registers or memory locations are selected for each variable used in the program.

→ An important task of code generation is assignment of registers to hold variables.

→ For ex previous optimized intermediate code, using Registers R1 and R2, can be translated to machine code as follows.

```
LDF    R2, id3.
MULF   R2, R2, ##60.0
LDF    R1,id2
ADDF   R1, R1,R2
STF    id1, R1
```

→ The first operand in each instⁿ specifies a destination. F in each instⁿ tell's us that it deals with floating point numbers.

Symbol Table Management

→ Compiler will record the variable names used in source program, and various info^n about attributes of each name.

→ The Attributes may provide Info^n about Storage allocated for a name, type, scope, and in case of functions, the name of arguments, Type of arguments, parameter passing methods and the type returned.

→ Symbol table is a data structure which contains a record for each variable, with fields for the attributes of that name.

→ Symbol table is designed in such a way that Compiler will find the record for each name quickly and stores and retrieves data from that name quickly.

The Science of Building a Compiler.

→ Compiler design is full of beautiful examples, where complicated real world problems are solved easily, mathematically by using abstraction

→ A compiler must accept all source programs that conforms to specification of programming languages. The programs can be large, possibly millions of lines of code.

→ While converting such big programs to Machine code the compiler must preserve meaning of the program.

Modelling in Compiler Design and Implementation.

Compiler design is a study of how to design a mathematical model and choose the right algorithm.

Some of most fundamental models are finite-state machines and regular expressions which are used to describe ~~[illegible]~~ lexical units of programs (keywords, identifiers etc)., And Context-Free Grammars used to describe syntactic structure of programming languages.

The Science of Code Optimization

- → Optimization in Compiler design, attempts to produce code that is most efficient than obvious code.
- → Optimization is complex because processor architecture became is very complex.
- → Optimization is important because, parallel Computers needs substantial Optimization. Otherwise their performance suffers by order of magnitude.

Programming Language Basics.

The Static/Dynamic Distinction.

→ If a programming language uses a policy that allows a compiler to decide an issue, then we say language uses static policy, or that issue can be decided at compile time.

→ A policy that allows a decision to made when we execute a program is said to be a dynamic policy or to require a decision at runtime.

Scope → scope of a variable x, is the region in which x is declared. A language uses static scope or lexical scope if we can say scope at time of declaration. In dynamic scope, same use of x could refer to different declaration of x.

Ex: Consider the use of term "static" in java, to declare a variable, as follows:

public static int x.

Variable is a name of memory location used to hold data value.

Above statement declares x as a static, i.e., only one variable is created, where all objects access same memory location while accessing this variable.

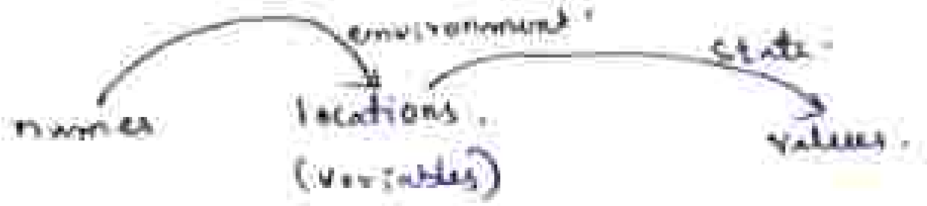
If static is omitted, all objects refer to different memory locations.

Environments and states

The association of names with location in memory, and then values can be described by two mapping which changes as the program runs.

1. **Environment** - mapping from names to memory location is known as Environment. Since variables refer to location (l-values). Environment is mapping b/w name and variable.

2. **State** → It is mapping from memory location to their values. State maps l-value to corresponding r-values in C terminology.



→ Most binding of names to locations is dynamic.

→ The bindings of locations to values is generally dynamic as well as static.

For ex:-

```
#define   ARRAYSIZE   1000
```

binds the name [illegible] ARRAYSIZE to a value
statically.

Static scope and Block Structure.

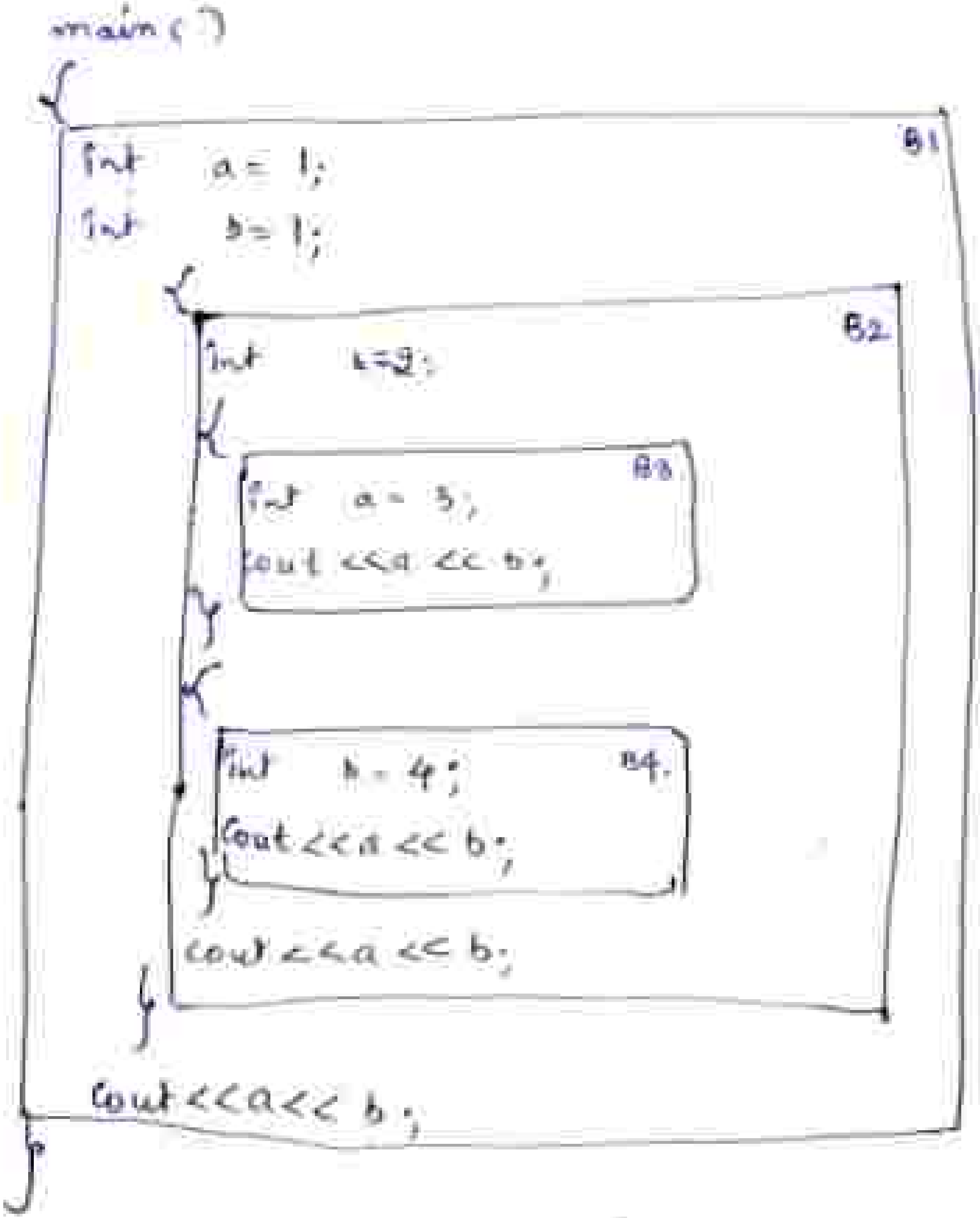
The scope rules for C are based
on program structure. Scope of a variable is
implicitly determined by where declaration appears
in the program. In languages like C++, C#
and java provides explicit control over scope through
use of keywords like public, private and
protected.

Static scope can be given by using blocks.
declarations and
Block is a grouping of '{' and '}' to denote a
braces
statements. C uses like Algol we
languages
block. In some a block.
'end' to denote
use 'begin' and

Blocks can be nested inside other blocks.
We say a variable declaration D belong to
a block B, if it is declared in B.
of a name x
If the declaration D is in B, Then
Scope of x is B, as well as in any
nested block. x will not have scope in

nested block if it is redeclared in that block.

Ex:- Consider the program shown in below figure, which has four blocks, with several definitions of a and b. each declaration initializes the variable the number of block to which it belongs.



```
main()
{
    int a=1;
    int b=1;
    {
        int b=2;
        {
            int a=3;
            cout<<a<<b;
        }
        {
            int b=4;
            cout<<a<<b;
        }
        cout<<a<<b;
    }
    cout<<a<<b;
}
```

For ex, Consider the declaration of 'int a=1' in B1, it Scope is entire block B1, except the nested block of B1, where 'a' is redeclared. B2 is immediately nested under B1, but does not contain redeclaration of ~~a~~ a. B3, which is nested under B1, has redeclaration of ~~a~~ a, So it is out of scope of declaration of a in B1, B4 does not contain redeclaration of a, so it is in scope of B1's declaration.

Consider print statement of B4, since B4 has redeclared b=4, so it prints ~~a~~ b value as 4, and it does not contain a, it surrounding block B2 also not contains a's declaration, B2's surrounding Block B1, contains a's declaration. So a's value is printed as 1.

Declaration	Scope.
int a=1	B1 - B3
int b=1	B1 - B2
int b=2	B2 - B4.
int a=3	B3.
int b=4	B4.

Explicit Access Control.

If p is an object of class with a member x, then p.x refers to access of object p to class member x. And if a class C with member x, its subclass C' can also access x, except if it restricted x.

The Object Oriented languages like C++, Java provide explicit access control through keywords public, private, protected. These keywords provide encapsulation by restricting access.

→ private limits the access to the members declared within the class and "friend" classes.

→ protected can members can be accessed from subclasses.

→ public members can be accessed from outside the class.

Unit-I

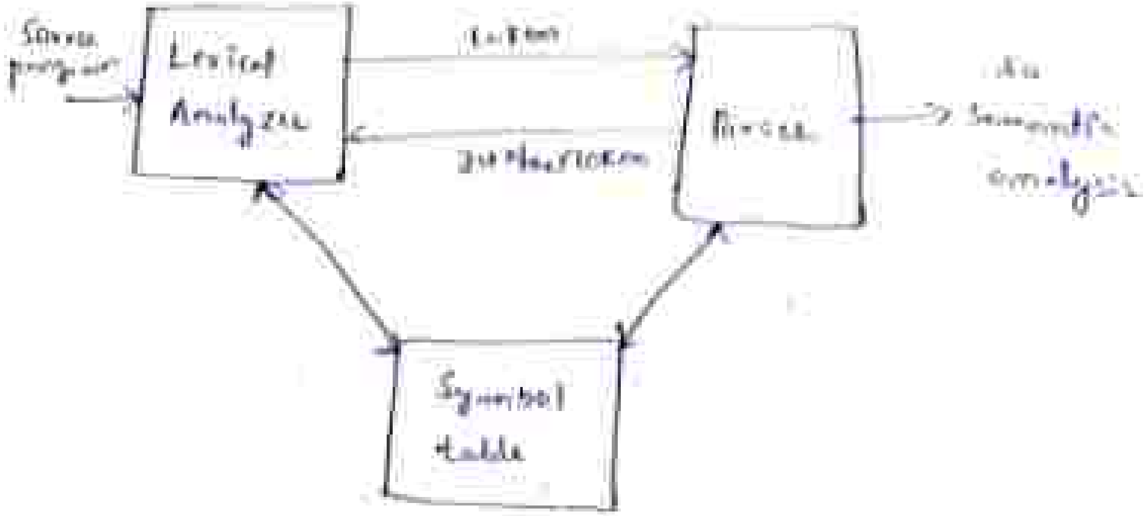
Chapter-2

Lexical Analysis

Role of the Lexical Analyzer

As it is the first phase compiler it's main task is to read input characters of source program, group them into tokens of lexemes, and produce as output sequence of tokens from each lexeme. The stream of tokens is sent to build a parser for syntax analysis. Lexical analyzer also interacts with symbol table. When lexical analyzer discovers a lexeme as an identifier, it enters the lexeme into symbol table. These interactions are shown in following figure.

Commonly the interaction is started by parser. The call from parser "getNextToken" command causes the lexical analyzer to read characters from input until it identifies next lexeme, and produces token from it, which it returns to parser.



As the Lexical analyzer is part of Compiler It will also do some other tasks besides Identification of Lexemes.

1. It will remove comments and white spaces (blank, newline, tab).
2. It will correlate error messages generated by Compiler with source program. It will associate a line number in source program with error message.
3. If the source program uses macros, The expansion of macros will be performed by Lexical Analyzer.

Lexical Analyzer consist of two processes. viz.

a) Scanning → Simply reads input, does not Concerned with tokenization.

b) Lexical Analysis → It is Complex, generate tokens from output of Scanner.

Lexical Analysis Versus Parsing

The reasons for separating lexical analysis from parsing are as follows

1. The design of compiler will be simplified. For ex, if comments and white spaces are not removed in Lexical Analyzer, the parser's task will become more complex.
2. Compiler's Efficiency is improved.
3. Compiler's portability is enhanced.

Tokens, Patterns, Lexemes.

→ Token is a pair consists of token name and optional attribute value. Token name represents kind of lexical unit. Token name works as input to parser.

→ pattern denotes type / form of lexeme of a token. For ex, if token is a keyword, the pattern will be character that forms keyword. For identifier the pattern is more complex

→ lexeme is a sequence of characters in source program that matches pattern for a token.

For ex: in below statement

printf("Total = %d\n", score);

printf, score matches the pattern for token id, and "Total= %d\n", is a lexeme matching literal.

Token	Informal Description	Sample lexemes
if	Characters i, f	if.
else	Characters e, l, s, e	else
Comparision	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters or digits	pi, score, d2.
literal	Anything ~~that~~ but surrounded by "'s.	"hello".

Examples of tokens.

Attributes for tokens.

When more than one ~~token~~ lexeme matches a pattern, we need additional infoⁿ about ~~p~~ matched lexeme and pattern.

For ex. for pattern of token "number" matches both 0 and 1. So it is important to know for code generator to know which lexeme is found in source program. Thus in many cases lexical analyzer sends token name and attribute value pair to parser that describes lexeme represented by token. The most important example is the token id. When we need to associate several infor with identifier like its lexeme, token, type, and where it can found in symbol table. Thus appropriate values for identifier is a pointer to symbol table entry for that identifier.

For ex, The ~~id~~ token names and associated attribute values for the Fortran statement

$$E = M * C ** 2.$$

are given as follows.

< id, pointer to symbol table entry for E >
< assign-op >
< id, pointer to symbol table entry for M >
< mult-op >
< id, pointer to symbol table entry for C >
< exp-op >
< number, integer value 2 >.

Input Buffering.

The task of reading lexeme from source program can be speed up by using buffers. It is a difficult task because we need to see next character, before we can sure that we have right lexeme. For ex, we can not be sure that we've have seen end of identifier until we see a character that is not a letter or digit, and any thing which is not a part of lexeme id.

Buffer pairs.

Buffers are used to reduce the overload involved in reading input characters. This scheme consists of two buffers.

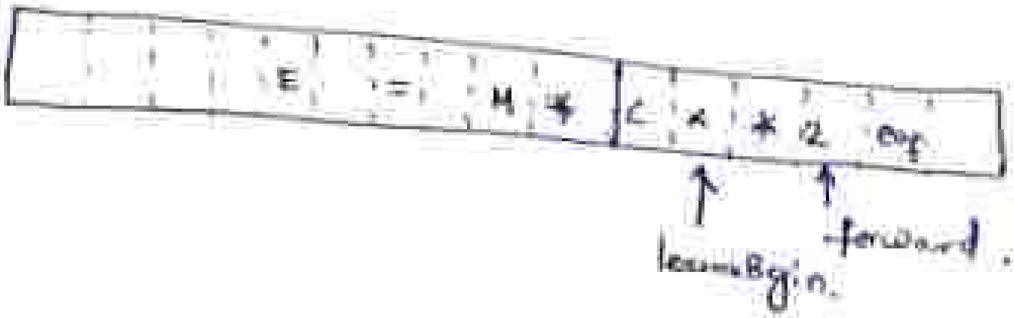
Each buffer size is same, say N, i.e., size of disk block e.g. 4096 byte. Using a read command we can read N characters into buffer using one system call at a time, rather than using one system call per character. If fewer than N characters remain in i/p file, special character 'eof' marks the end of source file, which is different from characters in source program.

Two pointers to input are maintained

1. pointer "LexemeBegin" marks the beginning of current lexeme, whose length we are attempting to determine.
2. Pointer "forward" scans until the pattern is matched.

Once the lexeme is matched, forward is set to character at right end. After the lexeme is found and passed to parser as token, "lexeme begin" is set to next character found immediately after the lexeme just found.

When "forward" pointer has reached end of the buffer, we must reload other buffer from the input, and move forward to beginning of newly loaded buffer.

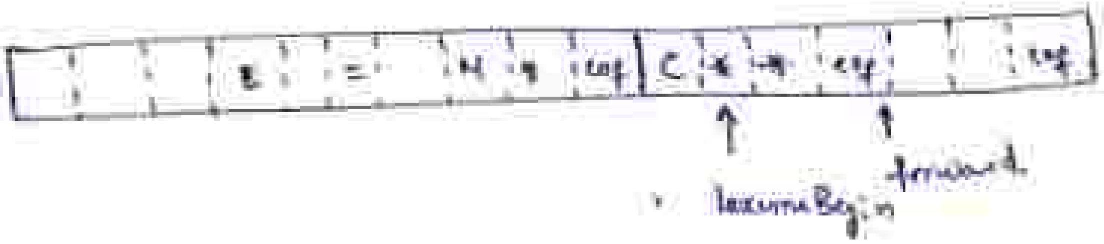


Sentinels.

For each character we read, we make two tests, one for end of the buffer, and second one to determine which character we read.

We can combine the buffer end test, and with test for new character read by using a special character known as sentinel character at the end. Sentinel is a special character which is not part of source program, and this is represented by eof.

Eof is placed at the end of the buffer, as well as at the end of entire i/p. As shown in below figure.



Recognition of Tokens.

The input string is compared with patterns, and then tokens are generated for ex. Consider below grammar

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number

The above grammar is a simple fragment of branching and conditional statements. This syntax is similar to pascal

⇒ relop is a comparisim operator like "=" for equals, and "<>" for not equals in pascal

Terminals of the grammar is which are if, then, else, relop, id and number are the names of tokens.

The patterns for these tokens are given by regular definitions as follows.

digit → [0-9]

digits → digit⁺

number → digits(. digits)?(E[+-]?digits)?

letter → [A-Za-z]

id → letter(letter|digit)*

if → if.

then → then.

else → else.

relop → < | > | <= | >= | = | <>

The keywords if, then, else are the lexemes, They are not identifiers, but their lexemes matches patterns of identifiers.

Another job of lexical analyzer is to remove white spaces, by using token "ws" which is defined as follows.

ws → (blank | tab | newline)⁺

When any of these are recognized as "ws", We do not return it to parser, rather the next character after whitespace is returned. i.e; Next token will be forwarded to parser.

Transition Diagrams

An intermediate step in lexical analyzer is we first convert patterns into flowcharts called as "Transition Diagram".

Transition diagram have a collection of nodes or circles called as states. Each state representation a condition that could occur during the process of scanning the input for matching the lexeme to pattern. State is a summary between lexemeBegin pointer and forward pointer.

Edges directs from one state of Transition diagram to other. Every edge is labled by one or more symbols.

If we are in state 's', and next input symbol is 'a', we look for an edge out of s labled by a. If such an edge is found, we advance 'forward' pointer, and enter into the state where edge leads. If every state has only one edge The transition diagram is deterministic otherwise non-deterministic.

Some important conventions about "transition diagram" are:-

1. Some states are called as accepting or final states. These states indicate that lexeme has been found. The accepting/final state is represented by double circle. If any action to be taken on this state like returning a token and attribute value to the parser, will be attached to accepting state.

2. If it is necessary to retract / move forward pointer backward to one position, we should place a * near accepting state.

3. One state is designated as "start state", initial state", which is indicated by a label start entering from nowhere.

Ex:- Following figure shows a transition diagram that recognizes lexemes matching the token "relop" (relational operator).

→ We begin at state 0.

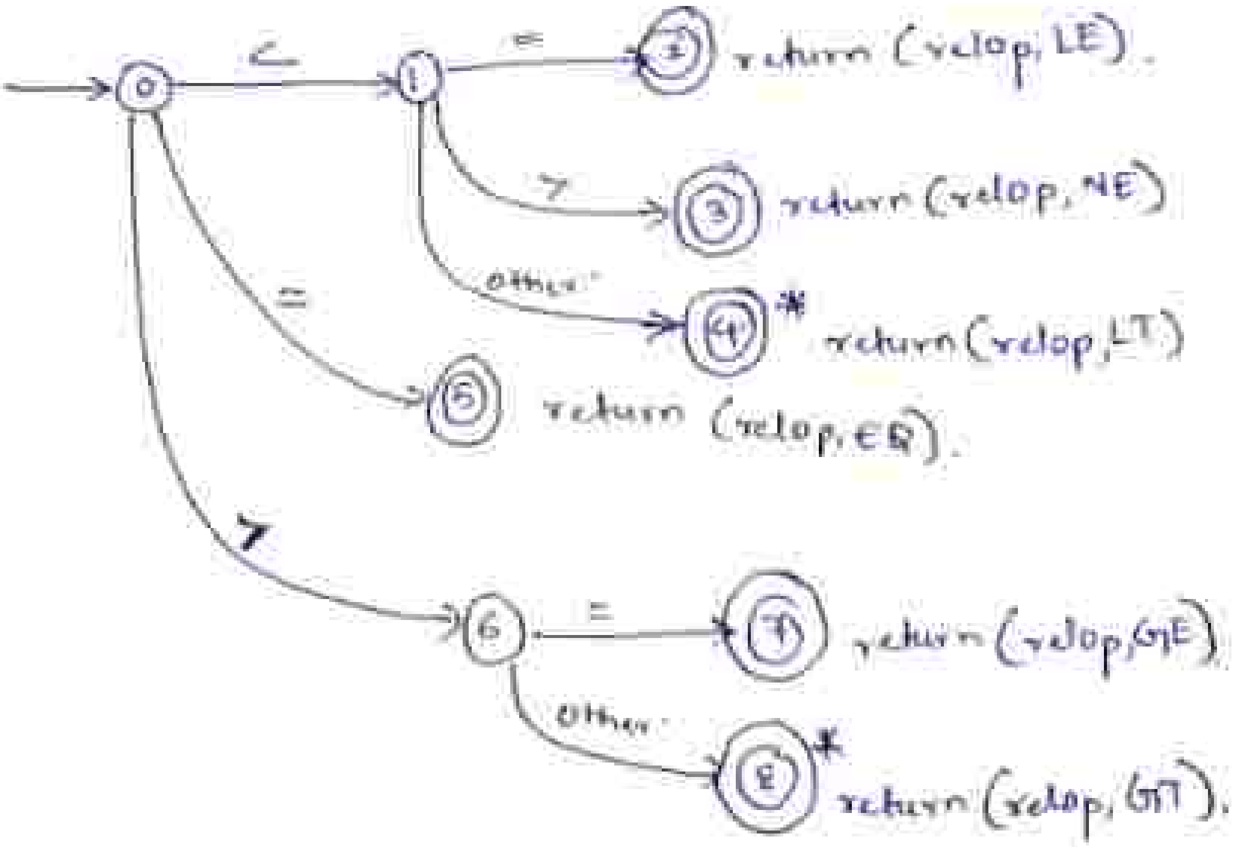
→ If input symbol is <, then among the lexemes that matches the pattern for relop we look for <, <>, <=.

* we therefore goto state 1, and looks for next character.

→ If next character is =, we recognize <= and enter into state 2 and return the token relop with attribute LE

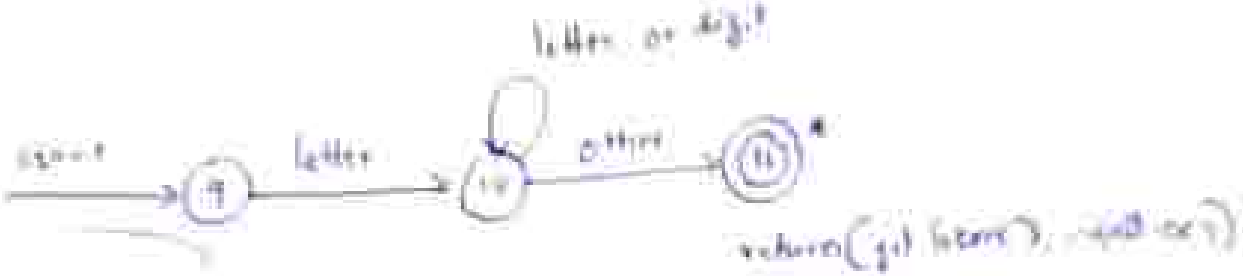
→ If in state 1, the next character is >, Then we have lexeme <>, and enter into state 3, and returns not equals operator token.

→ On Any other character, The lexeme is <, and we enter into state 4, and returns that info. State 4 has * to indicate that we must retract the input one position.



Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers is some what different from recognizing operators. Keywords like if and then are reserved. They are not identifiers. Following transition diagram identifies keywords if then else etc.



There are Two Ways, to recognize reserved words that looks like identifiers.

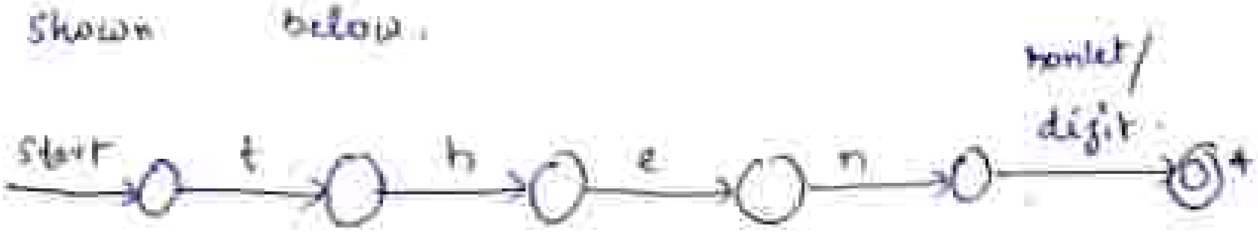
Install the reserved words in symbol table initially, A field of the symbol table [illegible]le that they are not identifier tells which token they represent

When we find an identifier, installID(), places the identifier in symbol table if it is not there already, and returns a pointer to the symbol table entry found.

→ Any identifier not in the symbol table during lexical analysis can not be a reserved word, so its token is id.

→ getToken(), examines the symbol table entry for lexeme found, and returns the token name either id or one of the keyword tokens that initially installed in the table.

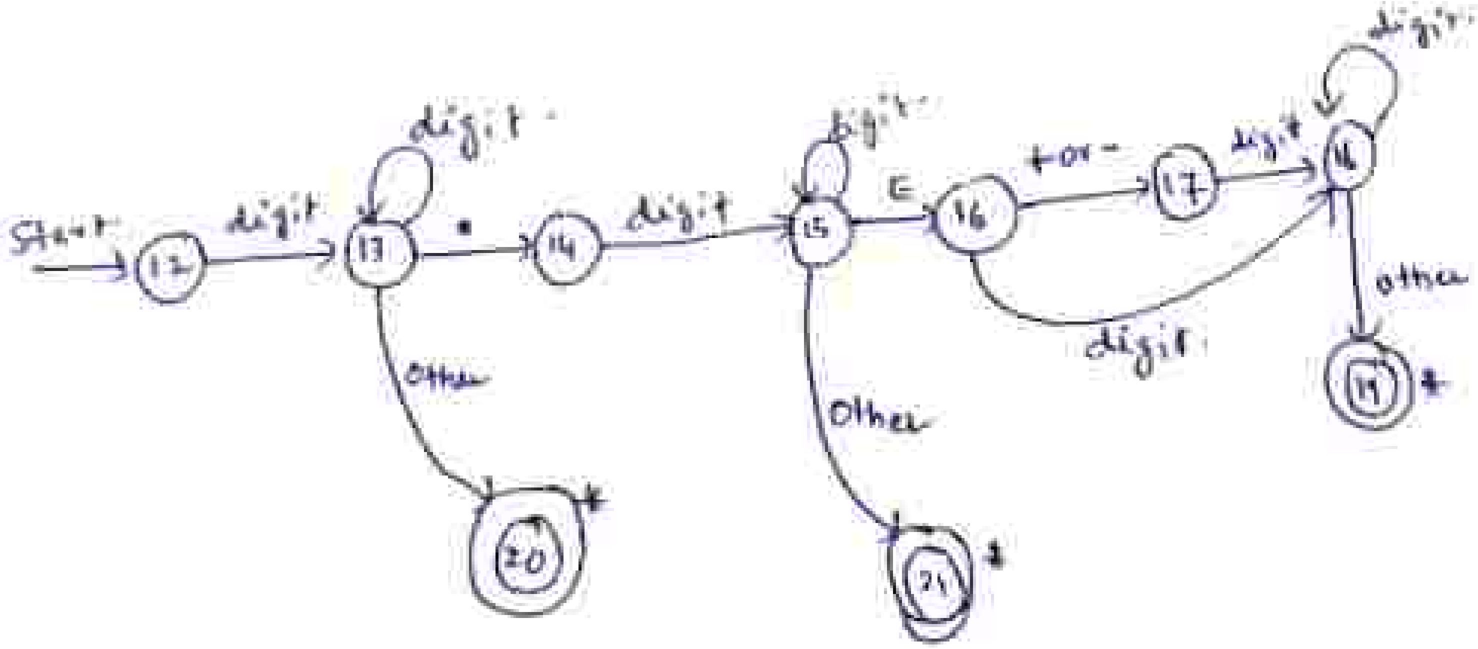
2. Create a seperate transition diagram for each keyword, an one for keyword "then" is shown below.



→ Transition diagram consist of states representing successive letters of keywords seen followed by test for "non-letter-or-digit" i.e., Any character that can not be a continuation of identifier. It is necessary to check the end of identifier, or else we return "then", when the correct token was 'id' for lexeme "thenextvalue". We If this approach is used, we should prioritize tokens so that reserved word tokens are recognized in preference to id, when lexeme matches both patterns.

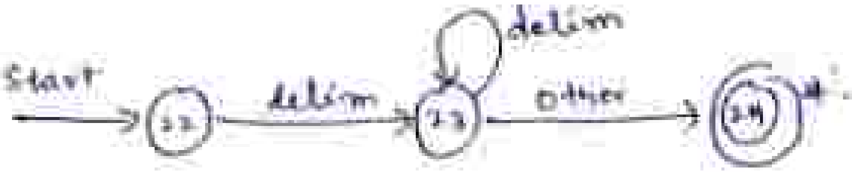
→ This approach is not used in our example, so the states in Trancition diagram Or numbered.

Trancition Diagram for Unsigned Numbers.



A Trancition Diagram for White Space

In following diagram we look for one or more white spaces, represented by delim. These delim characters could be blank, tab, newline and characters which are not considered to be part of any token.



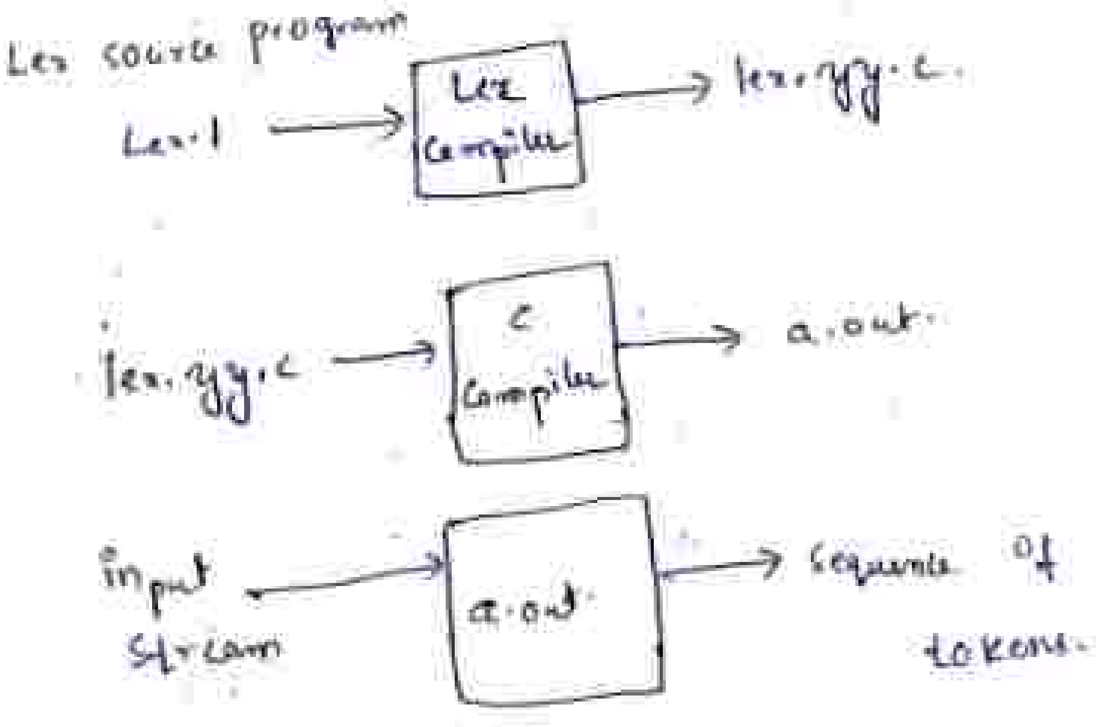
The Lexical-Analyzer Generator Lex.

Lex or Flex, allows to specify a lexical analyzer, which describes patterns for tokens.

The input notation for Lex tool is known as the Lex language, and the tool is known as Lex Compiler. Lex Compiler transforms input patterns into transition diagram and generates code in file called as lex.yy.c.

Use of Lex.

Following figure shows how Lex is used.



→ An input file lex.l which is written in Lex language describes lexical Analyzer to be generated.

→ Lex compiler transforms Lex.l to a C program, which is named as lex.yy.c

→ This C File is compiled by C Compiler, to a file a.out.

→ This a.out file takes the input stream of source program and generates sequence of tokens.

→ a.out is a function in C, that returns an integer, which is code for token names, The Attribute value, i.e; either a numeric code, or pointer to symbol table entry is placed in a global variable "yylval" which is shared between Lexical Analyzer and parser. so a.out returns ~~attribute~~ Token name and attribute value.

Structure of Lex Program.

Lex program has following pattern

declaration
%%
Transition rules
%%
Auxiliary Functions.

→ Declaration section consists of declaration of variables and constants.

→ The Transition rules will have the form

pattern { action }

→ pattern uses definitions of declarations in declaration section. The Actions fragments of code written in C.

→ The Third sections holds additional functions used in Actions.

Lexical Analyzer created by Lex, when called by parser, reads input string one character at a time, until longest prefix matches any one of the pattern P_i, it then executes Action A_i. A_i returns a single value Token Name to parser, and a shared variable yylval passes additional info about lexeme found if needed.

If P_i describes white space, lexical Analyzer proceeds to find additional lexemes which match to a pattern, and corresponding Action is returned to parser.

<u>Example</u> Following is a lex program that recognizes tokens of below figure and returns Token Found.

Lexeme	TokenName	Attribute Value
Any ws	—	—
if	if	—
Then	Then	—
else	else	—
Any id	id	pointer to table entry
Any Number	number	pointer to table entry.

<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Finite Automata.

Lex is a tool, which converts input program to lexical Analyzer. The heart of transition is known as Finite Automata.

Finite automata are graphs, like Transition diagrams, with following differences.

1. Finite Automata are recognizers, They say "yes" or "no" about each possible string.

2. Finite Automata is of two types.

   i. Non Deterministic Finite Automata (NFA).

   A symbol can label many edges out of same state, and ϵ is an empty state string, is a possible label.

   ii. Deterministic Finite Automata (DFA)

   Each state, for each symbol as input, has exactly one edge with symbol leaving that state.

Both DFA, and NFA can recognize same language known as regular languages.

Non Deterministic Finite Automata (NFA)

NFA consists of

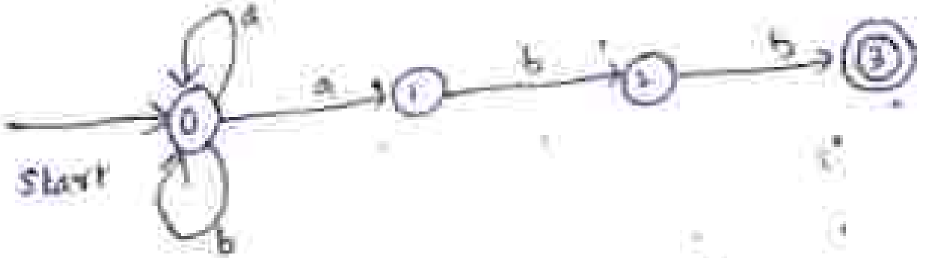
1. A finite set of states S.
2. A set of input symbols Σ, known as input alphabet, ϵ is an empty input string, which is never a member of Σ.
3. A Transition function δ, which gives for each state and symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.
4. A start state s_0, from S. (or initial state)
5. A set of states F, known as accepting states (or final states).

We can represent NFA or DFA by using transition graph, where nodes are states, labeled edges represent transition function. This graph is same like Transition diagram but with following differences.

a). The same symbol, can label edges from one state to several different states.

b) An edge may represent by label ϵ, the empty string, in addition to symbols of input string.

__Example__ :- An NFA, recognizing the language of regular Expression $(a|b)^*abb$ can be given as follows. This example describes all strings of a's and b's ends with abb



This NFA, accepts all strings ends with abb.

Trancition Table.

NFA can also represented by Transition table, whose rows corresponds to states, and columns corresponds to input symbol and ε. The entry in the table for a given state and input is the target state. If the trancition function has no info^n about state input pair, we put ϕ in the table.

The Transition table for previous NFA can be given as follows.

State	a	b	ε
0	{0,1}	{0}	ϕ
1	ϕ	{2}	ϕ
2	ϕ	{3}	ϕ
3	ϕ	ϕ	ϕ

Accepting input Strings.

→ NFA accepts input string x, if only there is some path in the transition graph from the start state to one of the Accepting state. Labels with ε along the path are ignored.

Ex:- The string aabb, of previous NFA is accepted, The path is from state 0 to 3 demonstrating the fact as follows.

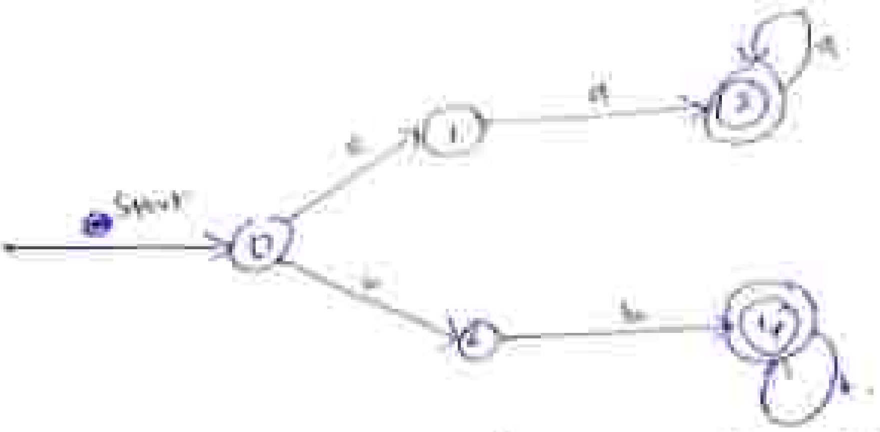
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3.$$

Several path for same string may lead to different states. for ex

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

This path leads to state 0, which is not final state, so string on this path is not accepted.

Ex:- 2) Following NFA accepts $L(aa^*|bb^*)$.



The String "aaa" is accepted because of path

$0 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$

Deterministic Finite Automata (DFA)

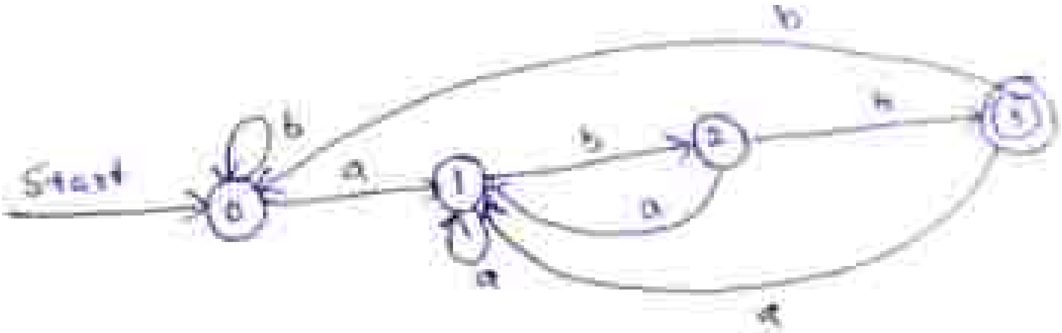
DFA is special case of NFA, where

1. There are no moves on input ε, and
2. For each state s, and input symbol a, There is exactly one edge labeled by 'a' Out of s.

In Transition Table of DFA, each entry is single state. So states are represented by ~~us~~ without using curly braces, which are used to represent sets.

Every regular expression and NFA can be converted to DFA accepting same language. DFA is used to build or simulate lexical analyzers.

Ex:- following figure represents DFA accepting language $(a|b)^*abb$, same as previous ex.



Given string "abab**b**" is accepted along the path

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Regular Expression to Automata.

Regular Expressions is a choice notation to describe lexical Analyzers and Other pattern processing Software.

Implementation of the required simulation of a DFA. Because NFA has choice of input symbol or an ε NFA's simulation is less straightforward than DFA. This is is often important to convert NFA to DFA that accepts same language.

Conversion of an NFA to DFA.

→ After conversion each state of DFA, corresponds to a set of NFA states. This technique is known as subset construction. This can be given by using following Algorithm

Algorithm: Subset construction of a DFA from NFA

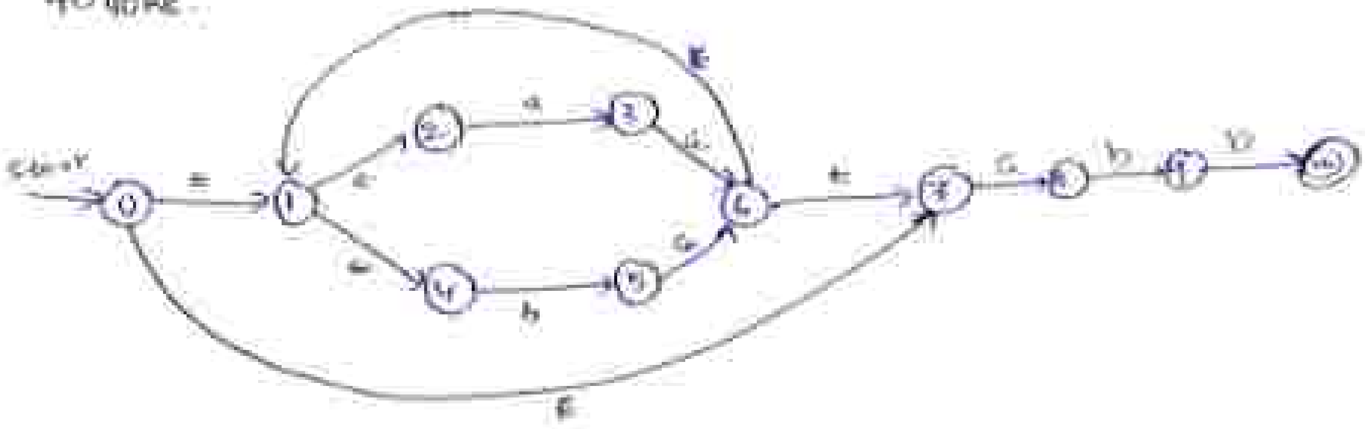
Input : An NFA N.
Output : A DFA D, accepting same Language N

Method :- → This algorithm constructs a Transition table $\underline{D_{tran}}$ for DFA.

→ Each state of DFA, also contains set of states of NFA, which has moves on a given input string. First we need to deal with null transitions of NFA. Following figure shows different functions that describes computations on states of NFA. s denotes single state of NFA and T is set of states of NFA.

Operation	Description
ε-closure(s)	→ set of NFA states reachable on ε-transitions alone.
ε-closure(T)	→ set of NFA states, reachable from set of states in set T on ε Transitions.
move(T,a)	→ set of states from T on a,

→ Following figure shows NFA, accepting $(a|b)^*abb$. It can be converted to DFA as follows.



→ ε-closure(0) is {0,1,2,4,7}, These are the States which are reachable from 0, on ε-transition. State 0, can be reached from itself using ε-transition. These set of state are equivalent to a set A, in DFA.

→. The input alphabet is {a,b}, we calculate/find ε-closure (move (A,a)) i.e; Dtrans [A,a], and ε-closure (move (A,b)) i.e; Dtrans [A, b].

→ In state A, only 2 and 7, have Transition on a to 3 and 8 respectively. Thus Mov (A,a) = {3,8}, and ε-closure({3,8}) = {1,2,3,4,6,7,8}

So:

$D_{trans}[A, a] = \epsilon\text{-closure}(move(A, a)) = \epsilon\text{-closure}(\{3, 8\})$

$= \{1, 2, 3, 4, 6, 7, 8\}$.

Call this state as B.

$D_{trans}[A, a] = B$.

⇒ Now $D_{trans}[A, b]$, among state of A, only 4 has transition on b to 5. so

$D_{trans}[A, b] = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$

Call this state as C.

$D_{trans}[A, b] = C$.

⇒ $D_{trans}[B, a] = \epsilon\text{-closure}(\{3, 8\})$, which is similar to B itself.

So $D_{trans}[B, a] = B$

⇒ $D_{trans}[B, b] = \epsilon\text{-closure}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}$

let $D_{trans}[B, b] = D$.

$D_{tran}[C,a] = \epsilon\text{-closure}(move(C,a)) = \epsilon\text{-closure}\{3,8\}$

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

$D_{tran}[C,a] = B.$

$D_{tran}[C,b] = \epsilon\text{-closure}(move(C,b)) = \epsilon\text{-closure}\{5,$ [illegible]$)$

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

$D_{tran}[D,a] = \epsilon\text{-closure}(move(D,a)) = \epsilon\text{-closure}\{3,8\}$

$= \{1,2,3,4,6,7,8\}$

$= B.$

$D_{tran}[D,b] = \epsilon\text{-closure}(move(D,b)) = \epsilon\text{-closure}(5,10)$

$= \{1,2,4,5,6,7,10\} = E$

$D_{tran}[E,a] = \epsilon\text{-closure}(move(E,a)) = \epsilon\text{-closure}(3,8)$

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

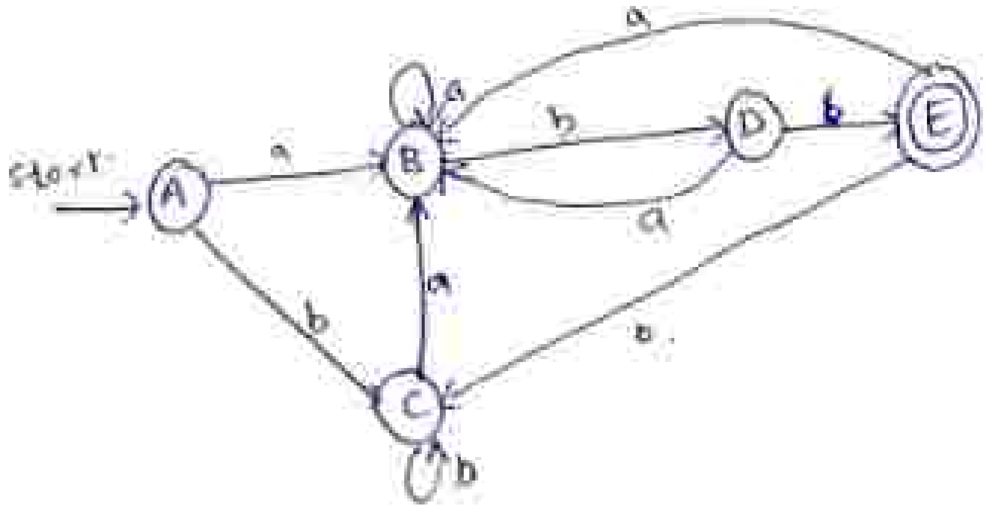
$D_{tran}[E,b] = \epsilon\text{-closure}(move(E,b)) = \epsilon\text{-closure}(5)$

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

State A is the start state, and state E, which contains state 10 of NFA, is only accepting state. The final D_{Trans} can be given as

NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C.
{1,2,3,4,6,7,8}	B	B	D.
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	E	B	C

The DFA for above NFA using subset construction is given as follows.



Design of a Lexical Analyzer Generator.

Following figure shows the architecture of lexical analyzer generated by Lex. The program use either finite automation which can be either deterministic or non deterministic.

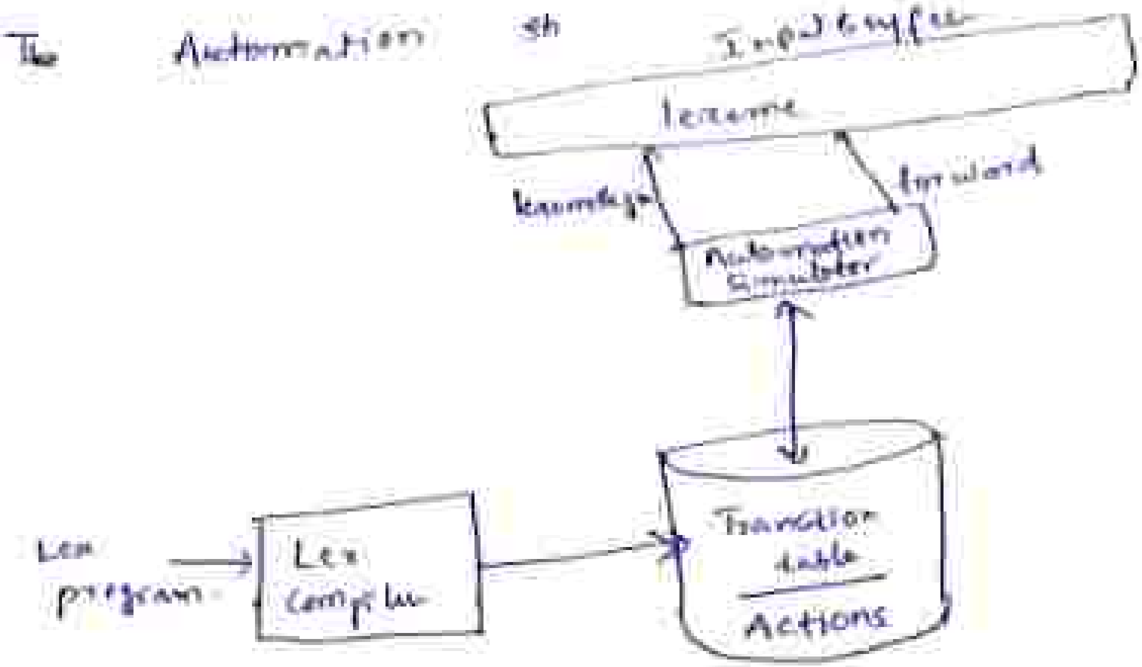
Following is a lex program which is turned into a Transition table and actions which will be used by Finite Automata Simulator.

They components include

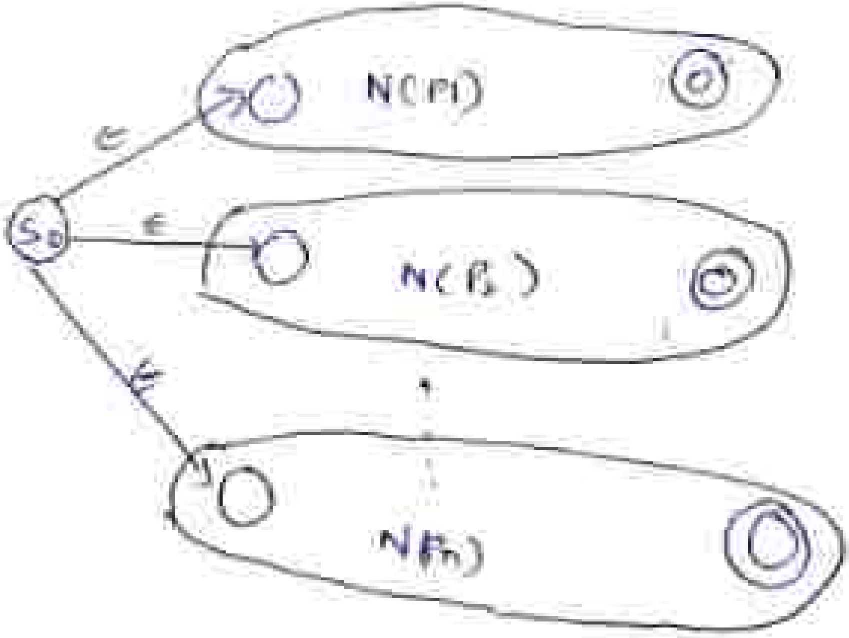
1. A Transition table for automation.
2. Functions passed directly through Lex to the output.
3. The actions from the input program, which are to be invoked at the appropriate time by simulator.

To construct an automation, we begin by taking each regular expression pattern in Lex program and converting it to NFA.

The Automation sh



The Automation will recognize all lexemes matching any of the patterns in source program with ε-transitions to from each start states of the NFA's N_i of pattern P_i as shown in below.



Example:

Consider following patterns.

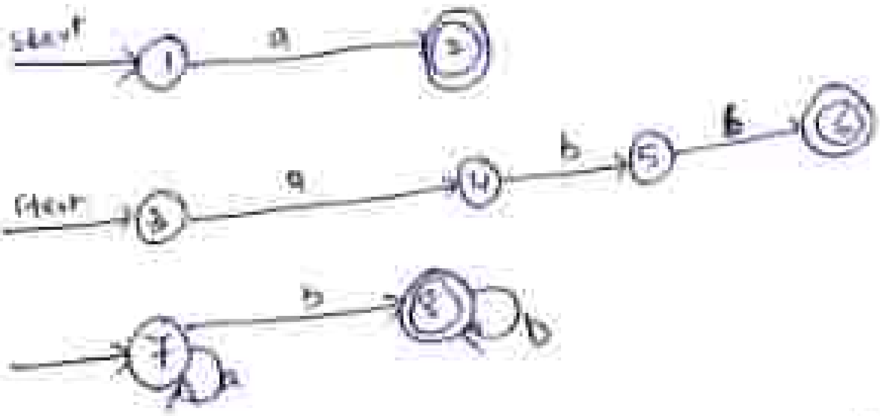
a { action A_1 for pattern P_1 }

abb { action A_2 for pattern P_2 }

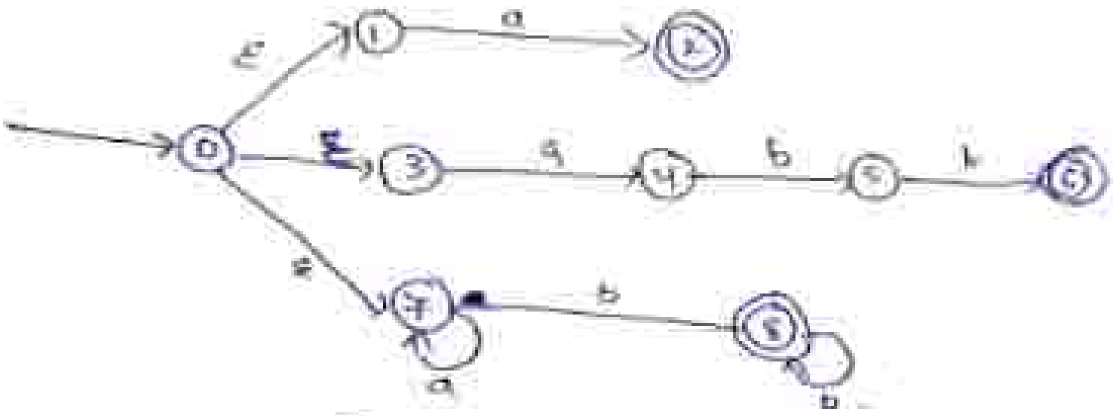
a^*b^+ { action A_3 for pattern P_3 }

There are some conflicts in this example. The pattern abb match P_2 and P_3 but we consider it as lexeme for P_2, because it is listed first.

Following figure shows NFA's that recognize three patterns. The last figure shows combination of all above three NFA's.



NFA's for a, abb, and a^*b^+



Combined NFA.

Pattern Matching Based On NFA's.

Lexical Analyzer which Stimulates NFA, reads the input from the begining referred to as "lexeme Begin", As it moves forward pointer, it calculates the set of states it is in each point.

When NFA reaches a point on input, where there are no next states, the next set of states will be empty. Then by seeing accepting states in transition we decide the longest prefix that is a lexeme matching the pattern.

We look backward in sequence set of states, until we find one or more accepting states. If there are more than one accepting state we take the state with earliest pi (pattern) in the list from Lexical Analyzer, move the forward pointer back to the end of lexeme, and perform the action A_i associated with pattern P_i.

Ex:- Consider patterns of given example, and input begins with "aaba". The below figure shows set of states of NFA of previous figure, we enter state {0,1,3,7} on ε-closure which is the initial state. After reading fourth i/p symbol, we are in empty set of states, since

There are no transitions out of state 8 on input a.

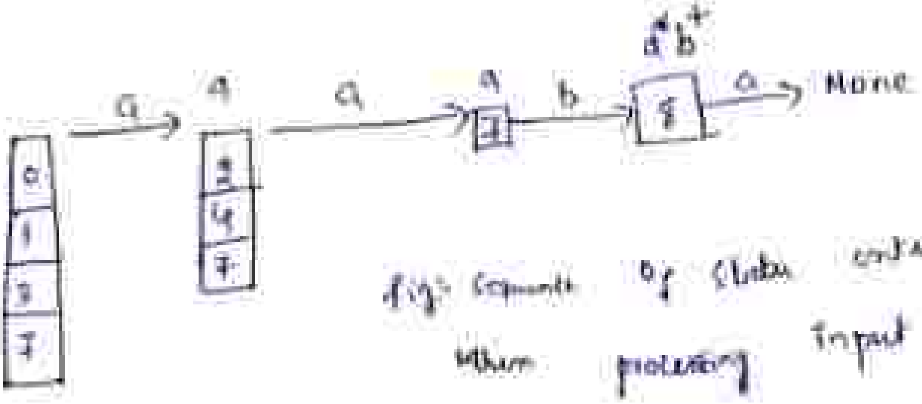


fig: sequence of states entered when processing input aaba.

Thus we backup from that position, looking for accepting states, we notice in above figure after reading first input symbol a, we are in state 2, which indicates pattern "a" is matched, but after reading aab, we are in state 8, which indicates a^*b^+ has been matched. Thus prefix aab, is the longest prefix up to an accepting state, we therefore selects "aab" as lexeme, and executes Action A_3, which returns the parser that token whose pattern $P_3 = a^*b^+$ has been found

Optimization of DFA Based Pattern Matchers

There are three algorithms that are to used to implement and optimize pattern matchers from Regular Expressions

1. First algorithm is used in LEX compiler, it constructs DFA directly from a regular expression, without constructing intermediate NFA.
2. Second algorithm minimizes the states of DFA, by combining the states that have the same future behavior.
3. The third algorithms produces more compact representations of transition tables than the standard, two dimensional table.

Important States of NFA

An NFA state is known as important if it has non-ε out transitions. The Subset. The subset construction uses only the important states in a set T, when it computes ε-closure (move (T,a)), The set of states reachable from T on a.

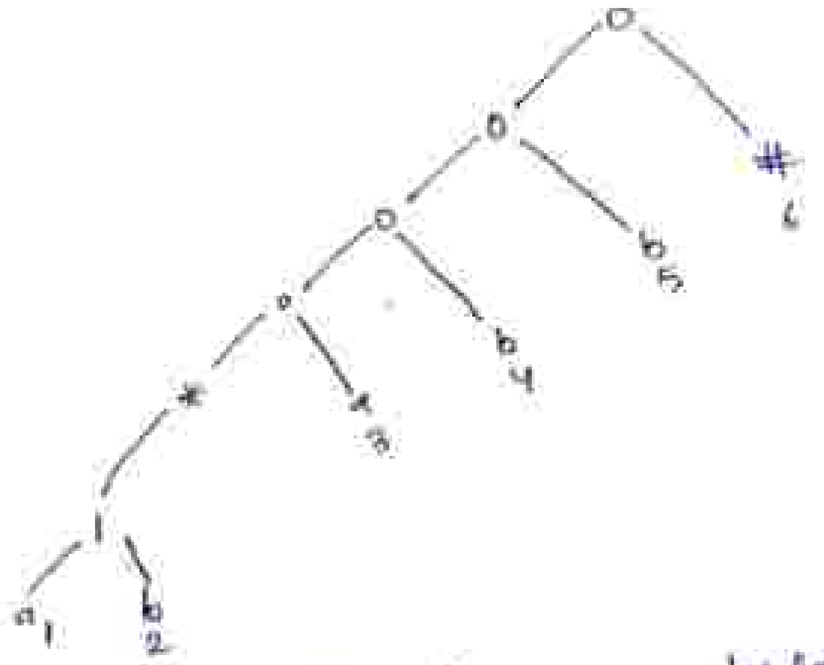
During subset construction, two sets of NFA states can be identified

1. Have the same important states, and.
2. Either both have accepting states or neither does.

Each important state corresponds to a particular operand in regular expression. The constructed NFA, has only one accepting state but does not have out-transitions. By concatenating right end marker #, to a regular expression(r) We give accepting state for r on # transition of #, making it important state of NFA for (r)#

The important states of NFA, corresponds to positions in the regular expression that holds symbols of alphabets. Regular expression can be represented by syntax Tree, where leaves denote operands, interior nodes denotes operators. Interior node are cat-node (dot), or-node (|), or star-node (*)

Following figure shows Syntax Tree for regular expression $(a|b)^*abb\#$



Leaves in syntax tree are labeled by ε or by alphabet symbol, if the leaf is not labeled by ε, we attach an integer, which refers to position of leaf / symbol in regular expression. One symbol can have many positions, for ex, a has position 1 and 3 positions corresponds to important state of NFA.

Functions Computed from the Syntax Tree

To construct DFA, directly from regular Expression, we construct Syntax Tree, and then compute four functions nullable, firstpos, lastpos and followpos, which are defined as follows.

1. nullable(n) ⇒ This function is true, for a Syntax Tree for node n, if the subexpression represented by n has ε in its language. The subexpression can be made null.

2. Firstpos(n) ⇒ is set of positions in subtree rooted at n, That corresponds to first Symbol.

3. last Symbol(n) ⇒ set of positions in the subtree rooted at n that corresponds to the last symbol of at least one string.

4. Followpos() ⇒ for a position P, there is a set of positions q in entire Syntax Tree, such that there is a String $x = a_1 a_2 \ldots a_n$ in $L((r)\#)$, where a_i is matched to position P, and a_{i+1} to position q.

Example: Consider the cat-node n, of previous Syntax Tree that corresponds to expression $(a|b)^*a$.

Nullable(n) ⇒ Nullable is false, for this node as it generates all the strings of a's and b's ending with a. On the other hand the star node, below it can be nullable, it may generate ε along with strings of a's and b's.

firstpos(n) → {1, 2, 3}, In string generated like 'aa', The first position corresponds to 1, in string 'ba', The first position corresponds to 2, And string with only 'a', first position corresponds to 3.

lastpos(n) ⇒ {3}, For all strings, generated from expression of node n, The last position is for 'a', and it is 3.

Computing nullable, firstpos, and lastpos.

We can compute nullable, firstpos and lastpos depending up on the height of tree. The following table represents the inductive rules for nullable and firstpos. The rules for lastpos are same as for firstpos.

Node (n)	Nullable (n)	firstpos (n)
A leaf node labeled ϵ	true	ϕ
A leaf with position i	false	$\{i\}$
An or-node $n = c_1 \mid c_2$	nullable (c_1) or nullable (c_2)	firstpos (c_1) $\cup$ firstpos (c_2)

A cat-node $n = c_1c_2$	nullable (c_1)
	and
	nullable (c_2)
		if (nullable (c_1))
		firstpos(c_1) $\cup$ firstpos(c_2)
		else
		firstpos(c_1)

A star Node $n = c_1*$	true	firstpos(c_1).

Example :- From all the nodes of previous syntax Tree only star-node is nullable, and non of The leaves are nullable because they corresponds to non-ε operands.

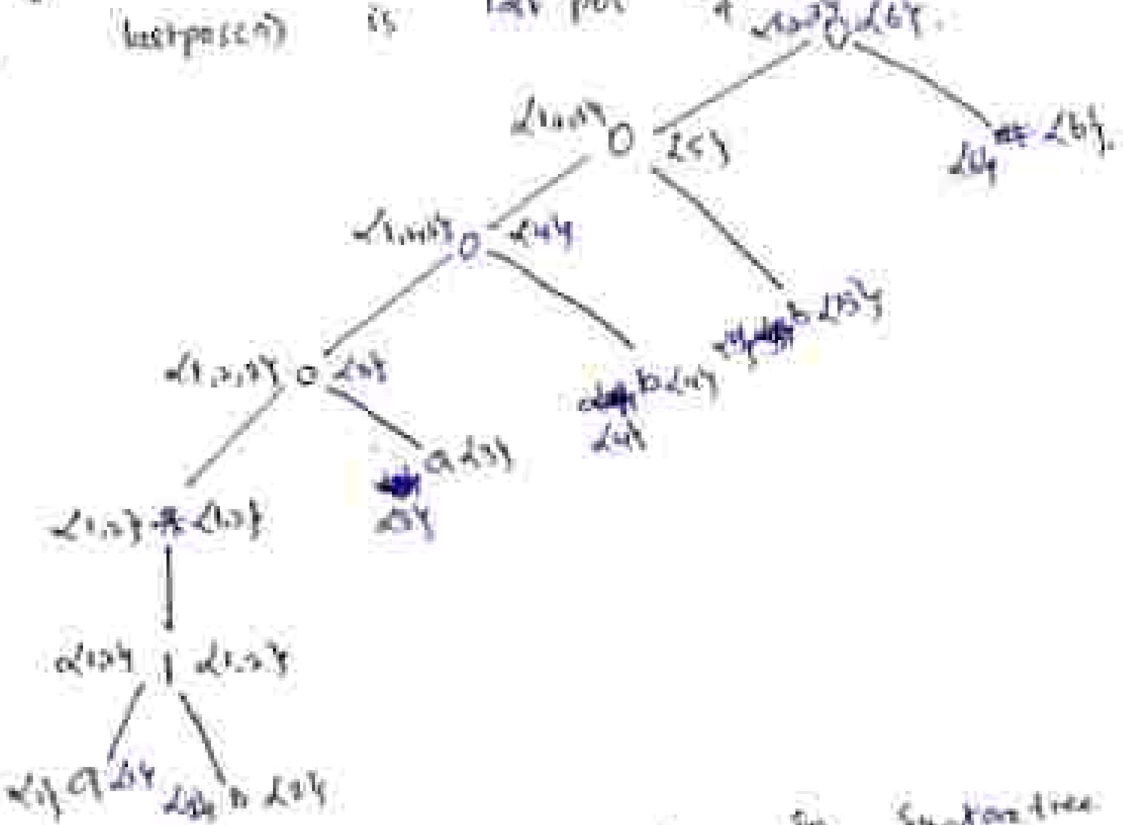
→ The 'or node' is not nullable, because neither of its children or null.

→ The star node is nullable because every star node is nullable.

→ The cat node is not nullable if atleast one child is non nullable.

The Computation of firstpos and last pos for each node is shown in below figure, with firstpos(n) to the left of node n, and lastpos(n) to its right. Each leaf has only itself for firstpos and lastpos, as the rule for non-ε leaves. For the or-node, we take the union of firstpos at the children and do the same for lastpos. The rule for the star-node says that we take value of firstpos or lastpos at any one child of that node.

For ex, consider lowest cat-node, consider it as n. It's left child is nullable, and so its first pos is {1,2}, and first pos of right child is {3}'s union will give i.e; {1,2,3} will give n's firstpos. The rule for lastpos is same like firstpos, with childs interchanged. that is to compute lastpos(n), we must check wether its right child is nullable, which is not. Thus lastpos(n) is last pos of its right child,



First pos and lastpos for nodes in Syntax tree for $(a|b)^* abb\#$.

Followpos

Follow pos can be applied to only cat node(•), and closure/star node (*). The rules for finding followpos are for a node n are:

if n is • then

First pos(c2) → last pos(c1)

i.e: copy first pos c2 to last pos of c1

if n is * then

Firstpos(n) → lastpos(n).

i.e copy firstpos of n i.e; * to last pos of *

* The final follow pos according to above two rules is shown in below table. First we find followpos for • and then for *.

Position n	followpos(n)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	ϕ.

Converting the Regular Expression to DFA.

We now put all the steps of our example to construct DFA, for regular expression $r = (a|b)^*abb$ We observed that nullable is true only for * node.

The value of firstpos for the root is $\{1,2,3\}$, so this is the start state of D, consider this state as A. i.e; $A = \{1,2,3\}$, Now we find $D_{trans}[A,a]$, and $D_{trans}[A,b]$. From the tree, Among the positions of A, 1 and 3 corresponds to a, and 2 corresponds to b

So

$$D_{trans}[A,a] = \text{followpos}(1) \cup \text{followpos}(3)$$
$$= \{1,2,3\} \cup \{4\}$$
$$= \{1,2,3,4\} \Rightarrow \text{let this state } B$$

$$D_{trans}[A,b] = \text{followpos}(2)$$
$$= \{1,2,3\} \Rightarrow \text{it is same as } A.$$

Similarly

$$D_{trans}[B,a] = \text{followpos}(1) \cup \text{follow}(3)$$
$$= \{1,2,3,4\} = B.$$

$$D_{trans}[B,b] = \text{followpos}(2) \cup \text{followpos}(4)$$
$$= \{1,2,3\} \cup \{5\}$$
$$= \{1,2,3,5\} = C.$$

$D_{tran}[C, a] = \{followpos\{1\} \cup followpos\{3\}\}$

$= \{1,2,3,4\} = B$

$D_{tran}[C, b] = followpos\{2\} \cup followpos\{4\} \cup followpos\{5\}$

$= \{1,2,3\} \cup \{5\} \cup \{6\}$

$= \{1,2,3,5,6\} = D.$

⇒ it is the final state as it has position of end marker.

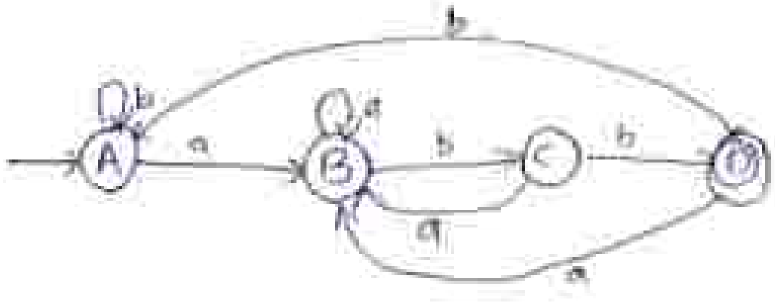
$D_{tran}[D, a] = followpos\{1\} \cup followpos\{3\}$

$= \{1,2,3,4\} = B.$

$D_{tran}[D, b] = followpos\{2\}$

$= \{1,2,3\} = A$

The DFA can be given as.

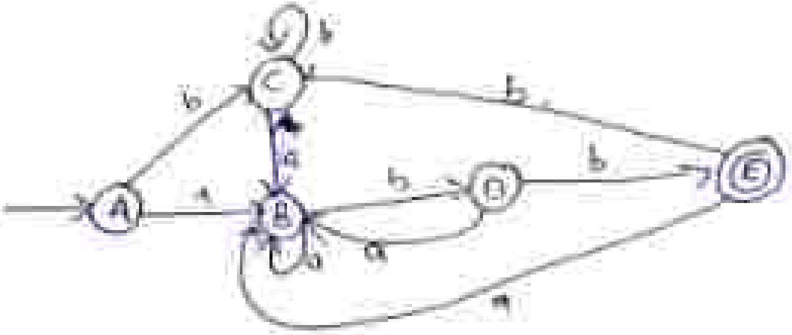


Minimizing The number of States of a DFA.

There can be many DFA's that recognize same language. If we implement a lexical analyzer using DFA, we prefer the DFA with few states. The minimization can be done as follows.

Method:-

Consider below DFA



Step1:- Seperate states in to two groups according to accepting state → {A,B,C,D}, {E}.

Step2:- Consider both groups and inputs a, b.
E can not be split, because it has only one state {E} remains in final group.

Step3:- In group {A,B,C,D}, on input a, each state goes to B. On input b, A,B,C goes to members of {A,B,C,D}, D goes to {E}, so we split {A,B,C,D}, as {A,B,C}, {D}. The final group contains ~~{A,B}~~ {D}, {E}

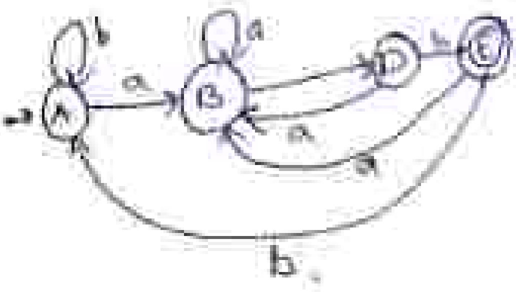
Step 4 :- Now in Group $\{A, B, C\}$, A and C all states goes to members of $\{A, B, C\}$ On input a, and on b, A and C goes to members of Group, B goes to D so we split $\{A, C\}, \{B\}$

Step 5 :- $\{A, C\}$, can't be splitted, because both goes to same state on input a and b.

The final Group consists of $\{A, C\}, \{B\}, \{D\}, \{E$

The minimum DFA, consist of four states. And we consider A as representative of $\{A, C\}$. The initial state is A, and final state is E. The Transition Table and DFA can be given as follows.

State	a	b
A	B	A
B	B	D
D	B	E
E	B	A



SYNTAX ANALYSIS:- (ROLE OF SYNTAX ANALYZER):-

→ Syntax Analyzer is also called as the parser, the parser obtains ~~the~~ stream of tokens from the lexical Analyzer

→ And, verifies the tokens syntactically, if the tokens are syntactically correct then it will generate a parse tree.

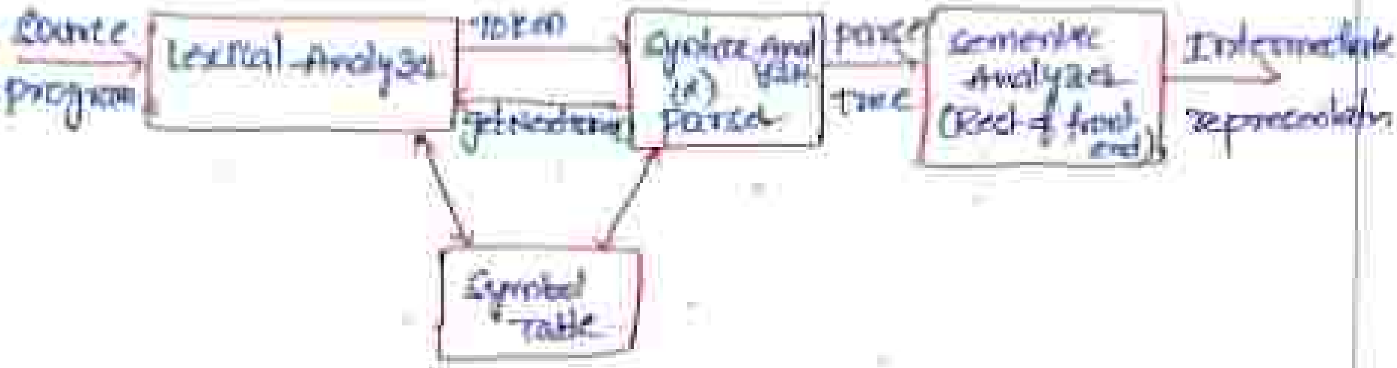

source program
Lexical Analyzer
token
get next token
Syntax Analyzer (or) Parser
parse tree
Semantic Analyzer (Rest of front end)
Intermediate representation
Symbol Table

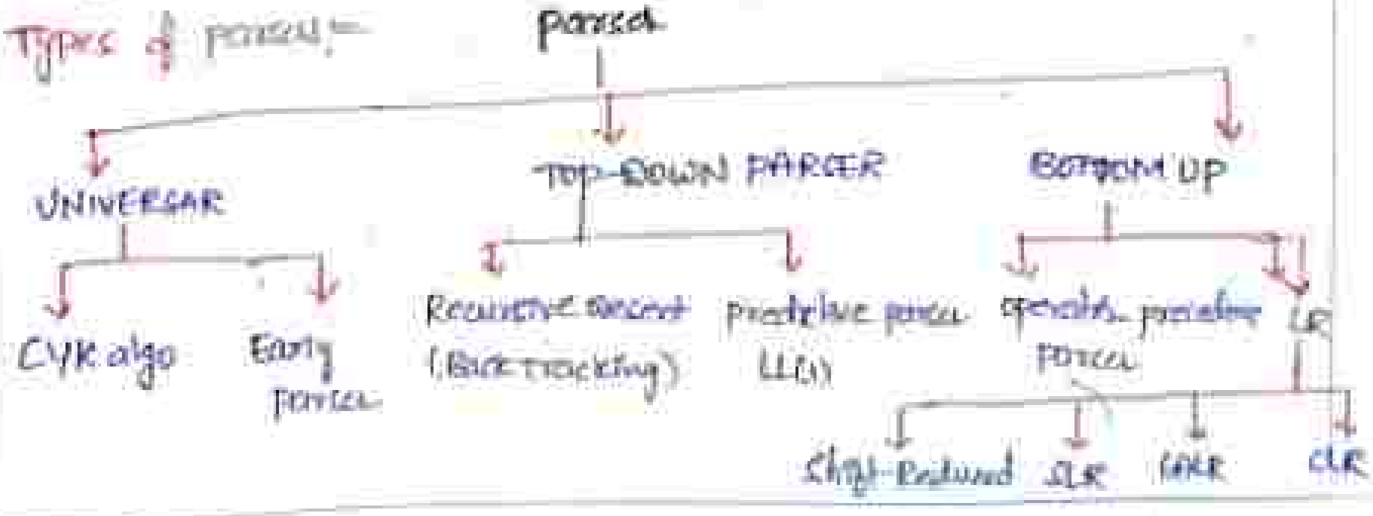
Fig:- position of parser in compiler model.

→ The main function of parser is to check the syntactic structure of i/p code, if the given i/p code is out of syntactic errors then it will generate a parse tree.

→ If i/p code is consist of syntax errors, then the ~~so~~ it reports an errors to user. Now, the user will correct it is responsibility of the user to correct those errors.

→ Symbol Table communicate with both lexical Analyzer & Syntax Analyzer for the token information.

Types of parser:-



→In top-down parser construct the parse tree from top(root) to the bottom (leaves)

→Bottom-up parser construct the parse tree from leaves to root (starts from i/p symbol and reduced to starting state)

Representative Grammars:-

Associativity and precedence are captured in the following grammar

$E \rightarrow E+T \,/\, T$
$T \rightarrow T*F \,/\, F$
$F \rightarrow (E) \,|\, id$

E → Expression
T → Terms, F → Factors that can be either parenthesized expression (or) id

These grammars belongs to LR grammars that are suitable for bottom-up parsing. This grammar cannot be used for top-down parsing

The following non-left recursive grammar will be used for top-down parsing

$E \rightarrow TE'$
$E' \rightarrow +TE' \,/\, \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \,/\, \epsilon$
$F \rightarrow (E) \,/\, id$

Syntax Error Handling:-

programming errors can occure at many different levels.

Lexical Errors:

→Include misspelling of identifiers, keywords, & operators

Eg:- identifier elipsesize instead of ellipsesize and missing quotes around text intended as a string. (Eg: "String")

Syntax Errors:- Misplaced semicolon(;), Extra & missing braces {, }.
Eg:- appearance of case statement without an enclosing switch

Semantic Errors:- Eg:- (a+(b*c) → missing parenthesis

→ Type mismatch b/w operator and operands.

Eg ① 2+a[i] → type mismatch.
② The return of a value in Java method with result type void.

logical Errors:-

→ Errors can be anything from incorrect reasoning on the part of programming.

Eg:- In "c" - the assignment operator instead of comparison operator.
if (c=a) ✗ if (c==a) ✓

Error Handler in parser:-

- should report the presence of errors clearly and accurately.
- should recovery from each error quickly enough to detect the subsequent errors
- should not slow down the processing of remaining programs

Error Recovery Strategies:-

(i) panic Mode Recovery:- Eg:- int a, b {
printf(" ");

→ The parser discards i/p symbols at a time until one of a designated set of synchronizing tokens is found.

→ The synchronizing tokens are delimeters, ";" & }

→ It is simple to implement and does not goto a loop

(ii) phrase level Recovery:-

→ The parser perform local correction on remaining i/p when the error is discovered.

→ The parser replaces the prefix of the remaining i/p by some string that allows the parser to carry on its execution

Eg:- replace a (,) comma by semicolon (;) , delete an extra semicolon, insert a missing semicolon(;)

(i) printf(" "), → printf(" ");

Disadvantage:- Error correction is difficult when actual errors occurs before the point of detection

(iii) Error production

→ common Errors that can be encountered, we can augment the grammar for the language with productions that generate erroneous constructs.

TE→

→ use a new grammar for the parser.

(iv) Global correction:-

→ The aim is to make some changes while converting incorrect i/p string to a valid string

- Given an incorrect i/p x & Grammar G find a parse tree for a related string w (using the given Grammar) such that no of changes (insertion/deletion) required to transform x to w is minimum
- Too costly to implement

CONTEXT FREE GRAMMAR:-

Context Free Grammar consists of 4-tuples

CFG=(V,T,P,S)

V → set of variables (or) Non-terminals
T → set of terminals
P → set of production Rules
S → start symbol.

(i) Non-terminal:-

→ Denotes set of strings

- uppercase letters early in alphabet A,B,C,...
- S → start symbol
- E,T,F (Expression - E, Term - T, Factor - F)

(ii) Terminal:-

→ Terminals are the basic symbols from which strings are formed

- Token-name is also called as terminal
- The following are terminal symbols.
  - ⓐ lowercase letters in the alphabets such as a,b,c.
  - ⓑ operator symbols such as +, *, ...
  - ⓒ punctuations ("{", "}", "(", ")", ",", ...) & digits 0,1...9

(iii) productions:- consists of

ⓐ A Non terminal called head or leftside of the production. This production defines some of the symbols denoted by the head

ⓑ body or right side → consisting of zero or more terminals and Non terminals

* uppercase letters late in alphabet X,Y,Z represent Grammar symbols either terminals or Non-terminals.

→ α, β, γ ... represent string of grammar symbol

→ A set of productions $A \to \alpha_1$, $A \to \alpha_2$ → can also written as $A \to \alpha_1 | \alpha_2 | \dots | \alpha_k$

Product $\overset{\text{head}}{A} \longrightarrow \alpha$

A ↓ Non-Terminal, α ↓ either terminal & Nonterminal $(V \cup T)^*$

Eg:- consider the grammar

$E \to E+E \quad E \to E*E \quad E \to I \quad I \to a$

$CFG = (\overset{V}{\{E, I\}}, \overset{T}{\{+, *, a\}}, \overset{P}{P}, \overset{S}{E})$

→ derive

Derivations:- Derivation is process of applying a sequence of production rules inorder to derive a string.

→ deriving an i/p string from the start symbol of Grammar in one or more steps by replacing the head of the production by the body of the production.

There are two types of Derivation.

(i) Left most Derivation (LMD)

(ii) Right most Derivation (RMD)

Left most Derivation:- In each step we have to expand left-most Non-terminal by one of the production body.

Eg:- $E \to E+E \,/\, E*E \,|\, -E \,|\, (E) \,|\, id$

Derive a string id+id*id

LMD:-

$E \xrightarrow{lmd} \underline{E}+E \quad (E \to E+E)$

$\underset{lm}{\Rightarrow} id+\underline{E} \quad (E \to id)$

$\underset{lm}{\Rightarrow} id+\underline{E}*E \quad (E \to E*E)$

$\underset{lm}{\Rightarrow} id+id*\underline{E} \quad (E \to id)$

$\underset{lm}{\Rightarrow} id+id*id \quad (E \to id)$

Right most Derivation:-

→ here, we have to expand the Right-most non-terminal by

RMD:-

$E \underset{rm}{\Rightarrow} E+\underline{E} \quad (E \to E+E)$

$\underset{rm}{\Rightarrow} E+E*\underline{E} \quad (E \to E*E)$

$\underset{rm}{\Rightarrow} E+\underline{E}*id \quad (E \to id)$

$\Rightarrow \underline{E}+id*id \quad (E \to id)$

$\Rightarrow id+id*id \quad (E \to id)$

Parse trees:

→ A Graphical representation of derivation is called parse tree.

There types of nodes in parse tree

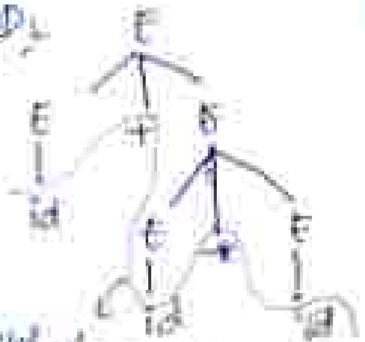
(i) Interior node → are Non terminals

(ii) children node → terminals

→ In parse tree the root node must be start symbol

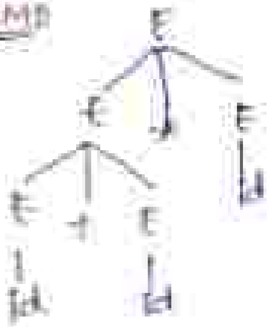
construct parse tree for i/p string w= id+id*id

LMD:



Yield of parse tree

RMD



Ambiguity:

→ If a grammar (CFG G=(V,T,P,S)) produces more than one parse tree, then that grammar is called as ambiguous.

(i) more than one LMD (different)

(ii) more than one RMD (different).

Eg $E \rightarrow E+E \mid E*E \mid (E) \mid id$ string $w = id+id*id$

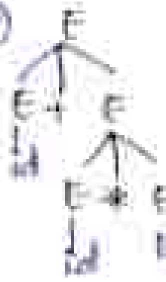
LMD1: $E \xRightarrow[lm]{} E+E \xRightarrow[lm]{} id+E \xRightarrow[lm]{} id+E*E \xRightarrow[lm]{} id+id*E \xRightarrow[lm]{} id+id*id$

RMD1: $E \xRightarrow[rm]{} E+E \xRightarrow[rm]{} E+E*E \xRightarrow[rm]{} E+E*id \xRightarrow[rm]{} E+id*id \xRightarrow[rm]{} id+id*id$

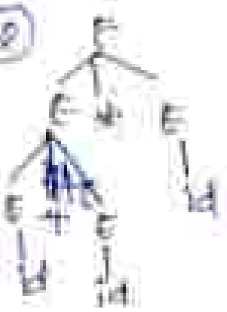
LMD2: $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$

RMD2: $E \xRightarrow[rm]{} E*E \xRightarrow[rm]{} E*id \xRightarrow[rm]{} E+E*id \xRightarrow[rm]{} E+id*id \xRightarrow[rm]{} id+id*id$

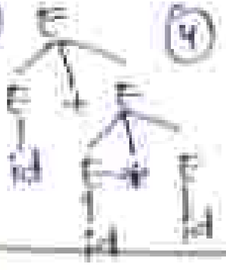
①



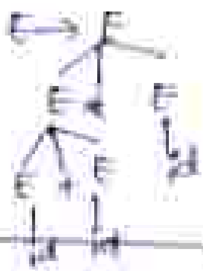
②



③



④



→ For the above string we get 2 LMD & RMD and 2 parse trees so, the given grammar is ambiguous

→ Top down parser can't handle ambiguous grammar, so we need to convert this grammar into unambiguous grammar.

*While converting ambiguous to unambiguous:-

→ The grammar should follow Associativity and precedence rules

*, /, +, − → left Associate (when an ~~operator~~ operand has ~~operand~~ operator on both sides, the operand should associate with left side operator.)

/, *, +, − → precedence order
1 2 3 4

② WRITING A GRAMMAR:-

→ Grammar is used to describe the syntax of programming language.

Lexical versus Syntactic Analysis:-

- Seperating syntactic structure into smaller and manageable components of lexical and non-lexical parts.
- lexical rules are simple to understand than grammar.
- We need notations to describe the grammar.
- RE are used to Realtime Examples to make them easy to understand.
- Grammars are used for describing nested structure such as balanced parenthesis, matching begin-end, if-then-else.

Eliminating Ambiguity:-

Properties of CFG

1. Ambiguous
2. Left Recursive
3. Left Factoring.

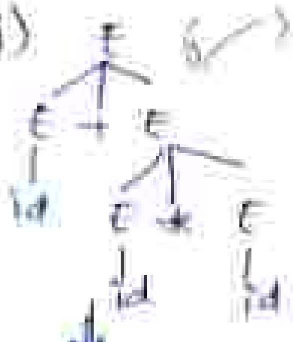
Elimination of ambiguity:-

Consider CFG:-

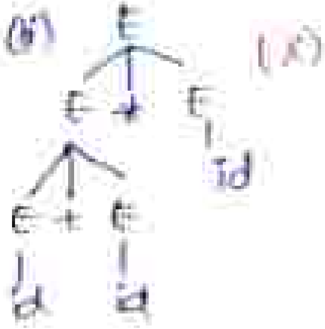
Eg: E → E+E | E*E | (E) | id

w = id+id*id (There are 2 parse trees for string)

(i)



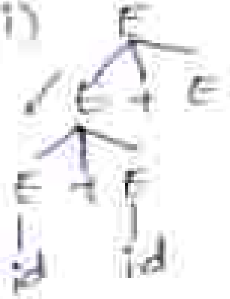
(ii)



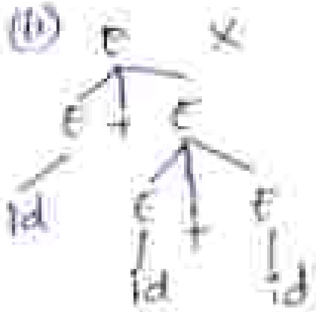
↳ This is the valid parse tree, because the "*" operator is appeared at bottom of parse, so it is evaluated first

Eg:- w=id+id+id

(i)



(ii)



→ two factors that are needed to be taken

(1) precedence

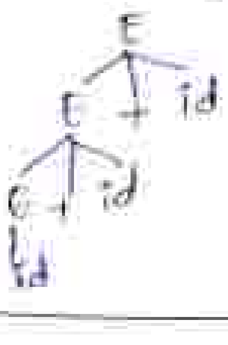
(2) left associativity → To ensure left associativity, we need to convert the grammar into left recursive

→ left recursive E → E+E (the leftmost symbol of RHS is equal to the LHS)

- In left recursive, the parse tree can be grown on left side only

E → E+id | id   id+id+id ⇒ (id+id)+id (left associative

left associativity can be achieved



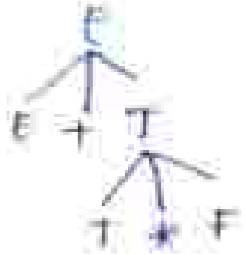
(id+id)+id

↓

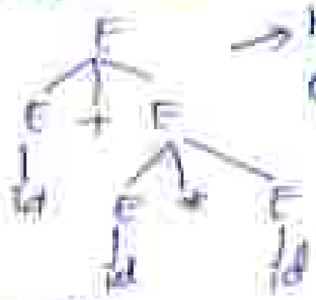
This expression is evaluated first, so that it is valid parse tree.

$E \rightarrow E+T$ (precedence)

$T \rightarrow T*F$



id+(id*id)



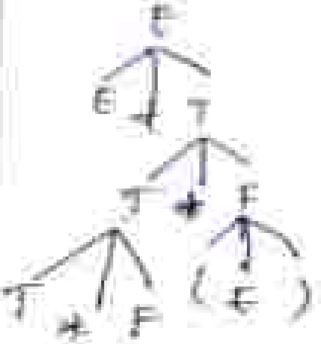
→ highest precedence operator is executing first

The Final ~~gen~~ unambiguous Grammar

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$



Dangling else :- In a program if there are more than one if-stmts then else part is matched with wrong if stmt, that will lead to wrong result, that is called dangling else.

Eg
```
if (expr1) — P1 → T
    if (expr2)    → T
        S1
P2 → else
        S2

other stmt
```

(i) if P1 is True Then it will goto P2 and
- check the condt 2nd condition, if it also True then it will execute S_1
  - if 2nd condition is false it will execute else part i.e S_2

(ii) if P1 → is false it has to execute the other statement

(iii) But here if P1 - False then it is executing else part i.e statement it is matching with wrong else.

- Consider the following dangling else grammar

stmt → if Expr then stmt

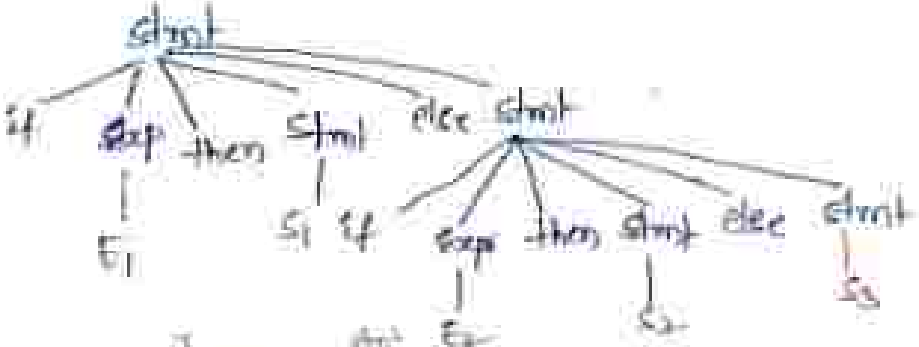
| if expr then stmt else stmt

| other

- The compound conditional stmt for the above grammar is

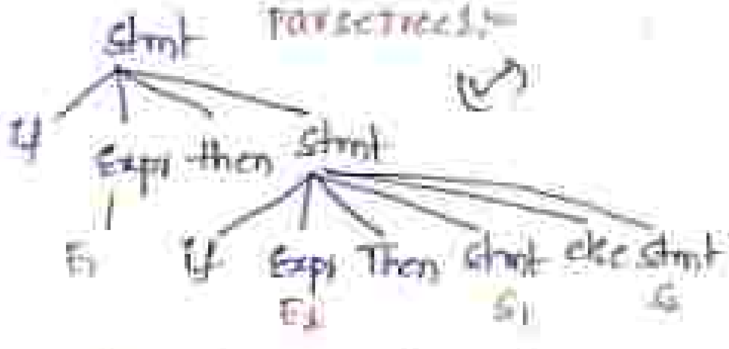
if E₁ then S₁ [else if E₂ then S₂ else S₃]

- The parse for this stmt

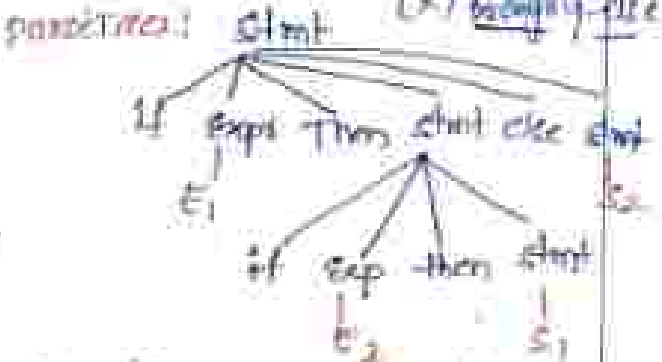


o/p string = If E₁ then [if E₂ then S₁ else S₂]

there exist two parse trees for the given string.


parsetree1:-

(✓)


parsetree2:

(×) dangling else

→ In these two parse trees, in programming language, the 1st parse tree is considered because, match the else statement with closest then. open here [If E₁ then [if E₂ then S₁ else S₂] → matched stmt

→ So, The given Grammar is ambiguous Grammar.

The unambiguous grammar will be

stmt → matched stmt | open stmt

matched stmt → if expr then matched stmt else matched stmt | other

open stmt → if expr then stmt

| if expr then matched stmt else open stmt.

matched stmt → perfect stmt
open stmt → simple if

(ii) Elimination of left Recursion:-

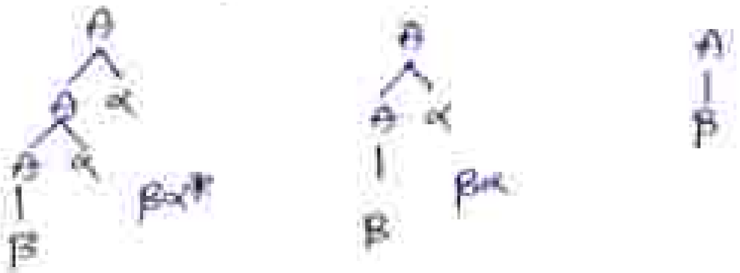
→ A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

→ Top-down parsing methods cannot handle this grammar, so we need to eliminate left recursion.

$A \overset{+}{\Rightarrow} A\alpha | \beta$ (left recursive) $\alpha, \beta \in (V \cup T)^*$

[if LHS is equal to the leftmost non-terminal of RHS]

The string derived from the above grammar.



$\text{string}(\omega) = \beta\alpha^*$
(any production that contains * symbol)

production $A \rightarrow A\alpha | \beta$ could be replaced by the following non-left-recursive productions.

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' | \varepsilon$

Eg:- $E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow (E)/id$ (left recursive grammar)

So, we need to eliminate left-recursion from the above grammar.

(1) $\underbrace{E}_{A} \rightarrow \underbrace{E}_{A}\underbrace{+T}_{\alpha}/\underbrace{T}_{\beta}$

$A = E$
$\alpha = +T$
$\beta = T$

$E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$

(ii) $T \to T*F \mid F$

$\underset{A}{} \quad A\alpha \mid \beta$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

(iii) $F \to (E) \mid id$ (Non left recursive)

After eliminating the left recursion from the grammar

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id \mid \varepsilon$

Eg 2:- $\underset{A}{S} \to \underline{\underset{A}{S}\,\underset{\alpha}{0S1S}} \mid \underset{\beta}{\underline{01}}$

$A \to A\alpha \mid \beta$

$S \to 01S'$

$S' \to 0S1SS' \mid \varepsilon$

Eg 3:- $S \to (L) \mid x$. No left recursion

$L \to L,S \mid S$

* Elimination of left recursion for multiple production.

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2 \cdots$

$$\boxed{\begin{array}{l} A \to \beta_1 A' \mid \beta_2 A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon \end{array}}$$

Eg:- $\underset{A}{Expr} \to \underset{A}{Expr}\underset{\alpha_1}{\underline{+Expr}} \mid \underline{\underset{A}{Expr}\underset{\alpha_2}{*Expr}} \mid \underset{\beta_1}{id}$

$Expr \to id\,Expr'$

$Expr' \to +Expr\,Expr' \mid *Expr\,Expr' \mid \varepsilon$

Eg 2:- $\underset{A}{S} \to \underset{A\alpha_1}{Sx} \mid \underset{A\alpha_2}{SSb} \mid \underset{\beta_1}{xS} \mid \underset{\beta_2}{a}$

$S \to xSS' \mid aS'$

$S' \to xS' \mid SbS' \mid \varepsilon$

Eg 3:- $S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$

$S \to Aa \mid b$ → No left recursion

$A \to Ac \mid Sd \mid f$

(1) Elimination of left factoring:-

→ A grammar contains production rule in the form

$$\boxed{A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \gamma_1 \mid \gamma_2}$$

Then that grammar contains left factoring.

→ TDP parsers can't handle left factoring the grammar contains left factoring.

→ we can eliminate the left factoring by replacing with the following production.

$$\boxed{\begin{array}{l} A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \\ A' \to \beta_1 \mid \beta_2 \end{array}}$$

Eg1:- $S \to \underbrace{iEtS}_{\alpha} \mid \underbrace{iEtS}_{\alpha}\underbrace{eS}_{\beta} \mid \underbrace{a}_{\gamma_1}$ (it contains left factoring)

$E \to b$

Here, ~~as~~ $A = S$, $\alpha = iEtS$ $\beta_1 = \epsilon$ $\beta_2 = eS$ $\gamma_1 = a$

$S \to iEtSS' \mid a$

$S' \to \epsilon \mid eS$

Eg2:- $\underset{A}{B} \to \underbrace{b}_{\alpha}\underbrace{B}_{\beta} / \underbrace{b}_{\beta_2}$

$B \to bB'$

$B' \to B \mid \epsilon$

Eg3:- $A \to \underbrace{a}_{\alpha}\underbrace{AB}_{\beta_1} \mid \underbrace{a}_{\alpha}A \mid \underbrace{a}_{\alpha}$

$B \to bB/b$

$A \to \underbrace{a}_{\alpha}\underbrace{AB}_{\beta_1} \mid \underbrace{a}_{\alpha}\underbrace{B}_{\beta_2} \mid \underbrace{a}_{\alpha}\underbrace{}_{\beta_3}$

$A \to aA'$

$A' \to AB \mid A \mid \epsilon$

Eg4:- $\underset{A}{X} \to \underbrace{X}_{\alpha}\underbrace{+X}_{\beta_1} \mid \underbrace{X}_{\alpha}\underbrace{*X}_{\beta_2} \mid \underbrace{D}_{\gamma_1}$

$X \to XX' \mid D$

$X' \to +X \mid *X$

Eg5:- $E \to T+E / T$

$T \to int / int*T / (E)$

$\underset{A}{E} \to \underbrace{T}_{\alpha}\underbrace{+E}_{\beta_1} / \underbrace{T}_{\alpha}\ \beta_2 = \epsilon$

$E \to TE'$

$E' \to +E \mid \epsilon$

$T \to \underbrace{int}_{\alpha}\ \underbrace{(}_{\beta_1}\ \underbrace{int}_{\alpha}\underbrace{*T}_{\beta_2} / \underbrace{(E)}_{\gamma_1}$

$T \to int\,T' \mid (E)$

$T' \to \epsilon \mid *T'$

Top down parsing:-

→ In Top-down parsing, The parse tree is constructed from root node to child node.

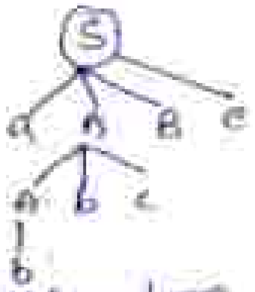
→ Top-down parser uses left-most derivation to derive the input string from the grammar.

→ TDP is starting from start symbol of grammar and reaching the i/p string

Eg:- Grammar is, derive the i/p string w=abbcde.

S → aABe
A → Abc/b
B → d

LMD:- S → aABe
aAbcBe
abbcBe
abbcde



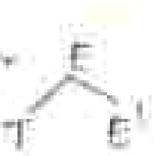
→ TDP constructed for the grammar if it is free from
- ambiguity
- left recursion.

→ The problem with TDP Parser is if we have more than one alternative for production, which alternative we should use. [E→E+T/T, T→T*F/F
F→(E)/id]
→ ambiguous, left recursion

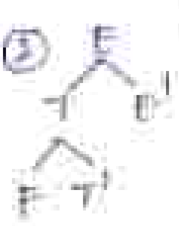
Eg:- E→TE' → unambiguous grammar
E'→+TE'
T→FT'
T'→*FT'/ε
F→(E)/id

→ Derive an i/p string w = id+id*id

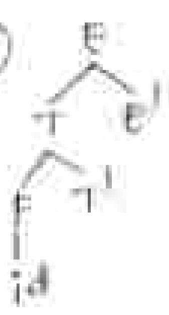
step 1:-



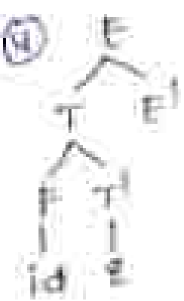
②



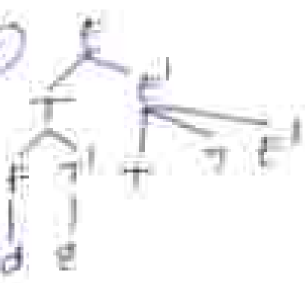
③



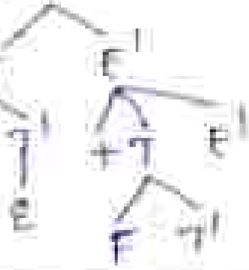
④



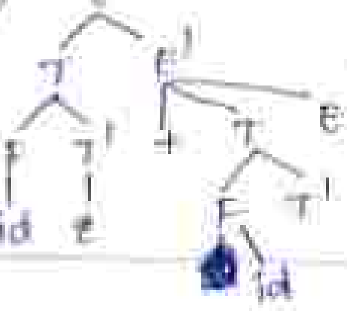
⑤



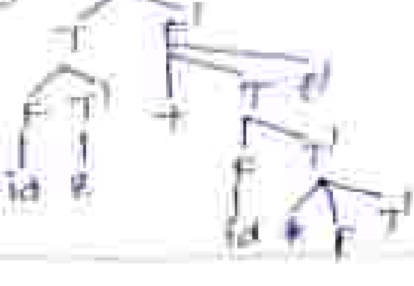
⑥



⑦



⑧



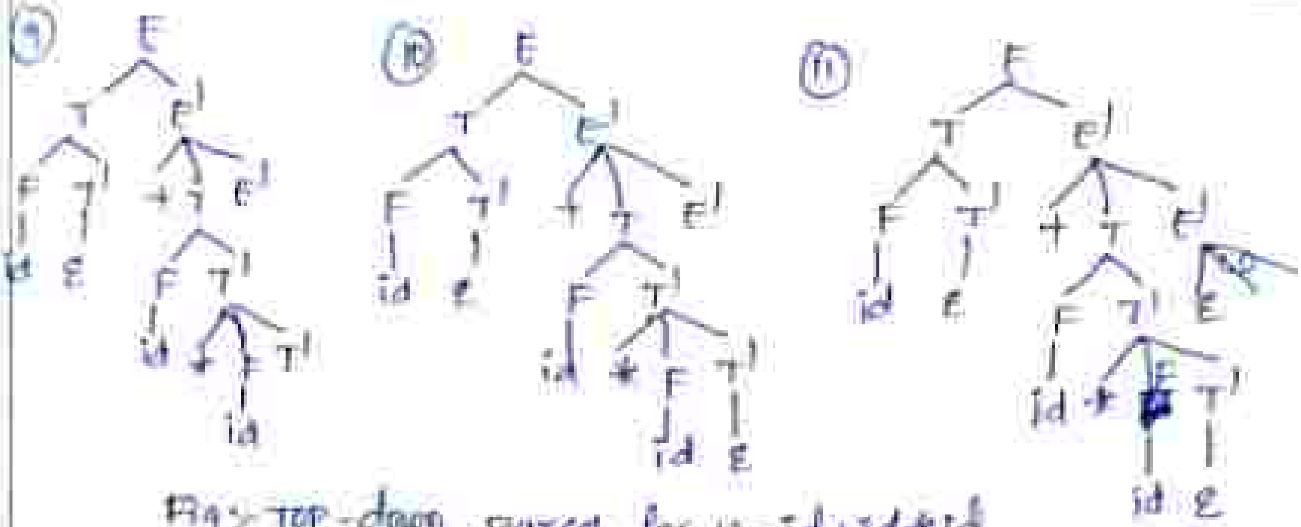


Fig:- top-down parse for w=id+id*id

(1) Recursive-Descent Parsing:= (with back tracking)

→ The construction of parse tree starts from root & proceed to children

→ Recursion:- A function which is called by itself.

Steps for constructing Recursive Descent parser:=

(i) If the i/p is a non-terminal, then call corresponding function

(ii) If the i/p is terminal, then compare terminal with i/p symbol. If both are same, the increment input pointer

(iii) If a Non-terminal contains more than one production then all the production code should be written in the corresponding function.

Eg:- $E \rightarrow iE'$ $E' \rightarrow +iE' \mid \varepsilon$

(In the given grammar we have 2-Nonterminals E, E'. So define two functions, same as C-lang functions)

```
E( )
{
 if (input=='i')
 input++;
 EPRIME( );
}
```

```
EPRIME( );
{
    if (input=='+')
    {
      ~~if (inp~~ input++;
    if (input==';')
     input++;
     EPRIM ( );
    }
  else
  return;
}
```

I/p string: (+;)

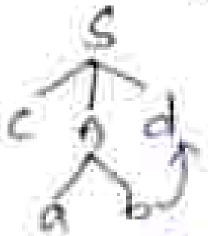
I/p string

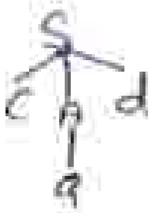
Ex2:- Grammar $E \to TE'$ $E' \to +TE' | \epsilon$
$T \to FT'$ $T' \to *FT' | \epsilon$ $F \to (E) | id$

```
E( )
{
 T( );
 EPRIME( )
}
EPRIME( );
{
 if (input=='+')
 {input++;
  T( );
  EPRIME( );
 }else
  return;
}
```

```
T( )
{
  F( );
  TPRIME( );
}
TPRIME( )
 {
  if (input=='*')
  {
   input++
   F( );
   TPRIME( );
  }
  else
   return;
 }
F( )
{
 if (input=='(')
 {input++;
  E( );
 if (input==')')
   input++;
 }
 else if (input=='id')
  input++;
}
```

Ex3:- $S \to cAd$
$A \to ab|a$
w= i/p string w=cad





(If None of the product for A satisfying the condition then we should check an another possibility for 'S')

```
input++
Eprime();
}
Eprime()
{
 if (input == '+')
  { input++;
  if (input=='i')
   input++;
   Eprime();
  }
  else
   return;
 }
```

i/p: id i+i$

Eg:- i/p = id+id

i/p = id+id.

E→TE'

E'→+TE'|ε

T→FT'

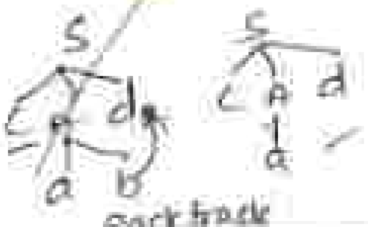
T'→*FT'|ε

F→(E)|id

```
E()
{
 T()
 Eprime();
}
Eprime()
{
 if(input=='+')
  input++
  T()
  EPRIME();
 }
 else
  return;
}
T()
{
  F()
  TPRIME();
}
TPRIME()
{
 if(input=='*')
  {input++;
  F()
  TPRIME();
  }
  else
   return;
}
```

```
F()
{
 if(input=='(')
 { input++;
   E();
 if(input==')')
   input++;
 }
 else if
 (input=='id')
   input++;
 }
```

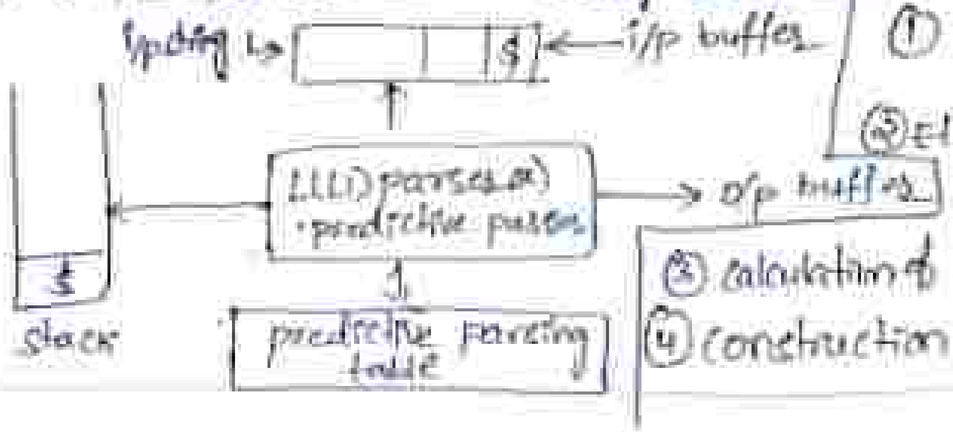
Eg S→cAd
A→ab|a   w=cad



(∵ If Non of the product for A is satisfying the condition then we should check another possibility for S)

(ii) predictive parsing: (or) LL(1) & Non recursive parser:-

LL(1) – means
- It scans the input from left to right by using leftmost derivation.
- LL(1) → 1 – no look ahead – at a time we are reading 1 character at a time



Steps for constructing LL(1) parser:-
① Elimination of left recursion
② Elimination of left factoring
③ calculation of FIRST() & Follow
④ construction of parsing table.

LL(1) parser → parsing algorithm

LL(1) parsing table – Data structure which is constructed from the LL(1) grammar

Stack :- Data structure used to store the grammar symbols

- Always the bottom of the stack is $

→ The end of i/p buffer is $

→ after $, the starting symbol of i/p is pushed to the stack

Inorder to construct the parsing table we should compute two functions ① FIRST() ② FOLLOW()

FIRST(X) :-

- if X is a terminal then FIRST(X) = {X}
- if X is a Non terminal & $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$ is a production for X.

FIRST(X) = FIRST(Y_1) &

if FIRST(Y_1) contains ε add FIRST(Y_2) to FIRST(X)

if all FIRST($Y_1 Y_2 \ldots Y_n$) contains ε then add ε to FIRST(X)

FOLLOW(B) :-

→ Always the follow() of starting symbol is $.

→ if $A \rightarrow \alpha B \beta$

Follow(B) = FIRST(β) &

if FIRST(β) contain ε or $A \rightarrow \alpha B$ then add Follow(A) to Follow(B)

Eg :- Given Grammar is

$E \rightarrow E+T / T$ – left recursion

$T \rightarrow T*F / F$ – left recursion

$F \rightarrow (E) / id$

So Eliminate left recursion ~~& left factoring~~ from these 2 productions

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

- FIRST (E) = FIRST (T) = FIRST (F) = { (, id)
- FIRST (E') = { +, ε }
- FIRST (T) = FIRST (F) = { (, id }
- FIRST (T') = { *, ε } • FIRST (F) = { (, id }

→ <u>Follow</u> (E) = ~~Eo~~ { $, Foll

$F \rightarrow (E) / id$

↓

Follow(E) = { $, FIRST ( )) } = { $,) }

~~E'~~ <u>Follow (E')</u> = { Follow (E) + Follow (E') }

$E \rightarrow TE'$ = { $,) }

$E' \rightarrow +TE' / \epsilon$

<u>Follow (T)</u> = ~~E→TE'~~ ~~Follow (E')~~ +

$E \rightarrow TE'$ Follow (T) = FIRST (E') + Follow (E')

= { +, $,) }

$E' \rightarrow +TE' / \epsilon$

<u>Follow (T')</u> ⇒ [Follow (T) + Follow (T')]

~~T~~ $T \rightarrow FT'$ ⇒ { +, $,) }

$T' \rightarrow *FT' / \epsilon$

<u>Follow (F)</u> = Follow (T') + FIRST (T') + Follow (T')

$T \rightarrow FT'$ = { *, +, $,) }

$T' \rightarrow *FT' / \epsilon$

(LL(1))
predictive parsing table :-

	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)		

w= id+id

Stack	Input	Action
E$	id+id$	E→TE'
TE'$	id+id$	T→FT'
FT'E'$	id+id$	F→id
idT'E'$	id+id$	
T'E'$	+id$	T'→ε
εE'$	+id$	
E'$	+id$	E'→+TE'
+TE'$	+id$	T→FT'
FT'E'$	id$	F→id
idT'E'$	id$	T'→ε
εE'$	$	E'→ε
$	$	accepted

The i/p: is parsed

Eg2:- S→(L)/a
L→L,S/S
w= (a)

Eg3:- S→aAB|bA|ε
A→aAb|ε
B→bB|ε
w= 'aabb'

S→iEtS|iEtSeS|a
E→b (Non LL(1))

Eg4: S→ACB|CbB|Ba
A→da|BC
B→g|ε
C→h|ε

Bottom-up parser:-

→ Bottom-up parser construct parse tree from child (bottom) node to root node (top).

→ Bottom up parser use RMD in Reverse order.

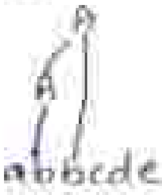
→ In Bottom-up parser the ~~pr~~ reduce the i/p string w to start symbol of grammar.

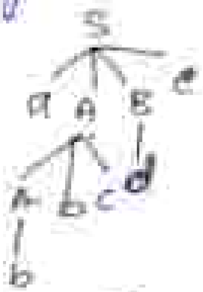
Eg:- $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input string w= abbcde





aAbcde (A→b)

aAde (A→Abc)

aABe (B→d)

S (S→aABe)

S→aABe

→aAde

→aAbcde

→abbcde

Eg2:- $E \rightarrow E+T \mid T$

$E \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

w= id*id

id*id (F→id)

F*id (T→id)

T*id (~~T→E~~)

~~E*id~~ T*id (F→id)

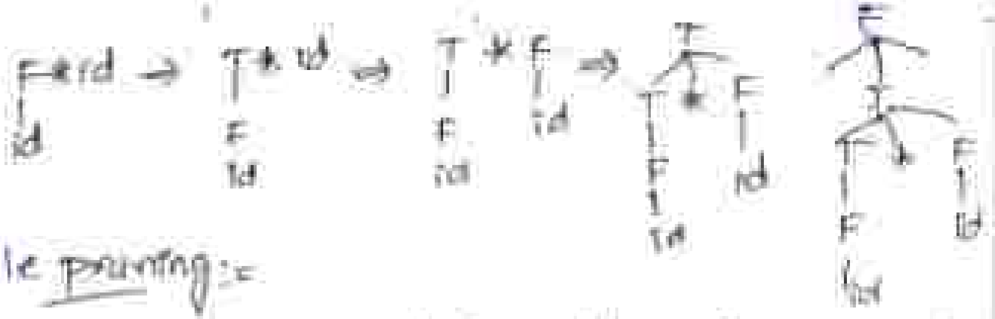
T*F (T→T*F)

T (E→T)

E

→ The problem with Bottom-up parser is when to reduce the ~~gr~~ to production.

→ Step: id*id → F*id → T*id → T*F ⇒



Handle & Handle pruning:-

Handle: Handle is a substring, which matches with right side of production ~~if~~

→ if Handle matches with the right side of production, then it is replaced with left hand side Non-terminal

Eg1:- $S \rightarrow aABe$ $w = abbcde$
$A \rightarrow Abc / b$
$B \rightarrow d$

Right sentential form	Handle	Reducing production
a<u>b</u>bcde	b	$A \rightarrow b$
a<u>Abc</u>de	Abc	$A \rightarrow Abc$
aAde	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$
S		

Handles during parse of $id_1 * id_2$

Eg2:- $E \rightarrow E+T / T$ $w = id_1 * id_2$
$T \rightarrow T*F / F$
$F \rightarrow (E) / id$

Right sentential form	Handle	Reducing production
<u>id_1</u>$* id$	id	$F \rightarrow id$
$F * id$	F	$T \rightarrow F$
$T * id$	id	$F \rightarrow id$
$T * F$	$T*F$	$T \rightarrow T*F$
T	T	$E \rightarrow T$

→ Handle pruning is obtained by Right most Derivation in reverse order.

(i) Shift-Reduce parser :-

→ Shift-Reduce parser use two Data structures ① Stack ② I/p buffer

① Stack - is used to store the grammar symbol

② Input buffer - holds the I/p string to be parsed

Stack	Input string
\$	w\$

Actions of shift-reduce parser:

1. shift — shift the next i/p symbol onto the top of the stack.
2. Reduce — If the top of stack is matched with right side of production, then it is reduced to corresponding nonterm
3. Accept — successful completion of parsing
4. Error — Discovers a syntax error and call error-recovery methods

Eg:- $E \to E+T / T$ $\quad w = id * id$
$T \to T*F / F$
$F \to (E) | id$

Fig:- configuration of shift-Reduce parser on i/p id*id

Stack	Input Buffer	Action
\$	id*id \$	shift
\$id	*id\$	Reduce by F→id
\$F	*id\$	Reduce T→F
\$T	*id\$	shift
\$T*	id\$	shift
\$T*id	\$	Reduce F→id
\$T*F	\$	Reduce T→T*F
\$T	\$	Reduce E→T
\$E	\$	Accepted

conflicts During shift-Reduce parsing

① shift-Reduce conflicts: cannot decide whether to shift or to reduce

② reduce-reduce conflict: if the two productions have same handle (searching).

Shift Reduce

Eg:- S→(L)/a
L→L,S/S   Parse the string (a,(a,a)) using SR-Parser.

Stack	I/p Buffer	parsing Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce S→a
$(S	,(a,a))$	L→S Reduce
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce S→a
$(L,(S	,a))$	~~Shift~~ Reduce L→S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce S→a
$(L,(L,S	))$	Reduce L→L,S
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce S→(L)
$(L,S	)$	Reduce L→L,S
$(L	)$	Shift
$(L)	$	Reduce S→(L)
$S	$	Accept

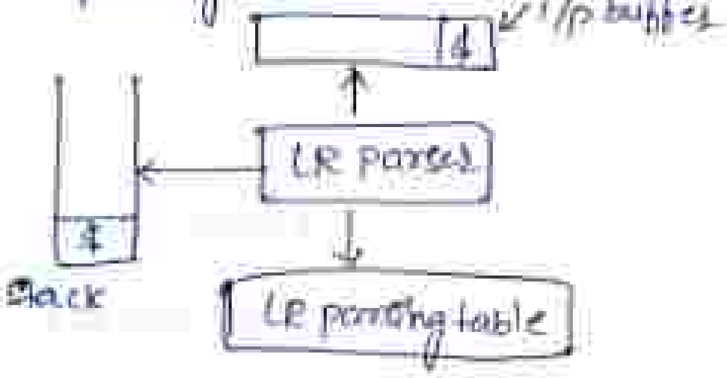
Eg:- S→TL;   ω= int id id ④ S→0S0/1S1/2
T→ int/float
L→ L,id/id   ω= 10201

Introduction to LR parsing:- LR parser is a type of Bottom-up parser.

→ LR(k) parser scans i/p from left to Right order.

→ It uses Right most Derivation in Reverse ~~order to~~ ~~derive the~~ to reduce i/p string to start symbol of grammar.

K → No of lookahead symbols that are used to make parsing decision



Stack – Datastructure used to store grammar symbols
i/p – " " " " " i/p string to be parsed
LR parser → uses algorithm to make decisions.
LR parsing table – is constructed by using LR(0) items
– These table uses two functions(s)
(i) closure (ii) Action.

Benefits of LR(k) parsing:-

→ most generic Non-Backtracking shift Reduced parsing Technique

→ These parser can recognize all programming languages for which CFG can be written.

→ They are capable of detecting syntactic errors as soon as possible while scanning of i/p.

Simple LR parsing:

Eg:-
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

① Augmented Grammar
~~$E' \rightarrow \cdot E$~~ $E' \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)/id$

Steps to construct SLR parser
① Introduce augmented grammar
② calculate cannonical collection of LR(0) items.
③ construct SLR parsing table by using (i) Goto (ii) Action function

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Goto $(I_0, E) - I_1$
$E' \rightarrow E \cdot$
$E \rightarrow E \cdot +T$

$(I_0, T) - I_2$
$E \rightarrow T \cdot$
$T \rightarrow T \cdot *F$

$(I_0, F) - I_3$
$T \rightarrow F \cdot$

$(I_0, () - I_4$
$F \rightarrow (\cdot E)$ $T \rightarrow \cdot T*F$
$E \rightarrow \cdot E+T$ $T \rightarrow \cdot F$
$E \rightarrow \cdot T$ $F \rightarrow \cdot (E)$ $F \rightarrow \cdot id$

$(I_0, id) - I_5$
$F \rightarrow id \cdot$

$(I_1, +) - I_6$
$E \rightarrow E+\cdot T$
$T \rightarrow \cdot F$
$T \rightarrow \cdot T*F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$(I_2, *) - I_7$
$T \rightarrow T*\cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$(I_4, E) - I_8$
$F \rightarrow (E \cdot)$
$E \rightarrow E \cdot +T$

$(I_4, T) - I_2$
$E \rightarrow T \cdot$
$E \rightarrow E \cdot +T$
$T \rightarrow T \cdot *F$

$(I_4, F) - I_3$
$T \rightarrow F \cdot$

$(I_4, () - I_4$
$F \rightarrow (\cdot E)$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$(I_4, id) - I_5$
$F \rightarrow id \cdot$

$(I_6, T) - I_9$
$E \rightarrow E+T \cdot$
$T \rightarrow T \cdot *F$

$(I_6, F) - I_3$
$T \rightarrow F \cdot$

$(I_6, () - I_4$
$F \rightarrow (\cdot E)$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$(I_6, id) - I_5$
$F \rightarrow id \cdot$

$(I_7, F) - I_{10}$
$T \rightarrow T*F \cdot$

$(I_7, () - I_4$
$F \rightarrow (\cdot E)$
$E \rightarrow \cdot E+T$
$T \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (F)$
$F \rightarrow \cdot id$

$(I_7, id) - I_5$
$F \rightarrow id \cdot$

$(I_8,)) - I_{11}$
$F \rightarrow (E) \cdot$

$(I_8, +) - I_6$
$E \rightarrow E+\cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$(I_7, *)$ ⑩

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5	~~S5~~	r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

1 $I_1 : E' \to E \cdot$

2 $I_2 : E \to T \cdot$

3 $I_3 : T \to F \cdot$ (r4)

4 $I_5 : F \to id \cdot$

5 $I_9 : E \to E + T \cdot$

$I_{10} : T \to T * F \cdot$

$I_{11} : F \to (E) \cdot$

Follow(E) =

$E' \to \cdot E$

1 $E \to E + T$

2 $E \to T$

3 $T \to T * F$

4 $T \to F$ (r5)

5 $F \to (E)$

6 $F \to id$

Follow(E') = \$

Follow(E) = {+,), \$}

Follow(T) = {\$, +,), *}

Follow(F) = {\$, +,), *}

$I_0 : S' \to S$

w= id *id+id

STACK	INPUT	ACTION
$0	id*id+id$	shift5
$0id5	*id+id$	reduce6 F→id
$0F3	*id+id$	reduce4 T→F
$0T2	*id+id$	s7
$0T2*7	id+id$	s5
$0T2*7id5	+id$	r6 F→id
$0T2*7F10	+id$	r5 T→T*F
$0T2	+id$	r2 E→T
$0E1	+id$	s6
$0E1+6	id$	s5 ~~[illegible]~~)
$0E1+6id5	$	r6 F→id
$0E1+6F3	$	r4 T→F
$0E1+6T9	$	r1 E→E+T
$0E1	$	acceptance

Eg-2 $S \rightarrow As/b$

$A \rightarrow sA/a$ w='abab'

→shift→ while performing shifting first we have to shift i/p symbol on to the top of the stack & then shift the state number.

→ reduce → while performing reduce operation (If F→id we reducing) if Right hand side contains one symbol we have to popped two symbols from top of the stack.

(ii) LCP LR(0) parser :-

steps for construct LR(0) parser

(1) Introduce Augumented grammar

(2) calculate canonical collection of LR(0) items.

(3) construct LR(0) parsing table by using (i) Goto (ii) action functions.

w - string - aabb$

Eg:- $S \to AA$
$A \to aA/b$

Augumented grammar
$S' \to S$
$S \to AA$

Closure:-
LR(0) items - I_0
$S' \to .S$
$S \to .AA$
$A \to .aA/.b$

Goto
(I_0, S) - I_1
$S' \to S.$

Goto (I_0, A) - I_2
$S \to A.A$
$A \to .aA/.b$

Goto (I_0, a) - I_3
$A \to a.A$
$A \to .aA/.b$

(I_0, b) - I_4
$A \to b.$

(I_2, A) - I_5
$S \to AA.$

(I_2, a) - I_3
$A \to a.A$
$A \to .aA/.b$

(I_2, b) - I_4 I_3,
~~accept~~
$A \to b.$

(I_3, a) - I_3
$A \to a.A$
$A \to .aA/.b$

(I_3, A) - I_6
$A \to aA.$

(I_3, b) - I_4
$A \to b.$

LR(0) parsing Table

	Action			Goto	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

writedown the states that contains "." at the end of the production.

$I_1 \to S' \to S.$
$I_4 \to A \to b.$
$I_6 \to A \to aA.$
$I_5 \to S \to AA.$

Given Grammar
$S \to AA$ - ①
$A \to aA$ - ②
$A \to b$ - ③

→The given string w = aabb ⇒ w = aabb$

~~Stack~~	~~Input~~	~~Action~~
~~$0~~	~~aabb$~~	~~shift 3~~
~~$0a3~~	~~abb$~~	~~shift 3~~
~~$0a3a3~~	~~bb$~~	~~r1 S→AA~~ ~~shift 4~~
~~$0a3a3b4~~	~~b$~~	~~reduce r3 A→b~~
~~$0a3a3~~	~~b$~~	~~shift 4~~
~~$0a3a3b4~~	~~$~~	

→ while performing shifting it we have to shift i/p symbol on to the top of the stack & then state numbers

Stack	Input	Action
$0	aabb$	S3
$0a3	abb$	S3
$0a3a3	bb$	S4
$0a3a3b4	b$	reduce r3 A→b
$0a3a3A6	b$	r2 A→aA
$0a3A6	b$	r2 A→aA
$0A2	b$	S4
$0A2b4	$	r3 A→b
$0A2A5	$	r1 S→AA
$0S1	$	Accepted.

String w=aabb is parsed though the grammar by using LR(0) parser.

Canonical LR parsing:-

Step 1:- Augmented grammar

Eg:- $S \to CC$ $\quad$ $S' \to S$

$C \to aC$ $\quad$ $S \to CC$

$C \to d$ $\quad$ $C \to aC$

$C \to d$

w = add

2:- LR(1) items = LR(0) + lookahead

I_0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a|d$
$C \to .d, a|d$

(I_0, S) – I_1
$S' \to S., \$$

(I_0, C) – I_2
$S \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

(I_0, a) – I_3
$C \to a.C, a|d$
$C \to .aC, a|d$
$C \to .d, a|d$

(I_0, d) – I_4
$C \to d., a|d$

(I_2, C) – I_5
$S \to CC., \$$

(I_2, a) – I_6
~~S~~ $S \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

(I_2, d) – I_7
$C \to d., \$$

(I_3, C) – I_8
$C \to aC., a|d$

(I_3, a) – I_3
$C \to a.C, a|d$
$C \to .aC, a|d$
$C \to .d, a|d$

(I_3, d) – I_4
$C \to d., a|d$

(I_6, C) – I_9
$S \to aC., \$$

(I_6, a) – I_6
$C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

(I_6, d) – I_7
$C \to d., \$$

Step 3:- construction of CLR parsing table

	Action			Goto	
	a	d	$	S	C
0	S_3	S_4		1	2
1			Accepted		
2	S_6	S_7			5
3	S_3	S_4			8
4	$C\rightarrow d$ r_3	$C\rightarrow d$ r_3			
5			$S\rightarrow CC$ r_1		
6	S_6	S_7			9
7			$C\rightarrow d$ r_3		
8	$C\rightarrow aC$ r_2	$C\rightarrow aC$ r_2			
9			r_2 $C\rightarrow aC$		

1: $S\rightarrow CC$
2: $C\rightarrow aC$
3: $C\rightarrow d$

I_1: $S'\rightarrow S.$, $
I_4: $C\rightarrow d.$, a/d
I_5: $S\rightarrow CC.$, $
I_7: $C\rightarrow d.$, $
I_8: $C\rightarrow aC.$, a/d
I_9: $S\rightarrow aC.$, $

LALR Table :- $I_3 = I_6$
$I_4 = I_7$
$I_8 = I_9$

	Action			Goto	
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

LALR parsing table

↓

w = add

Stack	I/p buffer	Action
$0	add$	S36
$0a36	dd$	S47
$0a36d47	d$	r_3 $C\rightarrow d$
$0a36C89	d$	r_2 $C\rightarrow aC$
$0C2	d$	S47
$0C2d47	$	r_2 $C\rightarrow d$
~~$0C2d~~	~~$~~	
$0C2C5	$	r_1 $S\rightarrow CC$
$0S1	$	Accepted

→ In LALR parser combine the ~~stat~~ same states that are having different lookahead symbols.

Canonical LR parser:- (CLR parser) ⑤ ⑯

Step 1:- Augmented grammar

$S \to CC$
$C \to cC$
$C \to d$

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

Step 2:- Calculating LR(1) items

$I_0 \to$
$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$
(I_0, S)

(I_0, S) — I_1
$S' \to S\cdot, \$$

(I_0, C) — I_2
$S \to C\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

(I_0, c) I_3
~~Case~~
$C \to c\cdot C, c|d$
$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$

(I_0, d) — I_4
$C \to d\cdot, c|d$

(I_2, C) — I_5
$S \to CC\cdot, \$$

(I_2, c) — I_6
$C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

(I_2, d) — I_7
$C \to d\cdot, \$$

(I_3, C) — I_8
$C \to cC\cdot, c|d$

(I_3, c) — I_3
$C \to c\cdot C, c|d$
$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$

(I_3, d) — I_4
$C \to d\cdot, c|d$

(I_6, C) — I_9
$C \to cC\cdot, \$$

(I_6, c) — I_6
$C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

(I_6, d) — I_7
$C \to d\cdot, \$$

Construction of CLR parsing table:-

	Action			Goto	
	c	d	$	S	C
0	s_3	s_4		1	2
1			Accepted		
2	s_6	s_7		5	
3	s_3	s_4		8	
4	r_3	r_3			
5			r_1		
6	s_6	s_7		9	
7			r_3		
8	r_1	r_1			
9			r_2		

$S' \to S$
1 $S \to CC$ — 1
2 $C \to cC$ — 2
3 $C \to d$ — 3

$I_1 : S' \to S\cdot, \$$
$I_4 : C \to d\cdot, c|d$
$I_5 : CC\cdot, \$$
$I_7 : C \to d\cdot, \$$
$I_8 : C \to cC, c|d$
$I_9 : C \to cC \$$

Parse the input string w= dd$

Stack	Input	String
$0	dd$	shift 4
$0d4	d$	reduce 3 C→d
$0C2	d$	shift 7
$0C2d7	$	reduce 3 C→d
$0C2C5	$	reduce 1, S→CC
$0S1	$	Accepted

↳ So, the given i/p string is accepted by the grammar.

↳

LALR parser :- (Look Ahead LR parser)

[illegible] steps to construct LALR parser

(1) Introduce Augmented grammar

(2) calculate the LR(1) items

(3) construction of LALR parse table

(4) parsing of given i/p string.

Eg:- Construct LALR parser for

S→CC

C→aC

C→d

and also parse the string w=add.

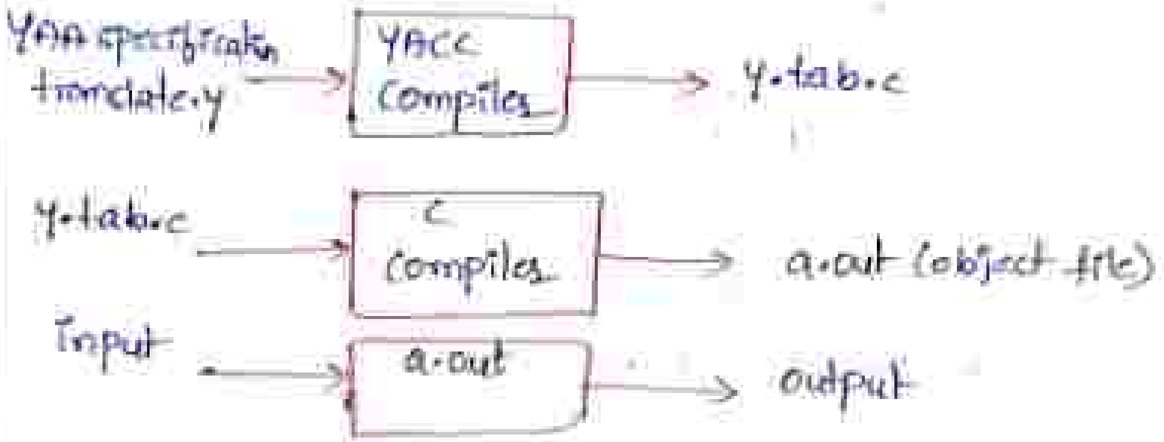
Parser generator:-

→ we use LALR parser generator YACC

→ YACC → "yet another compiler compiler"

→ YACC is a tool to generate parser, YACC accepts all tokens as i/p and produces parse tree as o/p.

The parser generator YACC:-



→ translate.y consists of YACC specification

Structure of YACC program:-

It has three parts

declarations
%%
translation Rules
%%
auxiliary functions

Declarations:-

→ used to declare the c variables and constants, & header files are also specified here.

Syntax:- %{
%}

Eg:- %{
int a,b;
const int a=20;
#include<stdio.h>
%}

(ii) Translation Rules:-

→ Translation Rules are enclosed between "%%" & "%%"

Syntax:- head → body1 | body2 | body... Eg:- E → aa | bb

head : body1 {semantic action}
 | body2 {semantic action}
 | body3 {semantic action}

$$ → represent the attribute value, to access the nonterminal value we use $$

$i →

→ In translation rules we use two symbols $$, $i

(ii) Auxiliary function:- $i → represent the ith symbol of body.

Eg:- E → E+T (If we want to access E, we have to use $1, + → $2, T → $3)

(iii) Auxiliary function:-

→ used to define "C" function

→ yylex is function is used here.

Eg:- YACC specification (program) of simple desk calculator.

Ex:- E → E+E/T
 T → T*F/F
 F → (E)/id

Declaration part:-

```
%{
    #include<ctype.h>
%}
% token DIGIT → [illegible] of CE
%%                    → it specifies i/p to desk calculator is an exp followed by \n
  line: expr '\n' {printf("%d\n", $1);}
  expr: expr '+' term {$$ = $1+$3;}
        |term
  term: term '*' factor {$$ = $1 * $3;}
        |factor
```

```
facts : '('expr')' {$$=$2;}
       | DIGIT ;
%%
yylex()
  {
    int c;
    c=getchar();
    if(isdigit(c))
   {
    yylval=c-'0';
    return DIGIT;
   }
    retur c;
 }
```

yylex → is a lexical analyzer function which takes our source program as input and produces token as o/p

1) Using Ambiguous Grammars

→ For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar

→ Another use of ambiguous grammar is in isolating commonly occurring syntatic constructs for special-case optimization, we can add new productions to grammar.

→ Sometimes ambiguity rules allow only one parse tree, in such case it is unambiguous, It is possible to design an LR parser that allows same ambiguity resolving choices.

Precedence & Associativity to Resolve conflicts:-

→ Consider ambiguous grammar with operators '+' & '*'

$$E \to E+E \mid E*E \mid (E) \mid id \quad —(1)$$

→ $E \to E+T$, $T \to T*F$, generates same language gives lower precedence to ~~*~~ '+' than *, makes left associative. [use ambiguous grammar]

→ ~~Ref~~ First we change associativity and precedence of operators + & * without disturbing above grammar.

→ Second, the parser for the unambiguous grammar will spend a substantial fraction of its time reducing by the productions $E \to T$ & $T \to F$

→ $E' \to E$, (1) is ambiguous there will be parsing-action conflicts when we try to produce LR parsing table.

$I_0: E' \to \cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot(E)$
$E \to \cdot id$

$I_1: E' \to E\cdot$
$E \to E\cdot +E$
$E \to E\cdot *E$

$I_2: E \to (\cdot E)$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot(E)$
$E \to \cdot id$

$I_3: E \to id\cdot$

$I_4: E \to E+\cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot(E)$
$E \to \cdot id$

$I_5: E \to E*\cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot(E)$
$E \to \cdot id$

$I_6: E \to (E\cdot)$
$E \to E\cdot +E$
$E \to E\cdot *E$

$I_7: E \to E+E\cdot$
$E \to E\cdot +E$
$E \to E\cdot *E$

$I_8: E \to E*E\cdot$
$E \to E\cdot +E$
$E \to E\cdot *E$

$I_9: E \to (E)\cdot$

LR(0) items of augmented exp. grammar

Conflict occur in I_7 & I_8 $E \to E+E\cdot$, $E \to E*E\cdot$

PREFIX	STACK	INPUT
E+E	0147	*id$

If i/p is id + id.

State	ACTION						GOTO
	id	+	*	(	)	$	
0	s3			s2			1
1		s4	s5			accept	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig :- parsing table for grammar

"Dangling-else" Ambiguity :-

stmt → if expr then stmt else stmt
       if expr then stmt
       other

Consider ~~The~~ grammar, an abstraction of this grammar where 'i' stands for if expr then, e stands for else and 'a' stands for "all other production".

$S' \to S$ (augmented grammar)

$S \to iSeS \,/\, iS \,/\, a$

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S\cdot$

I_2: $S \to i\cdot SeS$
$S \to i\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a\cdot$

I_4: $S \to iS\cdot eS$

I_5: $S \to iSe\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS\cdot$

Fig:- LR(0) states for augmented grammar

if expr then stmt —(2)

Ambiguity arises in I_4 shift/reduce conflict

$S \to iS\cdot eS$ calls for a shift of e

FOLLOW(S) = {e, \$}, $S \to iS\cdot$ call for reduction by [illegible]
$S \to iS$ on i/p 'e'

In (2) should we shift else onto stack or reduce if expr then stmt, we should shift else because it is associated with previous then.

SLR parsing table is constructed.

STATE	ACTION				GOTO
	i	e	a	$	S
0	S2		S3		1
1				accepted	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r1	

Input is iiaea, At line (5) state 4 selects the shift action on ip input e, whereas at line (9) state 4 calls for reduction by S → iS on input $.

STACK	SYMBOLS	INPUT	ACTION
(1) 0		iiaea $	shift
(2) 02	i	iaea $	shift
(3) 022	ii	aea $	shift
(4) 0223	iia	ea $	shift
(5) 0224	iiS	ea $	reduce by S → a
(6) 02245	iiSe	a $	shift
(7) 022453	iiSea	$	reduce by S → a
(8) 022456	iiSeS	$	reduce by S → iSeS
(9) 024	iS	$	reduce by S → iS
(10) 01	S	$	accept

Fig:- parsing actions on input iiaea

Syntax Directed Definition (SDD):

→ Semantic Analysis phase & ICG phase uses SDD's.

→ SDD is contex Free Grammar with semantic Rules and attributes

$\boxed{SDD = CFG + \text{semantic Rules}}$

→ SDD is also called as "attribute grammar"

→ Attributes are associated with grammar symbols and semantic Rules are associated with productions.

→ If 'x' is a symbol and 'a' is one of attribute then x.a denotes value at node 'x'.

→ Attributes may be numbers, string, data types & references etc

Eg

production	semantic Rules
$E \to E+T$	$E.val = E.val + T.val$ (Here E → is grammar symbol val → is attribute of E)
$E \to T$	$E.val = T.val$
$D \to TL$	$L.inh = T.type$

↳ Semantic Rules have two Notations (i) SDD (ii) SDT

Types of attribute:-

① **Synthesized Attribute:**- If a node takes value from its children node then it is called as synthesized Attribute.

Eg:- $A \to BCD$

$A.s = B.s$
$A.s = C.s$
$A.s = D.s$

} A → is synthesized attribute.
parent node takes value from child nodes
A → be a parent node
B, C, D → are children node.

(ii) **Inherited Attributes:**- If a node takes value from its sibling & parent then it is called as inherited attributes.

D → TL
Eg

production   semantic Rules

Eg:- $A \to BCD$ → C is inherited attribute

$C.i = A.i$ → Here C is taking value from its parent A.
$C.i = B.i$ → C is taking value from its sibling B
$C.i = D.i$ → C is " " " " " D

Eg:-

production	semantic Rules
$L \rightarrow E n$	$E.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

Fig: SDD for a simple desk calculator

Evaluating an SDD at Nodes of parse Trees:-

Annotated parse Tree:-

→ A parse tree which contain values at each node is known as Annotated parse tree.

→ A parse tree is constructed inorder to evaluate the attribute value at each node of parse tree.

→ If an attributed is synthesized,

- 1st evaluate the val attribute at all the children node.
- evaluate the value attribute at parent node.
- synthesized attributes, attributes are evaluated in bottom-up manner.

Production	Semantic Rules
$A \rightarrow B$	$A.S = B.i$ $B.i = A.S + 1$

These rules are circular, it is impossible to evaluate A.S without first evaluating B.i at same node.

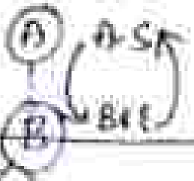


Fig: The circular dependency of A.S and B.i on one node

Eg1:-
construct an SDD for ~~a~~ simple desk calculator Grammar and construct parse tree for 3*5+4n.

Grammar-

production	SDD semantic Rule
L→En	L.val = E.val
E→E+T	E.val = E.val + T.val
E→T	E.val = T.val
T→T₁*F	T.val = T₁.val * F.val
T→F	T.val = F.val
F→(E)	F.val = E.val
F→digit	F.val = digit.lexval

Fig: SDD for simple desk calculator

Eg2: construct annotated parse tree for 2+3*4.

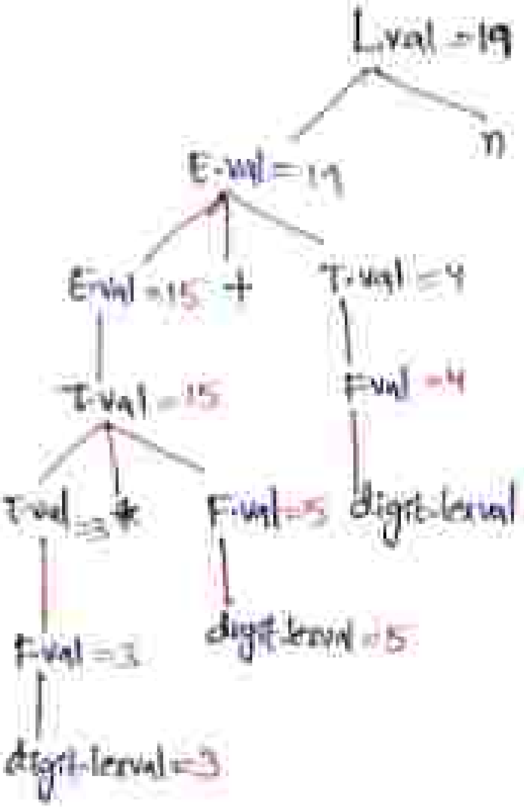


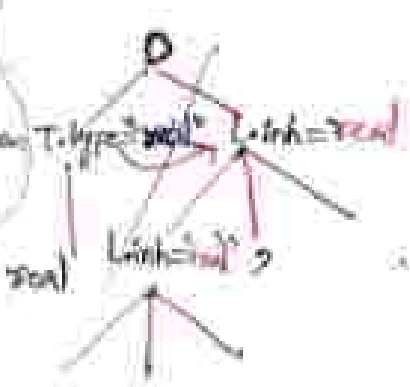
Fig: Annotated parsetree for 3*5+4n

SDD ↳ Annotated parse tree contains values at each node.
↳ To construct Annotated parse tree, we have to perform top-down left to Right traversing, if there is reduction execute corresponding action.

Eg2:

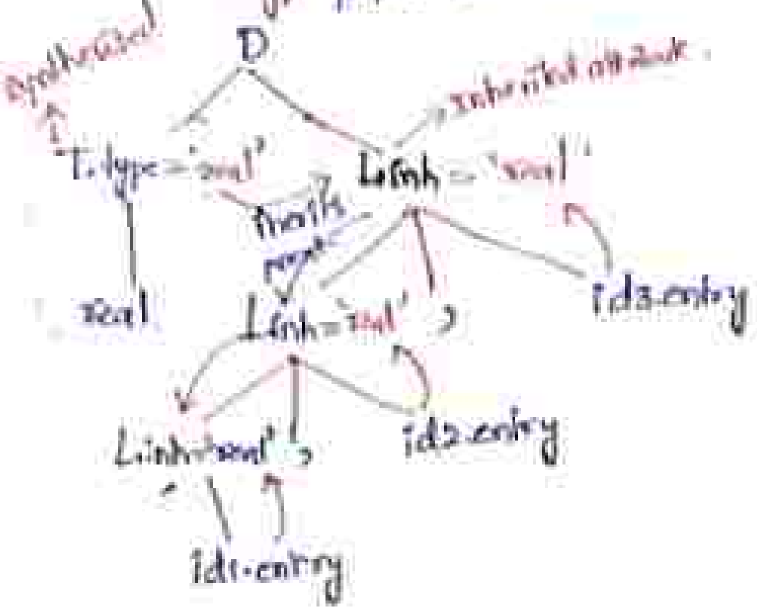
production	semantic Rules
D→TL	L.inh = T.type → (L) inherits type T
T→int	T.type = "integer"
T→real	T.type = "real"
L→L₁,id	L₁.inh = L.inh → here id needs to be given T.type addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

Parse tree:-



↳ In dependency graph we have ~~edges~~ directed edges

Fig:- Annotated parse tree for id1, id2, id3



⑥ Evaluating order SDD's:-

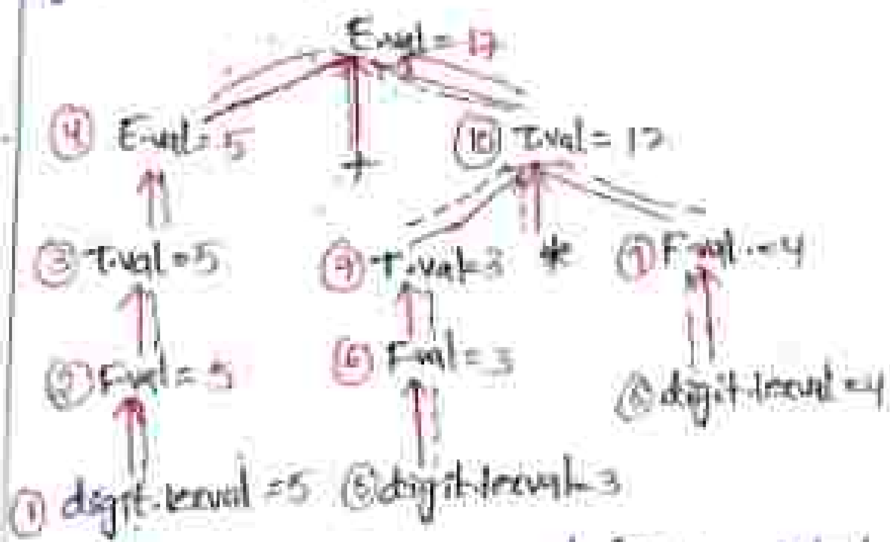
→ Dependency graph is used to determine an evaluation order for the attribute given

Dependency Graph:- → A directed graph that represents the interdependency b/w synthesized and inherited attribute at node in the parse tree

→ Dependency Graph represents the flow of information among the attributes in parse tree.

→ used to determine the evaluation order for attribute in a parse tree. (which semantic action should execute first)

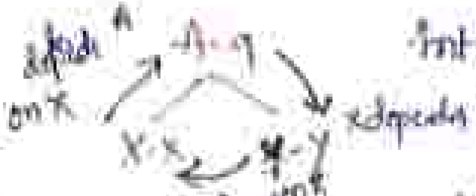
→ An Annotated parse tree shows the value of attributes, a dependency graph determines how those values can be computed.



production	semantic Rules
E→E+T	E.val=E.val+T.val
E→T	E.val = T.val
T→T*F	T.val=T.val*F.val
T→F	T.val=F.val
F→digit	F.val=digit.lexval

Fig:- Dependency graph for Annotated parse tree for 5+3*4

→ Edges in dependency graph show the interdependency b/w synthesized & inherited attributes at nodes in the parse tree.



Error: cyclic dependence

int a
D→TL

D→ int a

→ Dependency graph can't be cyclic

→ I DG there is a edge towards the depended node to originating node.

Eg:-

Production	Semantic Rules
T→FT'	T'.inh = F.val T.val = T'.syn
T'→*FT'₁	T'₁.inh = T'.inh * F.val T'.syn = T'₁.syn
T'→ε	T'.syn = T'.inh
F→digit	F.val = digit.lexval

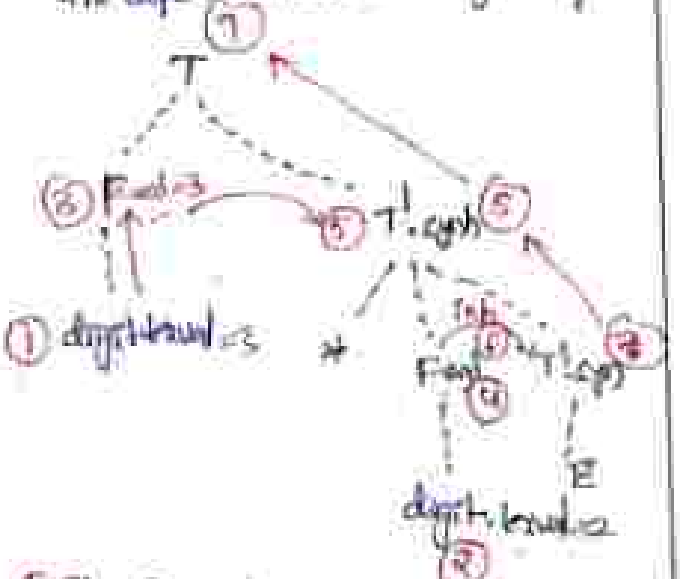
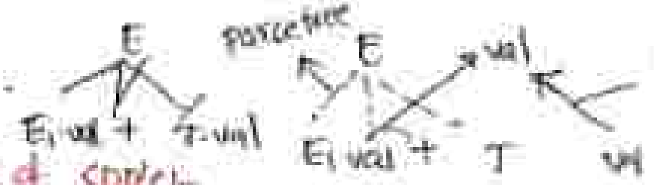


Fig:- Dependency graph for 3*5

Production	Semantic Rule
E→E₁+T	E.val = E₁.val + T.val



solid line → dependency
--- line → parse tree

Fig:- E.val is synthesized from E₁.val and T.val.

Types of SDD's:-

S-attributed Definition:-

* A SDD that use only synthesized attributes is called as S-attributed SDD.

Eg A→BCD

A.S = B.S
A.S = C.S
A.S = D.S

Eg:-

production	semantic rules
L→En	L.val = E.val
E→E+T	E.val = E.val + T.val
E→T	E.val = T.val
T→T*F	T.val = T.val * F.val
T→F	T.val = F.val
F→(E)	F.val = E.val
F→digit	F.val = digit.lexval

Eg:- SDD for S-attributed Definition

* semantic actions are placed at the end of the production.

	production	semantic action
Eg:-	E→E+T	{ E.val = E.val + T.val }

→ It is also called as postfix "SDD"

→ Attributes are evaluated with Bottom-up parsing

L-Attributed SDD:-

* A SDD that use both synthesis attributes & inherited attributes is called as L-attributed SDD.

* In L-attributed SDD, Inherited attributed is restricted to inherit from parent & left sibling only.

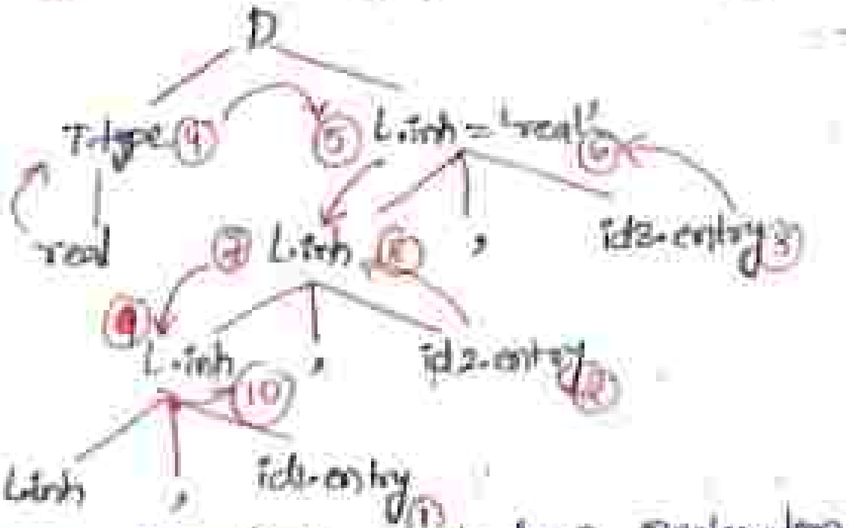
Eg:- $A \to XYZ \{ Y.i = A.i,\ Y.s = X.s,\ Y.s = Z.x \}$ production semantic

* semantic actions are placed anywhere on R.H.S. Eg= $E \to E + T \{\}$ $E \to \{\} E + T$

* attributes are evaluated by traversing parse tree depth first, left to right order.

Production	Semantic Rules
$D \to TL$	$L.inh = T.type$
$T \to int$	$T.type = integer$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$ addtype(id.entry, L.inh)
$L \to id$	L.inh - addtype(id.entry, L.inh)

Eg:- SDD for simple type declaration for L-attributed SDD



Eg:- Dependency graph for a Declaration float id_1, id_2, id_3

② Application of Syntax Directed Translation:-

Syntax Directed Translation (SDT):-

↳ SDT is a CFG together with semantic actions.

↳ Semantic actions are enclouse in '{', '}'

↳ Semantic actions are placed at any where on RHS of productions

eg: A→ABC →ABC{}
→A{}BC
→{}ABC

↳ semantic actions specifies in which order the expression is executed.

Eg:-

Production	Semantic action
A→B+C	{printf ('+');}

Eg2:-

production	Semantic action	
E→E+T	{printf ('+');}	①
E→T	{ }	②
E→T*F	{printf ('*');}	③
T→F	{ }	④
F→num	{printf(num.val);}	⑤

Eg3:-

production	Semantic action
E→E+T	{E.val=E.val+T.val}
T	{E.val=T.val}
T→T*F	{T.val=T.val*F.val}
F	{T.val=F.val}
T→num	{E.val=num.lvalue}

2+3*4

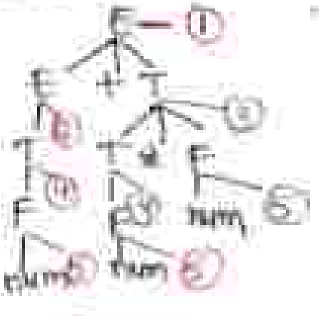


Fig:- SDT

Eg:- SDT for evalution of Expression.

Applications of SDT:-

Construction of syntax Tree:-

→ syntax tree is an intermediate representation

→ The nodes in syntax tree is implemented by objects with suitable no of fields

→ Each field will have an op-field that is the lable of the node.

We can construct the syntax tree by using the following functions.

(1) Mknode (op, left, Right)

② mkleaf (id, entry to symbol table)

③ mkleaf (num, value)

→ If the node is leaf, an additional field hold the lexical value for the leaf

leaf (op, val) -- create leaf object

→ If Node is an operator, create an object with first field - op and k additional for the k children $c_1 \ldots c_k$

Ex:- construct the syntax tree for the following grammar for the expression x*y-5+3.

Step 1:- construct SDD for the given grammar

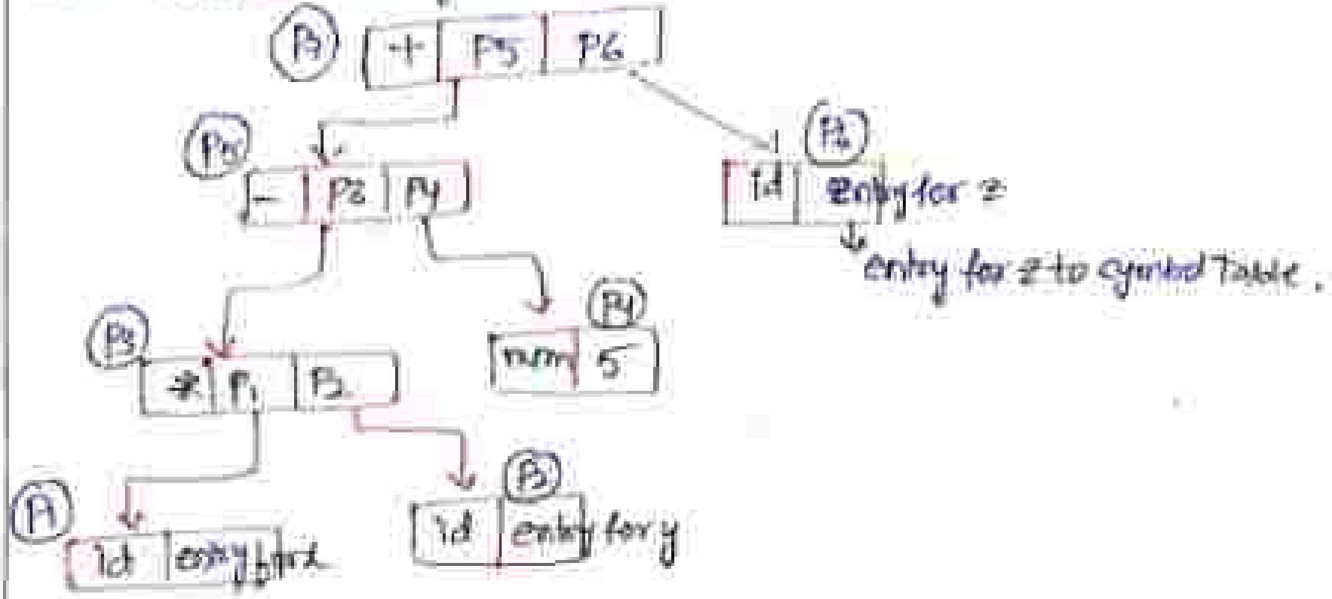
production	Semantic rule
$E \to E_1 + T$	E.node = mknode ('+', E_1.node, T.node) (or) E.node = Newleaf ('+', E_1.node, T.node)
$E \to E_1 - T$	E.node = mknode ('-', E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = mkleaf (id, id.entry) (or) T.node = newleaf (id, id.entry)
$T \to num$	T.node = mkleaf (num, num.val)

Step 2:-

symbol	operation
x	P_1 = mkleaf (id, entry-x)
y	P_2 = mkleaf (id, entry-y)
*	P_3 = mknode (*, P_1, P_2)
5	P_4 = mkleaf (num, 5)
–	P_5 = mknode (–, P_3, P_4)

z	P_6 = mkleaf (id, entry-z)
+	P_7 = mknode ('+', P_5, P_6)

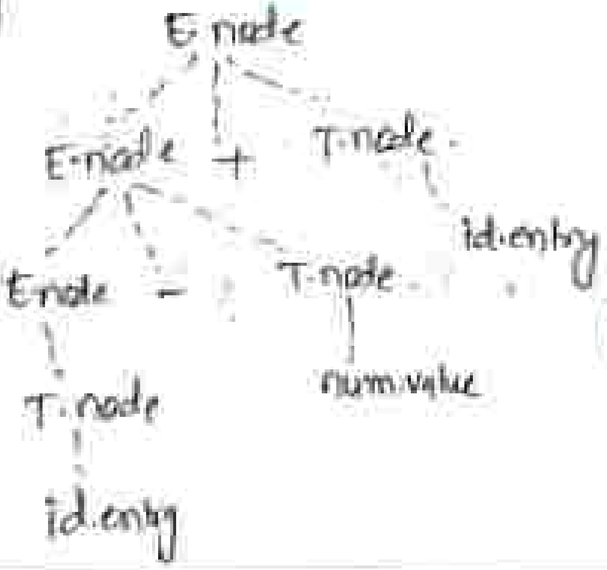
Step 3:- construct the syntax tree.

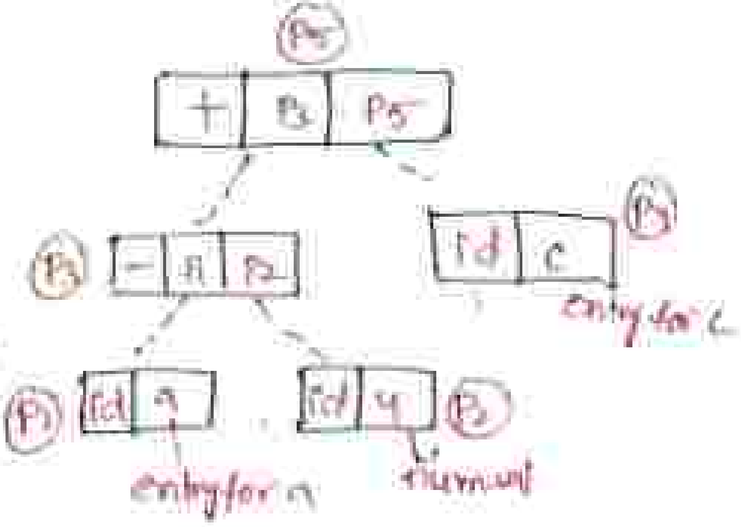


Eg 2:- a-4+c

symbol	operation
a	P_1 = mkleaf (id, entry-a)
4	P_2 = mkleaf (num, 4)
−	P_3 = mknode ('−', P_1, P_2)
c	P_4 = mkleaf (id, entry-c)
+	P_5 = mknode ('+', P_3, P_4)

Constructing syntax tree:-



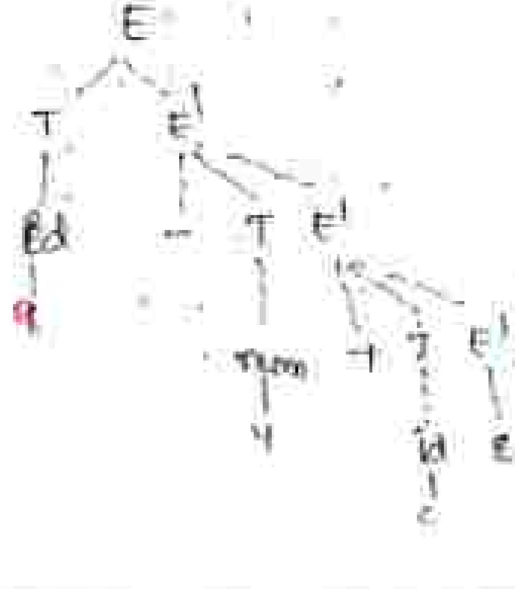


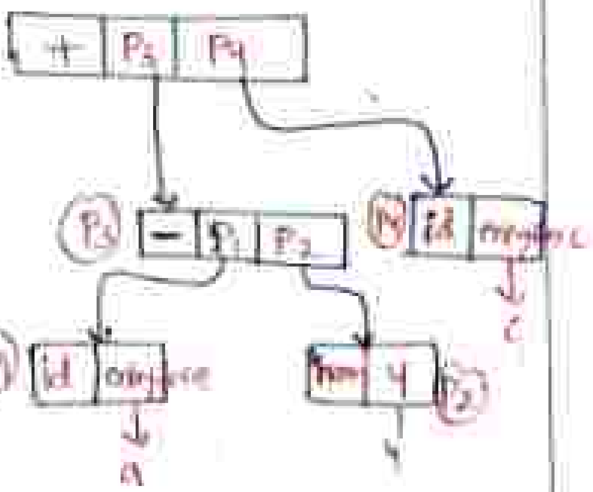
Eg3:- production	Semantic Rules
$E \to TE'$	E.node = E'.syn E'.inh = T.node
$E \to +TE_1'$	E_1'.inh = newnode('+', E'.inh, T.node) E'.syn = E_1'.syn
$E' \to -TE_1'$	E_1'.inh = newnode('-', E'.inh, T.node) E'.syn = E_1'.syn
$E' \to \varepsilon$	E'.syn = E'.inh
$T \to (E)$	T.node = E.node
$T \to id$	T.node = newleaf(id, id.entry)
$T \to num$	T.node = newleaf(num, num.val)

Eg:- a-4+c

symbol	operation
a	P_1 = mkleaf(id, entry-a)
4	P_2 = mkleaf(num, 4)
-	P_3 = mknode('-', P_1, P_3)
c	P_4 = ~~E~~ mkleaf(id, entry-a)
+	P_5 = mknode('+', P_3, P_4)

Syntax tree





③ Syntax Directed translation schemes:-

→ A SDT is a context free Grammar combined with semantic actions.

→ semantic actions are also called as program-fragments.

→ semantic actions are embedded with production body.

→ Any SDT can be implemented by constructing parse-tree and then performing the actions in a left-to-Right depth-first order.

SDT's are used to implement two important classes of SDD's

1) The underlying grammar is LR-parsable, & the SDD is is S-attributed

2) The underlying grammar is LL-parsable, and the SDD is L-attributed.

Postfix Translation schemes:-

→ It is used to convert infix expression to postfix expression

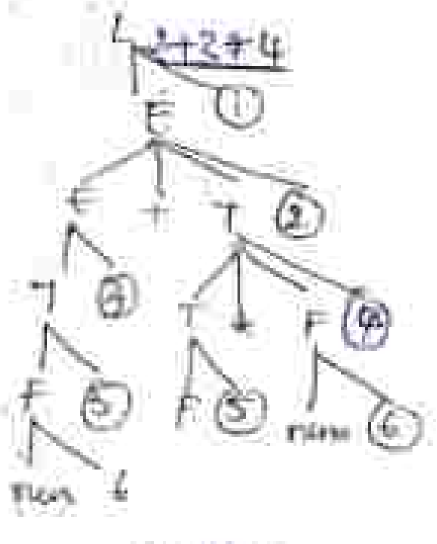
→ Infix expression - operators appears b/w operands

postfix expression - operators appears after the operands.

Eg:-

Production	Semantic actions
L → En	{ print(E.val); } ①
E → E+T	{ print('+'); } ②
E → T	{ } ③
T → T*F	{ print('*'); } ④
T → F	{ } ⑤
F → num	{ print num.val; } ⑥

fig:- SDT for desc calculator



2 34*+

IInd method to convert the Prefix infix expression to postfix expression

Eg:- $E \to E+T \mid T$ for = 1+2+3
$T \to num$

- here we are having only operation, so that we need check the left recursion in the grammar
- If Grammar contains left recursion we we need to convert the to eliminate left recursion from the grammar

production	SDT
$E \to E+T$	{printf('+');}
$E \to T$	{ }
$T \to num$	{print num.value}

SDT for given grammar

so the given Grammar contains left recursion

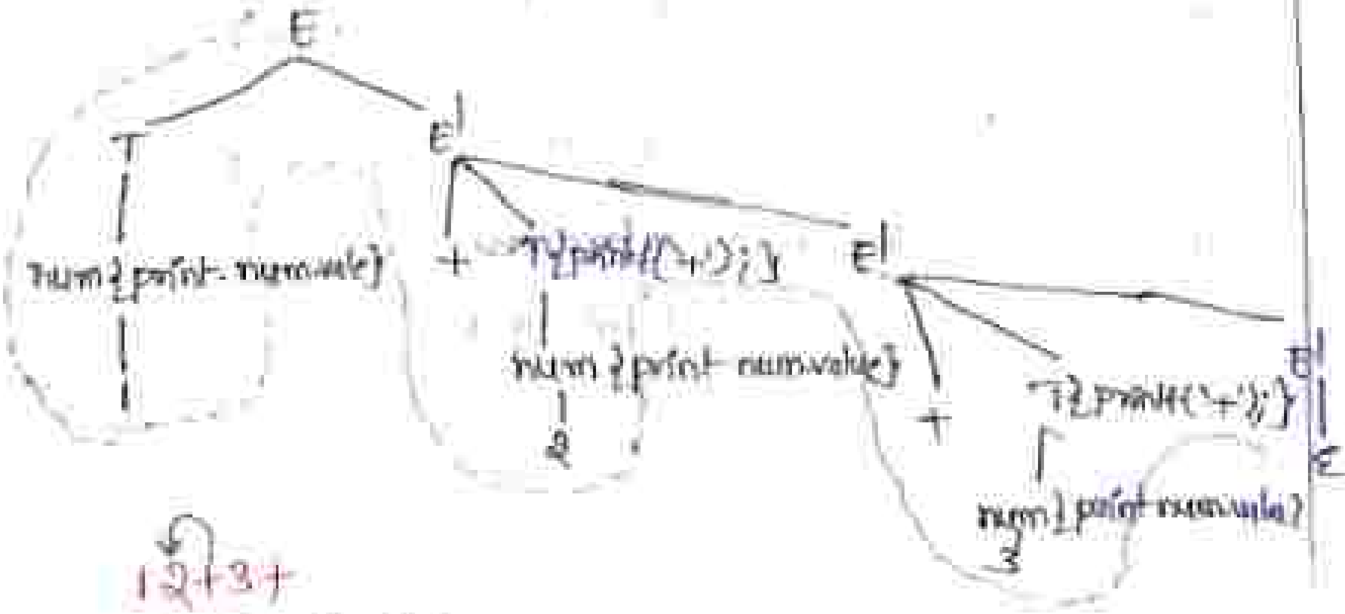
$\underset{A}{E} \to \underset{A}{E}\underline{+T}/T$
$T \to num$

$\underset{A}{E} \to \underset{A}{E}\underbrace{+T\{printf('+');\}}_{\alpha} / \underset{\beta}{T}$

$A \to A\alpha \mid \beta$

$\boxed{\begin{array}{l} A \to \beta A^{1} \\ A^{1} \to \alpha A^{1} \mid \epsilon \end{array}}$

The grammar after eliminating the left recursion
$$\left\{\begin{array}{l} E \to TE^{1} \\ E^{1} \to +T\{printf('+');\}E^{1} \mid \\ E^{1} \to \epsilon \\ T \to num\{print\ num.value\} \end{array}\right.$$



1 2 + 3 +

(1+2)+3 ⇒ 6

Eg:- Give translation scheme that convert infix expr to postfix expression for the following grammar and also generate annotated parse for input string 2+6+1

Grammar → $\underset{A}{E} \to \underset{A}{E}\underbrace{+T \text{ \{print('+')\}}}_{\alpha} / T_{\beta}$

$T \to 0|1|2|\cdots 9$ (It contains left recursion)

$A \to A\alpha|\beta$

$A \to \beta A'$

$A' \to \alpha A'|\epsilon$

$E \to TE'$

$E' \to +T$ {print('+');} E'

$E' \to \epsilon$

] The Grammar after eliminating left recursion

$T \to 0$ {print('0');}

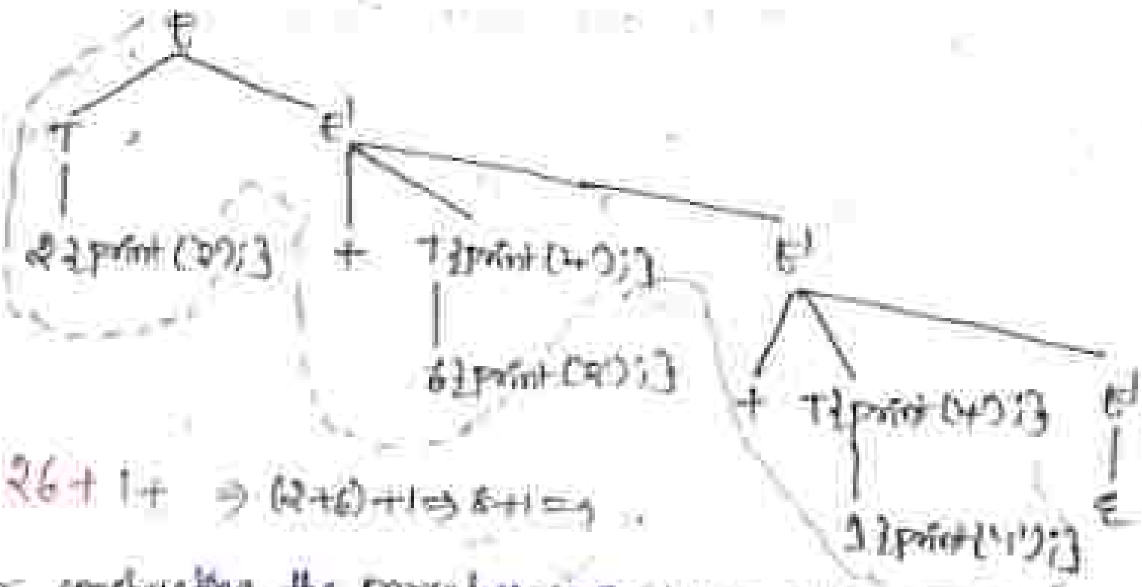
$T \to 1$ {print('1');}

|

$T \to 9$ {print('9');}

Fig: SDT to convert infix to postfix infix postfix

Annotated Parsetree:- The given i/p string 2+6+1 ⇒ 26+1+



26+1+ ⇒ (2+6)+1 ⇒ 8+1 = 9

→ after constructing the parse tree ~~parse~~ traverse the parse from top-down and left to right manner

$L \rightarrow E_n$ { print (E.val); }

$E \rightarrow E_1 + T$ { E.val = E_1.val + T.val; }

$E \rightarrow T$ { E.val = T.val; }

$T \rightarrow T_1 * F$ { T.val = T_1.val × F.val; }

$T \rightarrow F$ { T.val = F.val; }

$F \rightarrow (E)$ { F.val = E.val }

$F \rightarrow$ digit { F.val = digit.lexval }

Fig: postfix SDT implementing the desk calculator

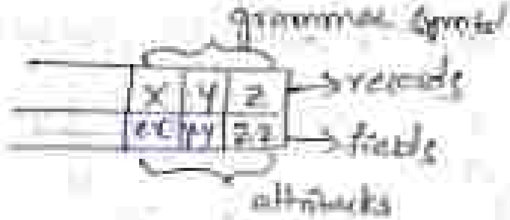
parser-stack implementation of postfix SDT's :-

→ postfix SDT's can be implemented during LR parsing by executing the actions when the reductions occur

→ The grammar symbols of each grammar-s

→ The attributes of each grammar symbols can be placed in stack during the parsing.

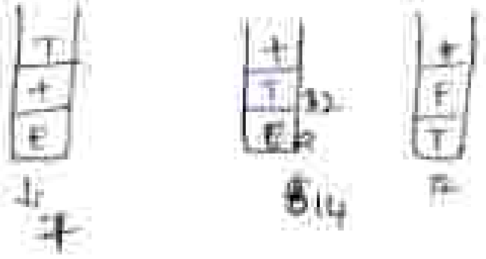
→ The parser stack contain records with fields for a grammar symbol.



If production A → XYZ

fig:- parser stack with a field for synthesized attributes

eg E→E+T {print(+);} & E→E+T 2+3*4



production	actions
L → En	{ print (stack[top-1]. val); top = top-1
E → E+T	{ stack [top-2]. val = stack [top-2]. val + stack [top].val; top = top-2; }
E → T	

T → T₁*F	{ stack [top-2]. val = stack [top-2]. val × stack[top].val;
	 top = top-2; }

T → F	

F → (E)	{ stack [top-2]. val = stack [top-1]. val;
	 top = top-2; }

F → digit	

Eg:- Implementing the desk calculator on bottom-up parsing stack.

SDT's with Actions Inside productions:-

→ ~~an~~ actions may be placed at any position ~~on~~ ~~Right~~ within the body of production.

Eg:- $B \rightarrow X\{a\}Y$

The action "a" is executed after recognizing the ~~B~~ "X"

- If the parse is bottom-up then we perform action "a" when "X" is appears on the top of the stack.
- If the parse is top-down, we perform a just before expanding the "Y"

Any SDT can be implemented as follows

1) Ignoring the actions, parse the i/p & produce a parse tree as a result.

2) examine each node and add additional node for corresponding action

3) perform the preorder traversal of the tree, and as soon as a node is labeled by action is visited, perform that action.

b

Eg:- parse tree for expression 3*5+4 with actions inserted we get if we visit the nodes in preorder, we get the prefix form of expression +*354

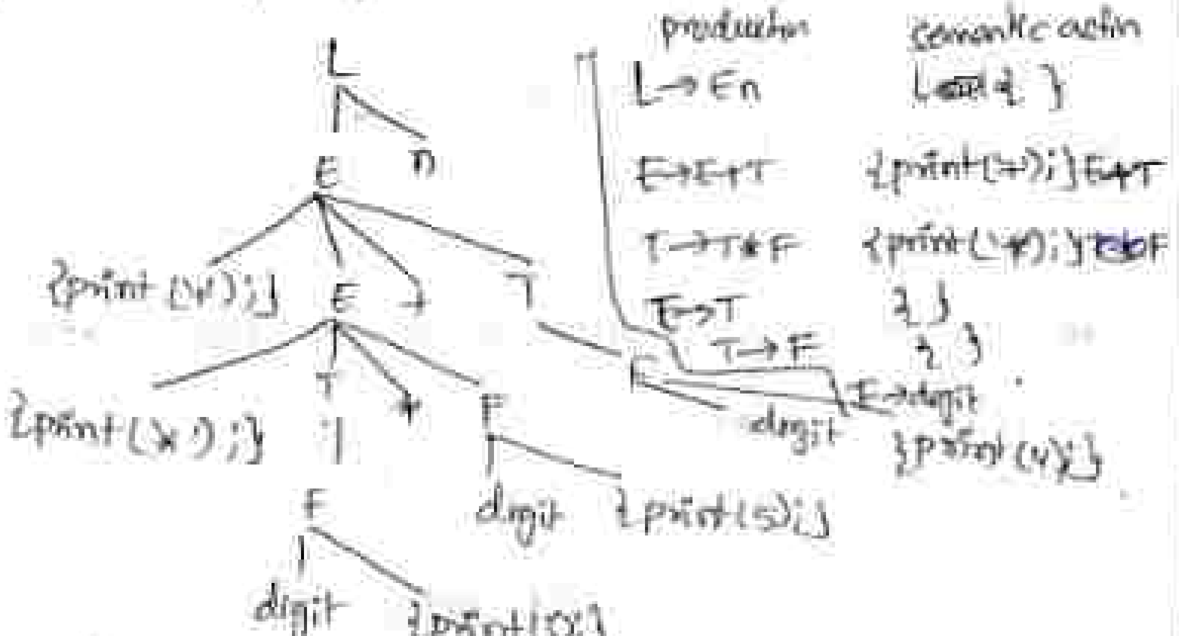


Fig: parse tree with actions embedded

Intermediate code generation:

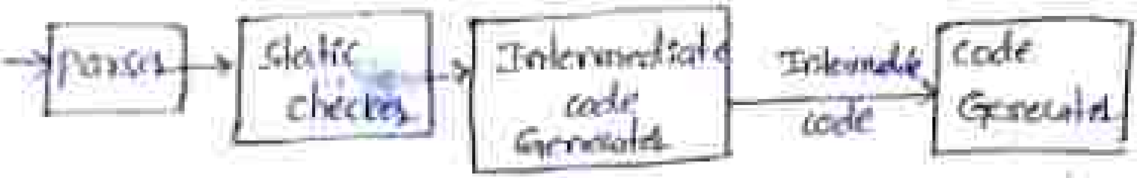
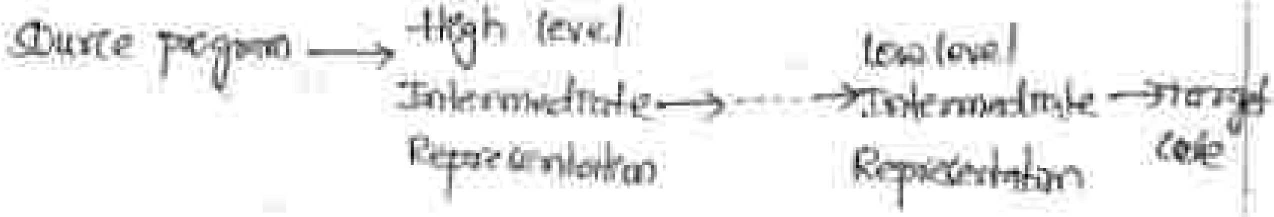


Fig:- logical structure of a compiler front end.

- →Intermediate code is used to translate the source code into the machine code.
- →In the above figure parsing, static checking and Intermediate code generation are done sequentially.
- → Static type checking includes type checking, which ensures that operators are applied to compatible operand.
- →ICG receives from the predecessor phase & semantic analyzer phase
- →It takes i/p in the form of an annotated syntax tree.
- → In process of translating a source program into target code compiler may construct a sequence of Intermediate representation


Source program
High level Intermediate Representation
low level Intermediate Representation
target code

- → Syntax trees are high level representation
- →A low level representation is suitable for machine dependent tasks like register allocation and instruction selection.

(1) Variants of syntax trees

Directed acyclic Graph (DAG) for Expressions:-

→ DAG is a Datastructure used for implementing transformations on basic blocks.

→ DAG Nodes in :

→ DAG represent the structure of a basic block.

- In DAG internal nodes represents operators and leaf nodes represent identifiers, constants.

operator
+
a
b

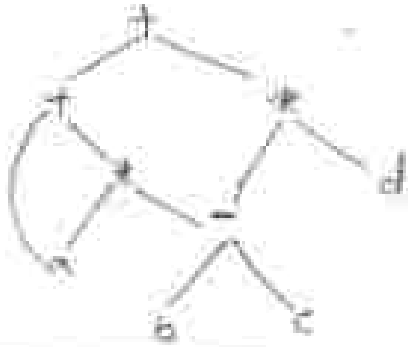
- Internal nodes represent the result of expression

→ The only difference b/w syntax tree and DAG is, In DAG a node has more than one parent.

Applications of DAG:-

- Determining the common subexpression
- Determining which names are used inside the block and computed outside, the blocks
- Determining which statement of the block could have their computed value outside the block.
- eg. Eliminating common subexpressions to simplify the code

Eg:- a+a*(b-c)+(b-c)*d



Syntax tree

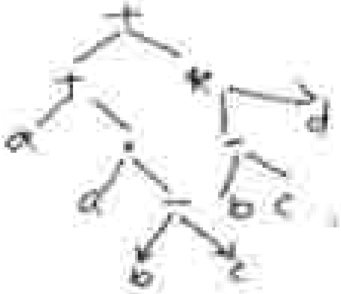


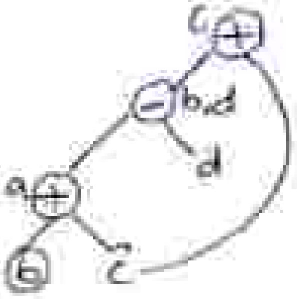
Fig:- DAG for expression a+a*(b-c)+(b-c)*d

production	Semantic Rules
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('−', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new leaf (id, id.entry)
$T \rightarrow num$	T.node = new leaf (num, num.val)

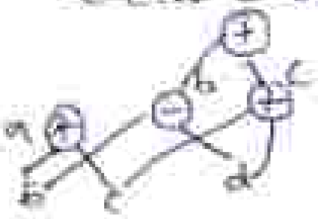
Fig:- SDD for to produce syntax tree & DAGs

1) P_1 = leaf (id, entry-a)
P_2 = leaf (id, entry-a) = p_1
P_3 = leaf (id, entry-b)
P_4 = leaf (id, entry-c)
P_5 = Node ('−', P_3, P_4)
P_6 = Node ('*', P_1, P_5)
P_7 = Node ('+', P_1, P_6)
P_8 = leaf (id, entry-b = P_3
P_9 = leaf (id, entry-c) = P_4
P_{10} = Node ('−', P_3, P_4) = P_5
P_{11} = leaf (id, entry-d)
P_{12} = Node ('*', P_5, P_{11})
P_{13} = Node ('+', P_7, P_{12})

Eg 2:- construct DAG for
1 a=b+c 3 c=b+c
2 b=a−d 4 d=a−b

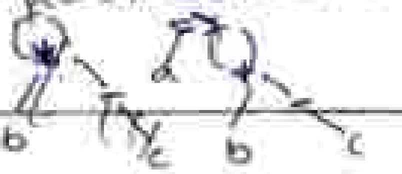


Eg 3:- 1 a=b+c 2 b=b−d
3 c=c+d 4 e=b+c

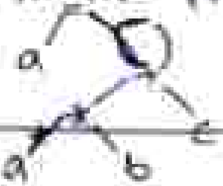


The value number Fig: steps for constructing the DAG.

Eg 2: a=b*−c+b*−c



Eg 3: a=(a*b+c)−(a*b+c)



3) The value Number method for constructing DAG:-

-> Nodes of syntax tree & DAG are stored in array of records.

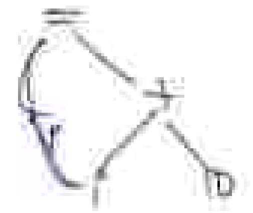
-> Each row of array represent one record.(node)

-> In each record first field is operation code, indicating the label of the node.

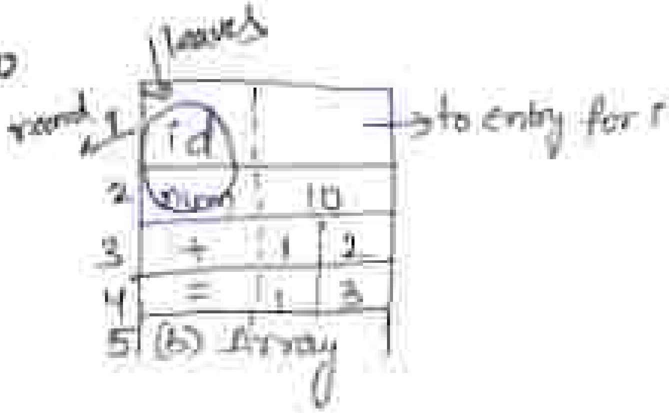
-> leaves has the leaves one additional field which holds the lexical value.

-> Interior nodes have two additional field indicating left and right children.

i = i + 10



(a) DAG for i=i+10


leaves
record
→ to entry for i

1	id		
2	num	10	
3	+	1	2
4	=	1	3
5			

(b) Array

Eg:- Nodes of a DAG for i=i+10 allocated in an array.

-> The array index is used for reference a node rather than a pointer.

-> Initially the array is empty.

-> First it searches for <id, lexval>, if it is not there, we will make new record and so on.

-> If already, record is present, it is just used for further records.

-> In the array, we refer to node by giving integer index of the record, for that node within the array this integer is called 'Value Number'. <op, value-num of left, value num of right>
↓
is also called as signature of node

Drawbacks:-

Search options; It takes time for every New/old record to search.

-> To overcome this we are using hash-functions, in which the nodes are put into "buckets" (hash table)

-> These buckets have only few nodes.

-> hashtable is a data structure that support the dictionaries

-> dictionaries are used to insert & delete elements of a set

-> dictionaries are used to determine whether a given element is currently in the set. this is don

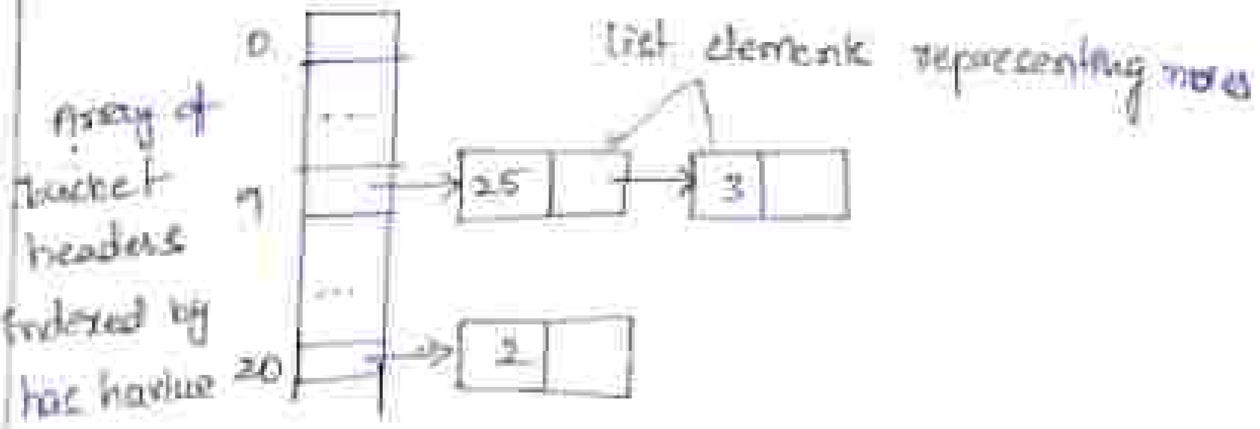
-> It search the element in less time and independent of the size of set.

TO construct a has table for node of a zone, ~~we~~ hashfunction "h" is used that computes the index of bucket

-> [illegible] The bucket index h(op, l,r)

-> bucket can be implemented as linked list.

-> An array indexed by hashvalue, hold the bucket headers, each of which point to first cell of list


0
Array of bucket headers indexed by has hashvalue
4
20
list elements representing nodes
25
3
2

Eg: data structure for seaching buckets

③ Three Address code:-

Intermediate code is three types

① syntax tree representation
② postfix Notation
③ Three address code.

In three-address code

① Each Instruction should contain atmost 3 addresses
② Each Instruction should contain 1 operator on RHS

Eg:- source language expression $x+y*z$ is converted into sequence of 3-address instructions

~~$t_1=0$~~ $t_1 = y*z$
$t_2 = x+t_1$

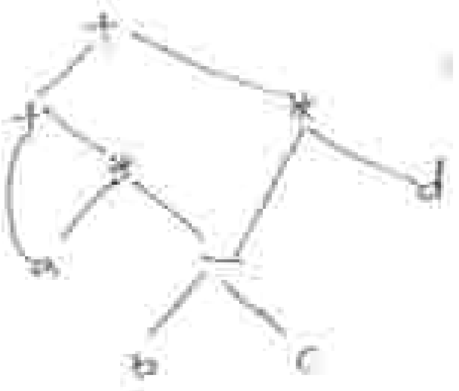
$t_1, t_2 \rightarrow$ compiler generated temporary variable.

→ 3-address code is linearized representation of syntax tree & DAG

Eg:- $a+a*(b-c)+(b-c)*d$

$t_1 = b-c$
$t_2 = a*t_1$
$t_3 = a+t_2$
$t_4 = t_1*d$
$t_5 = t_2+t_4$

fig(a) Three address code



Fig(b):- DAG

→ Three address code is represented in 3-ways

① Quadruple ② Triple ③ Indirect Triples

~~types of 3-address code~~

Address and Instructions:-

→ 3-address code can be implemented by using records with fields for the addresses

→ records are called quadruples and triples.

The address can be one of the following:

- A name → (source program names are addresses in 3-address code and names are replaced by pointer to the symbol table entry)
- A constant
- compiler generated temporary variables.

List of the common three-address instruction forms:-

(i) Assignment instruction → $x = y \text{ op } z$, where x, y, z - addresses

(ii) → $x = \text{op } y$ where op - is unary operation $(-, \sim)$

(ii) copy instruction → $x = y$

(iii) unconditional jump → goto L

(iv) conditional jump → if x goto L

→ if false x goto L

→ if x relop y goto L

(v) procedure call → y = call p, n

return y

p → is the address of starting of line of procedure p

n → argument address.

y save return value

(vi) Indexed copy instruction → $x = y[i]$, $x[i] = y$ } x, y, i are the variables

(vii) Address and pointer assignment $x = \&y$

$x = *y$

$*x = y$

Three address code

Quadruples :-

3-address code is implemented as objects(or) records with fields for the operator and operand.

→ Quadruple has 4-fields (i) op (ii) arg1 (iii) arg2 (iv) result

Eg:- * Instruction like $x = y$ & $z = -y$ do not use arg2

* operator like param use neither arg2 nor result

* conditional & unconditional jumps put the target table in result.

Eg :- $a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

$\} \; t_5 \Rightarrow t_1$
$= t_2 + t_3$

(a) Three address code

	op	arg1	arg2	result
(0)	−	c		t_1
(1)	*	b	t_1	t_2
(2)	−	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

(b) Quadruples

→ The disadvantage of quadruples is too many temporary variables are needed, it require more amount of memory

→ Inorder to overcome we are using Triples.

Triples :- Triples has only three fields ① op ② arg1 ③ arg2

Eg:- $a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$

	op	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
5	=	a	(4)

(b) Triples representation of $a = b * -c + b * -c$;

→ using triples we refer results of an operation to x op y by the position, rather than by an explicit temporary variable.

Indirect Triples:-

Indirect triples consist of listing of pointers to triples, rather than a listing of triples themselves.

→ with triples the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result; This problem doesn't occure with indirect triples.

Instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	−	c	
1	*	b	(0)
2	−	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Fig: Indirect triple representation of three address code

Static single assignment form:- (SSA)

SSA is an specialcase of 3-address code. SSA is an intermediate representation that facilitates certain code optimizations → In SSA each assignment to a variable should be specified with distinct names

→ Eg:-

p = a+b	p_1 = a+b
q = p-c	q_1 = p_1 - c
p = q*d	p_2 = q_1 * d
p = e-d	p_3 = e - p_2
q = p+q	q_2 = p_3 + q_1
(a) Three-address code	(b) static single assignment form.

Eg:- Intermediate program in three-address code SSA

→ The least no. of temporary variables required to create 3-address code in SSA.

* A variable can only be initialized one in L.H.S
* A variable which is initialized in L.H.S could only used R.H.S

⑨

Control Flow:-

Simple if, if-else, else-if, switch, for, while, do-while.

The translation of statements such as if-else-statement and while-statement is tied with translation of Boolean Expressions.

→ Boolean Expressions are used to.

① change the flow of control

② compute the logical values

Boolean expr can used as conditional expr in stmts that alter the flow of control for eg: if (E) S,

↓ A Boolean Expression can represent true & false as value.

Boolean Expressions:-

Boolean Expressions are composed of boolean operators

& &&, ||, and !

→ Boolean Expressions are generated by the following grammar

$B \rightarrow B || B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$

→ AND and OR are left associative

→ "NOT" has higher precedence than AND & OR

Short circuit code:- (Jumping code)

In short-circuit code, the boolean operators &&, || and ! are translate into jumps.

→ In short-circuit code the 2nd argument (expression) is evaluated only if 1st argument does not suffice to determine the value of Expression.

Eg:- if (x<100 || x>200 && x!=y) x=0;

In this translation the BE is true if control reaches label L_2. If the expression is false, control immediately to L_1, skipping L_2 and the assignment x=0.

if x<100 goto l_2 Eg2: (x==y || y==z)

ifFalse x>200 goto l_1

ifFalse x != y goto l_1

l_2: x=0

L_1:

fig: Jumping code

Flow of control statements:

→ Translation of Boolean Expression control-statements into three-address code.

Grammar $S \rightarrow$ if (B) S_1 (A) $S \rightarrow$ if (B) then S_1

↳ condition (or) Boolean expression

$S \rightarrow$ if (B) S_1 else S_2

$S \rightarrow$ while (B) S_1

∴ Grammar for simple if, if-else, while statements

① $S \rightarrow$ if (B) then S_1 (B) is evaluated 1st

Code for simple if:	Semantic Rule:-

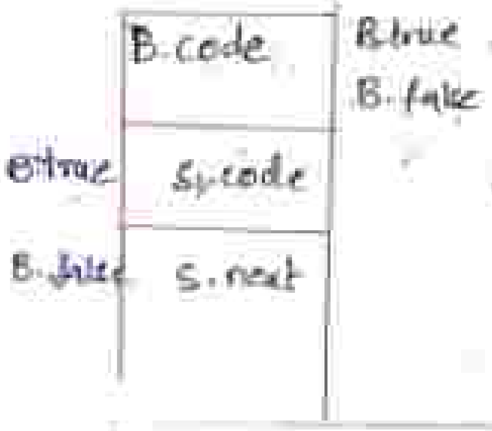
 B.code
B.true
B.false
B.true
S_1.code
B.false
S.next
	B.true = newlabel( )
	B.[illegible] = S_1.next = S.next B.false = S_1.next = S.next
	Intermediate code:-
	S.code = B.code || label (B.true) || S_1.code

Fig:- SDD for simple if-statement

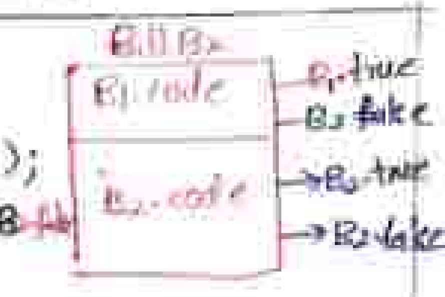
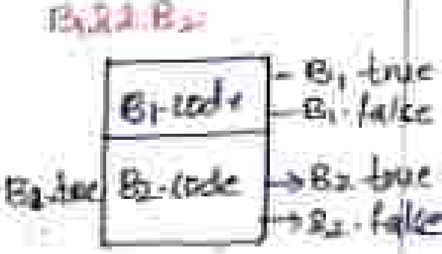
if ( ) ↳ i.e., newlabel( ) - function produce three address code for B.true.

{

}

S

Control flow translation of Boolean expressions (#)
Write intermediate code for Boolean expression (#) (or) (or) SDT for Boolean exp

production	semantic rules
B → B1 || B2	{ B1.true = B.true; B1.false = newlabel(); B2.true = B.true; B2.false = B.false; Intermediate code: B.code = B1.code || label(B1.false) ||B2.code  B1||B2 B1.code B2.code B1.true B1.false B2.true B2.false B.false
B → B1 && B2	{ B1.true = newlabel(); B1.false = B.false; B2.true = B.true; B2.false = B.false; B.code = B1.code || label(B1.true) || B2.code }  B1&&B2 B1.code B2.code B1.true B1.false B2.true B2.false B1.true

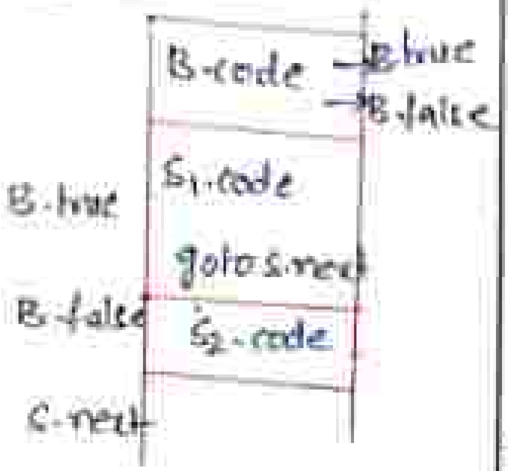
B → !B1	{ B1.true = B.false;
	B1.false = B.true;
	B.code = B1.code }

B → true	{ B.code = gen('goto' B.true); }
B → false	{ B.code = gen('goto' B.false); }
B → E1 relop E2 E1 [a<b] if a<b goto E.true goto E.false	{ B.code = E1.code || E2.code || gen('if' E1 relop E2 'goto' B.true || ~~gen('if' E1 relop E2 goto~~ || gen('goto E.false

E → (E1)	{ E1.true = E.true;
	E1.false = E.false;
	E.code = E1.code; }

S → if (B) then S_1 else S_2

code for if-else:



Semantic rules for if-else stmt:

B.true = newlable()

B.false = newlable()

S_1.next = S.next

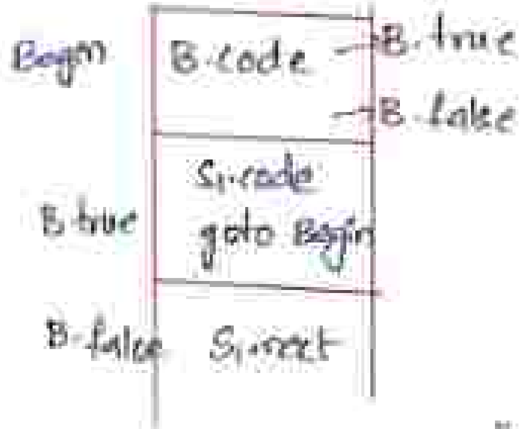
S_2.next = S.next

S.code ~~Three address~~ Intermediate code:

S.code = B.code || label (B.true) || S_1.code
|| gen ('goto' S.next)
|| label (B.false || S_2.code

(iii) while (B) then S_1

code for while



Semantic Rules

Begin = newlabel()

B.true = newlabel()

S_1.next = begin

B.false = S.next

Intermediate code

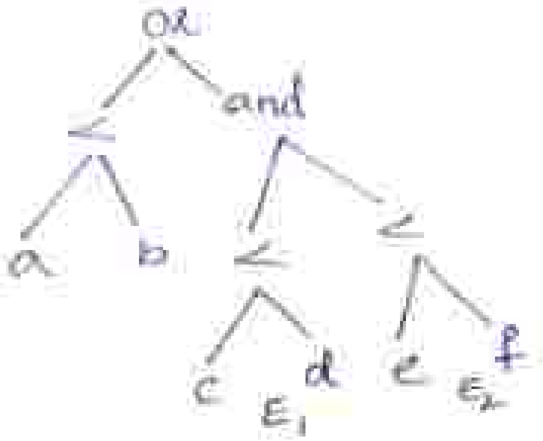
S.code = label (Begin) || B.code
|| label (B.true) || S_1.code
|| gen ('goto' begin)

production	Semantic rules

P → S	S.next = newlabel ()
	P.code = S.code || label (S.next)

S → assign	S.code = assign.code
S → S_1 S_2	S_1.next = newlabel () S_2.next = S.next S.code = S_1.code || label (S_1.next) || S_2.code

eg ① $a<b$ or $c<d$ and $e<f$



```
if a<b goto E.true
  goto E1
E1: if c<d goto E2
  goto E.false
E2: if e<f E.true
  goto E.false
```

eg-② if ($x<100$ || $x>200$ && $x!=y$) $x=0$

```
    if x<100 goto L2
    goto L3
L3: if x>200 goto L4
     goto L1
L4: if x!=y goto L2
    goto L1
L2: x=0
L1:
```

Types and Declarations:-

*Type checking uses logical rules to decide about the behaviour of program at runtime.

*It also ensures that types operand match type expected by the operators

Eg:- "&&" operation Java expect its two operands to be boolean

int + float → type error

→ Determine the storage needed

"Translation Application:

compiler translate a type of name into storage

compiler also determines the amount of storage required to store the type name at run time.

"Type Expression:-

type Expression is either a basic type or formed by applying an operator called type constructs to a type Expression.

→T.E are used represent the structure of type,

→T.E are primitive datatypes.

→Type name:- Is a Type Expression Eg:- ~~int~~ typedef abc int.

type Expressions are of two types.

(i) Basic type:- Basic type for language are int, real, boolean, char float, and void. A special type, type-error is used to indicate type error

Eg:- int x;   Eg:- type abc int;
int a;   is $\boxed{a=b;}$ depend language
abc b;

(ii) type constructs (or) TypeName:-

↳ Type constructs applied to list of type expressions.

↳ types are formed applying an operator called type constructs to type expression.

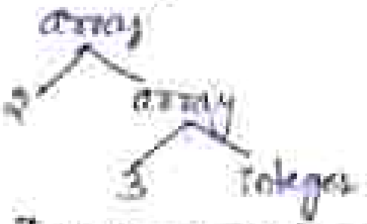
① **Arrays**:- Arrays are specified as array (I, T) where I → Int (or) range of integers
T → Type.

Eg1:- ~~For~~ In "C" Declaration "int a[100]" identifies type of "a" to be array(100, Integer) ↓ int (100) a;

Eg2:- T.E for int [2][3], "2 array with 3 integers".

obj name array (2, array(3, Integer)) →how many objects

↓ Type Expression



Type Expression for int[2][2]

Eg3:- int a[100], b[50];

$\boxed{a=b}$ × type check.

→ A type expression can be formed by applying array constructs to a number & type expression.

(ii) **Record**:- A record is a data structure with name fields.

→ A type expression can be formed by applying a record type constructs to the field name and their types.

Eg:- Struct st
{
int s;
float f;
};
Struct st s1;
record (s1, (int, float))
record (s1, Integer)
record (s1, float)

Eg:- Named records are product with named elements for record structure with 2 named fields length (an integer) and word (of type array char[10]) the record is of type.

record ((length × Int) × (word × array(10, char)))

struct
{ int length
char word[10]
};

→ Type Expression may contain variable whose values are T.E Eg: int a=5;

Product:- S and T are a T.E then their cartesian product S×T is a type expression Eg:- int × int

Function:- Function maps a collection of types to another represented by D→R, where D is domain R is range of function.

Eg:- int f (int x, char y, float z)
{ return m;
}

Domain = (int × char × float)
Range - o/p: int

→ T.E "int×int → char" represent a function that takes 2 integers & returns a char value

Type equivalence:-

→ Two type are said to be equivalent if and only if an operand of one type in an expression is substituted for one of the other type, without type conversion.

Type equivalence are of two types.

i) Name equivalence:-

The two type expression are said to be name equivalence if they they have same name & label.

Eg1:-
```
typedef int value
typedef int total
      :
value  var1, var2;
total  var3, var4;
```

Eg2:
```
type def struct Node
{
 int x;
 } node;
 node  * first, second;
 struct node  *last1, *last2;
```

→ in the above eg1: var1 and var2 are name equivalence because their types are same.

→ var3 & var4 also Name equivalence.

→ but var1 & var3 are not name equivalent because their types are different

ii) Structural equivalence:-

→ If two expression are the basic type (a)

→ Formed by applying the same constructor to structurally ~~type~~ equivalent types then those expression are called structurally equivalent

(i) It checks the structure of type

(ii) Determines equivalence by whether they have same constructs applied to structurally equivalent types

Eg:- type array (I_1, T_1) and array (I_2, T_2) structurally equivalent if $I_1 = I_2$ & $T_1 = T_2$

$I_1 \rightarrow$ Index of array

$T \rightarrow$ type of

array a[100], b[50]

array a[100], b[100]

&

structurally equivalent

Eg1:- type def int value ← x

typedef int number ← y

x : array (50, int)

y : array (100, int)

Eg2:-

S_1	S_2	Equivalence	Reason Reason
char	char	S_1 is equivalent to S_2	similar basic type
pointer (char)	pointer (char)	S_1 is equivalent to S_2	similar constructs pointer to the char type.

Declarations :-

$D \rightarrow T$ id ; D|ε

$T \rightarrow B\ C$ | record '{D}'

$B \rightarrow$ int / float

$C \rightarrow$ ε/[num] C

D → Sequence of declarations.

T → basic ~~B~~ & array and record types

~~B~~ C → 'component' - generates zero or more integers within the brackets.

- Array type consists of basic type specified by B, followed by array component C.

  Eg:- int [10][11]

- Record type is sequence of declaration for field of the record all surrounded by curly braces

  record {int, a}

Storage layout for local Names:-

→ compiler converts the typenames into the storage.

→ and determines the amount of storage needed to store the typename at runtime.

→ at compile time we can use these amount to assign a type name to relative address.

relative address = offset + program counter.

→ Relative & types are saved to symbol table entry for type name.

→ Data of varying length such as string or whose size cannot determined until runtime such as dynamic array.

→ The width of a type is no. of storage units needed for objects of that type.

SDT computes types and their widths for basic and array types.

T → B {t = B.type ; w = B.width;}

C {T.type = C.type ; T.width = C.width;}

B → int {B.type = integer; B.width = 4;}

B → float {B.type = float; B.width = 8;}

C → ε {C.type = t ; C.width = w;}

C $\xrightarrow{\text{array}}$ [num] C_1 {C.type = array (num.value, C_1.type) C.width = num.value × C_1.width;}

Fig:- SDT for computing their types & widths

→ These declaration are represented with DAG & parse tree

Eg:- parse tree for int [2] [3]

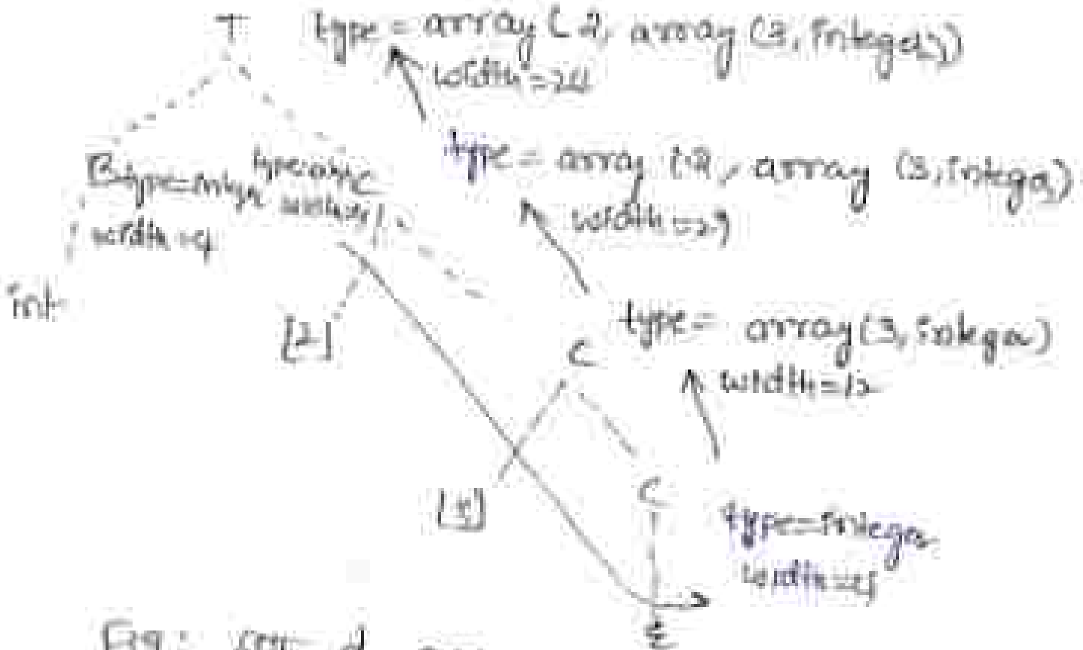


Fig: SDT of array types

Sequence of Declarations:-

→ In procedures all the declaration are passed at a time.

→ all Declarations in single procedure to be p as a group.

$P \rightarrow_{D}$ {offset = 0;}

$D \rightarrow$ T id; {top.put(id.lexeme, T.type, offset);
(or) id T; offset = offset + T.width;}
D; D

$D \rightarrow \epsilon$

offset — is variable to keep back of the next available relative address.

D → T id ; D — creates a symbol table entry for Executing top.put (id.lexeme, T.type, offset)

top — The current symbol table

top.put → Res creates a symbol table entry for id.lexeme with type & relative address.

Field in Records & classes:-

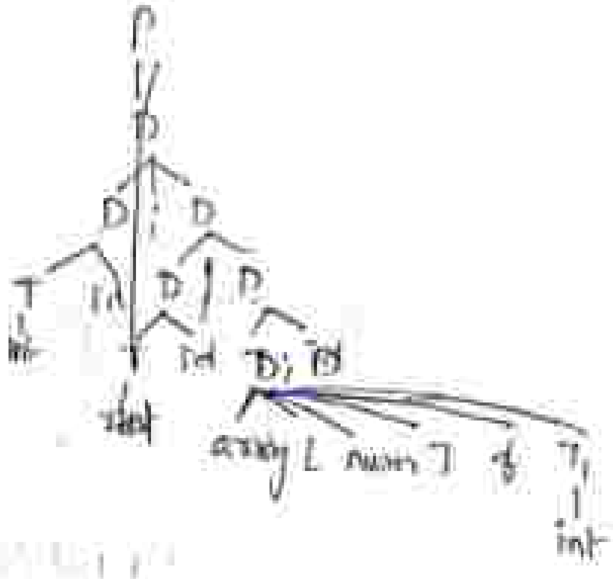
T → record '{D}'

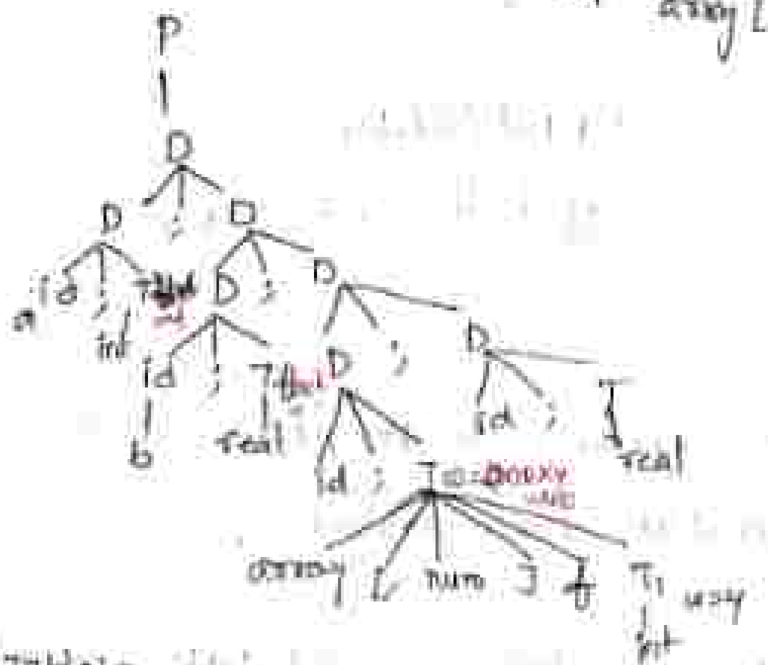
The field in this record is type specified by the sequence of Declaration grouped by D.

- The field names with record must be distinct
- The offset & relative address for fieldname is relative to the data area for that record

Eg:-

```
{
 a:int;
 b:real;
 c:array[10] of integer;
 d:↑real;
}
```





Symbol table:-

Name	type	offset	
a	integer	0 [0-3]	offset=0+4=4
b	real	4	offset=4+8=12
c	array(10,int)	12	offset=12+40=52
d	real	52	

Type checking :-

→ Compiler checks whether the program is following type rules or not.

→ information about data types is maintained & Computed by compiler.

→ Type checker is a module of a compiler devoted to typechecking tasks.

→ To do typechecking a compiler needs to asign a TE to each component of source program.

→ Compiler determines TE conform to collection of logical rules that is called type system for the source language.

→ Typechecking catch the Errors in program.

→ Assign types to values.

→ Simple situation :- Check types of objects & report a type error in case of a violation.

→ more complex :- Incorrect types may be corrected (type Coercing).

Static	Dynamic
→ Type checking done at compile time.	→ perform during program Execution.
→ properties can be verified before program run.	→ permits programmer to be less concern with c, pascal strongly
→ can catch many Common Errors.	→ Mandatory in some situations such as array; bounds check.
→ Desirable when faster Execution importance	→ more robust and clearer code

Eg: pascal, c-type

→ Type checking have been used to improve the Security of System.

Rules for Type checking:-

Type checking has two forms

i) Synthesis

ii) Interference

i) Type Synthesis :-

→ It derives the expressions from the types of its subexpressions.

→ It must be declared before they are used.

Ex:- The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2.

If f has type s → t and x has type s, then expression f(x) has type t

Ex:-
```
add(int a, float b)
  {

  }
fn1(float c, int d) → t
  {
    add(2,2.5)
  }
```

$\boxed{\text{add(float, int)}}$ [∵ int changes to float, float changes to int]

→ Here f and x denote expression, s→t denote a function from s to t.

→ This rule for functions with one argument. Carries over to functions with several arguments.

interference
ii) Type inference :-

→ It generally determines the type of language construct from the way it is used.

→ Ex:- $E_1 + E_2$ i.e., $\underset{int}{2} + \underset{int}{5}$ = Datatype will be int

$\underset{(string)}{abc} + \underset{(string)}{abc}$ = Datatype will be string

→ There is no need to declare variables

→ Type inference are used in meta languages.

If f(x) is an expression then for some α and β, f has a type α → β and x has type α

Type Conversion or type Casting:-

→ A type cast is basically a Conversion from one type to another.

→ There are two types of Conversions

1) Implicit type Conversion
2) Explicit type Conversion

1) Implicit type Conversion:- (smaller to bigger)

→ If a compiler Converts one data type into another type of data automatically.

→ There is no data loss

Ex:-
```
short a = 20;
      int b = a ; // Implicit Conversion
```

Assign:- bool → char → short int → int → long → floa

2) Explicit type Conversion :-

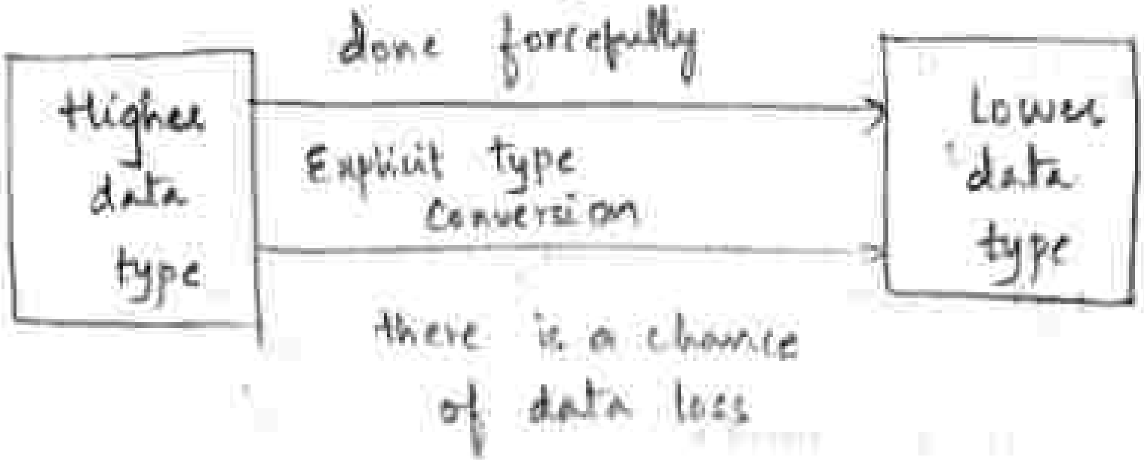
→ when data of one type is Converted explicitly to another type with the help of predefined functions.

→ There is a data loss.

→ Conversion done forcefully

→ Some Conversions cannot be made Implicitly

int to short (∵ int range is more than short so there is a chance of data loss)



Ex:- $t_1 = (\text{float})2$

$t_2 = t_1 * 3.14$

Ex:-
```
if (E1.type = integer and E2.type = Integer)
                    E.type = integer;
else if (E1.type = float and E2.type = Integer) ---
                    E.type = float;
  - - - -
```

→ two Conversions i) Widening Conversions

ii) Narrowing Conversions.

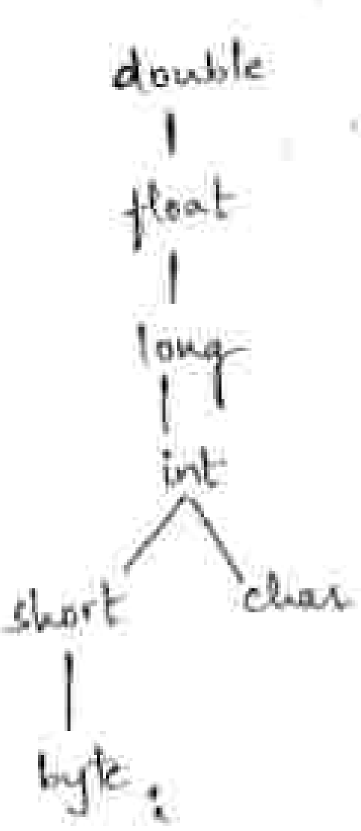
→ widening conversions generally preserve information

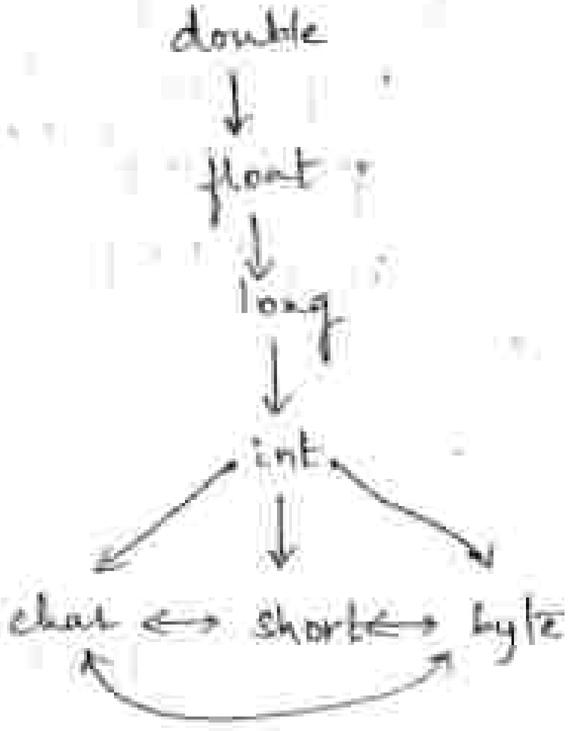
→ narrowing conversions generally lose information

→ widening rules – any lower can be widened to higher type.

→ a char can be widened to int or float but char cannot be widened to short.

→ narrowing rules – a, type s can be narrowed to type t if there is a path from t.





1) $\max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum of two types in widening hierarchy.

2) widen(a, t, w) generates type conversions if needed to widen the contents of an address a of type t into a value of type w.

```
Addr{ widen( Addr a, Type t, Type w)
    if (t=w) return a;
  else if (t = integer and w= float) {
        temp = new Temp();
        gen('temp'=' '(float)' a);
        return temp;
    }
     else error;
 }
```

→ Semantic Action $E \to E_1 + E_2$

```
E→E1+E2 { E.type = max (E1.type, E2.type);
          a1 = widen(E1.addr, E1.type, E.type);
          a2 = widen (E2.addr, E2.type, E.type);
        E.addr = new Temp();
        gen( E.addr '=' a1 '+' a2); }
```

Overloading of Functions and operators:-

An overloaded Symbol has different meanings depending on its context. overloading is resolved when a unique meaning is determined for each occurence of a name.

Ex:- The + operator in Java denotes either string concatenation or addition, depending on the type of its operands.

```
void err() {---}
void err(String s) {- -}
```

Intermediate code for switch statements (or) Three address code translation of switch statements

Switch statement syntax:

```
switch (E)
{
 case V1 : S1
 case V2 : S2
   ....
 case Vn-1 : Sn-1
 default : Sn
}
```

Eg:-

```
switch (x+y)
{ case1 : a=a+2;
         break;
  case4 : b=b*5;
         break;
  case6 : c=c/2;
         break;
  default : d=d-2;
         break;
}
```

Translation of switch statement:

```
Code to evaluate E into t
 goto test
 L1 : code for S1
      goto next
 L2 : code for S2
      goto next
      ----
      ----
 Ln-1 : code for Sn-1
        goto next
 Ln : code for Sn
      goto next
test : if t = V1 goto L1
       if t = V2 goto L2
       ----
if t = Vn-1 goto Ln-1, goto Ln
next:
```

Three address code:-

```
if t = Vn-1 goto Ln-1
goto Ln
next:
1. t1 = x+y
2. If (t1 = 1) goto 8
3. If (t1 = 4) goto 11
4. If (t1 = 6) goto 14
5. t2 = d-2
6. d = t2
7. goto Next
8. t3 = a+2
9. a = t3
10. goto Next
11. t4 = b*5
12. b = t4
13. goto Next
14. t5 = c/2
15. c = t5
16. goto Next
17. Next
```

```
if $t_1 = V_{n-1}$ goto $L_{n-1}$
    goto Ln
  next:
```

Intermediate code for procedures:- (A) Three Address code

```
D → define T id (F) {S}
F → ε | T id, F
S → return E;
E → id (A);
A → ε | E, A

float add( )
        (int a)
        (int a, int b)
  {
     return add( );
  }
```

→ S → adds stmts that returns the value of an expression.

→ E → adds function calls with actual parameters A.

Non-terminals D and T generates declarations and types.

→ Function definition generated by D consists of keyword define, a return type, the function name, Formal parameters in parenthesis and function body consisting of statements.

→ Non-terminal F generates zero or more formal parameters.

where Formal parameters consist of a type followed by identifier.

→ Non-terminal S & E generate statements and expressions.

→ In three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call followed by call itself and the parameters are passed by value.

Eg:- If the given function is in the form of

```
P(A1, A2, A3, --- An)
 param A1
 param A2
   |
 param An
 call P,n
```

P → is function name.

n → no. of arguments.

Eg:- n = f(a[i]);

Translated into three-address code as follows

1) $t_1 = i * 4$

2) $t_2 = a[t_1]$

3) param t_2

4) t_3 = call f, 1

5) $n = t_3$.

→ The first 2-lines compute the value of expression a[i] into temporary t_2,

→ line 3 makes t_2 an actual parameter for the call on line 4 of f with one parameter.

→ line 5 assign the value returned by the function call to t_3.

→

Functions types:-

→ The type of function must encode the return type and types of the return type and the types of the Formal parameters.

→ Let "void" be a special type that represent no parameter or no return type.

→ whenever the function is called the function name is name entered into the symbol table for use in the rest of the program.

→ The Formal Parameters are stored in the Activation Record. For storing formal parameters the Activation Records are used

Eg2:-

```
void main()
{
 int x, y
  ---
  ---
 swap(&x, &y);
}
void swap(int *a, int *b)
{
 int t;
 t=*b;
 *b=*a;
 *a=t;
```

Three address code

```
1. call main
2. param &x
3. param &y
4. call swap, 2
```

Eg3:-

```
float add() & float add(int a)
  float add(int a, float b)
  {
  return add() & return add(x),
     return add (x,y);
  }
```

formal paramts

Actual paramts

UNIT - V

Machine - Independent optimizations:- The principal sources of optimization, Introduction of Data-flow analysis, Foundations of Data-flow analysis, constant propagation, partial Redundancy elimination, Loops in flow Graphs.

Optimization is the process of transformation of code to an efficient code. Efficiency is in terms of space requirement and time for its execution without the changing the meaning of the given code.

The following constraints are to be considered while applying the techniques for code improvement.

① The transformation must preserve the meaning of the program, that is, the target code should ensure semantic equivalence with source program.

② Program efficiency must be improved by a measurable amount without changing the algorithm used in the program.

③ When the technique is applied on a special format, then it is worth transforming.

Optimization can be classified as

1) Local optimizations:-

Optimizations performed within a single basic block are termed as local optimizations. These techniques are simple to implement and does not require any analysis since we do not require any information relating to how data and control flows.

(2) Global optimizations:-

Optimizations performed across basic blocks is called global optimization. These techniques are complex as it requires additional analysis to be performed across basic blocks. This analysis is called data-flow analysis.

Optimization techniques can be applied to intermediate code or on the final target code. It is a complex and a time-consuming task that involves multiple sub phases, sometimes applied more than once.

- * most compilers allow the optimization to be turned off to speed up compilation process.
- * To apply optimization it is important to do control flow analysis and data flow analysis followed by transformations.

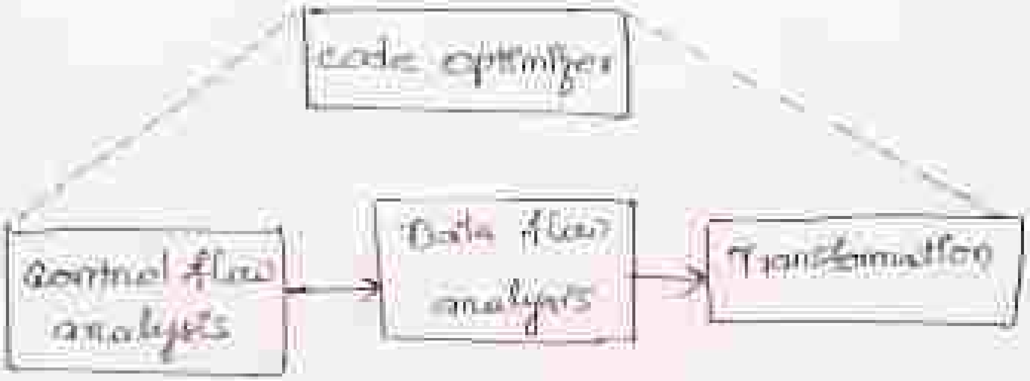


Fig: code optimization model

Control flow Analysis:— It determines the control structure of a program and builds a control flow graph.

Data flow Analysis:— It determines the flow of scalar values and builds data flow graphs. The solution to flow analysis propagates data flow information along a flow graph.

Transformation:— Transformations help in improving the code without changing the meaning or functionality.

Flow Graph:—

A graphical representation of three address code is called flow graph. The node in the flowgraph represent a single basic block and the edges represent the flow of control. These flow graphs are useful in performing the control flow and data flow analysis and to apply local or global optimization and

Code generation

Basic Block:- A basic block is a set of ~~[illegible]~~ consecutive statements that are executed sequentially. once the control enters into the block then every statement in the basic block is executed one after the other before leaving the block.

Ex: For the statement $a = b + c * d/e$ the corresponding ~~■~~ set of three address code is.

$t_1 = c * d$
$t_2 = t_1/e$
$t_3 = b + t_2$
$a = t_3$

All these statements correspond to a single basic block.

Procedure to Identify the Basic Blocks.

Given a three address code, first identify the leader statements and group the leader statements with the statements up to the next leader.

* To identify the leader use the following rules:

1. First statement in the program is a leader.
2. Any statement that is the target of a conditional or unconditional statement is a leader statement.
3. Any statement that immediately follows a conditional / unconditional statement is a leader statement.

Example:- Identify the basic blocks for the following code fragment

```
    main()
    {  int i=0, n=10;
       int a[n];
       while ( i<=(n-1))
       {   a[i] = i * i;
           i = i+1;      }  return; }.
```

The three address code for initializing function is as follows;

(1) i := 0 ⟶ leader1 using rule 1
(2) n := 10
(3) t_1 := n-1 ⟶ leader2 using rule 2
(4) if i > t_1 goto (12)
(5) t_2 := i*i ⟶ leader3 using rule 3
(6) t_3 := 4*i
(7) t_4 := a[t_3]
(8) t_4 := t_2
(9) t_5 := i+1
(10) i := t_5
(11) goto (3)
(12) return. ⟶ leader4 using rule 3.

```
1) i=0
2) n=10
```

B1 — Basic Block1 includes statements (1) and (2)

```
(3) t1 := n-1
(4) if i>t1 goto (1
```

B2 — Basic block2 includes statements (3) and (4)

```
(5) t2 = i*i
(6) t3 = 4*i
(7) t4 = a[t3]
(8) t4 = t2
(9) t5 = i+1
(10) i := t5
(11) goto (3)
```

B3. — Basic block3 includes statements (5) - (11)

```
(12) return
```

B4 — Basic block4 includes statement (12)

Fig: Basic Blocks.

Flow Graph :-

Flow graph shows the relation between the basic block and its preceeding and its successor blocks. The block with the first statement is B1. An edge is placed from block B1 to B2, if block B2 could immediately follow B1 during execution or satisfies the following conditions.

→ The last statement in B1 is either conditional or unconditional jump statement that is followed by the first statement in B2 or

→ The first statement in B2 follows the last statement in B1 and is not an unconditional / conditional jump statement

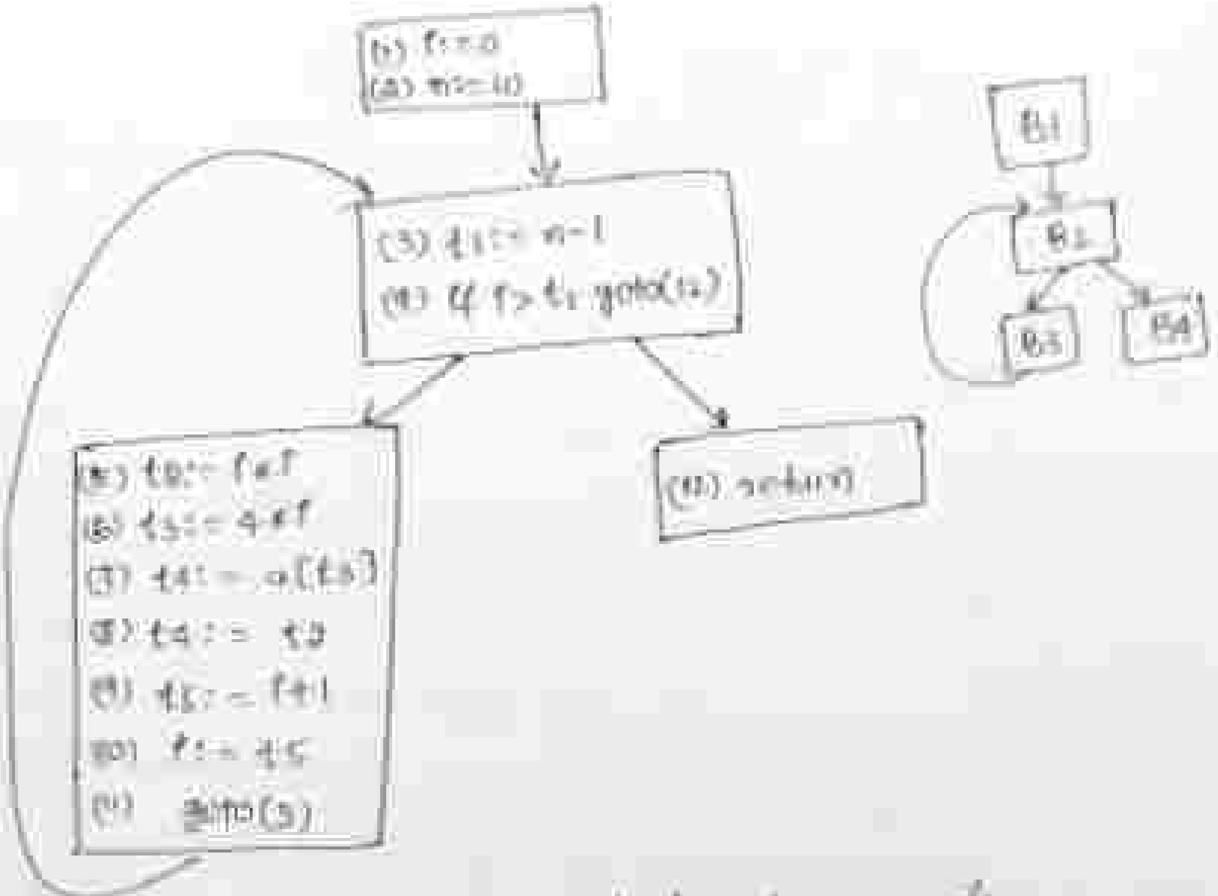


Fig: flow graph for above example.

DAG representation of Basic Block

A DAG is a useful data structure for implementing transformations within a basic block. It gives pictorial representation of how values computed at one statement are useful in computing the values of other variables.

* It is useful in identifying common sub-expressions within a basic block.

* A DAG has nodes, which are labeled as follows

① The leaf nodes are labeled by either identifiers or constants. If the operators are arithmetic then it always requires two r-value

② The labels of interior nodes correspond to the operator symbol.

Note: The DAG is not the same as a flow graph. Each node in a flow graph is a basic block, which has a set of statements that can be represented using DAG.

=

Construction of DAG

To construct DAG, we process each statement of the block. If the statement is a copy statement, that is, a statement of the form a=b, then we do not create a new node, but append label a to the node with label b.

* If the statement is of the form a=b op c, then we first check whether there is a node with same values as b op c, if so, we append the label a to that node. If such node does not exist then we first check whether there exists nodes for b and c which may be leaf nodes or an internal nodes if recently computed, then create a new node for 'op' and add it the left child b and the right child as 'c'.

* Label this new node with a. This would become the value of a to be used for next statements; hence, we mark the previously marked nodes with a as a_0.

=

Example :- Let us consider basic block B1 to B4 and construct DAG for each block step by step

B1

```
(1) f := 4
(2) n := 10
```

B2

```
(3) t1 := n-1
(4) if f>t1 goto(12)
```

B3

```
(5) t2 := f*f
(6) t3 := t2*f
(7) t4 := a[t3]
(8) t4 := t1
(9) t5 := f+1
(10) f := t5
(11) goto (3)
```

B4

```
(12) return.
```

For Block B1

Fig: DAG for Block B1

* For the statement in B1, first we create a leaf labeled 4 and attach identifier f to it

For Block B2

For first statement, which we first create nodes for n and 1, then create node for operator (−) and label it as t1.

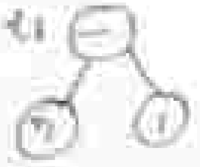
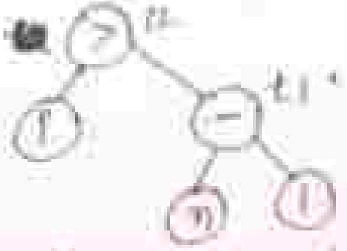


Fig (a)

DAG for first statement in Block B2.



Fig(b): DAG for block B2

* Fig(b) shows the, second statement, which is a conditional statement, we create a node for operator > and label it as t2 as when this condition is satisfied it should go to statement 12.

For Block B3

* For the first statement, we create nodes for f and use as right child, then create node for operator (*) and label it as t2.

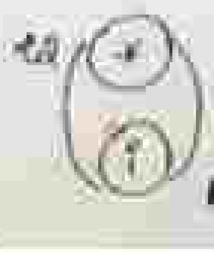


Fig: (c) DAG for Block B3.

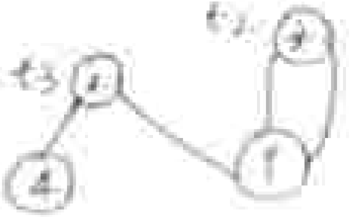
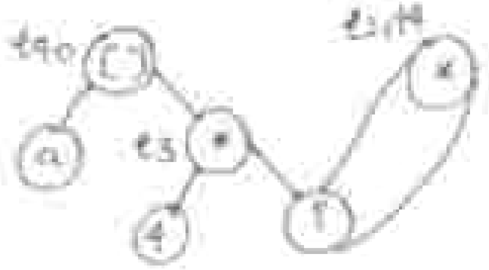
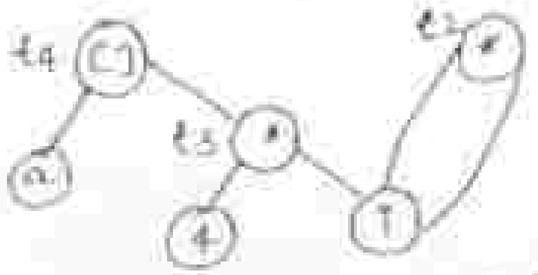


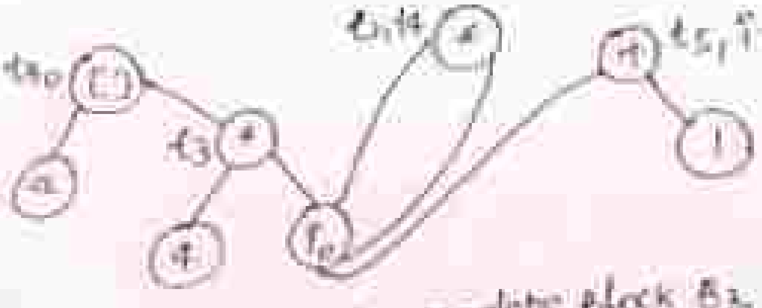
Fig: DAG for two statements in block B3.



Fig(c): DAG for three statements in block B3.



Fig(d): DAG for four statements to block B3.



Fig(e): DAG for complete block B3.

Principle Sources of optimization

A transformation of a program is called local if it is applied within a basic block and global if applied across basic blocks.

* there are different types of transformations to improve the code and these transformations depend on the kind of optimization required

① Function - preserving transformations are those transformations that are performed without changing the function it computes.

* These are primarily used when global optimizations are performed.

② Structure - preserving transformations are those that are performed without changing the set of expressions computed by the block.

* many of these transformations are applied locally.

③ Algebraic transformations are used to simplify the computation of expression set using algebraic identities.

* These can replace expensive operations by cheaper ones, for instance multiplication by 2 can be replaced by left shift.

Function preserving Transformations.

The following techniques are function-preserving transformations

1) common sub-expression Elimination
2) copy propagation
3) Dead code elimination
4) constant propagation.

(1) common sub-expression Elimination :-

An expression E is said to be common sub expression if E is computed before and the variables in the expression are not modified since its ~~comp~~ computation. If such expression is present then the re-computation can be avoided by using the result of the previous computation

* This technique can be applied both locally and globally

* We need to maintain a table to store the details of expressions evaluated so far and use this information to identify and eliminate the re-computation of the same expression.

* The common sub-expression elimination can be done within basic block by analyzing and storing the information of expression in the table until the operands in the expression are redefined

* If any operand in the expression is redefined, then Remove the expression from the table

```
c = a+b
d = m*n
e = b+d
f = a+b
g = -b
h = b+a
a = j+a
k = m*n
j = b+d
a = -b
if m*n go to L
```

⟹

```
t1 = a+b
c = t1
t2 = m*n
d = t2
t3 = b+d
e = t3
f = t1
g = -b
h = t1
a = j+a
k = t2
j = t3
a = -b
if t2 go to L
```

Program code before and after common sub expression elimination

* The above figure shows the optimized code on applying common sub expression elimination technique locally

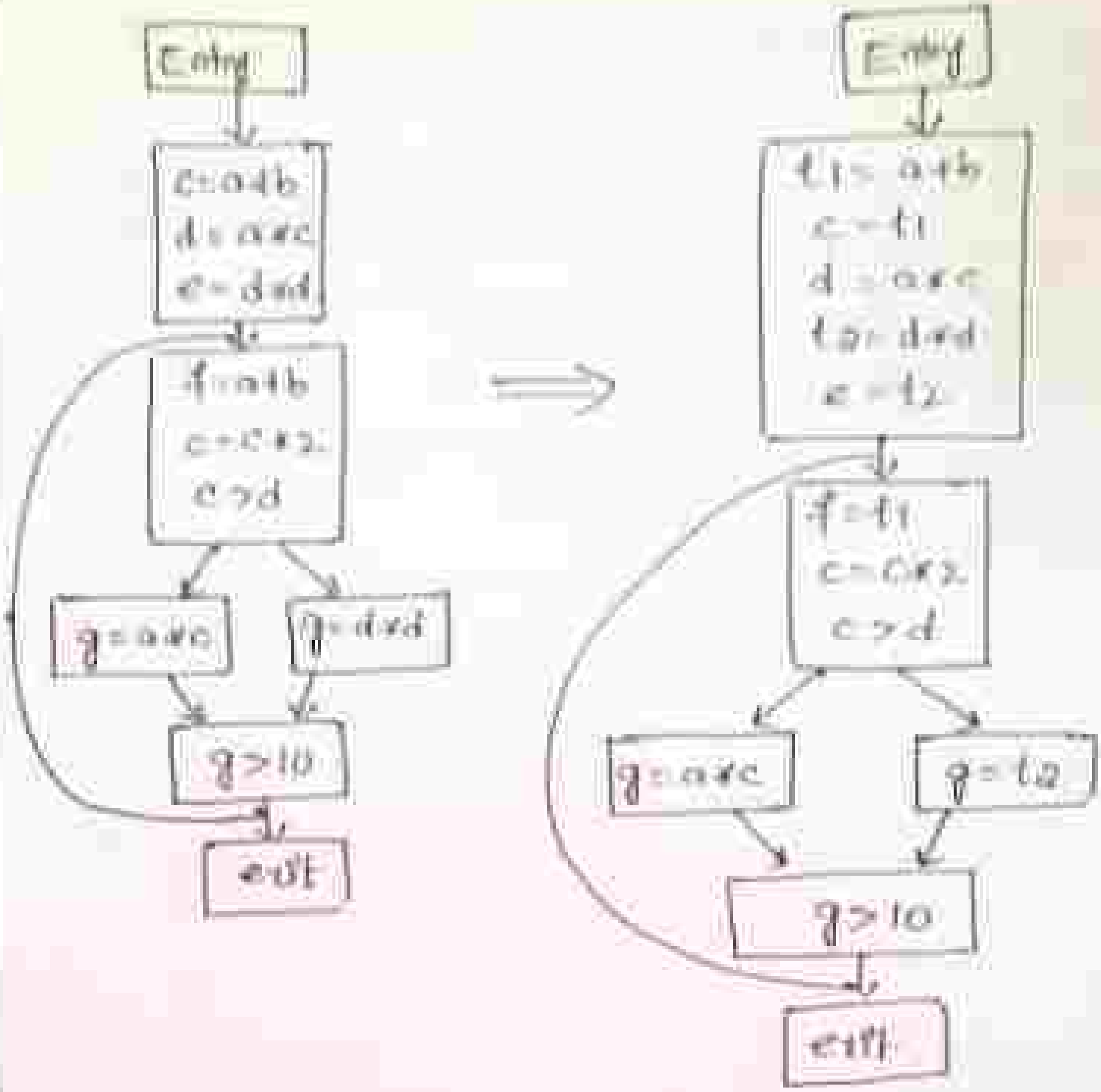


Fig: Example for global common sub expression Elimination

The above example shows the common-sub expression elimination globally by replacing the evaluated expression with a new temporary variable t1 and t2 and these variables are used whenever the same expression is used.

(ii) Copy propagation:-

A copy statement is a statement that is in the form a=b.

* copy propagation technique is applied to replace the later use of variable 'a' with the use of 'b' if original values of 'a' do not change after this assignment

This technique can be applied locally and globally.

- This optimization reduces runtime stack size and improve execution speed.
- To implement this, we need to maintain a separate table called copy table, which holds all copy statements. While traversing the basic block, if any copy statement is found, add this information into the copy table.
- While processing any statement, check the copy table for variable a, which can be replaced variable b.
- If there is an assignment statement that computes the value of a then remove the copy statement from the copy table.
- Copy propagation helps in identifying the dead code.

Example:- Let the basic block contain the following set of statements.

y = x
z = y+1
W = y
Y = W+z
Y = W

Statement NO	Instruction	Updated Instruction	copy table content
1.	y = x	y = x	{(y,x)}
2.	z = y+1	z = x+1	{(y,x)}
3.	W = y	W = x	{(y,x),(W,x)}
4.	y = W+z	y = x+z	{(W,x)}
5.	y = W	y = W	{(W,x), (y,x)}.

- In the first statement 1, since it is a copy statement, the information is added in to the copy table.
- Statement 2 since the value x [illegible] from y, hence get(y, table) would return x and the statement is modified

Replacing y with x.

- The third statement is a copy statement and since y is to replace x, we insert into the copy table (x) to be replaced by x
- When statement 4 is processed, y value is different and hence, details of y are removed from the copy table.
- The fifth statement is a copy statement and is inserted into the copy table.

(iii) Dead code elimination :-

Code that is unreachable or does not affect the program is said to be dead code. Such code requires unnecessary CPU time, which can be identified and eliminated using this technique.

Example program :-

```
1)  int var1;
2)  void sample()
3)  {
4)    int i;
5)    i = 1;  /* dead store */
6)   var1 = 1; /* dead store */
7)   var2 = 2;
8)   return;
9)   var1 = 3; /* unreachable */
10) }
```

The code fragment after dead code elimination

```
int var1;
void sample()
{
    var1 = 2;
    return;
}.
```

→ Here the value of i is never used, and the value assigned to var variable in statement 6 is dead store and the statement 9 is unreachable. These statements can be eliminated as they do not affect the program execution.

(IV) constant propagation :-

Constant propagation is an approach that propagates the constant value assigned to a variable at the place of its use.

For example, given an assignment statement x=c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.

* This approach is similar to copy propagation and is applied at first stage of optimization. This method can analyze by propagating constant value in conditional statement, to determine whether a branch should be executed or not, that is, identifies the dead code.

Example: Let us consider the following example.

```
pi = 22/7
void area_per(int r)
{
   float area, perimeter;
    area = pi*r*r;
    perimeter = 2*pi*r;
     print area, perimeter;
  }
```

In this example, we can notice some simple constant propagation results, which are as follows

→ In line 1 the variable pi is constant and its value is the result of 22/7, which can be computed at compile time and has the value 3.413.

→ In line 5 the variable pi can be replaced with the constant value 3.413.

→ In line 6 since the value of pi is constant, the partial result of the statement can be computed and the statement can be modified as perimeter = 6.285*r;

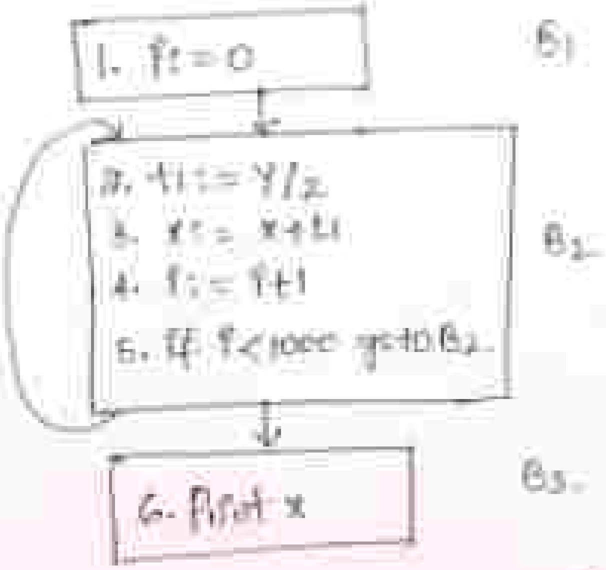
Loop optimization Techniques:

In most of the programs, 90% of execution time is inside the loops; hence loop optimization is the most valuable machine - independent optimization and are good candidates for code replacement.

Let us consider the example of a program that has a loop statement

```
for (int i=0; i<1000; i++)
    x= x+y/z;
print x;
```

The optimized code of this program is represented in flow graph as



Flow graph After code optimization.

The total number of statements that are executed are as follows

Statement Number	Frequency of Execution	Total No. of Statements
1	1	1
2-5	1000	4,000
6	1	1

On the whole, the number of statements that are executed are 4002. Instead if the code is written as follows then the total number of statements that are executed are only 1002.

```
1.    t1 = y/2
2.    x = x + t1
3.    x = x + t1
4     x = x + t1
5.    ...
6.    ..
1001. x = x + t1
1002. print x
```

The execution is three times better for later than the first form, but the space requirement is more. Inefficient code is generated in the loops for various reasons like

→ Induction variable usage to keep track of iteration

→ Unnecessary computations made inside the iterative loops that are not effective.

→ Use of high strength operators inside the loop.

The important loop optimizations are

① Elimination of loop invariant computations

② Strength reduction

③ Code motion

④ Elimination of induction variables

(1) A loop invariant computation:—

A loop invariant computation is one that calculates the same value every time the statements in loop are executed. & moving such loops [illegible] computational statement outside the loop leads to a reduction in the execution time.

Algorithm for elimination of loop invariant code

Step1: Identify loop-invariant code

Step2: An instruction is loop-invariant if, for each operand:

(i) The operand is constant, OR

(ii) All definitions for all that operands that reach the instruction inside the loop are made outside the loop, OR

(iii) There is exactly one definition made using loop invariant variables inside the loop for an operand that reaches the instruction.

Step3:- move it outside the loop

(i) move each instruction I to newly created preheader if and only if it satisfies these requirements of the loop.

Example:-

```
for (int i=0; i<1000; i++)
      x = x + y/z;
print x;
```

In the above example, the value y and z remains unchanged as there is no definition for these variables in the loop hence, the computation of y/z remains unchanged, which can be moved above the loop and the same code can be rewritten as follows

```
t1 = y/z;
for (int i=0; i<1000; i++)
  x = x + t1;
print x;
```

(3) Induction Variables :—

Induction variables are those variables used in a loop, their values are in lock-step, and hence, it may be possible to eliminate all except one.

* There are two types of induction variables — basic and derived.

* Basic induction variables are those that are of the form

$$\boxed{I = I \pm c}$$

where I is loop variable and C is some constant.

* Derived induction variables are those that are defined only once in the loop and their value is linear function of the basic induction variable.

For example :—

$$\boxed{J = A * I + B}$$

Here J is a variable that is dependent on the basic induction variable I and the constants A and B. This is represented as a triplet (I, A, B)

* Induction variable elimination follows three steps.

i) Detecting induction variable.

ii) Reducing the strength of induction variable.

3) Eliminating induction variable.

Induction Variable Elimination :—

Some loops contain two or more induction variables that can be combined into one induction variable.

Example:- The code fragment below has three reduction variables (i1, i2 and i3) that can be replaced with one reduction variable, thus eliminating two reduction variables.

```
int  a[SIZE];
int  b[SIZE];
void f(void)
{
  int i1,i2,i3;
  for (i1=0, i2=0, i3=0; i1< SIZE; i1++)
      a [i2++] = b[i3++];
      return;
}.
```

After reduction variable elimination the code is rewritten as/

```
int a[SIZE]
int b[SIZE]
void f(void)
{ int i1;
  for (i1=0; i1< SIZE; i1++)
      a[i1] = b[i1];
}. return;
```

* Reduction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both runtime performance and code space.

Machine - Dependent optimization

This optimization can be applied on target machine instructions. This includes register allocation, use of addressing modes, and peep hole optimization.

* Instructions involving register operands are faster and shorter; hence, if we make use of more registers during target code generation, efficient code will be generated.

Peephole optimization:-

Generally code generation algorithms produce code, statement by statement. This may contain redundant instructions and suboptimal constructs. The efficiency of such code can be improved by applying peephole optimization, which is simple but effective optimization on target code.

- The peephole is considered a small moving window on the target code. The code in peephole need not be contiguous. It improves the performance of the target program by examining and transforming a short sequence of target instructions.
- The advantage of peephole optimization is that each improvement applied increases opportunities and shows additional improvements. It may need repeated passes to be applied over the target code to get the maximum benefit.

(1) Redundant loads and stores

The code generation algorithm produces the target code, which is either represented with single operand or two operands or three operands.

- Let us assume the instructions are with two operands. The following is an example that gives the assembly code for the statement

$x = y + z$.   1. MOV y, R_0
2. ADD z, R_0
3. MOV R_0, x

Instruction 1 moves the value of y to Register R_0. Second instruction performs the addition of value in z with the register instruction performs the addition of value in z with the register content and the result of the operation is stored in the register.

- The third instruction copies the register content to the location x. At this point the value of x is available in both location of x and the register R_0.

If the above algorithm is applied on the code a=b+c, d=a+e then it generates the code given below:

1. Mov b,R_0
2. ADD c,R_0
3. mov R_0,a
4. mov a,R_0
5. ADD e,R_0
6. mov R_0,d

Here we can say that 3 and 4 are redundant load and store instructions. These instructions will not affect the values before or after their execution. Such redundant statements can be eliminated and the resultant code is as follows:

1. mov b,R_0
2. ADD c,R_0
3. ADD e,R_0
4. mov R_0,d

(ii) <u>Algebraic simplification</u>:-

There are few algebraic identities that occur frequently enough and are worth considering. Look at the following statements.

x := x+0
x := x+1

They do not alter the value of x. If we keep them as it is, later when code generation algorithm is applied on it, it may produce six statements that are of no use. Hence, such statements whether they are in three address code or target code can be removed.

(iii) Deadcode Elimination:-

Removal of unreachable code is an opportunity for peephole optimization. A statement immediately after an unconditional jump in a statement that never get a chance to be executed can be identified and eliminated. Such code is called the dead code.

Ex:
```
# define x=0
 ....
 if(x)
 {
   -  print value

 } --
```

If this is translated to target code as

```
If x=1 goto L1
       goto L2
L1: print value
L2: ....
```

then, value will be never printed. So whatever code inside the body of "if(x)" is dead code; hence it can be removed.

(iv) Reduction in Strength:-

This optimization mainly deals with replacing expensive operations by cheaper ones for example

→ $x^2 \Rightarrow x*x$.
→ fixed-point multiplication and division by a power of 2.
⇒ shift
→ floating point division by a constant ⇒ floating-point multiplication by a constant.