Compiler Design

- Basics of a compiler
- Lexical Analysis
- Syntax Analysis
- Syntax Directed Translation
- Intermediate code generator
- Code Optimization
- Runtime Environment

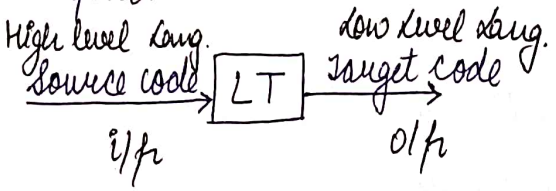
Marks : 4 to 9

Reference
- Ullman

SECTION 1 : Basics of Compiler

Language Translator

It takes one language as input and produces another language as output.



Types of Language Translator

1. Compiler

Compiler takes the source code as input and produces the target code as output,

If this target code is an executable code then it will be called by the user to process the inputs for producing the output.

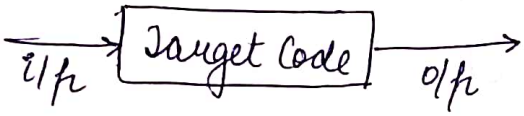
The compiler executes entire code at a time, if any error occurs at any line all of them will be given/ produce at a time.

Error diagonism in compiler is not as easy compared to interpreter but output producing in compiler is faster.

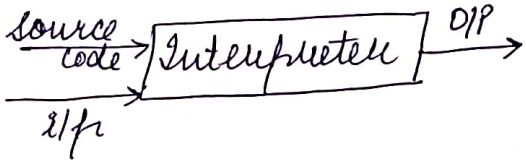
Compiler is an offline process.

The end user can't make the modification easily in the source program as an executable code will be given to the end user.

Ex : C, C++, C#, Pascal.



2. Interpreter



It takes the source code as input and produces the direct output.

It will not produce any intermediate language as in the case of compiler.

The interpreter executes the code line by line. If any error present at any line, immediately that error will be produce to the user, until the user resolves that error the interpreter will not move to the next line.

Error diagonism in interpreter is easy but output generation by interpreter is slower than the compiler.

The interpreter executes the source code and simultaneously it process the inputs also. That is why interpreter is an online process.

If the inputs are changed the interpreter executes the source program again.

The end user can make the modifications easily in the source program, as the entire source program is given to end user.

Ex: LISP, Python, PERL, ROBY

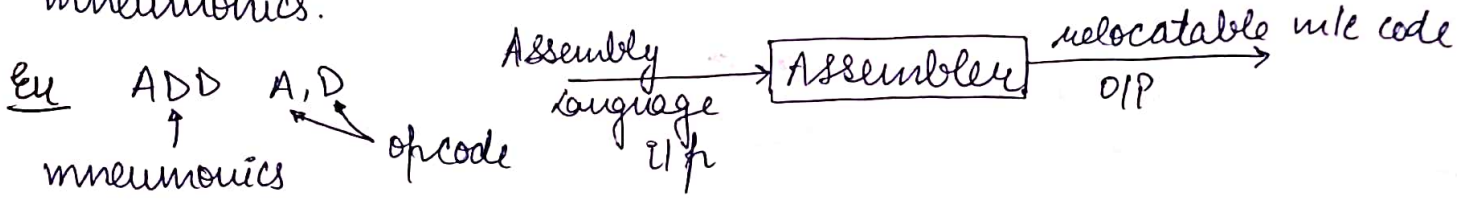
3. Assembler

It takes assembly language code as input and produces relocatable machine code as output which can be loaded in main memory which is ready for execution

Assembler is a compiler of assembly language.

Assembly language uses opcode for its instruction.
Opcode basically gives the information about the instruction.

The symbolic representation of opcode is called as mnemonics.



Based on the pass structure assembler can be divided into two types

i) One-Pass Assembler

The whole conversion of assembly language into machine code will be done in one step.

ii) Two-Pass Assembler / Multipass Assembler

These assemblers convert the assembly language code into machine code in two steps:

- In first step, they create symbol table, opcode table and location counter
- In second step, using the values in this table assembly language code into machine code.

Symbol Table

Symbol table stores the variables or symbol table is used to store the variables and their attributes.

Opcode Table

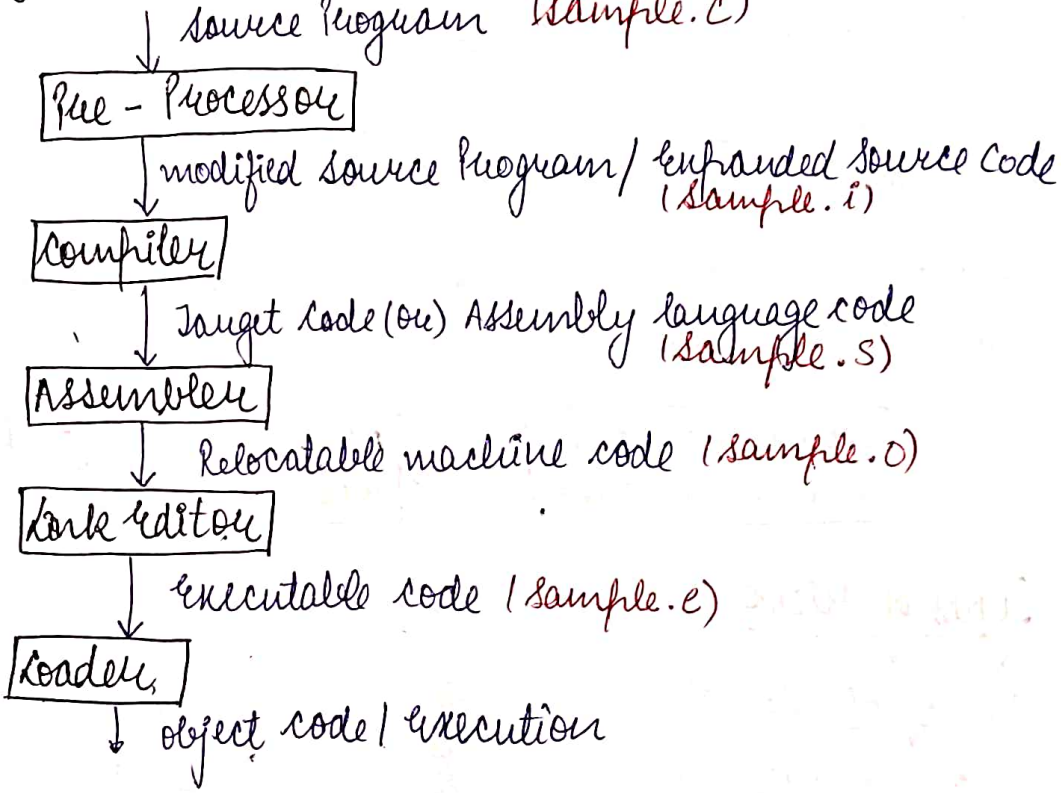
It is used to store the mnemonics and their corresponding values.

Location counter

It keeps the track of information about the location of where the current instruction will be stored.

⇒Assembly is a machine dependent i.e. the assembler which is designed for one machine will not works for another machine.
Ex: GAS, GNU.

Language Processing System



(#) Pre-Processor

It takes the source code as input and produces expanded source code as output if the program is to large it may be divided into small programs (modules) and will be stored in different file. It is optional.

- The Pre-Processor will take care of all theses modules.
- The commands used in the Pre-Processor are called pre-processor directives and they begin with symbol "#"

Example for pre processors:
#include, #define, #if, #elif, #ifdef, #ifndef, #else, #error, #undef, #pragma etc.

#FILE INCLUSION

Syntax: #include <file name>

The pre processor delete all #include statements by placing the code for the file name at the specified location. This is called file inclusion.

#MACRO

If any set of instruction can repeatedly used in the program then instead of repeating several times we can define them as a micro and a name will be given to it.

Whenever these instruction are to be repeated we will call by its micro name. These micro calls are expanded before compilation

Micro processor is a system software which does the job of micro expansion.

Types of Micro

1. Object like Macro

Eg
```
#define a,10
#undefine a
#define a,15
```

2. Function like macro

Eg
```
# define add (a,b) a+b
add (5,10)
```

• define concate (x,y) x##y
→ Invocation of macro → Result of Macro
concate (Node easy) (Node easy)

CONDITIONAL DIRECTIVES

```
if def a
#
#else
# define a,10
#end if
```

[#if def #if #else #endif
#ifndef #elif #error]

⊙ Pragma: It is used for calling a function before and after a main function.

Compiler

Compiler takes the expanded source code as input and produces assembly language code as output.

→ Compilation is an optional.

Eg: HTML, JAVA Script they do not required compilation

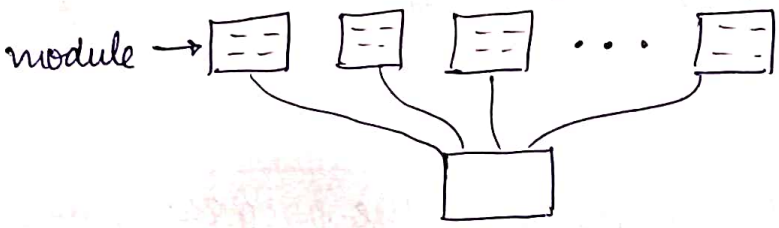
Assembler

It takes the target code as input and produces relocatable machine code as output which has to be loaded in main memory

Link editor

If the program is too large it may be divide into small programs (modules) and will be stored it different files.

- At the time of execution all these modules must be link as a single module.
- This will be done by link editor.
- The link editor also resolves the external references
  If a code in one file refers a location in another file is called as external reference.



- The high level languages will have some built in libraries and header files.
- A program in high level language may contain some library function whose definitions are stored in built in libraries like link editor links these functions to their definition.

- If the corresponding definition is not found in built in libraries then the link editor informs to the compiler generates an error.

Types of link editor

1. Static Linking
   This linking is done before execution of the program.
2. Dynamic Linking
   This will be done at the time of execution of the program.

Loader

Loader loads the executable program from secondary memory into main memory and initiates the execution of the program by transferring the control to the starting instruction of the program

Functions of the loader

1. Allocation
2. Realocation
3. Linking [Optional]
4. Loading

- Allocation

The loader allocates the required space in the main memory for the executable program.

- Relocation

There are some address dependent locations in the program, such address constants must be place according to the specified location.

Relocation means it is a mapping from Virtual address space to physical address space.

Relocation means allocating the address different from originally specified.

There are two types.

1. Static Relocation
which can be done at the time of loading of the program.

2. Dynamic Relocation
which can be done at executing of the program i.e. at run time.

- **Loading**

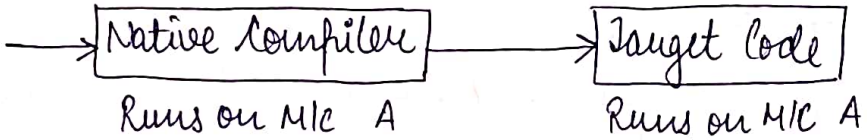
The loader loads the executable program into main memory.

#Type of compiler:

1. Native compiler
2. Cross compiler
3. Incremental compiler
4. Source to source compiler
5. Load and Go compiler
6. Byte code compiler
7. Just in time (JIT) compiler.

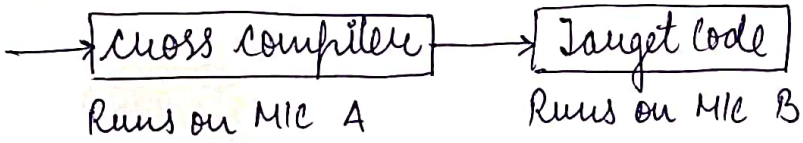
- **Native Compiler**

It is a compiler which produce the output target code which works on same machine where the native compiler runs on.



- **Cross Compiler**

Cross compiler works on one machine or runs on one machine and produce the output the target code which runs on different machine.

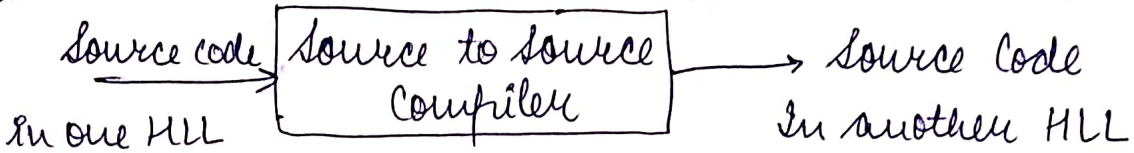


- **Incremental Compiler**

It is a compiler which executes only modified source code rather than executing the entire code.

- **Source to Source Compiler**

It converts one high level language into another high level language.



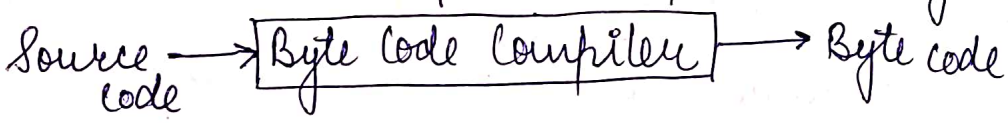
It takes one high level language as input and takes another high level language as output.

- **Load and go compiler**

It take source code as input and directly executes the program i.e. compiling, assembling, linking and loading will be done by load and go compiler.

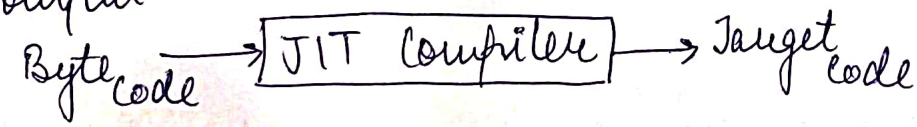
- **Byte Code Compiler**

It takes source code as input and produces byte code as output.

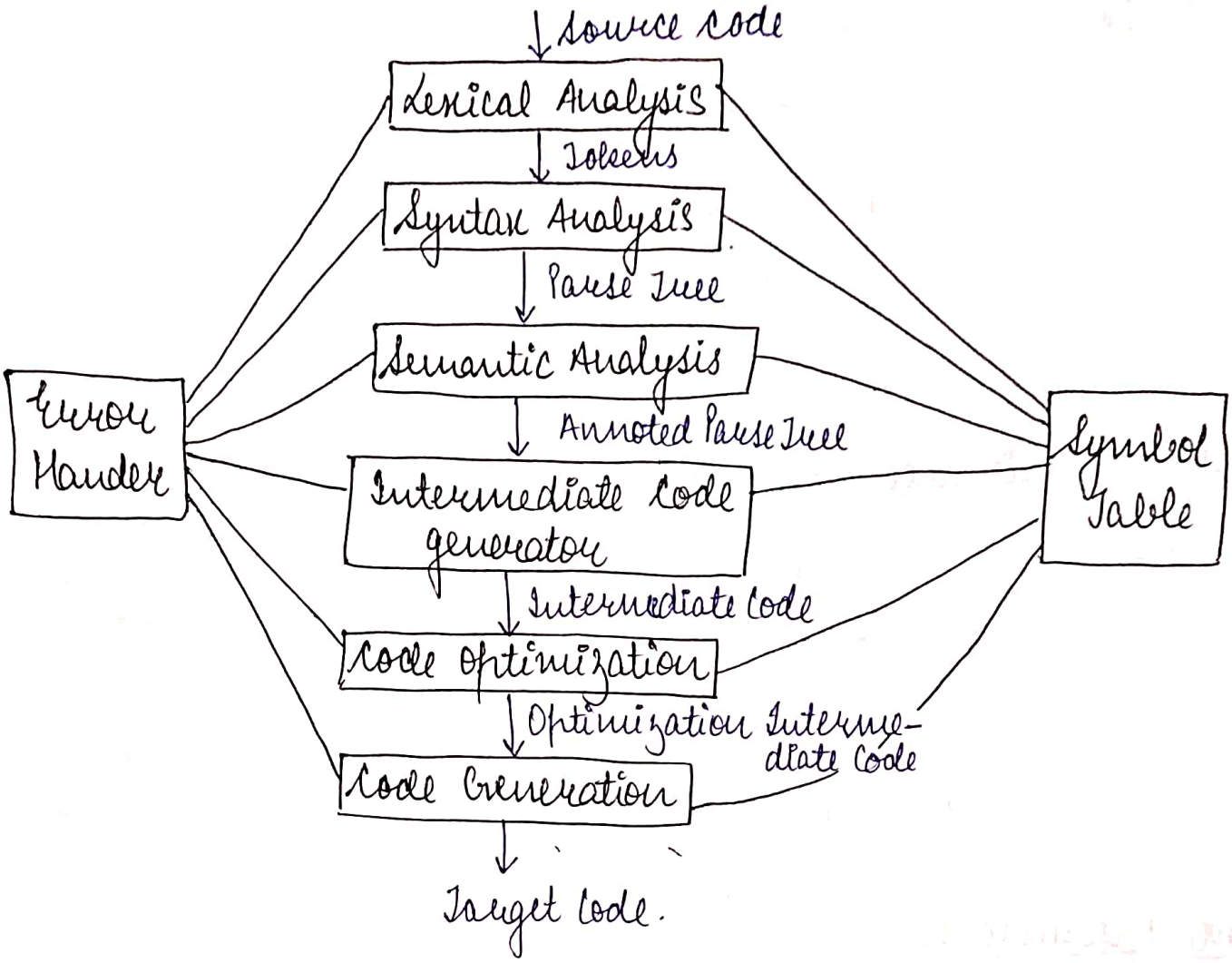


- **JUST-IN-TIME (JIT) Compiler**

It takes the byte code as input and produces the target code as output



PHASES OF COMPILER



- **Lexical Analysis**

Ex: a = b + c * 10

id operator id operator id operator constant

↓

LA

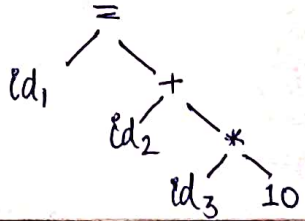
- **Syntax Analysis**

Ex: $id_1 = id_2 + id_3 * 10$

↓

S.A

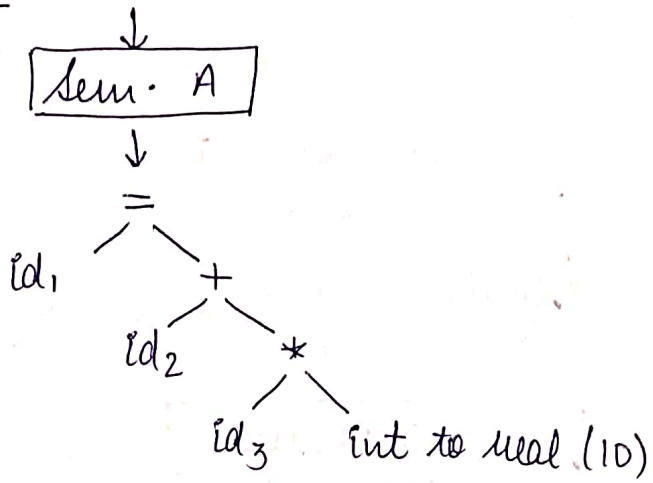
↓



Pass: Pass means grouping two or more consecutive phase of compiler into one module.

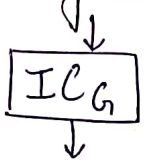
- **Semantic Analysis**

  Ex:



- **Intermediate code generator**

  Ex:



[3 address code]

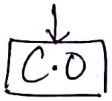
$t_1 = id_3 * 10.00$

$t_2 = id_2 + t_1$

$id_1 = t_2$

- **Code optimization**

  Ex:

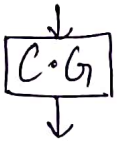


$t_1 = id_3 * 10.00$

$id_1 = id_2 + t_1$

- **Code generation**

  Ex:




```
MOV R, #10.00
MUL R, id3
ADD R, id2
MOV id1, R
```

- **Error Handler**

  error detection + Report + error Recovery Action

° Symbol Table

It is a data structure which contains the information about the variables and their attributes.

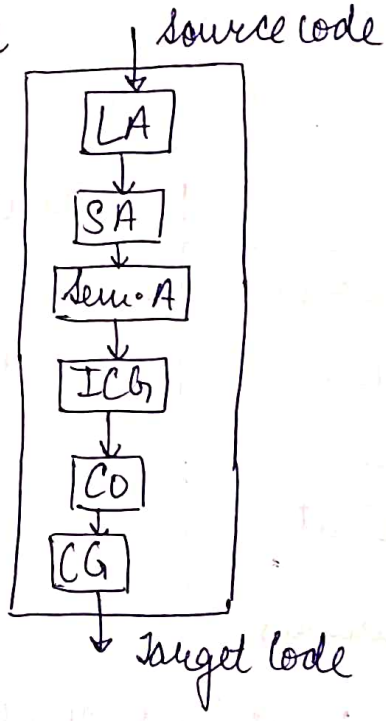
- Symbol table will be created during the 1st 3-phases of the compiler.
- Symbol table can be used by the last 4-phases of the compiler

Types of compiler

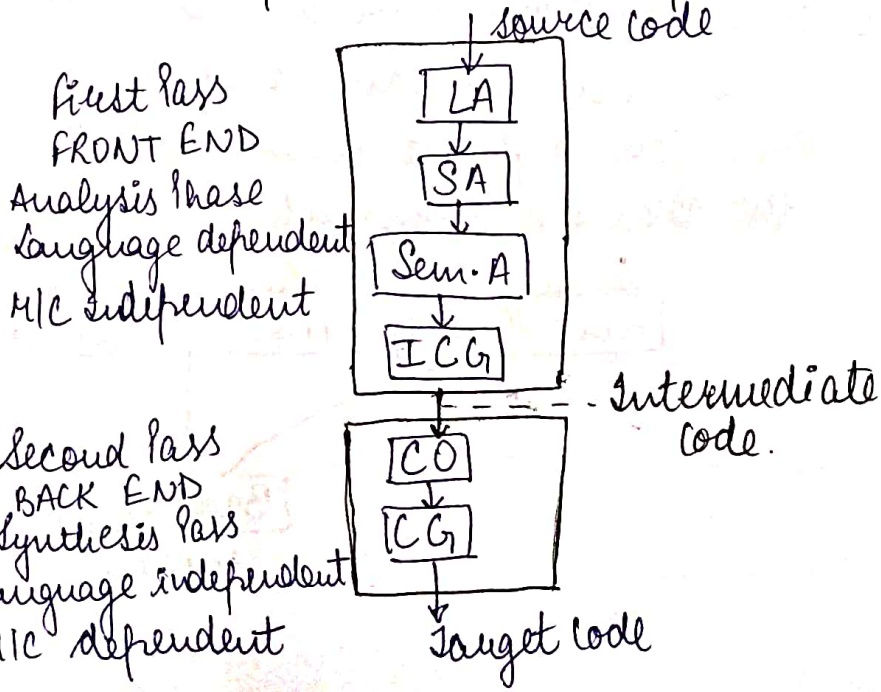
Based on the pass structure the compiler divided into two types:

1) Single Pass Compiler / Narrow Compiler

- In single pass compiler all the phases of compiler will be grouped as a single module.
- At the time of execution all these phases will be stored in MM. Thus, it required more memory as all the phases store at a time.
- Single pass compiler is faster but less efficient.



2) Multi Pass Compiler / Two Pass Compiler / wider Compiler

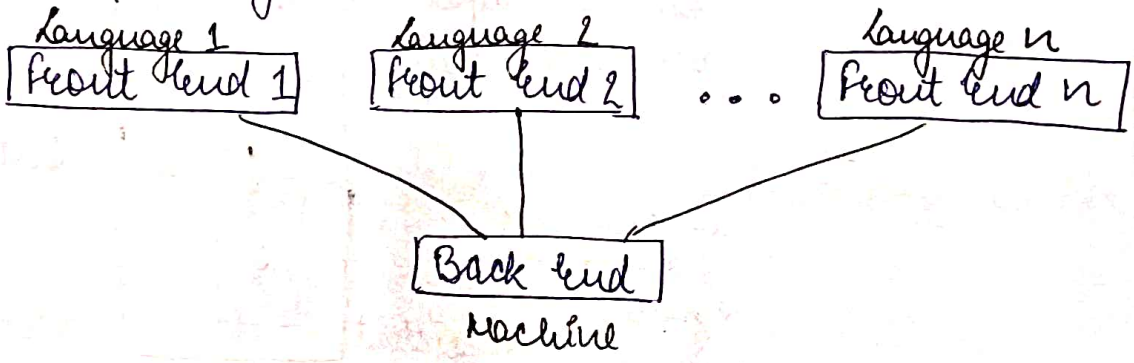


- A multipass compiler when its executes only the first pass on front end will be placed in main memory.
- After finishing the execution of first pass the output (i.e. intermediate code) will be placed on its temporary file then the first pass will be remove from the main memory
- Then the second pass or back end on the compiler will be stored in the main memory and it will read the i/p which is in temporary file. Thus multi pass takes more time, because of these read & write operation on temporary file.
- But it takes less memory and more efficient.

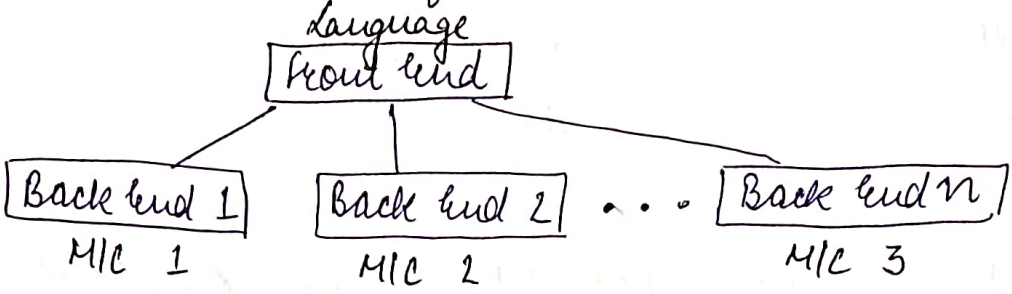
	Single pass compiler	Multi pass compiler
Memory	More	Less
Speed	More	Less
Time	Less	More
Efficient	Less	More
Examples	C, Pascal	C++, C#, Java

⊙ Using multipass compiler two problem can be solved

1. If we want to design a compiler for different languages which runs on same machine then design different front ends for different languages but only one back end for the corresponding machine.



2) Suppose, we want to design a compiler for same language which runs on different machine then design one front end and different back end for the different machine.



Ques The keywords of a language are recognized during?

Sol Lexical Analysis

Ques In a compiler the data structure which is used to manage the information about the variables and their attributes is?

Sol Symbol Table

Ques An ideal compiler should be!

a) Smaller in size
b) Takes less time for compilation
c) Must produce an object code which executes faster
d) None

Sol a, b, c

Ques A link editor is a program that

a) Matches external names of one program with location of other program
b) Acts as link b/w the text editor and user
c) Matches the parameters of the macro definition with the location of the micro call
d) All.

Sol a

Ques An instruction is a programming language i.e. replaced by a sequence of instructions prior to compiling is known as?

a) Literal
b) Label
c) Procedure
d) Macro

Sol d) Macro

Ques A System Program that set up an executable program which is ready for execution in main memory is

a) Linker
b) Loader
c) Assembler
d) All

Sol b) Loader

Ques which of the following is used for grouping of character into tokens?

Sol Lexical Analyser

Ques In a compiler the module that checks every charac. of the source text is?

Sol Lexical Analyser.

Chapter -1

Ques → 1, 2, 5, 6, 7, 8, 9, 11, 13, 15, 17, 16, 25, 3
a d a d c a c d b d c c a

Page 61
Q-1

$t_1 = 60.00$
$t_2 = id_3 * t_1$
$t_3 = id_2 * t_2$
$id_1 = t_3$

O/P: $t_1 = id_3 * 60.0$
$id_1 = id_2 + t_2$

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

LEXICAL ANALYSER

- Every characters of the source text is scanned by the lexical analysis and the following functions will be done by it.
  - Divide into tokens
  - Ignores comment
  - Ignores white space characters like blank space & tab space
  - Counts the number of lines of the source program
  - Creates a symbol tables
  - Produces lexical errors if any with line number and column number
  - Interacts with syntax ~~Anal~~ Analyser

The other name of lexical analyser is Scanner.

→ The terminologies used in lexical analysis are:
  - Token
  - Pattern
  - Lexeme.

1. Tokens

It describe the category of the input string i.e.

KW , OP , PM , const , identifier
[Keywords] [operations] [Punctuation Marker]

It is pair consist of <Token Name, Attribute Value>
Attribute Value is optional which points to the entry of the identify in the symbol table.

2. Pattern

- It is set of rules which describes the token.
- An identifier can be any combination of alphanumeric symbols.
  Only special symbol allowed in identify is '_' (underscore)
- The first symbol of the identifier must be either alphabet or underscore.

3. Lexeme

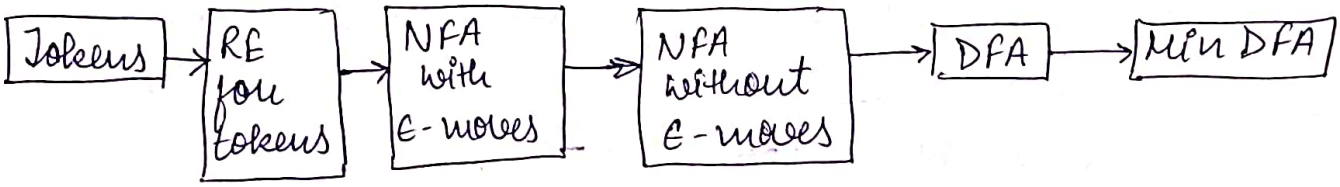
The sequence of characters in the source program that matched with the pattern of the tokens.

Eg $a = b + c * 2$

Lexeme	Token
a	$<id, 1>$ or id_1
=	$<=>$
b	$<id, 2>$ or id_2
+	$<+>$
c	$<id, 3>$ or id_3
*	$<*>$
2	$<2>$

1	a	- - - -
2	b	- - - -
3	c	- - - -

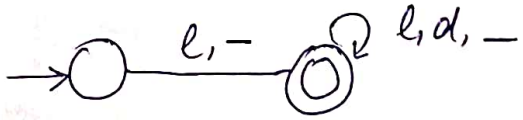
Note :- The program used in lexical analysis is finite automata
- The tokens are recognized by regular expression using finite automata.



⊛ Identification of tokens

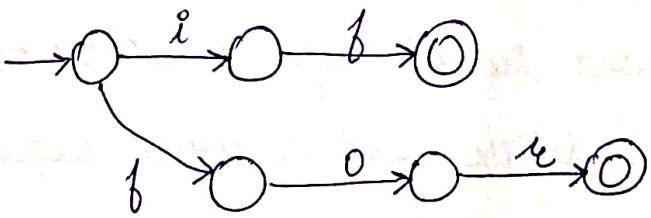
1. Letters → a/b/.../z/A/B/.../Z/
2. Digits → 0/1/2 ... /9
3. _ → Underscore

$(l/\_)\ (l/d/\_)^*$

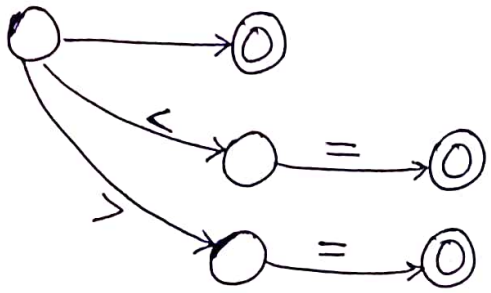


⊛ Identification of Keywords

Eg



⊛ Identification of relational operator



⊛ Symbol Table

It is a data structure which contains all information about the variables and their attributes, function name, classes and objects.

The data structure to implement the symbol table are linear list, binary search tree and hash table.

⊛ Lexical errors

If the scanned group of characters doesn't match with any rules of the token then lexical analyser throws an error.

Types of lexical errors

1. Unterminated Comments

   //_____

   /* - - - - - - (*/)

2. Unmatched string
   whenever you open the double coute and you have to close

   " - - - - - (")

3. Illegal character present
   Any illegal char present it will throw error.
   @ is not allowed

4. Invalid identifier

   eg int a1

   int 1a
   ↳ invalid identifier

5. Invalid constant

int a = 2.5 ✓ This will check by Semantic Analyser

int a = $10; ×

6. Exceeding the length of the identifier

⊛ Error Recovery Methods

1. Panic mode error recovery

The extra char deleted until the valid code recovered into the function

```
fooo i=0; i<n; i++)
function name
f1 = (a, b, c)
```

```
fooo i=0; i<n; i++)
```

In this technique the successive group of character will be deleted from the remaining input string until it forms the well formed token.

Ex: fooo → fooo i=0; i<n; i++)

- Delete

It delets only one last character of the input string

Ex: intt → int

- Insert

It inserts the one missing character

Ex: Print → Print

- Replace

It replaces one character by another character

Ex: iit → int

- Transpose

A transpose two successive character

Ex: fuo → foo

Interacts with Syntax Analyser

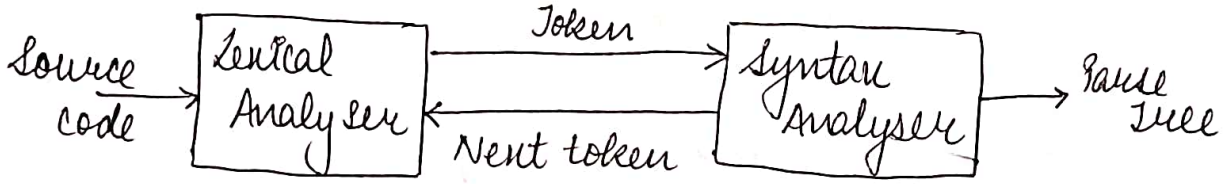
~~A token will be sent~~

The lexical analyser produce the token and it will be passes to the syntax analyser. The syntax analyser checks whether it is well formed token or not. If it is well formed then it sends a request to the lexical analyser for the next token.

Lexical analysis and syntax analysis simultaneously works together.

Why should we seperate the lexical analysis from syntax analysis?

i) To simplify the design of the compiler
ii) To improve the efficiency of the compiler
iii) To increase the portability of the compiler



Q) Check whether the lexical analyser can produce errors for the following statements.

1. float x, y;
2. itn a = b1 -
3. if (a > 5)
   a = a * 2
   else
   a = a + 2
4. for (i = 0; i < n)
5. printf ("Dehradun is a beautiful place);
6. scanf ("%d", a)
7. int 1x, 2x;
8. int x = @ 10;
9. a = b * c /* Delhi
10. int x = 5.35;

1. float x, y; ← No lexical error
   id id PM id PM

2. itn a = b = ← No lexical error
   id id op id op

3. if (a > 5
       a = a * 2
   else
       a = a + 2
   } No lexical error

4. for (i = 0; i < n) ← No lexical error

5. printf ("Dehradun is a beautiful place); ← lexical error
   unmatched string

6. scanf ("%d", a); ← No lexical error

7. int 1x, 2x; ← lexical error
   invalid identifier

8. int x = @ 10; ← lexical error
   Invalid constant

9. a = b * c /* Delhi lexical error
   ↑ unterminated comment

10. int x = 5.35 No lexical error
    ↓ op const

Ques) Find the number of tokens produced by the lexical analyser for the following statement?

1. int x, y; 5

2. if (a > b) else 19
   a = a1 + b; b = b1 * 10;

3. int aaa ; 3

4. int aaa; 3

5. int a aa; 4

6. `int a a a;` 5

7. `for (i=0; i<=10; i++);` 14

8. `printf("Made easy is the best institute");` 5

9. `scanf("%d%d", a, b);` 9

10. `scanf("%d", &a);` 8

11. `a+++b---c%b&&d;` 12

12. `int a=10;`
`float b=10.53;`
`printf("%d %d", ++a**-, b++);` 24

13. `main()`
`{ int *a, b;`
`b=10; /* a=b+c;`
`a=&b;`
`printf("%d", a); */`
~~`b=**a;`~~
`b=* /* pointer */ a;`
`}`

20

→ ignored comment

14. `int main()`
`{`
`/* a=b+c*d;`
`y=x*y-z*s;`
`"Hello students" */`
`a=b**x÷y;`
`}`

15

→ ignored

Ques lexical error?

1. int a = 0 97 ; Number starting with 0 represent Octal which is 0-7 but it has 9 so invalid
   Invalid const.
2. int a = 0b123 ;
   Invalid const.
3. int a = 0x9Ah ;
   Invalid const
   All lexical error.

1. Number starting with 0 represents Octal → 0-7 but it has 9 so invalid
2. Number starting with 0b represents binary but it has 2,3 so invalid
3. Number starting with 0x represents hexadecimal but it has h so invalid.

Ques In lexical analysis the modern computer language such as JAVA needs which one of the following machine models in a necessary and sufficient sense

a) Finite Automata
b) Push Down Automata
c) Turing Machine
d) All.

Sol a) Finite Automata

If it is not given necessary and sufficient then we can go for d) All.

P-64

Q-27 a) $T_1 T_2 T_3$ $\underbrace{\text{b}}_{T_1}\underbrace{\text{baa}}_{T_2}\underbrace{\text{cabc}}_{T_3}$

b) $T_1 T_1 T_3$ $\underbrace{\text{b}}_{T_1}\underbrace{\text{baa}}_{T_1}\underbrace{\text{cabc}}_{T_3}$

c) $T_2 T_1 T_3$ $\underbrace{\text{bbaa}}_{T_2}\underbrace{\text{c}}_{T_1}\underbrace{\text{abc}}_{T_3}$

d) $T_3 T_3$ $\underbrace{\text{bbaacabc}}_{T_3}\underbrace{}_{T_3}$

longest possible prefix

∴ (d)

Q-28 'A'
consider as one.

Q-31 >>=
consider as one.

Q-30 fl /*gate*/ oat

↑ ignore

in t gate;

(2) in
t gate;

Q-14
```
main()
{ in/*comment t x;
  floa/*comment*/t gate;
}
```

∴ both (c)

Semantic — in t x;
undeclared variable.

Syntax. — comment not closed. & above too
in — it should end with semi colon.
t gate;
in t gate; — there should be comma or any operator in between

SYNTAX ANALYSIS

The tokens produced by lexical analysis will be given as input to the syntax analysis, in syntax analysis all these tokens will be grouped together to form an hierarchical structure using the rules of the context free grammar That structure is called as parse tree.

The Syntax Analyser checks the syntax of the token if it is not satisfying the proper syntax then it will through an error.

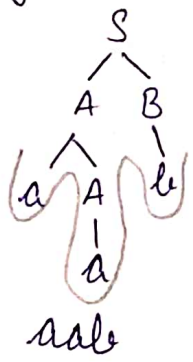
⊙ Left Most Derivation (LMD):

$G = (\{S, A, B\}, \{a, b\}, P, S)$

$P = \{S \to AB,\ A \to aA/a,\ B \to bB/b\}$

If we apply a production every time to the left most non terminal only is called as LMD

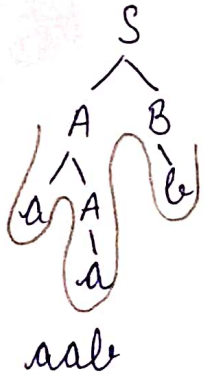
S
AB
aAB
aaB
aab



⊙ Right Most Derivation (RMD):

If we apply a production every time to the right most non terminal only is called as RMD

S
AB
Ab
aAb
aab



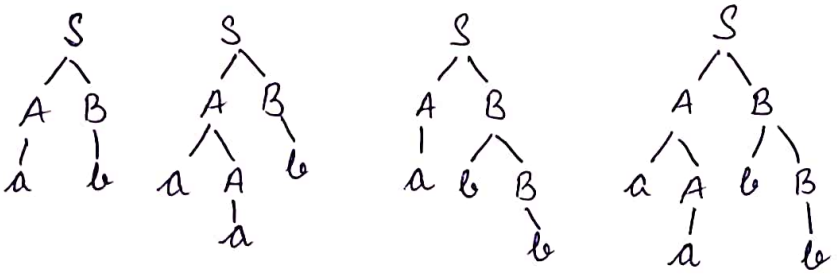
Ambiguity

A grammar G_1 is said to be ambiguous if it can derive atleast one string which is ambiguous.

A string $w \in T^*$ is said to be ambiguous if it has two or more different parse trees.

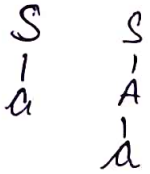
Eg $G_1 = (\{S, A, B\}, \{a, b\}, P, S)$

$P = \{S \to AB,\ A \to aA/a,\ B \to bB/b\}$ is unambiguous.



Ques Ambiguous or not?

$G_1 = \{S \to A/a,\ A \to a\}$ is ambiguous.



$L(G_1) = \{a\}$

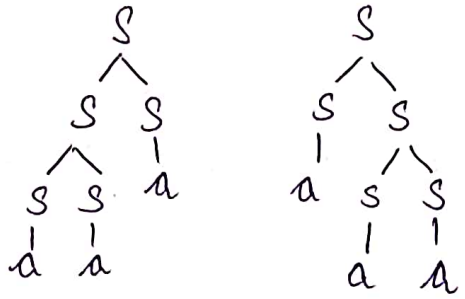
Note: If a grammar is ambiguous it is not suitable for any kind of parsing technique except backtracking and operator precedence parsing.

If we want to apply the remaining technique also, we must eliminate the ambiguity from the content free grammar. But there is no algorithm to eliminate the ambiguity from the grammar.

Hence elimination of ambiguity from a grammar is undecidable.

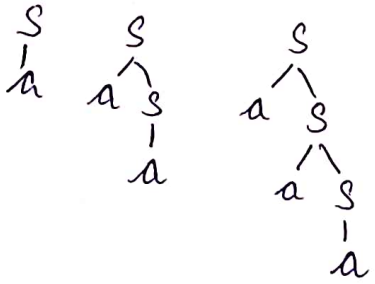
$G' = \{S \to a\}$ is unambiguous for the above ambiguous grammar

Ques $G = \{ S \to SS / a \}$ is ambiguous

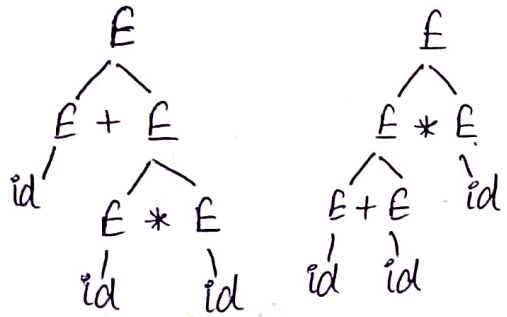


$L(G) = \{ a^m / m > 0 \}$

$G' = \{ S \to a / aS \}$ is unambiguous



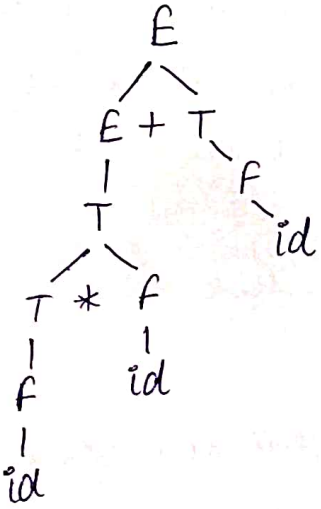
Ques $G = \{ E \to E+E / E*E / id \}$ is ambiguous



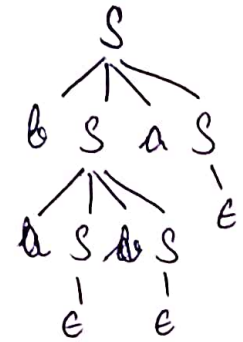
$id + id * id$ $\quad\quad$ $id + id * id$

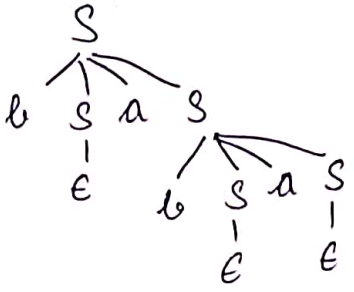
$* > +$

$G' = \{ E \to E+T/T,\ T \to T*F/F,\ F \to id \}$ is unambiguous



Ques $G = \{ S \to aSbS / bSaS / \epsilon \}$ is ambiguous.





Recursive CFGs

Left recursive CFGs

$A \to A\alpha / \beta$

↓ convert into right recursive [eliminate left recursion]

$A \to \beta A'$

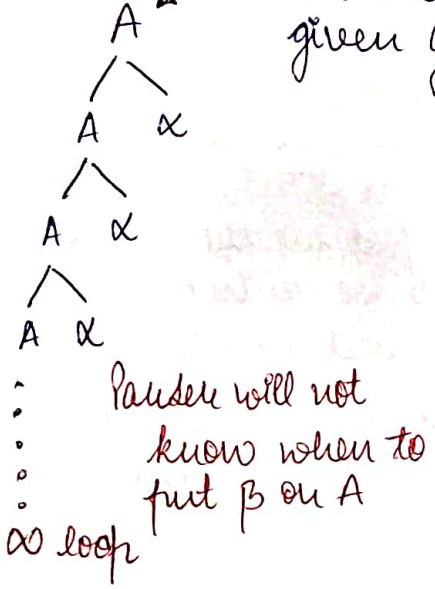
$A' \to \alpha A' / \epsilon$

Right recursive CFGs

$A \to \alpha A / \beta$

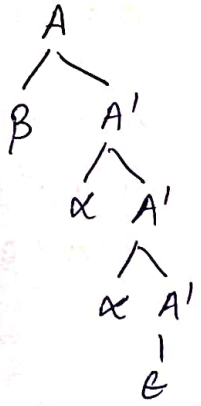
Note: We are eliminating left recursive CFGs so that it should not fall into infinite loop in the top down parsing.

In the case of top down parsing if the grammar is left recursive it may fall into infinite loop. Therefore we must eliminate the left recursion from the given grammar



β	α	α	$

Now right recursive →



$G = \{ A \to A\alpha_1 / A\alpha_2 / A\alpha_3 \ldots / A\alpha_n / \beta_1 / \beta_2 / \beta_3 \ldots / \beta_n \}$

eliminate left recursive.

↓

$G' = \{ A \to \beta_1 A' / \beta_2 A' / \beta_2 A' \ldots / \beta_n A'$

$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A' \ldots / \alpha_n A' / \epsilon \}$

Ques eliminate left recursion from the following grammar

1. $G = \{ \underset{A}{S} \to \underset{A}{S}\underset{\alpha}{a} / \underset{\beta}{b} \}$

$G' = \{ S \to bS'$

$S' \to aS' / \epsilon \}$

2. $G = \{ S \to Sa / Sb / a / b / \epsilon \}$

$G' = \{ S \to aS' / bS' / S'$

$S' \to aS' / bS' / \epsilon \}$

3. $G = \{ S \to Aab / bS / a$

$A \to AbS / Sb / ab / \epsilon \}$

It is having 3 left recursion production

$\underline{A} \to \underline{A}bS$

$\underline{S} \to Aab$

$\to \underline{S}bab$

$\underline{A} \to Sb$

$\to \underline{A}abb$

$G' = \{ S \to Aab / bS / a,$

$A \to \underset{A}{Ab}\underset{\alpha_1}{S} / \underset{A}{A}\underset{\alpha_2}{abb} / \underset{\beta_1}{bSb} / \underset{\beta_2}{ab} / \underset{\beta_3}{\epsilon}$

In $A \to Sb$
we have put all S prod. to make direct recursion.

$G'' = \{ S \to Aab / bS / a,$

$A \to bSbA' / abA' / A'$

$A' \to bSA' / abbA' / \epsilon$

When we put all the S prod in $A \to Sb$ we didn't get S as left most in right side. Therefore we didn't put all A prod in $S \to Aab$

Ques $G_1 = \{ S \to Sb / Aa / b$
$A \to Aa / Sab / a / b / \epsilon \}$

Sol

$G_1' = \{ S \to AaS' / bS'$
$S' \to bS' / \epsilon$
$A \to Aa / Sab / a / b / \epsilon \}$

$G_1'' = \{ S \to AaS' / bS' \quad , S' \to bS' / \epsilon$
$A \to Aa / AaS'ab / bS'ab / a / b / \epsilon \}$

$G_1''' = \{ S \to AaS' / bS' \ , \ S' \to bS' / \epsilon$
$A \to bS'abA' / aA' / bA' / A'$
$A' \to aA' / aS'abA' / \epsilon \}$

$G = T * id E' / T$
$G' = + T G' / \epsilon$
$T = T - E / T * id E' * T / id$
$T = id T'$
$T' = - E T' / * id E' * T T' / \epsilon$

Now there is not indirect left recursion

Ques $G_1 = \{ S \to SbA / Aas / bS / a / \epsilon ,$
$A \to Ab / Sa / b \}$

Sol $G_1' = \{ S \to SbA / Aas / bS / a / \epsilon$
$A \to SaA' / bA'$
$A' \to bA' / \epsilon \}$

$G_1'' = \{ S \to SbA / SaA'as / bA'aS / bS / a / \epsilon$
$A \to SaA' / bA'$
$A' \to bA' / \epsilon \}$

$G_1''' = \{ S \to bA'asS' / bSS' / aS' / S'$
$S' \to bAS' / aA'asS' / \epsilon$
$A \to SaA' / bA'$
$A' \to bA' / \epsilon \}$

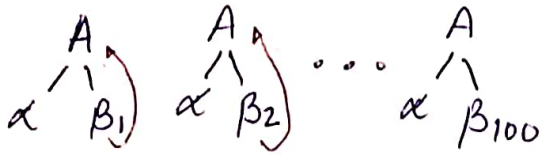
Ques $G_1 = \{ E \to E + T / T * id ,$
$T \to T - E / E * T / id \}$

Sol $G_1' \{ E = T * id E'$
$E' = + T E' / \epsilon$
$T \to T - E / T * id E' * T / id \}$

$G_1'' \{ E = T * id E'$
$E' = + T E' / \epsilon$
$T = id T'$
$T' = - E T' / * id E' * T T' / \epsilon \}$

⑥ Left factor

$$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 \cdots / \alpha\beta_n$$

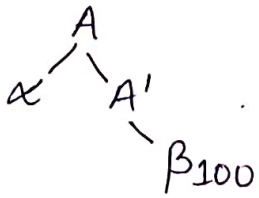


α	β100	$

So eliminate left factor → To avoid the back tracking.

$G_1' = \{ A \to \alpha A'$

$A' \to \beta_1 / \beta_2 / \beta_3 \cdots / \beta_n \}$



To eliminate the back tracking process, we must eliminate the left factor in the case of top down parsing.

Ques $G_1 = \{ S \to \underset{\alpha}{a}\underset{\beta_1}{S} / \underset{\alpha}{a}\underset{\beta_2}{b} / \underset{\alpha}{a} / b \}$ Eliminate the left factoring.

$G' = \{ S \to aA' / b$

$A' \to S / b / \epsilon \}$

Ques $G_1 = \{ S \to \underset{\alpha}{a}\underset{\beta_1}{S}b / \underset{\alpha}{a}\underset{\beta_2}{SA} / \underset{\alpha}{a}\underset{\beta_3}{SbA} / \underset{\alpha}{a}\underset{\beta_4}{A} / b$

$A \to \underset{\alpha}{a}\underset{\beta_1}{A} / \underset{\alpha}{a} / b \}$

$G_1' = \{ S \to aS' / b$

$S' \to \underset{\alpha}{Sb} / \underset{\alpha}{SA} / \underset{\alpha}{SbA} / A$

$A \to aA' / b$

$A' \to A / \epsilon \}$

$G'' = \{ S \to aS' / b$

$S' \to SS'' / A$

$S'' \to \underset{\alpha}{b} / A / \underset{\alpha}{b}\underset{\beta_1}{A}$

$A \to aA' / b$

$A' \to A / \epsilon \}$

$G''' = \{ S \to aS' / b$

$S' \to SS'' / A$

$S'' \to bP / A$

$P \to A / \epsilon$

$A \to aA' / b$

$A' \to A / \epsilon \}$

PARSING

- Top Down
  - Back tracking (Brute Force Technique)
  - Without back tracking (Predictive Parsing)
    - Recursive Descent Parsing
    - Non recursive Descent Parsing (OR) LL(1) parsing
- Bottom Up
  - Operator Precedence Parsing
  - LR(K) Parsing
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1) or LR(1)

Top Down

- From Root to leaves
- Uses Left Most Derivation
- Uses Derivation Process

Eg $G = \{S \to AB,\ A \to aA/a,\ B \to bB/b\}$

$\underline{S}$
$\underline{A}B$
$a\underline{A}B$
$aa\underline{B}$
aab

Bottom Up

- From leaves to root
- Uses Right Most Derivation in reverse
- Uses Reduction Process.

Eg
aab
$a\underline{A}b$
$\underline{A}b$
$A\underline{B}$
S

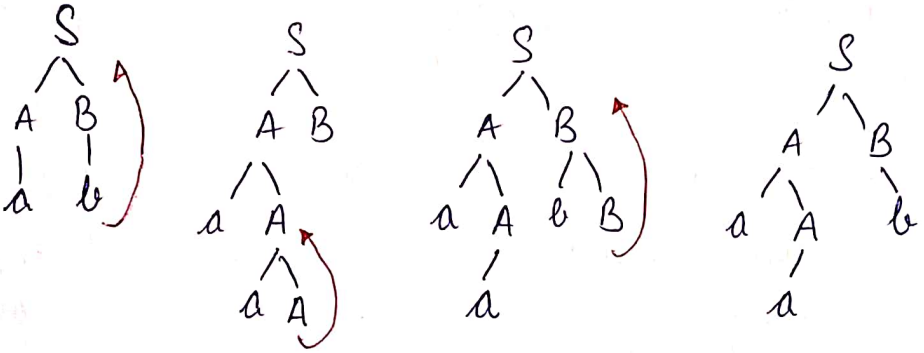
Back Tracking Parsing

[Brute force Technique]

$G_1 = \{ S \to AB, A \to aA|a, B \to bB|b \}$

String aab

a	a	b	$



Backtracking is a process of trying different productions until it get the required string

If the grammer is having so many productions and if the length of the string is too large then the parser has to go for several tries.

Because of the several tries it requires / takes more time. Hence, we don't prefer back tracking process.

Without Back Tracking

1. Recursive Decent Parsing

Recursive Decent Parsing is a program that consist set of procedures one for each non terminal. The execution begins with procedure of the start symbol and if its procedure body has scanned complete string then the execution stops and by announcing the successful parsing has done.

Consider a production of the form

$A \rightarrow X_1 X_2 X_3 \ldots X_K$

```
A()
for(i=1 to K)
{
  if (Xi is a non terminal)
      call the procedure Xi();
  else if (Xi is matches with current i/p symbol)
      advance the i/p symbol;
  else
      error;
}
```

$G = \{E \rightarrow iE'$
$E' \rightarrow +iE'/\epsilon\}$

```
E()
{
1 if(l=='i')
2 {
3 match('i');
4 E'();
5 }
```

```
E'()
{1 if(l=='+')
2 {3 match('+');
4 match('i');
5 E'();
6 }
 else
   return;
 }
}
```

Note: Inherently ambiguous Language.

A language is said to be inherently ambiguous if there exist no unambiguous grammar for that language, i.e. if any grammar is ambiguous for that language, the language is said to be inherently ambiguous.

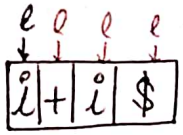
The term inherently ambiguous is used for language not grammar.

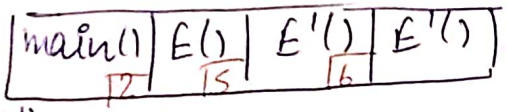
A regular language is never ambiguous but regular grammar can be ambiguous.

```
match(char t)
{
  if(l==t)
    l=getch();
  else
    printf("error");
}
```

```
main()
{
1 E();
2 if(l=='$')
    printf("successful passing");
}
```

Eg





main()
call E()

~~main~~ E()
if (l==i)
match(i)

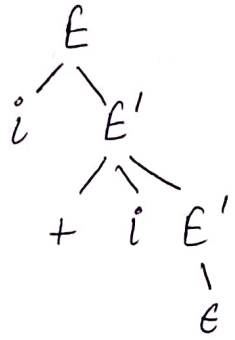
call E'()

E'()
if (l==+)
match +
match i
call E'()

E'()
if false
else → return

else
return

go to main
if (l=='$')
successful



Ques $G = \{ S \to aS / b \}$

Note: i) If a grammar is ambiguous it is not suitable for recursive descent parsing.

~~If a grammar is~~

ii) A left recursive grammar is not suitable for recursive ~~grammar~~ descent parsing

Eg $A \rightarrow A\alpha$

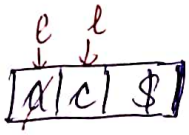
```
A()
{ A()
  match('a');
```

main | A() | A() | A() ...

It will fall into ∞ loop
As infinitely it is calling itself.

iii) A left factor grammar is not suitable because it may give an error message.

Eg $S \rightarrow ab \mid ac$




```
S()
{ if(l == 'a')
  { match('a');
    match('b');
  }
else
  if(l == 'a')
  { match('a');
    match('c');
  }
```

← true.
then match(a)
match(b) false
will print error message.
But the string is valid.

FIRST Set

First (A) is the set of all terminals that are the first symbols on RHS of the arrow in every production of 'A'

Procedure

1. First (a) = {a}
2. If $A \to a \Rightarrow$ First (A) = {a}
3. If $A \to \epsilon \Rightarrow$ First (A) = {ϵ}
4. If $A \to BC \Rightarrow$ First (A) = First (B) if First (B) doesn't contains 'ϵ'
5. If $A \to BC \Rightarrow$ First (A) = First (B) − {ϵ} ∪ First (C)

Ques Find the First set of each non terminal of the following grammar

1. $G_1 = \{ S \to ABC$
   $A \to aA \mid b$
   $B \to bB \mid \epsilon$
   $C \to c \}$

   First (S) = {a, b}
   First (A) = {a, b}
   First (B) = {b, ϵ}
   First (C) = {c}

2. $G = \{ S \to ABC$
   $A \to bB \mid \epsilon$
   $B \to aB \mid c$
   $C \to cC \mid \epsilon \}$

   First (S) = {b, a, c}
   First (A) = {b, ϵ}
   First (B) = {a, c}
   First (C) = {c, ϵ}

3. $G = \{ S \to SaA \mid \epsilon$
   $A \to BbC \mid \epsilon$
   $B \to Ba \mid c$
   $C \to cC \mid \epsilon \}$

   First (S) = {a, ϵ}
   First (A) = {c, ϵ}
   First (B) = {c}
   First (C) = {c, ϵ}

4. $G = \{ S \to ABC$
   $A \to Bb \mid \epsilon$
   $B \to Aa \mid \epsilon$
   $C \to cC \mid \epsilon \}$

   First (S) = {a, b, c, ϵ}
   First (A) = {a, b, ϵ}
   First (B) = {a, b, ϵ}
   First (C) = {c, ϵ}

5. $G_1 = \{ S \to ACB / CbB / Ba$
$A \to da / BC$
$B \to g / \epsilon$
$C \to h / \epsilon \}$

First(S) = { d, g, h, ε, b, a }
First(A) = { d, g, h, ε }
First(B) = { g, ε }
First(C) = { h, ε }

6. $G_1 = \{ S \to AaB / bC$
$A \to BC / bA$
$B \to aB / \epsilon$
$C \to a / SA \}$

First(S) = F(A) ∪ {b} = {a, b} ∪ F(C) ∪ {b}
First(A) = {a, ε} ∪ F(C) ∪ {b} = {a, b} ∪ F(C)
First(B) = { a, ε }
First(C) = {a} ∪ F(S) =

When there is loop
terminal are { a, b }
First of any non terminal contains → { a, b, ε }.
max

First(S) = {b} ∪ F(A)
First(A) = {a, b} ∪ F(C)
First(B) = { a, ε }
First(C) = { a } ∪ F(S)

Now First(A) don't have ε
Assume First(A) contains ε

First(S) = $\underbrace{\{b\}}_{bC} \cup \underbrace{\{a, b, \not{\epsilon}\} \cup \{a\}}_{AaB} = \{a, b\}$.

So First(S) will never contain ε.

First(C) = {a} ∪ {a, b} = {a, b}.
First(A) = {a, b} ∪ {a, b} = {a, b}.

∴ Ans
First(S) = {a, b}
First(A) = {a, b}
First(B) = { a, ε }
First(C) = {a, b}

7. $G_1 = \{S \to ASb/\epsilon$ First(S) = {a, b, ε}
$A \to aA/Bb$ First(A) = {a, b}
$B \to aB/Ab/\epsilon\}$ First(B) = {a, b, ε}

8. $G_1 = \{S \to SABB/Ca/\epsilon$ First(S) = {a, b, c, ε}
$A \to ABc/bA/\epsilon$ First(A) = {b, c, ε}
$B \to CB/bA/cC$ First(B) = {b, c}
$C \to AC/\epsilon\}$ First(C) = {b, c, ε}

9. $G_1 = \{S \to aA/BS/AB/\epsilon$ First(S) = {a, b, c, ε}
$A \to Sb/CA/a$ First(A) = {a, b, c}
$B \to BC/bA/\epsilon$ First(B) = {a, b, c, ε}
$C \to cA/Sc/\epsilon\}$ First(C) = {a, b, c, ε}

Qus Consider the following production $S \to ABC$
Let the number of elements in first(A), First(B), and First(C) are x, y, z respectively. Then the number of elements in First(S)

i) If all the elements in first(A), First(B), First(C) are different and all these first set contain ε
ii) All these F(A), F(B), F(C) contains {a, ε} & remaining all elements are different

Sol i) $F(S) = F(A) - \{\epsilon\} \cup F(B) - \{\epsilon\} \cup F(C) - \{\epsilon\} + \{\epsilon\}$ ← At last
$= x - 1 + y - 1 + z + 1 - 1$
$= x + y + z - 2$

ii) $F(S) = F(A) - \{\epsilon\} \cup F(B) - \{a, \epsilon\} \cup F(C) - \{a, \epsilon\} + \{\epsilon\}$
↑ As added in A; ↑ As added in A; ↑ At last

$= x - 1 + y - 2 + z - 2 + 1$
$= x + y + z - 4$

FOLLOW Set

Follow (A) is the set of terminals that present immediately ~~wherever 'A' is~~ on the RHS of 'A' wherever 'A' is on the RHS of the arrow.

Procedure

1. Follow(S) = { $ } if S is the start symbol
2. If $A \to \alpha B \beta \Rightarrow$ Follow(B) = First(β) if First(β) doesn't contain 'ϵ'
3. If $A \to \alpha B \beta \Rightarrow$ Follow(B) = First(β) $- \{\epsilon\} \cup$ Follow(A)
4. If $A \to \alpha B \Rightarrow$ Follow(B) = Follow(A)

Ques find the follow of each non terminal of the following grammar

1. $G = \{ S \to AaAb / BbBa$
   $A \to aA / \epsilon$
   $B \to bB / \epsilon \}$

   Follow(S) = { $ }
   Follow(A) = { a, b }
   Follow(B) = { a, b }

2. $G = \{ S \to ABC$
   $A \to Aa / Bb / \epsilon$
   $B \to bB / \epsilon$
   $C \to cC / \epsilon \}$

   Follow(S) = { $ }
   Follow(A) = { a, b, c, $ }
   Follow(B) = { b, c, $ }
   Follow(C) = { $ }

3. $G = \{ S \to ACB / CbB / Ba$
   $A \to dc / BC$
   $B \to g / \epsilon$
   $C \to h / \epsilon \}$

   Follow(S) = { $ }
   Follow(A) = { h, g, $ }
   Follow(B) = { h, a, g, $ }
   Follow(C) = { h, g, b, $ }

4. $G = \{ S \to aBS / AaC / A+C$
   $A \to -B*C / BC / \epsilon$
   $B \to SaA / BA / \epsilon$
   $C \to C* / SAa / \epsilon \}$

   First(B) = { a, +, −, * }
   First(A) = { −, ε, a, +, * }
   First(B) = { a, +, −, *, ε }
   First(C) = { ε, *, a, +, − }

   Follow(S) = { $, a, +, −, * }
   Follow(B) = { a, +, −, * }
   Follow(A) = { a, +, −, * }
   Follow(C) = { a, +, −, *, $ }

5. $G = \{ S \to AaBC / SaB$ — Follow(S) = {a, $}
$A \to BCb / \epsilon$ — Follow(A) = {a, b, c, $}
$B \to aA / \epsilon$ — Follow(B) = {a, b, c, $}
$C \to cCa / \epsilon \}$ — Follow(C) = {a, b, $}

LL(1) Parsing

- L → Look ahead symbol of length 1
- L → Uses Left Most Derivation
- L → Left to right scan of the i/p string.

Construction of LL(1) parsing table

We consider every production of the form $A \to \alpha$ and proceed as follows

1. Write $A \to \alpha$ in $M[A, x]$ where $x \in First(\alpha)$
2. If $First(\alpha) = \{\epsilon\}$, then write $A \to \alpha$ in $M[A, y]$ where $y \in Follow(A)$
3. The unfilled entries in the table are called as Syntax error.
4. If all the entries in the table are single then the grammar is LL(1)

5. Construct LL(1) parsing table for following grammar

$G = \{ S \to AB, A \to aA / b, B \to bB / \epsilon \}$

Sol

	a	b	$
S	S→AB	S→AB	Syntax error
A	A→aA	A→b	Syntax error
B	Syntax error	B→bB	B→ε

$S \to AB$
$M[S, x]$
$x = First(AB)$
$\{a, b\}$

$A \to aA$
$M[A, x]$
$x = First(aA)$
$= \{a\}$

$A \to b$
$M[A, x]$
$x = First(b)$
$\{b\}$

$B \to bB$
$M[B, x]$
$x = First(bB)$
$= \{b\}$

$B \to \epsilon$
$M[B, y]$
$y = Follow(B)$
$= \{\$\}$

In the above table, all the entries are single
Therefore the grammar is LL(1).

Ques $G_1 = \{ E \to TE',\ E' \to +TE'/\epsilon$
$T \to FT',\ T' \to *FT'/\epsilon$
$F \to (E)/id \}$

	+	*	(	)	id	$
E	×	×	$E \to TE'$	×	$E \to TE'$	×
E'	$E' \to +TE'$	×	×	$E' \to \epsilon$	×	$E' \to \$$
T	×	×	$T \to FT'$	×	$T \to FT'$	×
T'	$T' \to \epsilon$	$T' \to *FT'$	×	$T' \to \epsilon$	×	$T' \to \epsilon$
F	×	×	$F \to (E)$	×	$F \to id$	×

⊛ Properties of LL(1) grammar
1. A left recursive grammar is not LL(1)
2. A left factor grammar is not LL(1)
3. An ambiguous grammar is not LL(1)
4. Every LL(1) grammar is unambiguous but every unambiguous grammar is not LL(1).
5. Every single production grammar is LL(1)

Eg $\{ S \to Sa/b \}$ // left rec. grammar
$S \to Sa$
$M[S, x] = First(Sa)$
$= b$
$S \to b$
Both in same cell
not LL(1)

$\{ S \to Aa/Ab,\ A \to \epsilon \}$ Left factored grammar
$S \to Aa \quad S \to Ab$
$M[S, x] = First(Aa)$
$= \{a\}$
$M[S, x] = First[Ab]$
$= \{b\}$
But LL(1)
// common part has ϵ production

6. If any productions are of the form, $A \to \alpha_1 / \alpha_2$ then,
If $First(\alpha_1) \cap First(\alpha_2) \neq \phi$
then grammar is not LL(1)

	a	b
A	$A \to \alpha_1$ $A \to \alpha_2$	$A \to \alpha_2$

Multiple entry in one block
∴ not LL(1)

Ex $A \to \alpha_1$ $\quad$ $A \to \alpha_2$

$M[A, \alpha] = First(\alpha_1) = \{a\}$ $\quad$ $M[A, \alpha] = First(\alpha_2) = \{a, b\}$

$\{a\} \cap \{a, b\} \neq \phi$ ∴ grammar not LL(1)

7. If any productions are of the form $A \to \alpha_1 / \alpha_2 / \alpha_3 \ldots / \alpha_n$ then,
If $First(\alpha_1) \cap First(\alpha_2) \cap First(\alpha_3) \ldots First(\alpha_n) \neq \phi$
If first set of any 2 productions having common elements then the grammar is not LL(1)

8. If any productions are of the form $A \to \alpha / \epsilon$, then
If $First(\alpha) \cap Follow(A) \neq \phi$
$\Rightarrow$ Not LL(1)

Ques Check whether the following grammar is LL(1) or not.

1. $G = \{S \to aS / Ab, A \to a\}$

Sol $First(aS) \cap First(Ab)$

$\{a\} \cap \{a\} \neq \phi$

$=$ Not LL(1)

2. $G = \{S \to Sa / ab\}$

It is left recursive grammar

$=$ Not LL(1)

(Or)

$First(Sa) \cap First(ab)$

$\{a\} \cap \{a\}$

$\neq \phi$

Not LL(1)

3. $G_1 = \{ S \to AaS / aA \ , \quad A \to bA / \epsilon \}$

	a	b	$
S	S→AaS S→aA	S→AaS	
A	A→ϵ	A→bA	A→ϵ

First (AaS) ∩ First (aA)

$\{b, a\} \cap \{a\}$

$= \{a\} \neq \phi$

Not LL(1)

4. $G_1 = \{ S \to aAB / bS \ , \ A \to AbS / \epsilon \ , \ B \to aB / \epsilon \}$

First (aAB) ∩ First (bS)

$\{a\} \cap \{b\}$

$= \phi$

First (AbS) ∩ Follow (A)

$\{b\} \cap \{b, a, \$\}$

$\{b\}$

Not LL(1)

(or)

$A \to AbS$ is left recursive

Not LL(1)

5. $G_1 = \{ S \to AXY / a \ , \ A \to +X*Y / \epsilon \ , \ X \to AX / \epsilon \ , \ Y \to aS / X \}$

$S \to AXY$

$\{+, \cancel{\epsilon}\} \cup \{+, \cancel{\epsilon}\} \cup \{a, +, \cancel{\epsilon}\} \cup \underbrace{\{a, +, *, \$\}}_{\text{Follow S}}$

First(A) First(X) First(Y)

$= \{a, +, *, \$\}$

$S \to a$

$= \{a\}$ $\quad \cap \neq \phi$ Not LL(1)

Page 66

Q-4 a) ambiguous
b) left recursive

Best one a) ambiguous

Q-44 $E \to E+T / E*T / T$ $\quad T \to T-f / f$ $\quad f \to f+2 / id$

$E \to TE'$ $\quad T \to FT'$ $\quad f \to id\ f'$

$E' \to +TE' / *TE' / \epsilon$ $\quad T' \to -FT' / \epsilon$ $\quad f' \to +2f' / \epsilon$

Now eliminate $f' \to \epsilon$

$f \to id\ f' / id$ $\quad$ ∴ (a)

$f' \to +2f' / \cancel{\epsilon}$

Q-2 $S \to iEtSS' / a$ $S' \to eS / \epsilon$ $E \to b$

$\{i\} \cap \{a\}$
ϕ

$\{e\} \cap \text{follow}(S')$
$\{e\} \cap \text{follow}(S)$
$\{e\} \cap \{e, \$\}$
$\neq \phi$
Not LL(1)

$S' \to eS$
$M[S', x]$
$x = \text{first}(eS)$
$= \{e\}$

$S' \to \epsilon$
$M[S', y]$
$y = \text{follow}(S')$
$= \{e, \$\}$

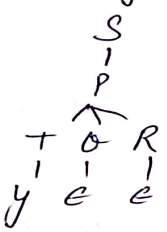
$\therefore$ (d).

Q-17 B is useless as it is not terminated
$\therefore$ (b)

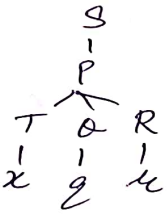
Q-18 (c) $\{S \to CA$
$A \to a$
$C \to b\}$ eliminate all the B production

Q-84 $S \to P$
$P \to TQR / QR$
$Q \to q / \epsilon$
$R \to u / \epsilon$
$T \to x / y$

a) y.



b) xqu



c) xyq
Not possible
$\therefore$ (c)

Q-93 $S \to ABA$ ← one production will have single entry.
$A \to Bc / dA / \epsilon$
$B \to eA$

	e	d	$	c
A	$A \to Bc$	$A \to dA$		
	$A \to \epsilon$	$A \to \epsilon$	$A \to \epsilon$	$A \to \epsilon$

First(A) = {e, d, ε}
~~Follow(A) = {e, $}~~
Follow(B) = {e, d, $, c}
Follow(A) = {e, d, c, $}

$\therefore$ 2

Q-107

$S \to X\ Y$

$X \to (S) / \epsilon M$

$Y \to \flat S / \epsilon$

$M \to \varrho X / \epsilon$

∴ a, b, c

a) $M[S, x] \Rightarrow x = First(XY) = \{(, \epsilon\}$

b) $Y \to \epsilon$

$M[Y, y] \Rightarrow y = Follow(Y)$

$= \{), \$\}$

c) $M \to \epsilon$

$M[M, y] \Rightarrow y = Follow(M)$

$= Follow(X)$

$= First\ Y$

$= \{\flat, \epsilon\} \cup Follow(S)$

$= \{\flat\} \cup \{), \$\}$

$= \{\flat,), \$\}$

Note: Every Regular language has LL(1) grammar.
Every regular grammar is not LL(1).

$L = \{a^n / n \geq 0\} \Rightarrow G_1 = \{S \to aS / \epsilon\}$ is regular grammar and also LL(1)

$= \{a\} \cap Follow(S)$

$= \{a\} \cap \{\$\} = \phi$

$L = \{a^n / n > 0\} \Rightarrow G_1 = \{S \to aS / a\}$ is not LL(1)

$G^1 = \{S \to aA, A \to aA / \epsilon\}$ is LL(1)

To construct regular grammar which is LL(1).
First construct DFA according to that grammar.
As it is define [state] on every input uniquely.

LL(K) Parsing

- L → Look ahead symbols of length 'k'
- Uses Left Most Derivation
- Left to right scan of the input string

$LL(1) \subset LL(2) \subset LL(3) \subset \dots LL(K)$

1. $G = \{ S \to \underline{aa}S / \underline{ab} / \underline{a} \}$ is left factor grammar
   $= \{aa\} \cap \{ab\} \cap \{a\}$
   $= \phi \Rightarrow$ the grammar is LL(2)

2. $G = \{ S \to aaS / aA / aBb \;,\; A \to bA / \epsilon \;,\; B \to \epsilon \}$
   $= \{aa\} \cap \{ab, a\} \cap \{ab\}$
   $= \{ab\}$
   $\neq \phi \Rightarrow$ Not LL(2)

Ques Which of the following is LL(3) but not LL(2) and not LL(1)

a) $\{ S \to aS / b \}$

b) $\{ S \to aaS / ab / b \}$

c) $\{ S \to aaaS / aab / ab / b \}$

d) None

Sol a) LL(1)

b) not LL(1) but LL(2)

c) is not LL(2), not LL(1) but LL(3)

∴ (c)

⊛ Bottom Up Parsing

⊙ **Handle:** A handle is a substring of a string that matched with any one of the right side of the productions, then that handle will be reduced with left side of the production.

$G = \{ E \to E+E / E*E / id \}$

Right sentential form	Handle	Production
id * id * id	id	E → id
E + **id** * id	id	E → id
E + E * id	E+E	E → E+E
E * **id**	id	E → id
E * E	E * E	E → E * E
E		

Handle Pruning

⊙ **Handle Pruning:** Bottom Up Parsing is a process of finding the handles and using them in the reduction to get the start symbol. This entire process of reducting the string to the start symbol is called as handle Pruning.

⊙ **Sentential Form:** Any string that can be derived from the start symbol is called as sentential form.

⊙ **Right sentential form:** The sentential form that occurs in the derivation of some sentence using RMD is called as right sentential form.

⊙ **Left sentential Form:** The sentential form that occurs in the derivation of some sentence using RMD is called as left sequential form.

⊙ Sentence: If all the symbols in the sentential form are terminals then it is called as sentence.

Every sentence is also sentential form but every sentential form is not sentence.

Eg
E
E+E
id+E
id+E*E
id+id*E
id+id*id. ← Sentence

} Sentential

(#) Bottom up Parsing [Shift reduce parsing]

⇒ In this we use 4 actions

Shift
Reduce
Accept
Error

⇒ In this 2 data structure will be used

i) Input Buffer → for storing the input strings

a	b	a	$

configuration : w$ where w ∈ T*

ii) Stack → To store the symbols of the grammar

Configuration :

Ques Consider the following grammar and parse the input string aabb using shift reduce parsing

$G = \{S \to aS/aA,\ A \to bA/b\}$

Stack	Input Buffer	Action
$	aabb$	Shift
$a	abb$	Shift
$aa	bb$	Shift
$aab	b$	Shift
$aabb	$	Reduce by $A \to b$
$aabA	$	Reduce by $A \to bA$
$aaA	$	Reduce by $S \to aA$
$aS	$	Reduce by $S \to aS$
$S	$	ACCEPT

⊙ Shift action = $|w| = 4$ [Always equal to length of a i/p string]

Reduce = 4

Accept = 1

Total = 9

⊙ Viable Prefixes

Set of prefixes ~~that appear~~ of a right sentential form that appear on top of a stack of a shift reduce parser

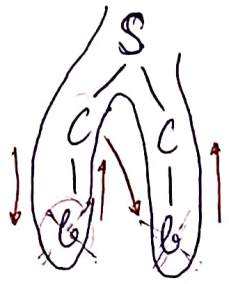
Viable Prefixes: $\{a, aa, aab, aabb, aabA, aaA, aS, S\}$

Ques Consider the following grammer

$G = \{S \to CC, C \to aC | b\}$

which of the following are viable prefixes?

a) ab b) bb c) Cb d) Ca e) abb



b – At last [left leaf]

C – Left subtree finish reduce delete b so C left

Cb – from left C and right leave b

CC – Now right subtree finish reduce delete b from right leave so C from left & right are there.

→ a

→ ab

At II C → b reduce so eliminate b

→ aC

At III All the left subtree reduce so eliminate

→ C

→ Cb

At IV reduce C → b so eliminate

→ CC

→ b

At I reduce C → b.

→ C

→ Ca

→ Cab

At II C → b reduce eliminate

→ CaC

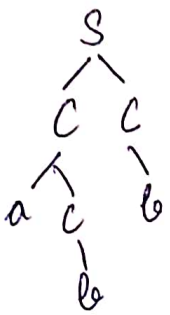
At III right subtree after C reduce!

→ CC

∴ (a), (c), (d) Ans

Method-2

Option(a)



Stack	i/p Buffer	Action
$	$	
$abb	b$	Reduce by C → b
$aC	b$	Reduce by C → aC
$C	b$	shift
$Cb	$	Reduce by C → b.
$CC	$	
$S		✓

i/p → abb

Option(b)

Stack	i/p Buffer	
$	$	
$bb	Cb$	
$bC	Cb$	×
$bCC		
$bS		Not reducable.

Option(c)

Stack	i/p Buffer	
$	bb$	
$b	b$	
$C	b$	
$Cb	$	✓
$CC		
$S		

Option(d)

Stack	i/p Buffer	
$		
$Ca	b$	
$Cab		
$CaC		✓
$CC		
$S		

Option(c)

Stack	i/p Buffer	
$Cb	$	
$CC	$	
$S	$	✓

Option(e)

Stack	i/p Buffer	
$		
$abb		
$abC		Not reducable.

Ques $G = \{E \to E+T/T,\ T \to T*F/F,\ F \to (E)/id\}$
What are variable prefixes?

a) $E+T*$ b) $id+id$ c) $E*id$ d) $T*F$

Sol

	Stack	Input buffer	Action
a)	$E+T*$		
	$E+T*id$		c) X
	$E+T*F$		d) $T*F$
	$E+T$ ✓		T
	E		E

b) $id+id$
$id+F$
$id+T$ X.
$E+T$
$T+T$
$E+T$
E

Ques $G_1 = \{S \to aAb/bS,\ A \to Bb/a,\ B \to Aa\}$

a) bbA b) aBb c) $bbaA$ d) Aab

a) bbA

b) aBb
aAb
S

c) $bbaAb$
bbS
bS
S

d) Aab x

∴ b, c

Operator grammar

A grammar is said to be operator grammar if it satisfies the following conditions.

i) No production should contain adjacent non terminals on RHS of ~~any production~~ the arrow.

$S \to aAbB$ ✓ $\quad\quad$ $S \to aABb$ ✗

ii) There should not be any null productions.

$A \to \epsilon$ ✗

Ex: $G_1 = \{S \to aAb / BbS,\ A \to bB/a,\ B \to ab\}$ is an operator grammar

$G_1' = \{S \to SAS/a,\ A \to +/-/*\}$ is not an operator grammar

$G_1'' = \{S \to S+S/S-S/S*S/a\}$ is an operator grammar.

Operator Precedence Parsing

4 Actions
- Shift
- Reduce
- Accept
- Error

2 Data structure
- Input Buffer — w $
- Stack — $

⊙ Construction of operator precedence parsing table.

Let x is the top most symbol in the stack and y is the present symbol in the input buffer then proceed as follows.

i) If $x \lessdot y$ then perform the shift operation

ii) If $x \gtrdot y$ then ~~perform~~ there will be a handle in the stack, then perform reduce operation with an appropriate production.

Ques Consider the following grammar and operator precedence table. Parse the input string id + id * id using operator precedence parsing.

$G = \{ E \to E+E / E*E / id \}$

	id	*	+	$
id	–	$\gtrdot$	$\gtrdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	–

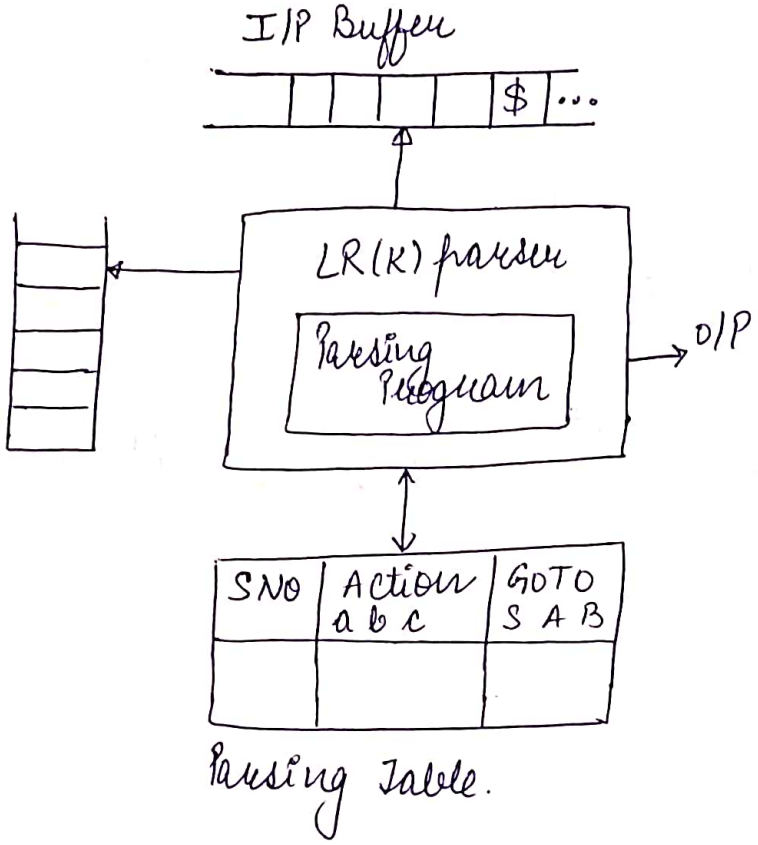
Note: id has the highest priority
$ has the lowest priority
Non terminal has less priority than terminals except $

Sol

Stack		Input Buffer	Action
$	$\lessdot$	id + id * id $	Shift
$ id	$\gtrdot$	+ id * id $	Reduce by E → id
$ E	$\lessdot$	+ id * id $	Shift
$ E +	$\lessdot$	id * id $	Shift
$ E + id	$\gtrdot$	* id $	Reduce by E → id
$ E + E	$\lessdot$	* id $	Shift
$ E + E *	$\lessdot$	id $	Shift
$ E + E * id	$\gtrdot$	$	reduce by E → id
$ E + E * E	$\gtrdot$	$	Reduce E → E * E
$ E + E	$\gtrdot$	$	Reduce E → E + E
$ E		$	ACCEPTED

LR(k) parsing

- L → look ahead symbols of length 'k'
- R → Uses Right Most Derivation in reverse
- L → Left to right scan of the string.



Parsing Table.

⊙ Augmented Grammar

Augmented grammar is a new grammar with a new start symbol S' such that S' derives S as a new production. The purpose of this grammar is, the parser will get terminated after successful parsing has done.

$G = \{S \to AB,\ A \to aA/a,\ B \to bB/b\}$

$G' = \{S' \to S,\ S \to AB,\ A \to aA/a,\ B \to bB/b\}$

A = XYZ

A = .XYZ

A = X.YZ

A = XY.Z

A = XYZ.

SLR(1) parsing

This parsing can be divided into 2 parts

1. Construction of canonical set of items (or) LR(0) items.
2. Construction of parsing table

1. Construction of canonical set of items (or) LR(0) items

It has 2 parts

i) Construction of closure operation.

ii) Construction of Goto operation

i) Construction of closure operation

a. Initially add $S' \rightarrow .S$

b. If $A \rightarrow \alpha . B\beta$ is in I_i and if $B \rightarrow \gamma$ is a production then add $B \rightarrow .\gamma$ in I_i

Ex $G = \{S \rightarrow AB, A \rightarrow aA/a, B \rightarrow b\}$

I_0

$S' \rightarrow .S$
$S \rightarrow .AB$
$A \rightarrow .aA$
$A \rightarrow .a$

I_i

$A \rightarrow \alpha . B\beta$
$B \rightarrow .\gamma$

In general

ii) construction of Goto operation

If $A \rightarrow \alpha .B\beta$ is in I_i then Goto $(I_i, B) = A \rightarrow \alpha B . \beta$

Here B can be a terminal or non terminal.

Ques Construct canonical set of items for the following grammer.

1. $G_1 = \{ S \to AB,\ A \to aA/a,\ B \to b \}$

I_0
- $S' \to .S$
- $S \to .AB$
- $A \to .aA$
- $A \to .a$

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

I_2: Goto(I_0, A)
- $S \to A.B$
- $B \to .b$

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

I_4: Goto(I_2, B)
- $S \to AB.$

I_5: Goto(I_2, b)
- $B \to b.$

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

I_7: Goto(I_3, a)

It is same as I_3

- $A \to a.A$
- $A \to a.$
- $A \to .aA$
- $A \to .a$

X Cancel.

2. $G_1 = \{ E = E+T/T,\ T \to T*F/F,\ F \to (E)/id \}$

I_0
- $E' = .E$
- $E = .E+T$
- $E = .T$
- $T = .T*F$
- $T = .F$
- $F = .(E)$
- $F = .id$

I_1: Goto(I_0, E)
- $E' \to E.$
- $E \to E.+T$

I_2: Goto(I_0, T)
- $E = T.$
- $T = T.*F$

I_3: Goto(I_0, F)
- $T = F.$

I_4: Goto$(I_0, ()$
- $F \to (.E)$
- $E \to .E+T$
- $E \to .T$
- $T \to .T*F$
- $T \to .F$
- $F \to .(E)$
- $F \to .id$

I_5: Goto(I_0, id)
- $F \to id.$

I_6: Goto$(I_1, +)$
- $E \to E+.T$
- $T \to .T*F$
- $T \to .F$
- $F \to .(E)$
- $F \to .id$

I_7: Goto$(I_2, *)$
- $T \to T*.F$
- $F \to .(E)$
- $F \to .id$

I_8: Goto(I_4, E)
- $F \to (E.)$
- $E \to E.+T$

Goto$(I_4, T) = I_2$
Goto$(I_4, F) = I_3$
Goto$(I_4, () = I_4$
Goto$(I_4, id) = I_5$

I_9: Goto(I_6, T)
- $E \to E+T.$
- $T \to T.*F$

Goto$(I_6, F) = I_3$
Goto$(I_6, () = I_4$
Goto$(I_6, id) = I_5$

Goto$(I_6, *) = I_7$

I_{10}: Goto(I_7, F)
- $T \to T*F.$

Goto$(I_7, () = I_4$
Goto$(I_7, id) = I_5$

I_{11}: Goto$(I_8,))$
- $F \to (E).$

Goto$(I_8, +) = I_6$

2. Construction of SLR (1) parsing table

It can be divided into 2 parts

→ Action part of the table will be filled as follows:

- If $A \to \alpha . a\beta$ is in I_i and if Goto $(I_i, a) = I_j$ then
  Set Action $[i, a] = S_j$ (shift - j)
- If $A \to \alpha .$ is in I_i then
  Set Action $[i, x]$ = Reduce by $A \to \alpha$
  where $x \in$ follow (A)
  Here $A \to \alpha .$ must not be an augmented production
- If $S' \to S.$ then and it is in I_i
  Set Action $[i, \$]$ = Accept

→ Goto part of the table will be filled as follows:

- If $A \to \alpha . B\beta$ is in I_i, and if
  Goto $(I_i, B) = I_j$ then
  Set GOTO $[i, B] = j$

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

Note: For final item, we will reduce that production in the follow of LHS of arrow

When terminal comes we will go for shift in Action part for the next state

When non terminal comes we will write only number for the going state in goto part.

R4

States	ACTION						GOTO		
	+	*	(	)	id	$	E	T	F
0	×	Syntax error	S_4		S_5		1	2	3
1	S_6					ACCEPT			
2	R_2	S_7		R_2		R_2			
→ 3	R_4	R_4		R_4		R_4			
4			S_4		S_5		8	2	3
5	R_6	R_6		R_6		R_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9	R_1	S_7		R_1		R_1			
10	R_3			R_3		R_3			
11	R_5			R_5		R_5			

In the above table all the entries are single. Therefore grammar is SLR(1)

Conflicts in SLR(1)

- Shift - Reduce [S/R conflict]

I_i | $A \to \alpha . a\beta$
 $B \to \gamma .$

$\Rightarrow \{a\} \cap \{follow(B)\} \neq \phi$

$\Rightarrow$ S/R conflict

$\Rightarrow$ Not SLR(1)

- Reduce - Reduce [R/R conflict]

I_i | $A \to \alpha .$
 $B \to \gamma .$

$\Rightarrow follow(A) \cap follow(B) \neq \phi$

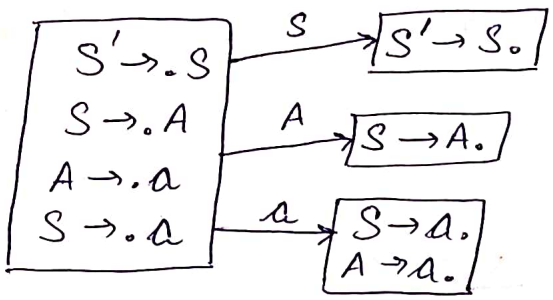
$\Rightarrow$ R/R conflict

$\Rightarrow$ Not SLR(1)

Ques check whether the following grammar is SLR(1) or not

1. $G = \{ S \to A/a,\ A \to a \}$

Sol



Follow(S) ∩ Follow(A)

$\{\$\} \cap \{\$\}$

$\neq \phi$

RR conflict

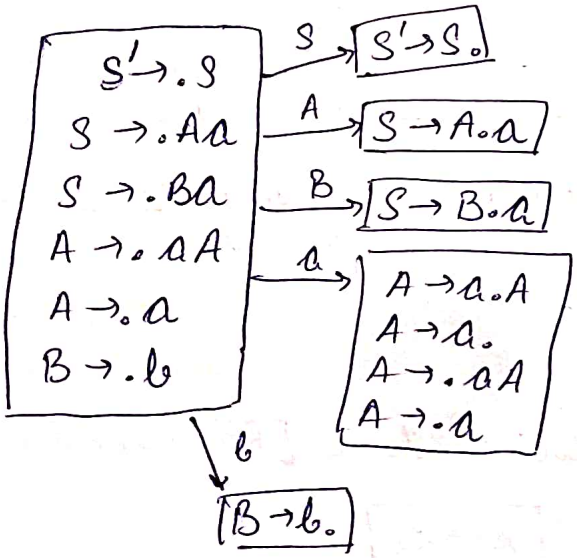
Not SLR(1)

Note: For augmented production, we write accept so no reduce-reduce or shift-reduce conflict.

⊙ For any conflict, we require atleast one final item.

⊙ When there is one production, there can't be any conflict.

2. $G = \{ S \to Aa/Ba,\ A \to aA/a.,\ B \to b \}$



$\Rightarrow \{a\} \cap$ Follow(A)

$\{a\} \cap \{a\}$

$\neq \phi$ S/R conflict

Not SLR(1)

3. $G_1 = \{ S \to Aa / bAc / Bc / bBa,\ A \to c,\ B \to c \}$

Sol

$S' \to .S$
$S \to .Aa$
$S \to .bAc$
$S \to .Bc$
$S \to .bBa$
$A \to .c$ —c→ $A \to c.$
$B \to .c$ —c→ $B \to c.$

follow(A) ∩ follow(B)

$\{a, c\} \cap \{a, c\} = \{a, c\} \neq \phi$ R-R conflict

Not SLR(1)

4. $G_1 = \{ S \to Aa / bAc / dc / bda,\ A \to c \}$

$S' \to .S$ —S→ $[S' \to S.]$
$S \to .Aa$ —A→ $[S \to A.a]$ —a→ $[S \to Aa.]$
$S \to .bAc$ —b→ $[S \to b.Ac,\ S \to b.da,\ A \to .c]$
$S \to .dc$
$S \to .bda$
$A \to .c$

$[S \to b.Ac]$ —A→ $[S \to bA.c]$ —c→
$[S \to b.da]$ —d→ $[S \to bd.a]$ —a→
$[A \to .c]$ —c→ $[A \to c.]$

$[S \to d.c]$ —c→ $[S \to dc.]$
$[S \to d.c]$ —c→ $[A \to c.]$

No conflict

It is SLR(1)

5. $G = \{ S \to aA / b,\ A \to Ab / \epsilon \}$

$S' \to .S$ —S→ $S' \to S.$
$S \to .aA$ —a→ $S \to a.A$
$S \to .b$
$A \to .Ab$
$A \to .$

$S \to a.A$ —A→ $S \to aA.$ —b→ $A \to Ab.$
$A \to A.b$

$S \to .b$ —b→ $S \to b.$

Final item

$\{b\} \cap$ follow(S)

$\{b\} \cap \{\$\} = \phi$

The grammar is SLR(1)

6. $G = \{S \to aS/Ab,\ A \to aB/b,\ B \to a\}$

$S' \to .S$ —S→
$S \to .aS$ —a→ $S \to a.S$ —a→ $S \to a.S$
$S \to .Ab$ $A \to .aB$ $A \to .b$ $B \to .a$ Final.
$A \to .aB$ $S \to .aS$ $S \to .Ab$ $A \to .aB$ $A \to .b$ $B \to .a$
$A \to .b$ $S \to .Ab$ $B \to .a$
$A \to .aB$
$A \to .b$
↓b ↘A

$\Rightarrow \{a, b\} \cap Follow(B)$
$\{a, b\} \cap \{b\}$
$\{b\} \neq \phi$
S/R conflict
Not SLR(1)

7. $G = \{S \to AS/b,\ A \to SA/a\}$

$S' \to .S$ —A→ $S \to A.S$ —S→ $S \to AS.$ Final
$S \to .AS$ $S \to .AS$ $A \to S.A$
$S \to .b$ $S \to .b$ $A \to .SA$
$A \to .SA$ $A \to .SA$ $A \to .a$
$A \to .a$ $A \to .a$ $S \to .AS$
$S \to .b$
↘S
$S' \to S.$
$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

$\Rightarrow \{a, b\} \cap Follow(S)$
$\{a, b\} \cap \{\$, a, b\}$
$\{a, b\} \neq \phi$
S-R conflict
Not SLR(1)

8. $G = \{S \to A*B/+B/a,\ A \to +SB/*,\ B \to Aa/+\}$

$S' \to .S$ —+→ $S \to +.B$ —+→ $S \to +.B$
$S \to .A*B$ $A \to +.SB$ $A \to +.SB$
$S \to .+B$ $S \to .A*B$ $B \to +.$
$S \to .a$ $S \to .+B$ $S \to .A*B$
$A \to .+SB$ $S \to .a$ $S \to .+B$
$A \to .*$ $A \to .+SB$ $S \to .a$
$A \to .*$ $A \to .+SB$
$B \to .Aa$ $A \to .*$
$B \to .+$ $B \to .Aa$
$B \to .+$

Follow(B) $\cap \{+, *, a\}$
$\{+\ldots\} \cap \{+, *, a\}$
$\{+\} \neq \phi$
S/R conflict
Not SLR(1)

Ques Consider the following augmented grammar

$S' \to S$
$S \to S\#CS$
$S \to SS$
$S \to S@$
$S \to \langle S \rangle$
$S \to a$
$S \to b$
$S \to c$

Let I_0 = closure $(S' \to .S)$, then no. of items in the set Goto(Goto(I_0, <), <) is?

I_0: $S' \to .S$

<

$S \to \langle .S \rangle$
$S \to .S\#CS$
$S \to .SS$
$S \to .S@$
$S \to .\langle S \rangle$
$S \to .a$
$S \to .b$
$S \to .c$

<

$S \to \langle .S \rangle$
$S \to .S\#CS$
$S \to .SS$
$S \to .S@$
$S \to .\langle S \rangle$
$S \to .a$
$S \to .b$
$S \to .c$

Ans (8)

LR(0) parsing

It can be divided into 2 parts

1. Construction of canonical set of items (or) LR(0) items
   It is same as construction of LR(0) items in SLR(1)

2. Construction of parsing table
   It is also same as construction of parsing table in SLR(1) except that if $A \to \alpha.$ is in I_i, then write Reduce by $A \to \alpha$ in entire Action part of i^{th} row of the table.

Conflicts in LR(0) parsing

- Shift-Reduce S/R conflict

I_i

$A \to \alpha.a\beta$
$B \to \gamma.$

S/R conflict
Not LR(0)

- Reduce-Reduce R/R conflict

$A \to \alpha.$
$B \to \gamma.$

R/R conflict
Not LR(0)

Note: A state having final item with shift or another final item will have conflicts.

Q) The following grammar is?

$G_1 = \{ S \to Aa/bAc/d,\ A \to d \}$

a) LR(0) but not SLR(1)

b) SLR(1) but not LR(0)

c) Both LR(0) ~~but not~~ and SLR(1)

d) Neither LR(0) nor SLR(1)

Sol

$S' \to .S$

$S \to .Aa$

$S \to .bAc$

$S \to .d$

$A \to .d$

$\xrightarrow{d}$ $\boxed{\begin{matrix} S \to d. \\ A \to d. \end{matrix}}$ Two final item ∴ R/R conflict in LR(0) not LR(0)

∴ (b)

$\downarrow b$

$S \to b.AC$ $\xrightarrow{A}$ □

$A \to .d$ $\xrightarrow{d}$ □

follow(S) ∩ follow(A)

$\$ \cap \{a, c\} = \phi$ ∴ SLR(1)

Ques $G_1 = \{ S \to SA/A,\ A \to a \}$

Sol

$S' \to .S$

$S \to .SA$

$S \to .A$

$A \to .a$

$\xrightarrow{S}$ $S' \to S.$ ← Not consider as final item as it is augmented

$S \to S.A$

$A \to .a$

$\xrightarrow{A}$ □

$\xrightarrow{a}$ □

No conflict

(c)

Ques $G_1 = \{ S \to AbB/a,\ A \to aA/a,\ B \to b \}$

$S' \to .S$

$S \to .AbB$

$S \to .a$

$A \to .aA$

$A \to .a$

$\xrightarrow{a}$ $S \to a.$

$A \to a.$

$A \to a.A$

$A \to .aA$

$A \to .a$

S/R & R/R conflict in LR(0) ⇒ Not LR(0)

$\downarrow A$

$S \to A.bB$

$\downarrow b$

$S \to Ab.B$

$B \to .b$

$\{a\} \cap follow(S) \cap follow(A)$

$\{a\} \cap \{\$\} \cap \{b\}$

$= \phi$

∴ (b)

Page-71

(b) Time.

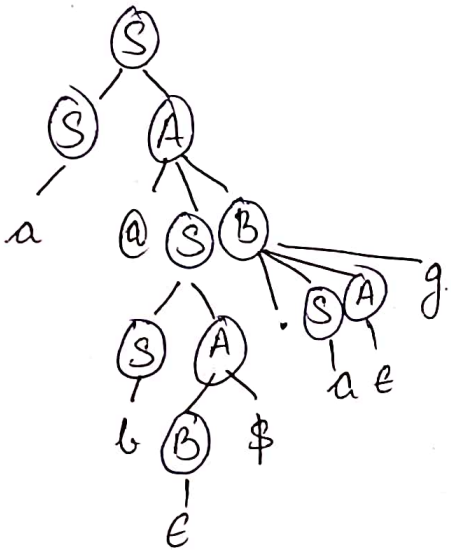
Q-46 $E \to AaBcD$
$A \to bA \mid e$
$B \to bB \mid e$
$D \to eD \mid g$

a) $a <\cdot b$
$E \to AaBcD$ ∴ False
$bAaBcD$

c) $a <\cdot e$
$E \to AaBcD$ ∴ False
$eaBcD$

d) ~~$b <\cdot e$~~ $c <\cdot b$ ∴ False
$AaBcD$
$bAaBcD$

Q-91



1D

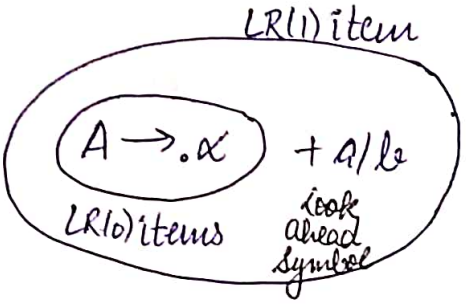
CLR(1) parsing

LR(0) items

LR(0) parsers — SLR(1) parsers

LR(1) items

LALR(1) parsers — CLR(1) parsers



LR(1) item = LR(0) item + LAS [look ahead symbol]

⇒ It can be divided into 2 parts

1. Construction of canonical set of items (or) LR(1) items
   It has 2 parts
   i) Construction of closure operation
      a. Initially add $S' \to .S, \$$
      b. If $A \to \alpha.B\beta, a$ is in I_i and if $B \to \gamma$ is a production then add $B \to .\gamma, x$ in I_i
         where $x \in First(\beta, a)$

I_i | $A \to \alpha.B\beta, a$
 $B \to .\gamma, x$ | where $x \in First(\beta a)$

Note: ϵ will never be present in look ahead

   ii) Construction of Goto operation
      a. If $A \to \alpha.X\beta, a$ is in I_i then
         $Goto(I_i, X) = A \to \alpha X.\beta a$
         Here X can be a terminal or non terminal.

2. Construction of parsing table

Construct LR(1) items for the following grammars

$G = \{S \to AB, A \to aA/a, B \to b\}$

I_0
$S' \to .S, \$$
$S \to .AB, \$$
$A \to .aA, b$
$A \to .a, b$

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

$Goto(I_0, A) = I_2$
$S \to A.B, \$$
$B \to .b, \$$

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

$Goto(I_2, B) = I_4$
$S \to AB., \$$

$Goto(I_2, b) = I_5$
$B \to b., \$$

$Goto(I_3, A) = I_6$
$A \to aA., b$

$Goto(I_3, a) = I_3$

9) Construct CLR(1) parsing table for the following grammar.

$G = \{S \to CC, C \to aC/d\}$

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

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

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

I_3: $Goto(I_0, a)$
$C \to a.C, d/a$
$C \to .aC, a/d$
$C \to .d, a/d$

I_4: $Goto(I_0, d)$
$C \to d., a/d$

I_5: $Goto(I_2, C)$
$S \to CC., \$$

I_6: $Goto(I_2, a)$
$C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

I_7: $Goto(I_2, d)$
$C \to d., \$$

I_8: $Goto(I_3, C)$
$C \to aC., d/a$

I_9: $Goto(I_3, a)$
$C \to a.C, a/d.$
$C \to .aC, a/d$
$C \to .aC, a/d.$

I_9: $Goto(I_6, C)$
$C \to aC., \$$

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

2. Construction of CLR(1) parsing table

It has 2 parts

1. ACTION
2. GOTO

→ Action part of the table will be filled as follows:

- If $A \to \alpha . a\beta, b$ is in I_i and if $Goto(I_i, a) = I_j$ then set $Action[i, a] = S_j$
- If $A \to \alpha ., a$ is in I_i, then set action $[i, a]$ = Reduce by $A \to \alpha$

  Here $A \to \alpha .$ must not be augmented production.
- If $S' \to S., \$$ is in I_i, then set $Action[i, \$]$ = Accept

→ Goto part of the table will be filled as follows:

- If $A \to \alpha . B\beta, a$ is in I_i and if $Goto(I_i, B) = I_j$ then set $Goto[i, B] = j$

1) $S \to CC$
2) $C \to aC$
3) $C \to d$

States	ACTION			GOTO	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

In the above table all the entries are single therefore the grammar is CLR(1)

Conflicts in CLR(1)

- Shift - Reduce [S/R conflict]

Ii $\boxed{\begin{matrix} A \to \alpha . a\beta, b \\ B \to \gamma ., a \end{matrix}}$

→ S/R conflict
→ Not CLR(1)

- Reduce - Reduce [R/R conflict]

Ii $\boxed{\begin{matrix} A \to \alpha ., a \\ B \to \gamma ., a \end{matrix}}$

→ R/R conflict
→ Not CLR(1)

Ques $G = \{S \to A/a,\ A \to a\}$

Sol

$S' \to .S, \$$
$S \to .A, \$$
$S \to .a, \$$
$A \to .a, \$$

$\xrightarrow{S}$ $S' \to S., \$$

$\xrightarrow{A}$ $S \to A., \$$

$\xrightarrow{a}$ $S \to a., \$$
$A \to a., \$$

Two final item with same look ahead. ∴ not CLR(1)

R/R conflict

Ques $G = \{S \to Aa,\ A \to Ba/ba,\ B \to b\}$

Sol

$S' \to .S, \$$
$S \to .Aa, \$$
$A \to .Ba, a$
$A \to .ba, a$
$B \to .b, a$

$\xrightarrow{b}$ $A \to b.a, a$
$B \to .b, a$

S/R conflict
⇒ not CLR(1)

Ques $G = \{S \to aSa/Aa,\ A \to aA/b\}$

Sol

$S' \to .S, \$$
$S \to .aSa, \$$
$S \to .Aa, \$$
$A \to .aA, a$
$A \to .b, a$

$\xrightarrow{a}$ $S \to a.Sa, \$$
$A \to a.A, a$
$S \to .aSa, a$
$S \to .Aa, a$
$A \to .aA, a$
$A \to .b, a$

$\xrightarrow{A}$ $A \to aA., a$
$S \to A.a, a$

S/R conflict
⇒ not CLR(1)

LALR(1) parsing

In CLR(1) those states having everything as same but with a different lookaheads, we will divide them into 2 different states.

But in LALR(1) parsing we will merge them into single state.

Construction of LALR(1) parsing table is same as CLR(1)

Que construct LALR(1) parsing table for the following grammer.

$G = \{ S \to CC,\ C \to aC \mid d \}$

I_0

$S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

I_1: Goto(I_0, S)

$S' \to S., \$$

I_2: Goto(I_0, C)

$S \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

I_5: Goto(I_2, C)

$S \to CC., \$$

I_{89}: Goto(I_{36}, C)

$C \to aC., a/d/\$$

I_3

$C \to a.C, d/a$
$C \to .aC, a/d$
$C \to .d, a/d$

+

I_6

$C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

$\Rightarrow$

I_{36}: Goto(I_0, a)

$C \to a.C, a/d/\$$
$C \to .aC, a/d/\$$
$C \to .d, a/d/\$$

I_4

$C \to d., a/d$

+

I_7

$C \to d., \$$

$\Rightarrow$

I_{47}: Goto(I_0, d)

$C \to d., a/d/\$$

Goto(I_2, a) = I_{36}
Goto(I_2, d) = I_{47}
Goto(I_{36}, C) = I_{89}
Goto(I_{36}, a) = I_{36}
Goto(I_{36}, d) = I_{47}

States	ACTION			GOTO	
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
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		

All the entries are single, so grammar is LALR(1)

Conflicts in LALR(1)

- Shift - Reduce
- Reduce - Reduce

Note: The techniques for conflict in LALR(1) are same as techniques for CLR(1)

Ques The following grammer is

$G = \{S \to Aa / bAc / Bc / bBa,\ A \to d,\ B \to d\}$

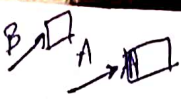
a) LALR(1) but not LR(1)

b) LR(1) but not LALR(1)

c) Both LALR(1) & CLR(1)

d) Neither LALR(1) nor LR(1)

Sol



$S' \to .S, \$$
$S \to .Aa, \$$
$S \to .bAc, \$$
$S \to .Bc, \$$
$S \to .bBa, \$$
$A \to .d, a$
$B \to .d, c$

$b \to$ $S \to b.Ac, \$$, $S \to b.Ba, \$$, $A \to .d, c$, $B \to .d, a$

$S \to$

$d \to$ I_i: $A \to .d, a$; $B \to .d, c$ — Two final item But look ahead diff

$d \to$ I_j: $A \to d., c$; $B \to d., a$ — Two final item but lookahead diff.

$\therefore$ LR(1)

Merge $I_i + I_j$

$A \to .d, a/c$
$B \to .d, a/c$

Two final item and look ahead same

$\therefore$ R/R conflict Not LALR(1).

$\therefore$ Not LALR(1) but CLR(1) [(b)]

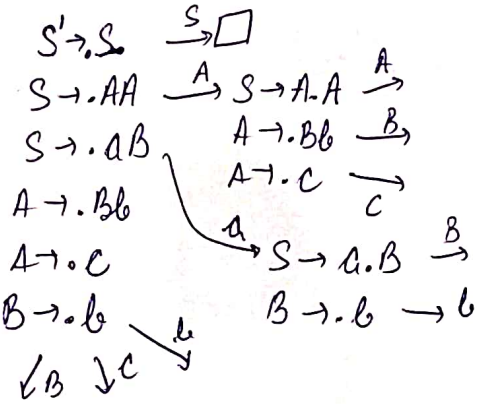
Ques $G = \{S \to AA/aB,\ A \to Bb/c,\ B \to b\}$

- LL(1)
- LR(0)
- SLR(1)
- LALR(1)
- CLR(1)

Sol $S \to AA/aB$, $A \to Bb/c$, $B \to b$ ✓ $\therefore$ LL(1)

$\{b, c\} \cap \{a\}$ $\{b\} \cap \{c\}$

ϕ ϕ



There are no conflicts for LR(0)
The grammar is in LR(0)

→ G is also SLR(1)
→ G is also LALR(1)
→ G is also CLR(1)

	CLR(1)	LALR(1)
S/R Conflict	Yes	Yes
	No	No
R/R Conflict	Yes	Yes
	No	Yes / No

$A \to \alpha . a\beta, a$
$B \to \gamma ., b$
No S/R

$+$

$A \to \alpha . a\beta, b$
$B \to \gamma ., a$
S/R conflict

$=$

$A \to \alpha . a\beta, a|b$
$B \to \gamma ., a|b$
S/R conflict

$A \to \alpha . a\beta, a$
$B \to \gamma ., b$
No S/R

$+$

$A \to \alpha . a\beta, b$
$B \to \gamma ., c$
No S/R

$=$

$A = \alpha . a\beta, a|b$
$B \to \gamma ., b|c$
No S/R

$A \to \alpha ., a$
$B \to \gamma ., b$
No R/R

$+$

$A \to \alpha ., c$
$B \to \gamma ., c$
R/R

$=$

$A \to \alpha ., a|c$
$B \to \gamma ., b|c$
R/R conflict

$A \to \alpha ., a$
$B \to \gamma ., b$
No R/R

$+$

$A \to \alpha ., c$
$B \to \gamma ., a$
No R/R

$=$

$A = \alpha ., a|c$
$B = \gamma ., b|a$
R/R

$A \to \alpha ., a$
$B \to \gamma ., b$
No R/R

$+$

$A \to \alpha ., c$
$B \to \gamma ., d$
No R/R

$=$

$A = \alpha ., a|c$
$B = \gamma ., b|d$
No R/R

- $LR(0) \subset SLR(1) \subset LALR(1) \subset CLR(1)$
- $LL(1) \subset CLR(1)$
- $LL(1) \subset LL(2) \subset LL(3) \ldots \subset LL(K)$
- $LL(2) \subset CLR(2)$
- $LL(K) \subset CLR(K)$
- Let the sizes of parsing tables of LR(0), SLR(1), LALR(1) and CLR(1) are n_1, n_2, n_3, n_4 respectively then
  $(n_1 = n_2 = n_3) \leq n_4$
- Among SLR(1), LALR(1), CLR(1), the most powerful is CLR(1) and most easy method is SLR(1)
- If a grammar is right linear grammar and if has null string then the grammar is not LR(0)

Eg: $S \to wA$
$S \to \epsilon$

$S' \to .S$
$S \to .wA$ shift op
$S \to .$ → Final item so in LR(0) we don't check the follow ∴ Reduce in whole row.

∴ not LR(0)

Ques $G_1 = \{ S \to aAS / Ba,\ A \to Ab / aS,\ B \to bB / \epsilon \}$

Sol $\{a\} \cap \{a, b\}$ — Left recursion

$\{a\}$ — Not LL(1)

Not LL(1)

$S' \to .S, \$$
$S \to .aAS, \$$
$S \to .Ba, \$$
$B \to .bB, a$
$B \to ., a$

SR conflict

not CLR(1) $\Rightarrow$ not LALR(1) ∴ None

not SLR(1)

not LR(0)

Q-2

Q-3 $S \to CC$

$S \to cC / d$

LL(1) ✓

$S' \to .S$ $\xrightarrow{S}$

$S \to .CC$ $\xrightarrow{C}$ $S \to C.C$ $\xrightarrow{C}$

$S \to .cC$ $C \to .cC$

$S \to .d$ $C \to .d$ $\xrightarrow{c}$

$\downarrow d$ $\xrightarrow{c}$ $C \to c.C$ $\xrightarrow{C}$

$C \to .cC$ $\xrightarrow{c}$

$C \to .d$ $\xrightarrow{d}$

Q-31

$S \to TA$

$A \to \epsilon / +TA$

$T \to i / n$

Follow(A) $\cap \{+\}$

$\{\$\} \cap \{+\}$

$= \phi$

∴ (1)

LL(1)

$\{i\} \cap \{n\}$

ϕ

$S' \to .S, \$$

$S \to .TA, \$$ $\xrightarrow{T}$ $S \to T.A, \$$

$A \to ., \$$ ← Final item But $ it will never make conflict as $ is not i/p symbol

$T \to .i, +/\$$ $\xrightarrow{i}$ $A \to .+TA, \$$

$T \to .n, +/\$$ $\xrightarrow{n}$

$A \to +.TA, \$$ $\xrightarrow{+}$

$T \to .i, +/\$$

$T \to .n, +/\$$

$A \to +T.A, \$ \to A$

$A \to ., \$$

$A \to .+TA, \$$

$T \uparrow \quad \downarrow +$

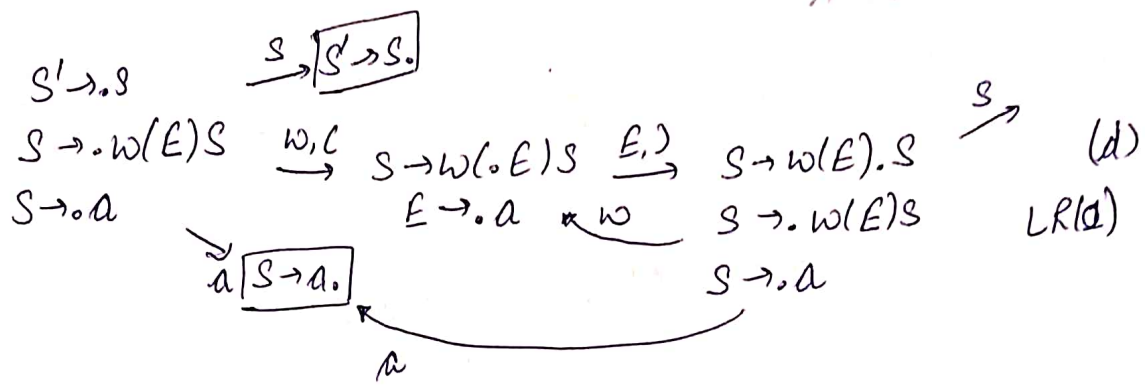
No conflict CLR(1) ✓ LALR(1) ✓

Q-#4 $S \to w(E)S/a$

$E \to a$

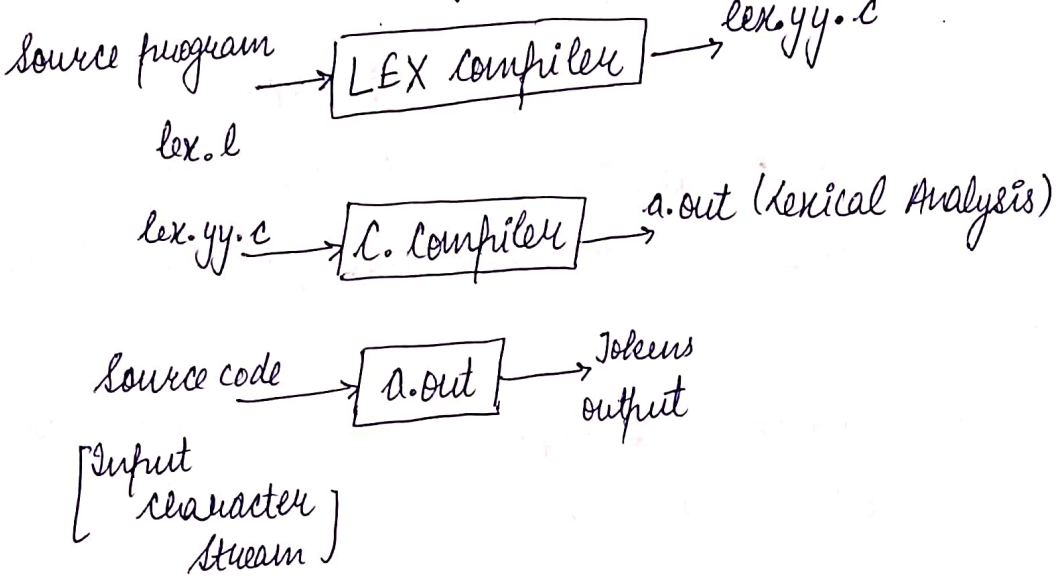
$S \to T.A$

$A \to .$ or $A \to a.$



LEX tool

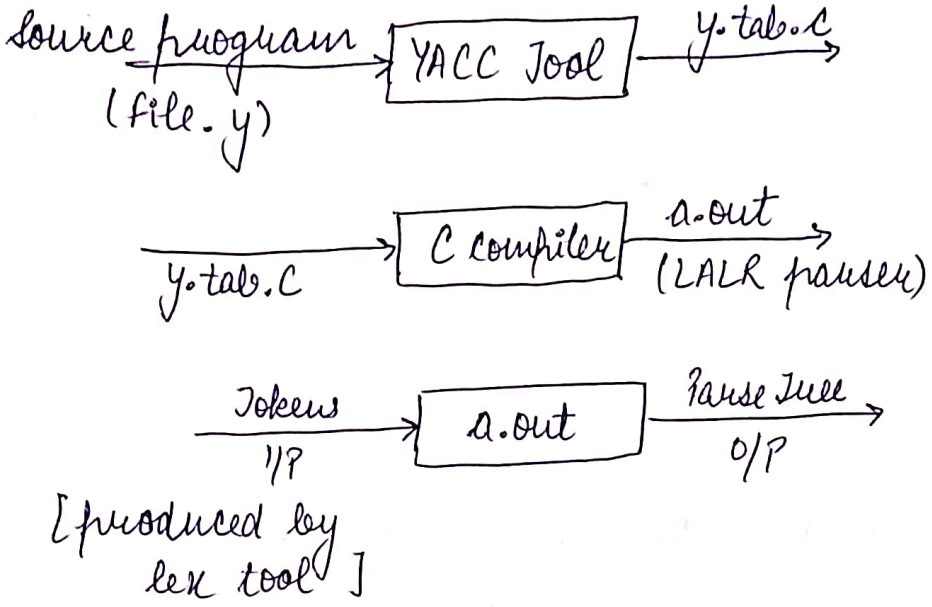
LEX tool is used to generate the lexical analyser.



- Lex is a tool which is used to generate the lexical analysis.
- The source program (i.e. which is written for developing the lexical analyser) will be given as input with file name lex.l to the lex compiler, then it produces the o/p lex.yy.c
- This lex.yy.c will be given as input to the C compiler, then the C compiler will produce the output as a.out which is the required lexical analyser.

YACC Tool

[Yet another compiler compiler]



Note: An LALR parser which is developed using YACC tool, if it detects shift-reduce conflict for any grammar, then it performs shift operation over reduce.

YACC tool evaluates the expression from right to left (i.e. right associative)

⊕ SYNTAX DIRECTED TRANSLATION [SDT]

The tokens produced by lexical analyser, will be given as input to the syntax analyser to produce a parse tree.

The syntax analyser can not check the meaning of the source code.

The meaning of the source code and type checking will be done in semantic analysis by adding some extra information to every production of the grammer.

The grammar with this extra information [i.e. semantic rules] is called as Syntax Directed Translation [SDT]

SDT = CFG + Semantic rules.

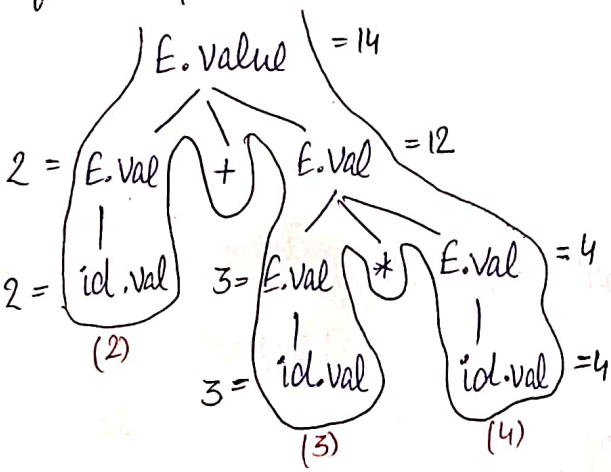
SDT

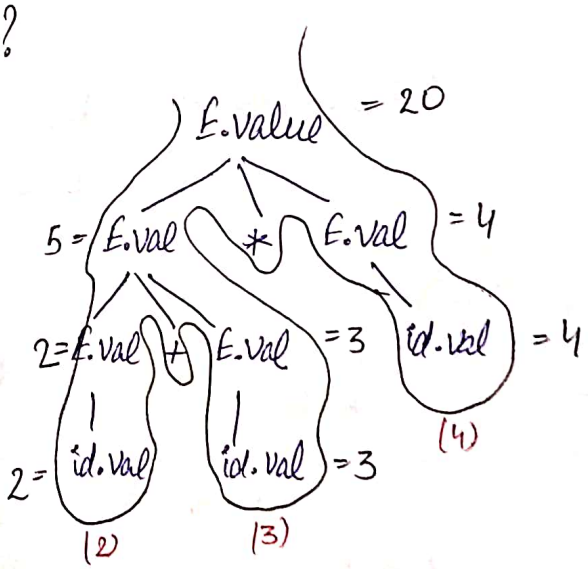
$E \rightarrow E_1 + E_2$ { E.value = E_1.value + E_2.value }

$E \rightarrow E_1 * E_2$ { E.value = E_1.value * E_2.value }

$E \rightarrow id$ { E.value = id.value }

If the i/p = 2+3*4 then O/P = ?

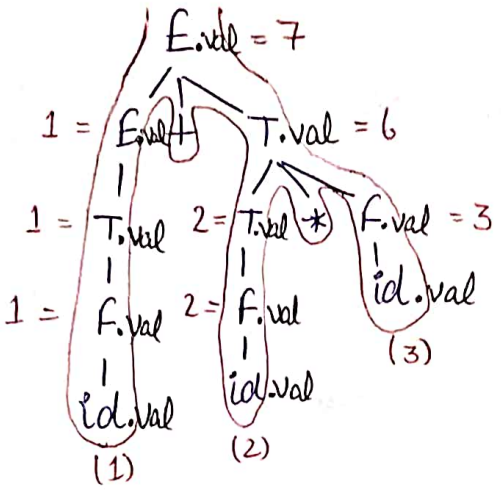




b) Consider the following SDT if the input is 1+2*3 then O/P is?

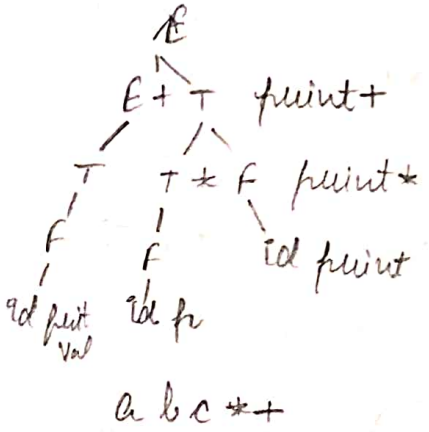
$E \rightarrow E_1 + T$	{ E.value = E_1.value + T.value }
$E \rightarrow T$	{ E.value = T.value }
$T \rightarrow T_1 * F$	{ T.value = T_1.value * F.value }
$T \rightarrow F$	{ T.value = F.value }
$F \rightarrow id$	{ F.value = id.value }

Sol

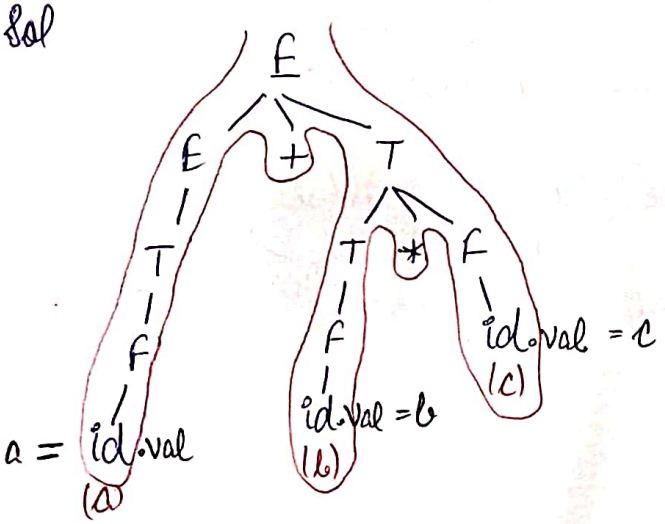


c) Consider the following SDT if the input string is a+b*c then output is?

$E \rightarrow E + T$	{ printf('+'); }
$E \rightarrow T$	{ }
$T \rightarrow T * F$	{ printf('*'); }
$T \rightarrow F$	{ }
$F \rightarrow id$	{ printf(id.val); }



Sol



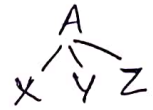
O/P = a b c * +

Types of attributes

- Synthesis attribute
- Inherited attribute

⊙ Synthesis attribute

The attribute whose value is evaluated in terms of attribute values of its children is called as synthesis attribute.



$A \rightarrow XYZ$

$A.S = f(X.S / Y.S / Z.S)$

↑ Synthesis attribute.

It is an attribute of a non terminal which is on the left hand side of a production.

⊙ Inherited attribute

The attribute whose value is evaluated in terms of attribute values of its present or siblings is called as inherited attribute.

$A \rightarrow XYZ$

$X.S = f(A.S / Y.S / Z.S)$

↑ inherited attribute

It is an attribute of a non terminal which is on the right hand side of a production.

Types of SDT

- S-attribute SDT
- L-attribute SDT

S-attribute SDT	L-attribute SDT
1. It uses synthesized attributes	1. It uses both synthesis and inherited attributes. If it is inherited attribute, it must inherit the values either from parent and/or left side of the siblings only.
2. Attributes are evaluated in bottom up approach as the parent value depends on its children	2. Attributes are evaluated in depth-first (top down) and left to right
3. Semantic actions will be placed at right most end of the RHS of the production $A \to XYZ \{\}$	3. Semantic actions can be placed anywhere on the RHS of the production. $A \to X\{\}YZ$, $A \to \{\}XYZ$, $A \to XY\{\}Z$, $A \to XYZ\{\}$

Q) The following SDT is

$A \to BC \{A.S = B.S\}$

$B \to AB_1 \{B_1.i = A.i\}$

a) S-attri
b) L-attri
c) Both (a) & (b)
d) None.

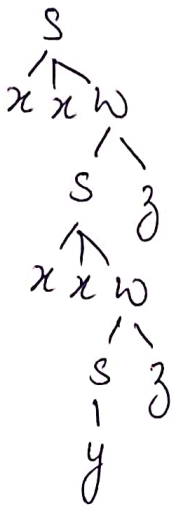
Sol b) L-attribute

Q) $A \to XSA_1 \{S.val = X.val,$
$X \to A_1SA \quad A_1.val = A.val\}$
$\{A_1.val = S.val, A.val = X.val\}$

d) None.

Ques $S \to xxW$ print("1"); }
$/y$ print("2"); }
$W \to Sz$ print("3"); }

xxxxyzz then o/p



O/P → 2 3 1 3 1

Ques

$E \rightarrow E_1 * T$ | T — $E.val = E_1.val * T.val$

$E.val = T.val$

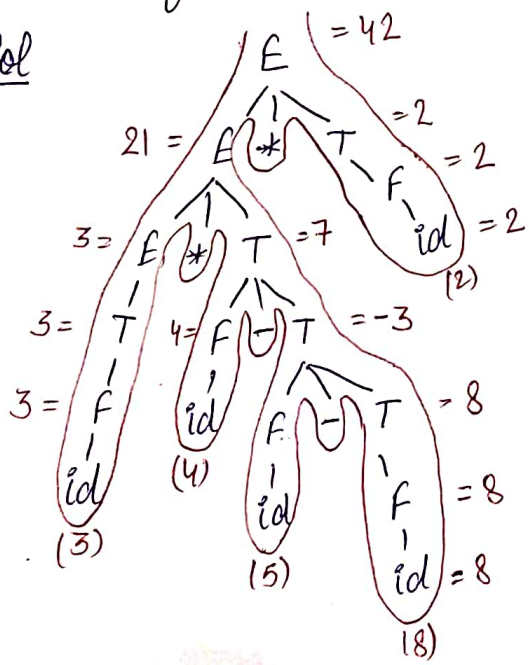
$T \rightarrow F - T_1$ | F — $T.val = F.val - T_1.val$

$T.val = F.val$

$F \rightarrow id$ — $F.val = id.val$

If input = 3 * 4 - 5 - 8 * 2 then?

Sol



go from the bottom to top in the production.

Operator which we are getting first has the higher priority.

- > *

Now find associative.

- is right associative.

* is left

3 * 4 - (5 - 8) * 2

3 * 4 - (-3) * 2

3 * (4 + 3) * 2

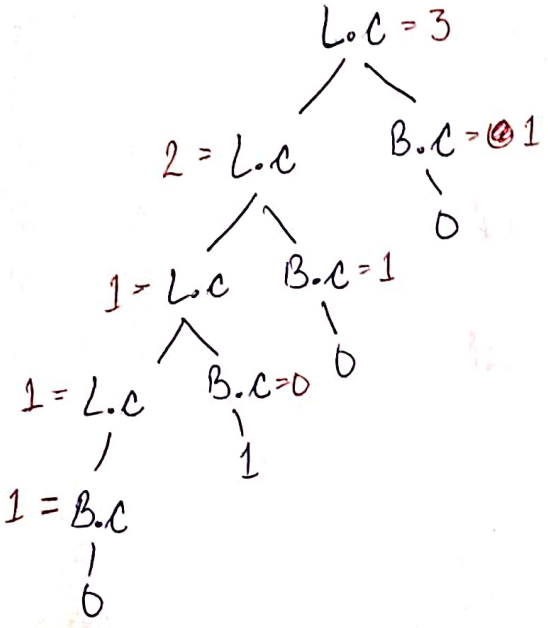
(3 * 7) * 2 = 21 * 2 = 42

Ques $L \to L_1 B$ { L.count = L_1.count + B.count }

$\quad / B$ { L.count = B.count }

$B \to 0$ { B.count = 1 }

$\quad / 1$ { B.count = 0 }



It count's the number of 0's in the input string.

① for count no of 1's

$B \to 0$ B.count = 0

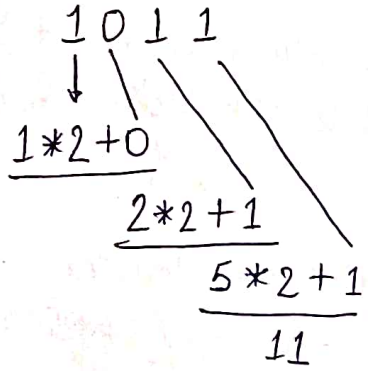
$/1$ B.count = 1

① for count no. of bits

$B \to 0$ B.count = 1

$/1$ B.count = 1

Ans Construct SDT to convert binary to decimal.

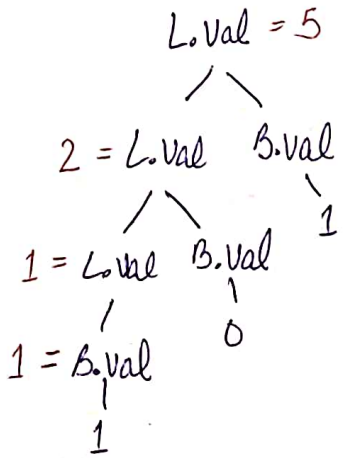


$L \to L_1 B$ { L.Val = L_1.Val * 2 + B.Val }

$\quad / B$ { L.Val = B.Val }

$B \to 0$ { B.val = 0 }

$B \to 1$ { B.val = 1 }



Convert ~~Bi~~ Binary to decimal with decimal places

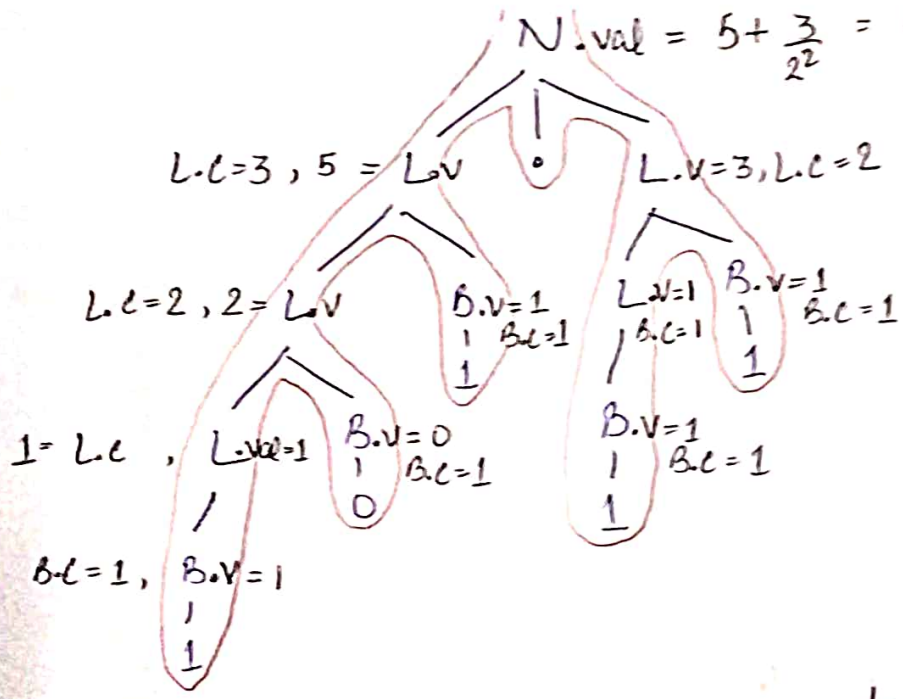
$N \to L_1 . L_2$ { $N.val = L_1.val + \frac{L_2.val}{2^{L_2.count}}$ }

$L \to L_1 B$ { $L.val = L_1.val * 2 + B.val$, $L.count = L_1.count + B.count$ }

/B { $L.val = B.val$, $L.count = B.count$ }

$B \to 0$ { $B.val = 0$, $B.count = 1$ }

/1 { $B.val = 1$, $B.count = 1$ }



$\frac{101 \cdot 11}{5 + \frac{3}{2^2}} = 5.75$

Construct SDT which can create a syntax tree for the following expression. $x = a + b * c$

$S \to f = E$ { $S.ptr = makenode(f.ptr, =, E.ptr)$ }

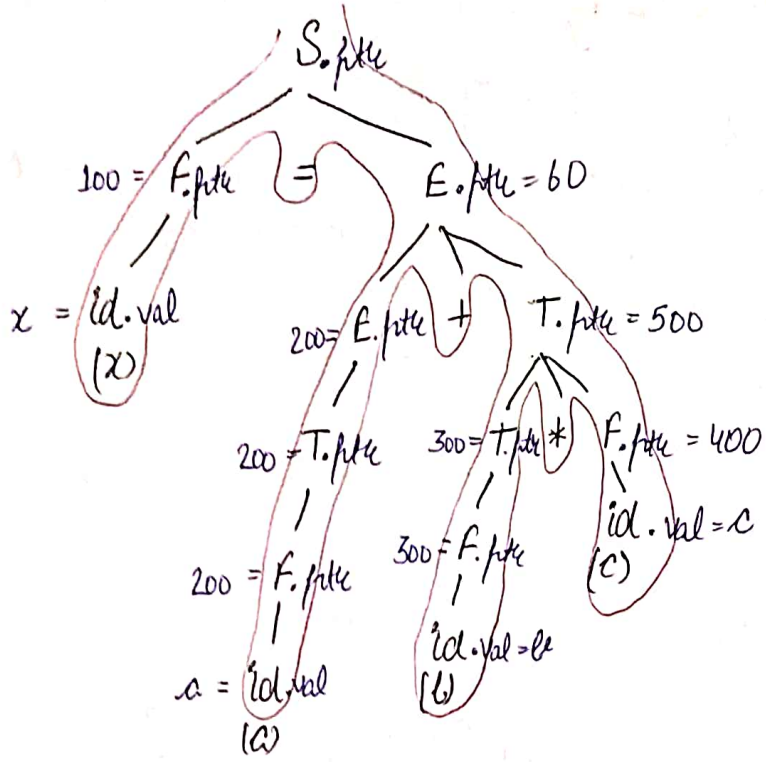
$E \to E_1 + T$ { $E.ptr = makenode(E_1.ptr, +, T.ptr)$ }

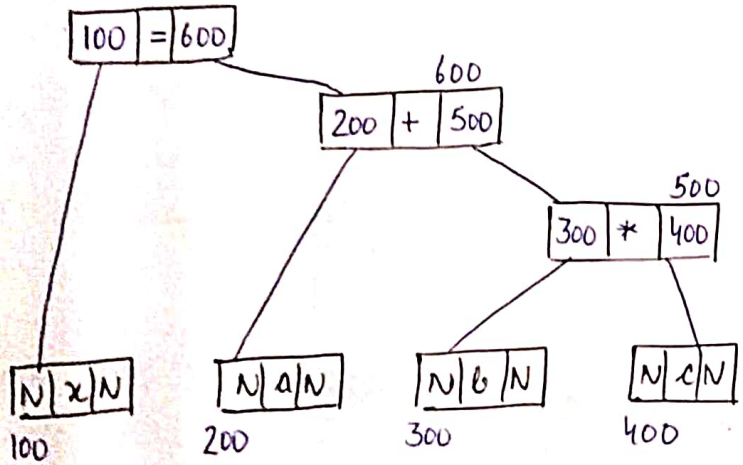
/T { $E.ptr = T.ptr$ }

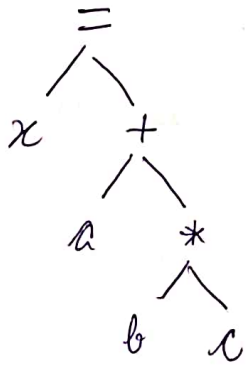
$T \to T_1 * f$ { $T.ptr = makenode(T_1.ptr, *, f.ptr)$ }

/F { $T.ptr = F.ptr$ }

$f \to id$ { $F.ptr = makenode(null, id.val, null)$ }







Ques Construct an SDT which can create a 3-address code for the following expression.

$x = a + b * c$

$S \rightarrow id = E$ { gen(id.name = E.place) }

$E \rightarrow E_1 + T$ { E.place = newtemp(); gen(E.place = E_1.place + T.place) }

/ T { E.place = T.place }

$T \rightarrow T_1 * F$ { T.place = newtemp(); gen(T.place = T_1.place * F.place) }

/ F { T.place = F.place }

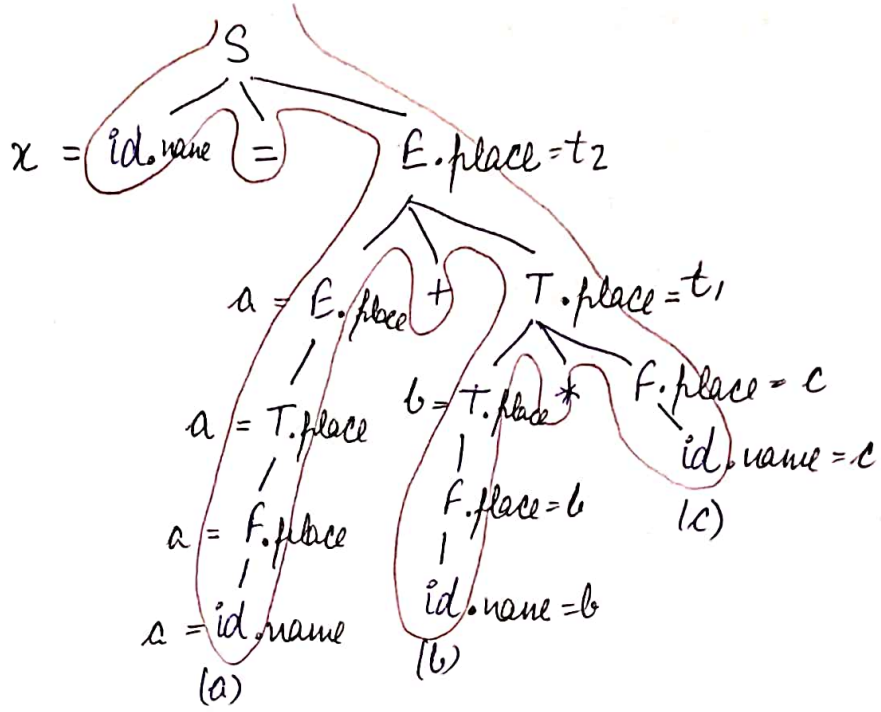
$F \rightarrow id$ { F.place = id.name }

$x = a + b * c$

O/P: $t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$



Ques If the I/P = $3 * 5 - 6 - 8 + 4 * 7$ then O/P = ?

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

/T { E.val = T.val }

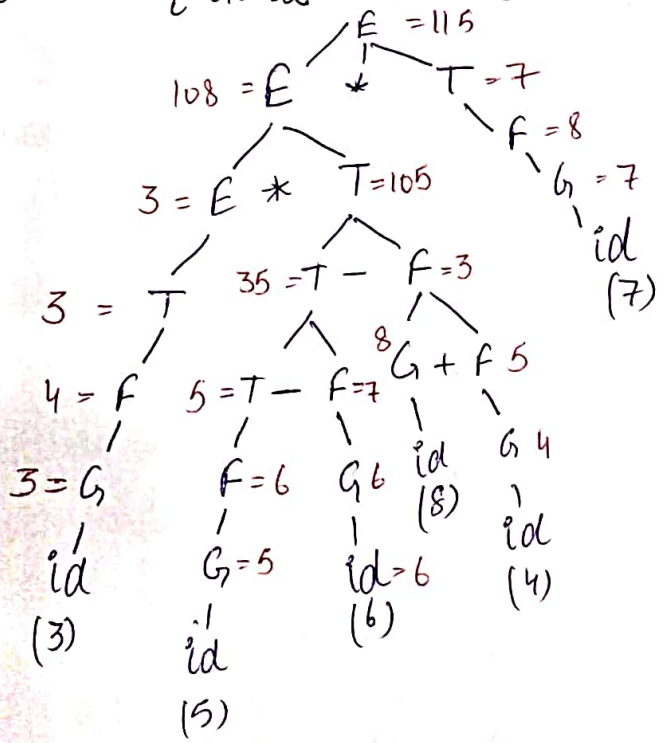
$T \rightarrow T - F$ { T.val = T_1.val * F.val }

/F { T.val = F.val − 1 }

$F \rightarrow G_1 + F$ { F.val = G_1.val − F.val }

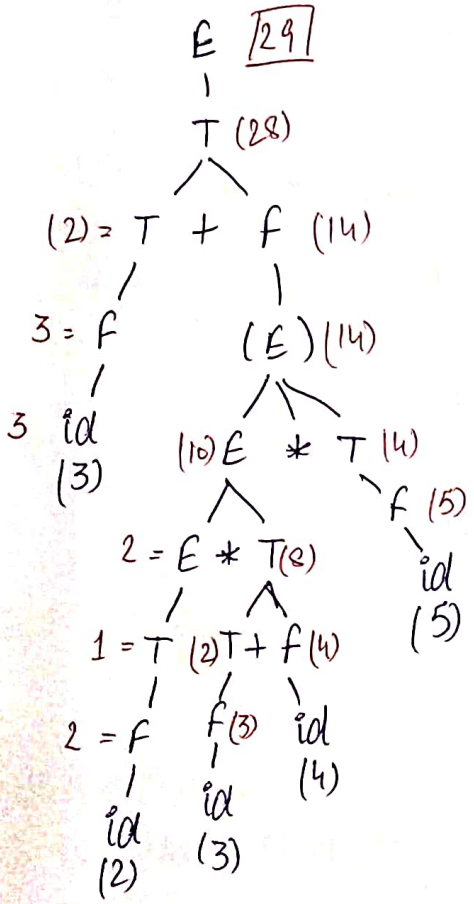
/G { F.val = G_1.val + 1 }

$G_1 \rightarrow id$ { G_1.val = id.val }



Q) If the i/p = 3+(2*3+4*5) then O/P = ?

$E \to E_1 * T$	$\{ E.val = E_1.val + T.val \}$
$/T$	$E.val = T.val + 1$
$T \to T_1 + F$	$T.val = T_1.val * F.val$
$/F$	$T.val = F.val - 1$
$F \to (E)$	$F.val = E.val$
$/id$	$F.val = id.val$



P-75

Q-77) $E \to E \uparrow T \,/\, T-E \,/\, T$

$T \to F+E \,/\, F*T \,/\, F$

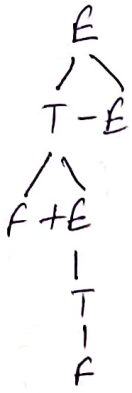
$F \to a$

$E \to T-E$

Here $T \to F+E$

we are getting E again in T

∴ (b)

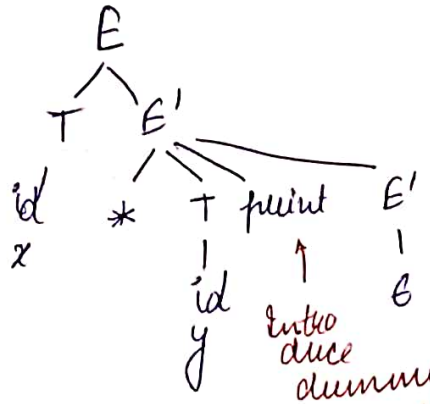


Q-79

$E \to TE'$

$E' \to *T$ { print (*) } E' / ϵ

$T \to id$ { print(id.name) }



Introduce dummy here. to convert to S. action should be at last.

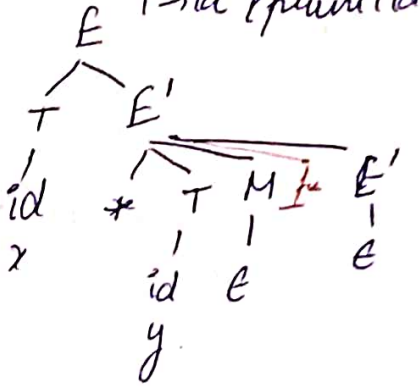
o/p → xy *

$E \to TE'$

$E' \to *TME' / \epsilon$

$M \to \epsilon$ { print (*) }

$T \to id$ { print(id.name) }



o/p → xy *

Semantic Errors

- Undeclared variables
- Incompatible type of operands
- Not matching of actual arguments ~~and~~ with formal one
- Undefined / Undeclared function name.

Error Recovery

Undeclared var → An entry of that variable will be done in symbol by semantic analysis.

Incompatible type of operands → Automatic type conversion will be done by semantic analysis.

Ques If the automatic type conversion is done by semantic analysis then find the final values of a, i, y from following code segment.

```
bool a = 5;        ⇒ a = True
int i = a;         ⇒ i = 1
    i = 2.53;      ⇒ i = 2
float y = i;       ⇒ y = 2.0
```

Ques Find the earliest error produced by the compiler for the following code segment.

1.
```
void main()
{
    int a=0;
    iff (a==0)
    {
        printf("compiler Design );
    }
}
```
← unmatched string
lexical analysis

2.
```
void main()
{
    int a=0;
    iff (a==0)
    {
        printf("compiler design");
    }
}
```
← syntax error [not ending with ;]

3.
```
void main()
{
    int a=0;
    iff (a==0);
    {
        printf("compiler design");
    }
}
```
← Semantic error [undeclared function name]

4.
```
void iff (int a);
void main()
{
    int a=0;
    iff (a==0);
    {
        printf("compiler design");
    }
}
```
// no syntax
no lexical
no semantic error

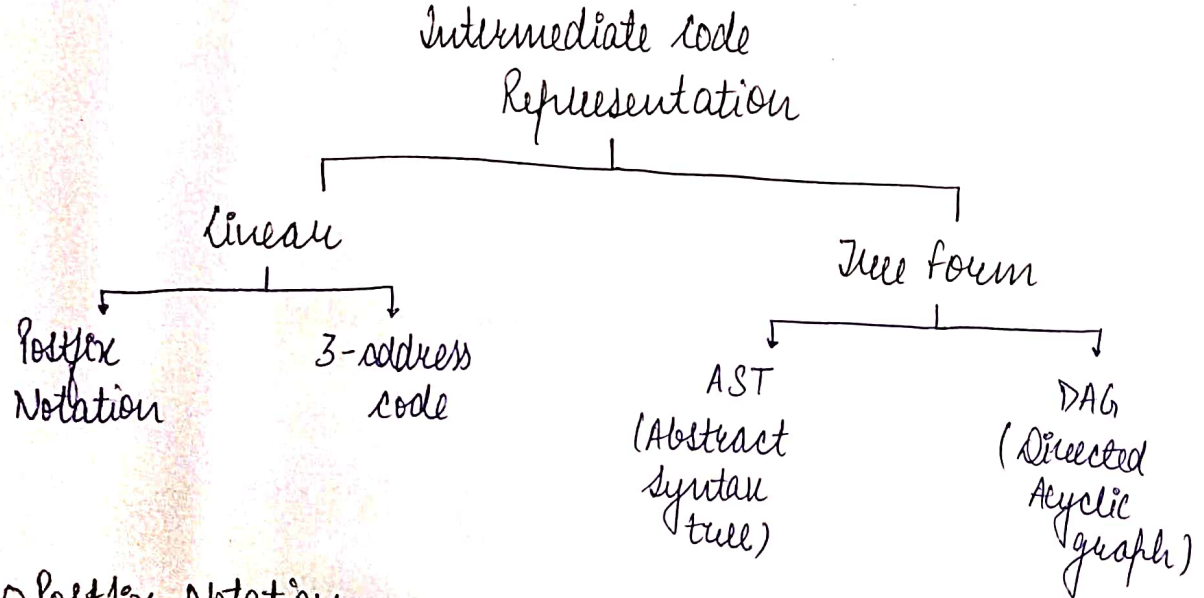
linking error

```
void iff ( int a);
 void main()
 { int a = 0;
    iff (a==0);
    { printf ("Compiler Design");
    }
}
void iff (int a)
    { return ;
    }
```

NO error

INTERMEDIATE CODE GENERATION (ICG)

- Intermediate code makes the code generation process simple and easy.
- It is a language between the source code and target code.
- The front end of a compiler produces independent intermediate code then the backend of a compiler uses this intermediate code to generate the target code.
- Because of machine independent intermediate code the portability will be enhanced.
- It is easier to apply the source code modifications to improve the performance of the source code by optimizing the intermediate code.



⊙ Postfix Notation

Operands first then operator.

a+b → ab+

b*c → bc*

a-b*c → abc*- [* > -]

ab-c* [- > +]

$x = ((a*b) + (a/b)) / ((a+b) * (a-b))$

$((ab*) + (ab/)) / ((ab+) * (ab-))$

$(ab * ab/ +) / (ab + ab - *)$

$x\ ab * ab/ + ab + ab - * / =$

⊙ 3 address code

In 3 address code in any statement there will be maximum 3 references.

Two for operands and one for result.

The sequence of 3 address statements is called as 3 address code.

Some valid 3 address statements

- $a = b$ op c [Relational assignment statement]
- $a =$ op c [Relational assignment statement]
- $a = b$ [copy statement]
- if (relation) Goto L [conditional jump]
- if (relation) Goto L [conditional jump]
- Goto L [unconditional jump]
- $a = b[i]$ } Array indexed assignment statement
  $a[i] = b$ }
- $a = \&b$ Address assignment statement
- $a = *b$ } Pointer assignment statement
  $*a = b$ }
- param x_1
  param x_2
  ⋮
  param x_n
  } The arguments to the procedure and function calls are defined by the param instruction.
- call p, n [An invocation of a procedure p that takes n arguments]

y = call P, n [An invocation of procedure P that takes n arguments. The result of the call to procedure P is return and stored in y]

return [Passes the control to the instruction following the call instruction that invoked the procedure P]

return x [It passes the control to the instruction following the call instruction that invoked the procedure P, the value of x is returned]

-exit [It takes no arguments and exits the program]

Que Write the 3 address code for the following statement.

1.
```
if (a>b)
    x = x+1
else
    y = y-1
```

```
if (a>b) Goto L1
y = y-1
Goto L2
L1 : x = x+1
L2 : ___
```

← If cond. false then control go to L1 directly else part.

← If we don't write this then else part y=y-1 will execute then L1 also. As sequential it will execute.

2.
```
for (i=0; i<n; i++)
{
    a = a * 5
}
```

Sol
```
i = 0
L1 : if (i>=n) Goto L2
     a = a*5
     i = i+1
     Goto L1
L2 : ____
```

3. $x = a + b * c + d$ where x, a are real and c, d, b are integer

$t_1 = b * c$

$t_2 = t_1 + d$

$t_3 = \text{int to real}(t_2)$

$x = a + t_3$

[wrong]

$t_3 = a + \text{int to real}(t_2)$

opr opr

max 1 opr can be used

Tree

Parse Tree Syntax tree

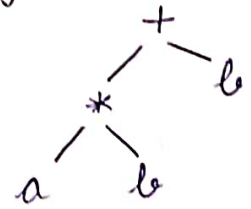
① Parse tree

$G = \{ S \to A + B$

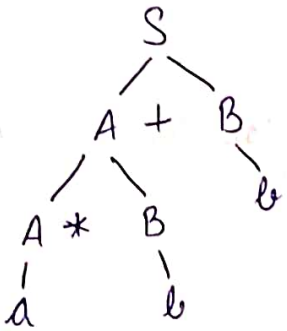
$A \to A * B / a$

$B \to b$

Syntax tree

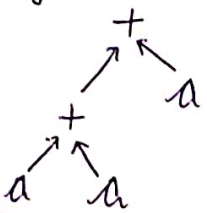


$a * b + b$

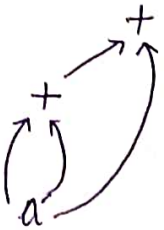


$a + a + a$

Syntax Tree



DAG [Directed Acyclic graph]



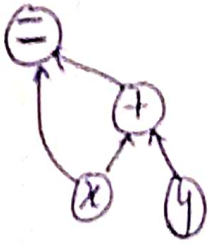
In DAG if a leaf node is one created instead of creating the same node again, we will make use of already created node

DAG is used for elimination of common subexpression.

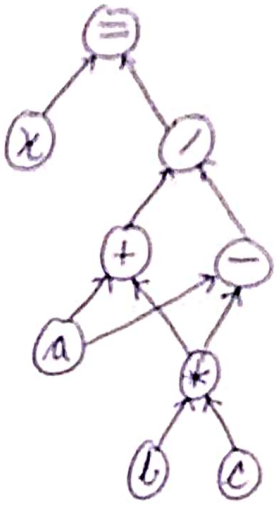
Ques Construct DAG for the following expression.

1. $x = x + y$
2. $x = (a + b * c) / (a - b * c)$
3. $a + (a * (b - c)) + ((b - c) * d)$

1.

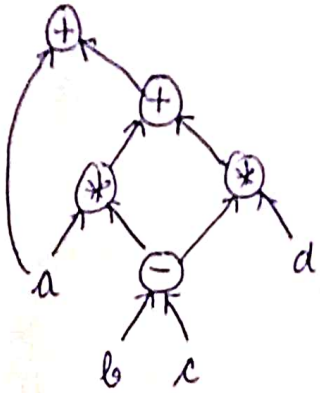


2.



In DAG we will not evaluate the same expr again Here using b+c in both only once.

3.



Q) The number of min nodes and edges in DAG for the following code segment.

$a = b + c$
$c = a + d$
$d = b + c$
$e = d - b$
$a = e + b$

① Common sub expression

Ex: $a = b * c$, $x = a + 5$, $y = b * c$, $z = b * c$

$a = b * c$
$x = a + 5$
$y = a$
$z = a$

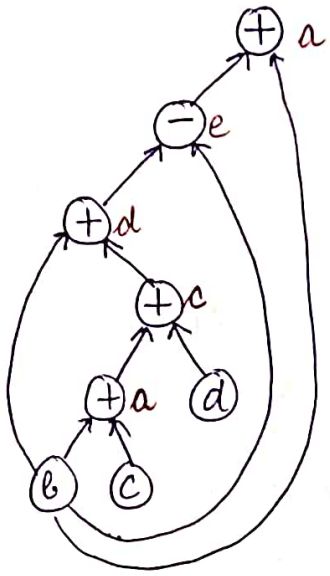
Ex

$a = b + c$, $x = a + y$, $y = b + c$, $b = a + x$, $z = b + c$

$a = b + c$
$x = a + y$
$y = a$
$b = a + x$
$z = b + c$

As ~~we are~~ here the value is changed $b = a + x$

Sol



It is DAG but not with minimum nodes & edges.

For minimum

Go bottom to top whatever come on LHS of arrow takes that. [Substitution]

$a = e + b$

$= d - b + b$

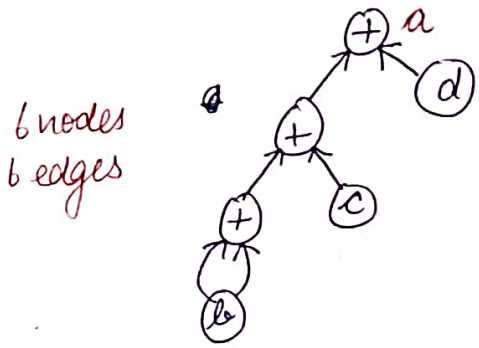
~~$b + c \quad c + b$~~

$= d$

$= b + c$

$= b + a + d$

$= b + b + c + d$



4 nodes
4 edges

2. $a = b + c$
$e = a + 1$
$d = b + c$
$f = d + 1$
$g = e + f$

Top to bottom

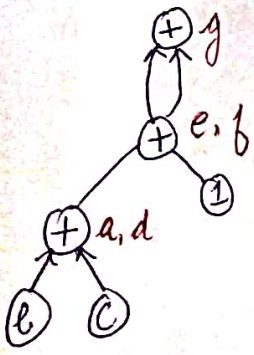
$a = b + c$
$e = a + 1$
$d = a$
$f = a + 1$
$g = e + f$

$\Rightarrow$

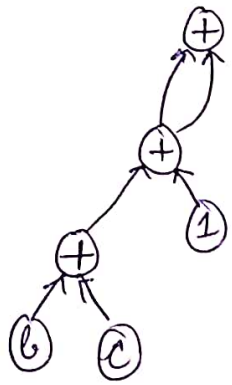
$a = b + c$
$e = a + 1$
$d = a$
$f = e$
$g = e + e$

Bottom to top

$g = e + f$
$= e + d + 1$
$e + b + c + 1$
$a + 1 + b + c + 1$
$b + c + 1 + b + c + 1$



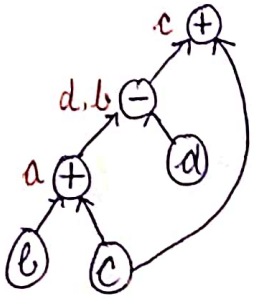
6 nodes, 6 edges



3. $a = b+c$
$b = a-d$
$c = b+c$
$d = a-d$

top to Bottom

$a = b+c$
$b = a-d$
$c = b+c$
$d = b$

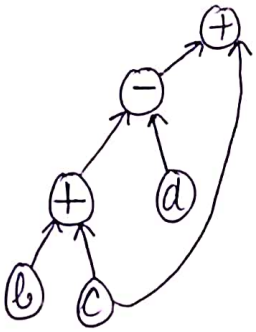


6 nodes, 6 edges.

Here if we go with $d = b-d$ we have to skip 2 expression.

Bottom to top

~~$d = a-d$~~
$c = \underline{b}+c$
$c = \underline{a}-d+c$
$= b+c-d+c$



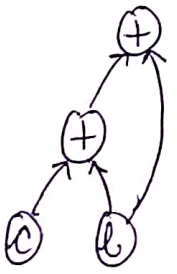
4. $a = c+b$
$b = b-d$
$c = b+d$
$d = a+c$

top to Bottom

No common subexpression.

Bottom to top

$d = a+\underline{c}$
$a + \underline{b} + d$
$\underline{a} + b - d + d$
$c + b + b$

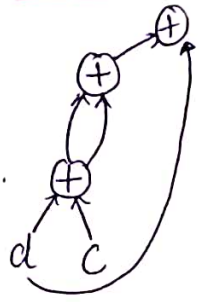


4 nodes, 4 edges

5. $a = b+c$
$b = a-b$
$c = d+b$
$b = c+d$
$d = b+c$

Bottom to top

$d = b+c$
$= \underline{c}+d+\underline{c}$
$= d+\underline{b}+d+d+\underline{b}$
$= d+\underline{a}-b+d+d+\underline{a}-b$
$= d+b+c-b+d+d+b+c-b$
$= \underline{d+c}+d+\underline{d+c}$



Control flow Graph (CFG)

Static Single Assignment [SSA] form

In SSA form every assignment must be made to a different variable.

In SSA form a variable must be defined before using.

Eg
$a = b+c$
$b = a-b$
$a = c+a$
$b = b-c$
$a = a+b$

$\Rightarrow$

$a_1 = b+c$
$b_1 = a_1-b$
$a_2 = c+a_1$
$b_2 = b_1-c$
$a_3 = a_2+b_2$

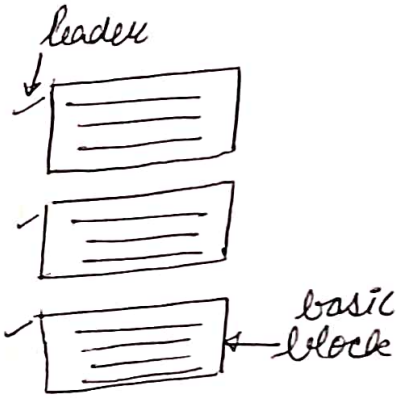
Control Flow graph (CFG)

⊙ Basic block

Basic block is a sequence of statements in which if the first statement is executed and all other statements in that basic block also will be executed ~~without~~ in the order of their appearance without any jump or halt in between

⊙ How to find the basic blocks

1. Find the leader statements.
2. Basic blocks will be formed with one leader statement and ends with the statement just before the next leader statement.



⊙ How to find the leader

1. The first statement is a leader.
2. The target of conditional or unconditional goto statement is a leader.
3. The statement that comes immediately after conditional or unconditional goto statement is a leader.

⊙ Control Flow graph

Control Flow graph is a directed graph between the basic block.

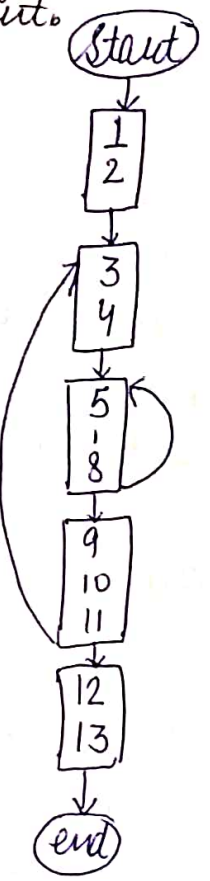
It gives the flow information between the basic blocks.

If block B_2 follows immediately after B_1 then there is a directed edge from B_1 to block B_2.

Ques Find the number of nodes and edges in the control flow graph for the following code segment.

1)

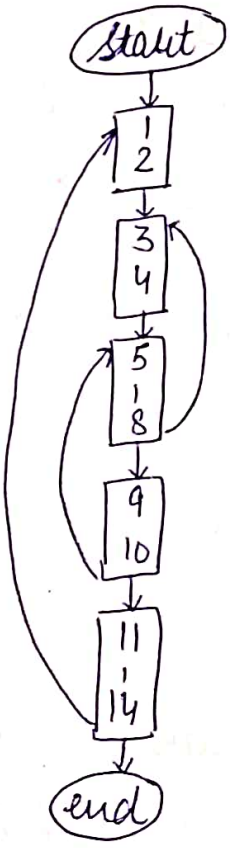
```
1  a = 1
2  b = 0
3  t1 = a + 5
4  t2 = x * a
5  t3 = 4 * b
6  a = a + t3
7  i = i + 1
8  if i<10 goto 5
9  j = j + 1
10 K = K + 1
11 if j<K goto 3
12 j = j + k
13 k = 0
```



7 nodes & 8 edges

2)

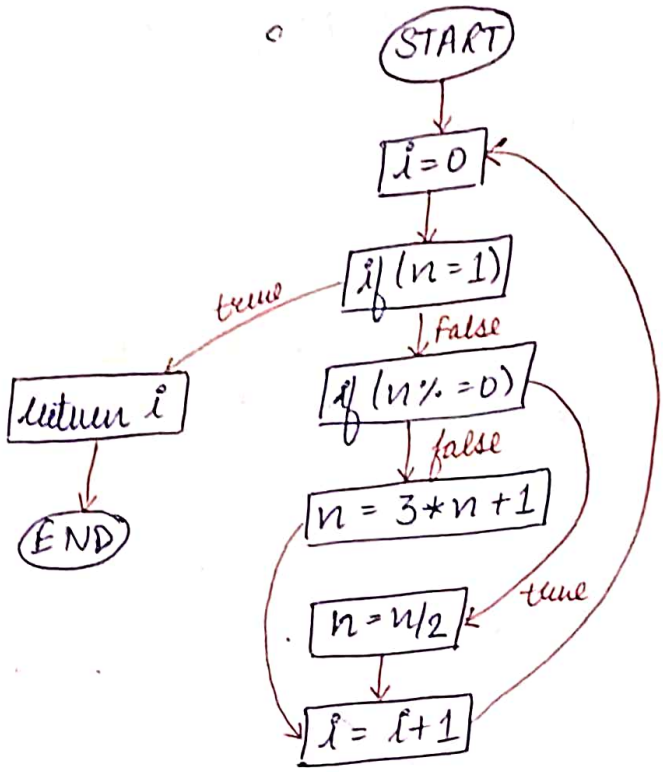
```
1  a = 1
2  b = 1
3  t1 = a + b
4  t2 = b * 10
5  t3 = t2 - t1
6  a = a + t1
7  b = a - b
8  if a > b goto 3
9  b = b + 5
10 if b < 10 goto 5
11 t4 = t2 - t3
12 a = t4 + t5
13 b = t1 + t2
14 if a < b goto 1
```



7 nodes
9 edges

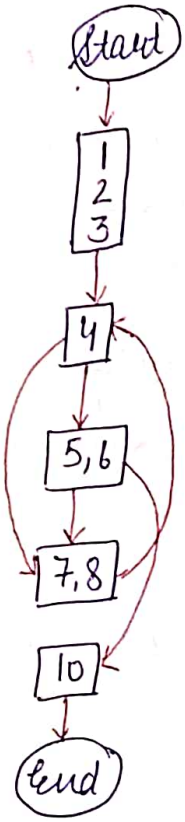
3)

```
    i = 0
L1: if (n = 1) goto L4
    if (n % = 0) goto L2
    n = 3 * n + 1
    Goto L3
L2: n = n/2
L3: i = i + 1
    Goto L1
L4: return i
```



4)

1. a = 10
2. b = 20
3. c = a - b
4. if c > d goto 7
5. a = a + b
6. if (a > d) goto 10
7. b = b ÷ 5
8. c = c + b
9. Goto 4
10. a = a + 4



we are not writing 9 statement as it is jump so just showing the arrow for Goto 4.

Implementation of 3-address code

1. Quadruple

operator operand1 operand2 Result

Eg $x = ((a*b) - (c\ d)) + ((c+d) * a)$

$t_1 = a*b$
$t_2 = c+d$
$t_3 = t_1 - t_2$
$t_4 = t_2 * a$
$x = t_3 + t_4$

SNO.	oper	op1	op2	Result
1.	*	a	b	t_1
2.	+	c	d	t_2
3.	−	t_1	t_2	t_3
4.	*	t_2	a	t_4
5.	+	t_3	t_4	x

Advantage :- we can access the values quickly using temporary variables from the symbol table
- Statements can be moved around

Disadvantage : using temporary variables more memory required.

2. Triples

operator operand 1 operand 2

S.No	oper	op1	op2
1.	*	a	b
2.	+	c	d
3.	−	(1)	(2)
4.	*	(2)	a
5.	+	(3)	(4)
6.	=	x	(5)

Advantage: Less space

Disadvantage: Statements can't be moved around.

3. Indirect Triple

SNO	
(i)	(1)
(ii)	(2)
(iii)	(3)
(iv)	(4)
(v)	(5)
(vi)	(6)

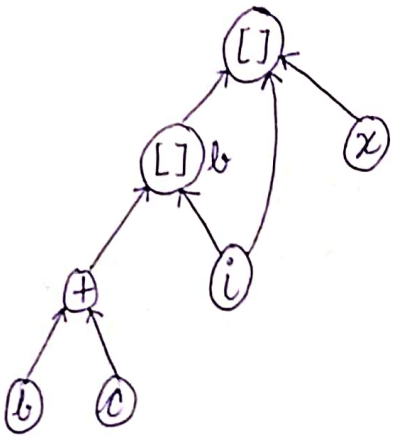
(1) = c+d
(2) = a*b
(5) = (a*b) - (c+d)

Advantage: Statements can be moved around.

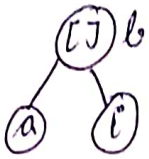
Disadvantage: Two times memory access.

Q-28.

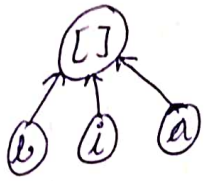
a = b+c
b = a[i]
b[i] = x



b = a[i]



b[i] = a



CODE OPTIMIZATION

Code Optimization means process of reducing the number of instructions if possible without affecting the outcome of the source code.

It means improvement of the program i.e. It increases efficiency of the program.

It takes less time and less space if the program is optimized and also it increases the speed of the execution of the program.

code optimization

Machine Dependent optimization

- It is based on the characteristics of the target machine.
- Can be achieved by allocating the sufficient resources to the program

Machine Independent Optimization

- It is based on characteristics of the source program
- Can be achieved by analyzing the program and finding alternative equivalent program

⊙ Where to apply code optimization

- Source code

  It involves optimization in changing the algorithm and loop structure.

  This can be done by programmer.

- Intermediate code
  It involves changing the address calculation and transforming the procedure calls involved.
  This will be done by the compiler

- Target code
  It involve in changing the, select, move, etc instructions
  This will be done by the compiler

⊙ Where to apply the optimization in source code.

- local optimization
  It will be apply within a basic block.

- Global optimization
  It can be applied among the basic blocks. i.e. within a control flow graph

- Inter Procedural Optimization
  It can be applied among the procedures i.e. within a program.

Machine Independent optimization technique

1. Common sub expression elimination
   It is a technique in which we avoid the recomputation of same expression.

$x = a+5$		$x = a+5$
$y = b-x$	$\rightarrow$	$y = b-x$
$z = a+5$		$z = a$
$c = b-x$		$c = y$
$d = a+b$		$d = a+b$

2. Constant Folding

It is a technique in which constant expressions are evaluated during compilation

$$x = 3*5 \quad \Rightarrow \quad x = 15$$
$$y = 4+9 \qquad\quad y = 13$$

3. Constant Propagation

It is a process of a constant variable will be replaced by its value.

$x = 3*5$
$y = x+6$ (CP: $y = 15+6 \Rightarrow y = 21$ CF)
$3 = y+10$ $\quad\Rightarrow\quad x = 15$, $y = 21$, $3 = 31$

4. Copy propagation

It is a technique in which using one variable instead of another variable

$x = b+5$
$y = x$
$3 = y+10$

$\Rightarrow$

$x = b+5$
$y = x$
$3 = x+10$

5. Dead code elimination

A code which never gets executed is called as dead code.

```
i = 1
if (i == 0)
{
   ≡
}
```

Dead code which never gets executed

$\Rightarrow$ $\quad i = 1$

6. Code Motion

It is a technique in which moving the statement from one place to another place.

7. Reduction in strength

It is a technique of reducing strength of certain operator by replacing with another operator.

$y = x * 2 \Rightarrow y = x + x$

8. loop Invariant

It is a technique in which avoiding the computation inside a loop.

```
for(i=0; i<n; i++)
{
    a = i + x/y;
}
```

⇒

```
b = x/y
for(i=0; i<n; i++)
{
    a = i + b;
}
```

```
for(i=0; i<n; i++)
{
    a = i + x/y;
    b = c + d;
}
```

⇒ (code motion)

```
b = c + d;   e = x/y;   // loop invariant
for(i=0; i<n; i++)
{
    a = i + e;
}
```

9. loop Fusion / Jamming

It is a technique of merging several loops into single loop

```
for(i=1; i<n; i++)
{
    x = x + i;
}

for(i=1; i<n; i++)
{
    y = i * 5;
}
```

⇒

```
for(i=1; i<n; i++)
{
    x = x + i;
    y = i * 5;
}
```

10. Loop Unrolling

It is a technique of writing the code 2 times thus number of jumps and tests will be reduced.

```
for(i=1; i<n; i++)
{
    a = a+b[i];
}
```

⇒

```
for(i=1; i<n; i++)
{
    a = a+b[i];
    i = i+1;
    a = a+b[i];
}
```

Liveness Analysis

A variable x is ~~let~~ said to be live at any statement if it is used either in that statement or anywhere in the subsequent statement in the program before it is defined.

i.e. a variable is said to be live if it is present anywhere on the RHS either in that statement or anywhere in the subsequent statements before it is present on the LHS

1)
```
a = b+c
c = a-b
d = b+a
a = a+c
d = b-c
```

Statement	a	b	c	d
1	X	L	L	X
2	L	L	X	X
3	L	L	L	X
4	L	L	L	X
5	X	L	L	X

2)
```
a = 1
b = a+5
c = a*b
d = c ÷ 10
e = a+b-c
a = b+c *d
return a*c
```

	a	b	c	d	e
1	X	X	X	X	X
2	✓	X	X	X	X
3	✓	✓	X	X	X
4	✓	✓	✓	X	X
5	✓	✓	✓	✓	X
6	X	✓	✓	✓	X
7	✓	X	✓	X	X

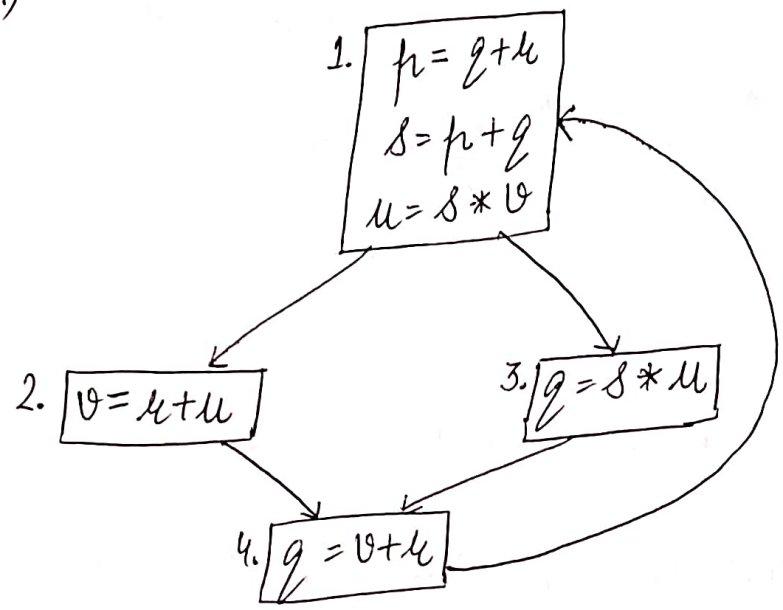
3)

```
a = 10
L: b = c + 5
   c = a + b
   a = a * b
   if a > 20 goto L
   return c
```

	a	b	c
1	×	×	✓
2	✓	×	✓
3	✓	✓	×
4	✓	✓	✓
5	✓	×	✓
6	×	×	✓

Note: When there is conflict for live and dead then give priority to live.

4)



IN = Set of variables live at the beginning of the block

OUT = Set of variables live at just after the block

USE/GEN = Variables that are used in block before any assignment. i.e. Variables that present on RHS in stmt but prior to that variables should not be present on LHS.

DEF/KILL = Variables that are assigned a value in block. i.e variables present on LHS

$IN = USE \cup (OUT - DEF)$

$OUT = \cup IN(S)$ S - Successor

Blocks/Node	USE	DEF	IN (first step)	OUT (first step)	IN (second step)	OUT (second step)
1.	{q, r, v}	{p, s, u}	q, r, v	r, u, s	q, r, v	s, u, v, r
2.	{r, u}	{v}	r, u	v, r	r, u	r, v
3.	{s, u}	{q}	s, u	v, r	s, u, v, r	r, v
4.	{v, r}	{q}	v, r	q, r, v	r, v	q, r, v

For explanation:

In first step IN

IN = USE ∪ (OUT − DEF)

out is not known

write only USE in IN

First step OUT

OUT = ∪ (IN(S))

Have the union of IN of the successor

Also for 1 = Union of 2,3 [IN]

2 = Union of 4 [IN]

3 = –

4 = Union of 1 [IN].

Second step IN

IN = USE ∪ (OUT − DEF)

Second step OUT = Have union of second step IN of the successor.

USE: Variable which are on RHS and union means before statement don't have that variable on LHS

DEF: Variables on LHS.

→ The variables that are live in block 2 = {x, u}

Block 3 = {s, u, v, x}

The variable that are live in both block 2 and block 3 = {x, u}

Short Cut

Node	p	q	r	s	u	v
1	×	✓	✓	×	×	✓
2	×	×	✓	×	✓	×
3	×	×	✓	✓	✓	✓
4	×	×	✓	×	×	✓

→ when live in block asked then take the corresponding row.

→ and when live out then take the union of successor.

Q) Find the minimum number of registers required to execute the program without memory spilling

$a = 1$	R_1
$b = 2$	R_2
$c = 3$	R_3
$d = a + b$	$R_1 = R_1 + R_2$
$e = c + d$	$R_1 = R_3 + R_1$
$f = c + e$	$R_2 = R_3 + R_1$
$b = c + e$	$R_3 = R_3 + R_1$
$e = b + f$	$R_3 = R_3 + R_2$
$d = 5 + e$	$R_2 = R_1 + R_3$
return $(d + f)$	return $(R_1 + R_2)$

when we are not using that variable again in RHS then we can reuse that register.

∴ 3 registers required.

Memory spilling : Moving a variable from a register to the memory is called memory spilling.

Thus that register can be allocated to another variable.

Machine Dependent Optimization

⊙ Peephole Optimization

This can be applied on small fraction of the code
It is performed on the very small set of instructions
in a segment of the code.

1. Redundant load and store optimization

Eg: Load R_0, X
Store X, R_0 ×

Here in between load and store there is no other instruction
we can delete the second instruction.

Eg: Load R_0, X
Load R_0, X ×

Here we can delete the second instruction

2. Use of machine idioms

MOV R_1, a
MOV R_2, 1
ADD R_1, R_2
MOV a, R_1

⇒ Incr (a)
INCR a

3. Flow Control Optimization

L_1: Goto L_2
|
L_2: Goto L_3
|
L_3: Goto L_4
|
L_4: —

⇒

L_1: Goto L_4

4. Constant Folding

5. Strength Reduction

Ques The following code segment has

1)
```
i = 0
b = 10
n = b*5      ⇒ n = 10*5
for(j=0; j<n; j++)
{
    a = 2*j  ⇒ a = j+j
    x = y+3  ← can write outside
}
```

a) Reduction in strength
b) loop invariant
c) constant folding
d) Dead code elimination

sol [a, b, c]

2)
```
i = 0
b = 5
n = b * 10   ⇒ n = 5*10
if(i==1)
{
    for(j=0; j<n; j++)
    {
        a = 2*j;
        x = y+3;
    }
}
```

Dead code.

sol [c), d)]