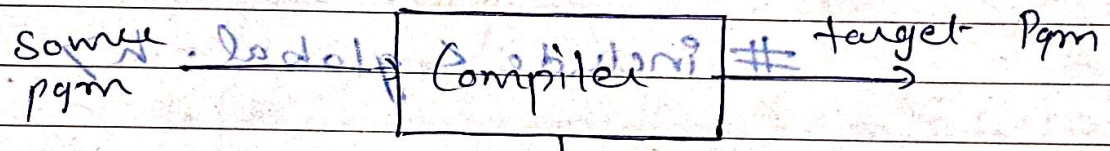
UNIT - 1

Introduction to Compilers

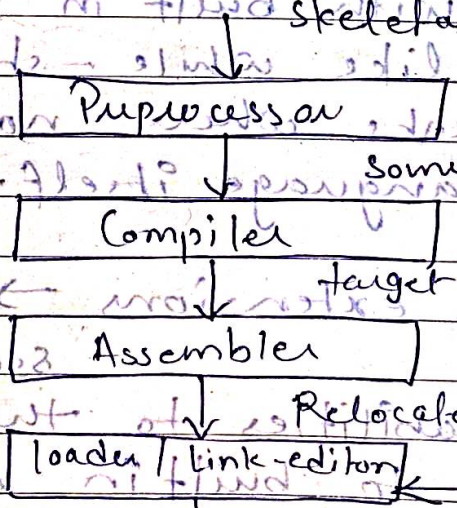
Compiler → A compiler is a pgm that reads a pgm written in one language - the source language & translates it into an equivalent pgm in another language - the target language.

An imp part of this translation process, the compiler reports to its user the presence of errors in the source pgm.



Error msgs.

Cousins of the compiler →



① Preprocessors ⟹

Preprocessor produce i/p to compilers. They may perform the following fun's —

i) Macroprocessing → A preprocessor may allow a user to define macros that are shorthands for longer constructs.

ii) File inclusion → A preprocessor may include header files into the pgm text.

```
# include <global.h>
```

iii) Rational preprocessors → These processors augment older langs with more modern flow of ctrl & data structuring facilities.

eg. such a preprocessor might provide the user with built in macros for constructs like while-statements or if statements where none exist in the pgm language itself.

iv) Language extensions → These processors attempt to add capabilities to the lang by what amts to built in macros.

eg. The lang. Equel is a database query lang. embedded in C. statements beginning with ## are taken by the preprocessor to be database access statements related to C, & are translated into procedure calls to subroutines that perform the database access.

Macro processors →
- → macro definition
- → macro use

TEX type setting system uses macro facility →

\define <macroname> <template> {<body>}

eg \define \JACM #1; #2; #3.
{ {\sl J.ACM}{\bf #1}: #2, PP. #3.}

call → \JACM 17; 4; 715-728.

output → J. ACM 17:4, pp. 715-728.

Explanation → A Macro name is any string of letters preceded by a backslash. The template is any string of characters, with strings of the form #1, #2, --- #9 regarded as formal parameters. These symbols may

also appear in the body, any number of times. For example the following macro defines a citation for the Journal of ACM.

```
\define\JACM #1;#2;#3.
  {{\sl J. ACM}{\bf #1}:#2,
   pp. #3.}
```

The macro name is JACM and the template is "#1;#2;#3." ; semicolons separate the parameters, and the last parameter is followed by a period. A use of this macro must take the form of that template, except that arbitrary strings may be substituted for the formal parameters.

Thus we may write –

```
\JACM{17;4;715-728.}
```

& we expect to see

J. ACM: **17**:4, pp. 715–728.

(italic: J. ACM; bold: 17)

The portion of the body {\sl J. ACM} calls for an italicized ("slanted") "J. ACM". Expression {\bf #1} says that the first actual parameter is to be made boldface; this parameter is intended to be the volume number.

(2) Assemblers → Some compilers produce assembly code, that is passed to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.

Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.

Ex

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

b = a+2

This code moves the contents of the address a into register 1, then adds the constant 2 to it, treating the contents of register 1 as a fixed-point no. & finally stores the result in the location named by b. Thus it computes b := a+2.

Two-Pass Assembly → The simplest form of assembler makes two passes over the input, where a pass consist of reading an input file once.

In the first pass, all the identifiers that denote storage locations are found

and stored in a symbol table. Identifiers are assigned storage locations as they are encountered for the first time.

Ex

identifier Address

a 0

b 4

(we have assumed that a word consisting of four bytes is set aside for each identifier & that addresses are assigned starting from byte 0)

In Second pass, the assembler scans the input again. This time it translates each operation code into sequence of bits representing that operat^n in machine language, and it translates each identifier representing a location into the address given for that identifier in a symbol table.

The output of the second pass is usually relocatable machine code, meaning that it can be loaded starting at any location L in memory, i.e. if L is added to all addresses in the code, then all references will be correct. Thus the output of the assembler must distinguish those portions of instructions that refer to addresses that can be relocated.

Steps

Ex ① MOV a, R1
Assembly instructions: ADD #2, R1
MOV R1, b

② identifier Address
Symbol table: a 0
b 4

③ Reg. Ordinary mode Immediate mode Memory Add. Relocatable bit

m/c code:
load ← 0001 01 00 0000 0000 *
Add ← 0011 01 10 0000 0010
store ← 0010 01 00 0000 0100 *

④ Absolute m/c code:
0001 01 00 0000 1111
0011 01 10 0000 0010
0010 01 00 0000 0011

In step ③ The first four bits are the instruction code with 0001, 0010 and 0011 standing for load store and add respectively. By load & store we mean moves from memory into a register & vice versa. The next two bits designate a register & refers to register in each of the three above instructions. The two bits after that represent a "tag" with 00 standing for

the "ordinary address mode" where the last eight bits refer to a memory address. The tag 10 stands for the "Immediate mode" where last eight bits are taken literally as the operand. * associated with the first & third instructions. This * represents the relocation bit that is associated with each operand in relocatable machine code.

Suppose that the address space containing the data is to be loaded starting at location L. The presence of the * means that L must be added to the address of the instructions. Thus if L = 00001111 then 00001 b would be at locations 15 and 19 respectively and the instructions would be

0001 01 00 00001111
0011 10 10000001 0
0010 01 10 00 00010011

in the absolute or unrelocatable machine code. Note that there is no * associated with the second instruction in (3) so L has not been added to its address in (4)

Loaders and Link-Editors ⟹

Usually, a program called a Loader performs the two functions of loading and link-editing. The process of loading consists of taking relocatable machine code, altering the relocatable addresses as discussed in previous example and placing the altered instructions and data in memory at the proper locations.

The link-editors allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations and one or more may be library files of routines provided by the system and available to any program that needs them.

(3)

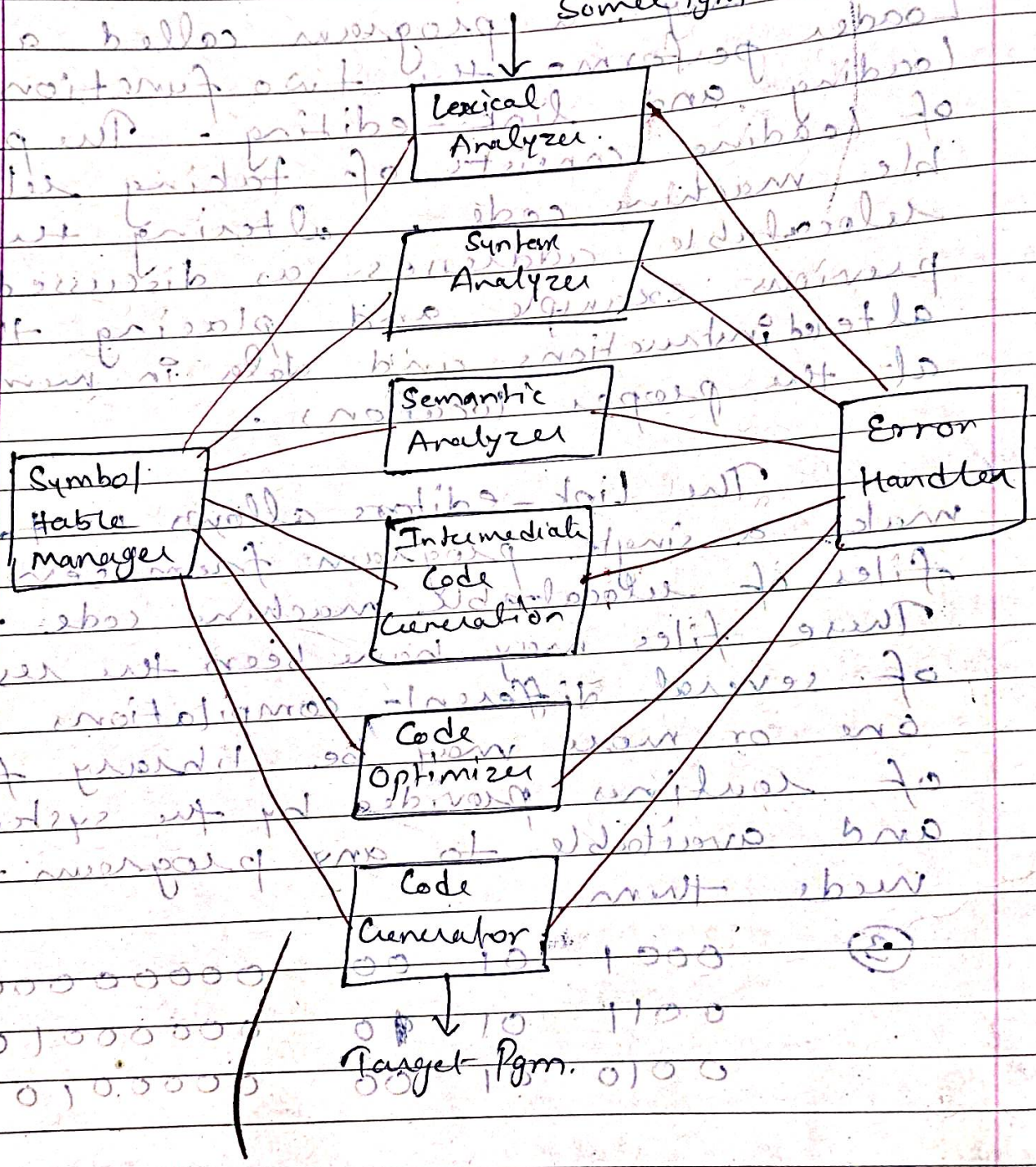
0001	01	00	00000000 *
0011	01	10	00000010
0010	01	00	00000100 *

(4)

0001	01	00	00001111
0011	01	10	00000010
0010	01	00	00010011

where L = 00001111

Phases of Compilers



Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

(1) Lexical Analysis → The lexical Analysis phase leads the characters in the source program and groups them into streams of tokens, each token represents a logically cohesive sequence of characters such as identifiers, operators and keywords. It is also called as scanning.

Ex position := initial + rate * 60

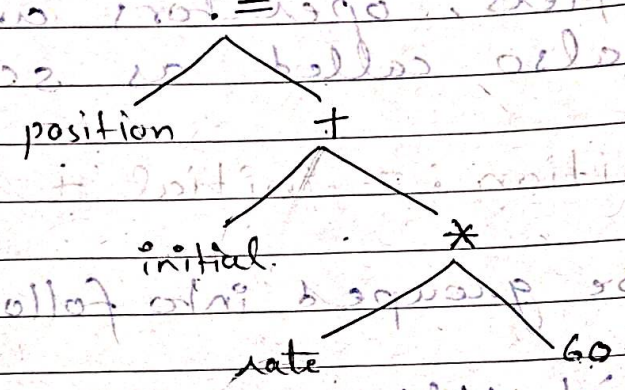
would be groupes into following token -

1. The identifier position
2. The assignment symbol :=
3. The identifier initial
4. The plus sign +
5. The identifier rate
6. The multiplication sign *
7. The number 60

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

(2) Syntax Analysis → Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source

program into grammatical phrases that are used by the compiler to synthesize output. Usually the grammatical phrases of the source pgm are represented by a parse tree such as the one shown below —

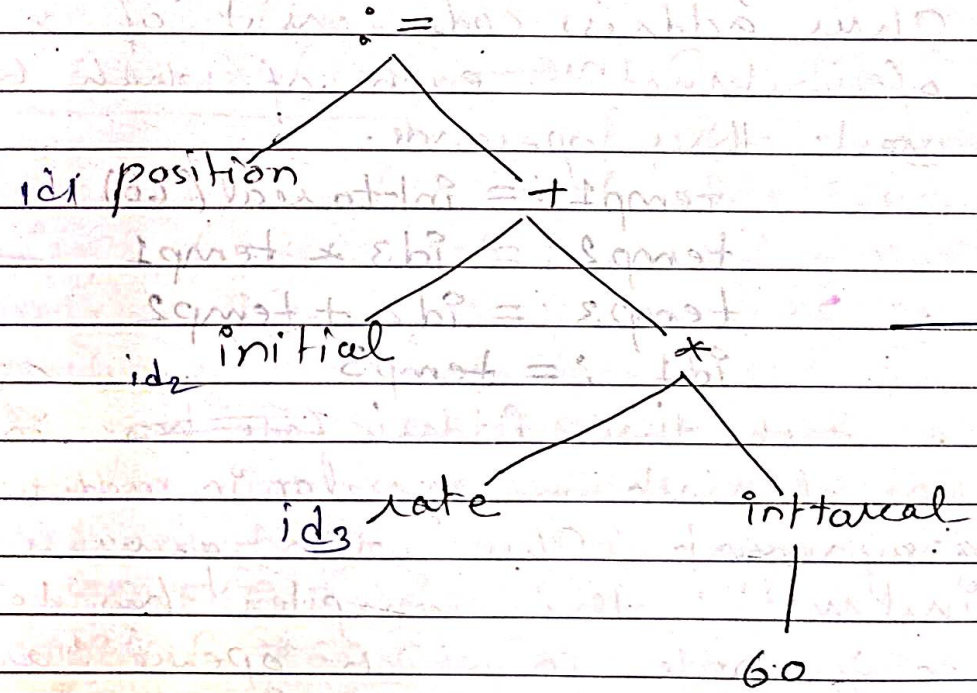


(3) Semantic Analysis → The semantic analysis phase checks the source pgm for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determine by the syntax analysis phase to identify the operators and operands of expressions and statements.

An important component of semantic analysis is type checking. Here the compiler checks that each operators has operands that are permitted by the source language specification.

Ex Inside a machine, the bit pattern representing an integer is generally different from the bit pattern for a real, even if the integer and the real number happen to have same value. For example, that all identifiers in fig ① have been declared to be reals & that 60 by itself is assumed to be an integer.

Type checking of fig ① reveals that * is applied to a real rate and an integer 60. The general approach is to convert the integer into real. This has been achieved in fig ② by creating an extra node for the operator "inttoreal" that explicitly converts an integer into a real.



②

(4) Intermediate Code Generation

After syntax & semantic analysis, some compilers generate an explicit intermediate representation of the source pgm. ~~We can think of this~~

This intermediate representation should have two important properties - it should be easy to produce & easy to translate into the target pgm. The intermediate representation can have a variety of forms. We consider an intermediate form called "Three address code" which is like the assembly language for a machine in which every memory location can act like a register. Three address code consist of a sequence of instructn each of which has almost three operands.

```
temp1 := int-to-real(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

Each three Address ~~Code has~~ instructn has at most one operator in addition to the assignment. Thus when generating these instructns the compiler has to decide on the order in which operatns are to be done.

(5) <u>Code Optimization</u> → The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```
— (4)

Compiler can deduce the conversion of 60 from integer to real representation can be done once & for all at compile time, so the "inttoreal" operation can be eliminated. Besides, temp3 is used only once, to transmit its value to id1. It then become safe to substitute id1 for temp3, whereupon the last statement of (3) is not needed & the code of (4) results.

(6) <u>Code Generation</u> → The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the pgm. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```
— (5)

Where R_1 & R_2 are register 1 and register 2 respectively. The first & second operands of each instruction specify a source & destination respectively. The F in each instruction tells us that the instructions deal with floating-point numbers.

This code moves the contents of the address id3 into register 2 then multiplies it with the real-constant 60.0. The # signifies that 60.0 is to be treated as a constant. The third instruction moves id2 into register 1 & adds to it the value previously computed in register 2. Finally, the value in register 1 is moved into the address of id1.

(7) <u>Symbol-Table</u> → A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly & to store or retrieve data from that record quickly.

An essential function of a compiler is to record the identifiers used in the source pgm & collect info about various attributes of each identifier. These attributes may provide info about the storage allocated for an identifier, its type, its scope & in the case of

position := initial + rate * 60

↓

Lexical Analyzer

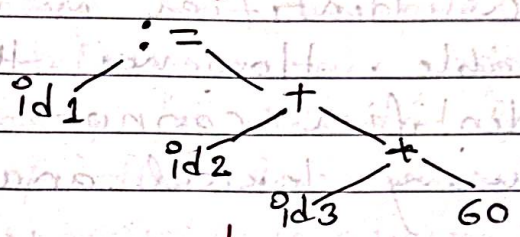
↓

id1 := id2 + id3 * 60

↓

Syntax Analyzer

↓



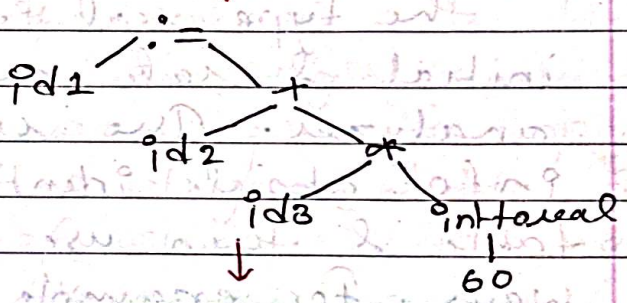
↓

Semantic Analyzer

↓

Symbol Table.

1	position	...
2	initial	...
3	rate	...
4		



↓

Intermediate Code Generator

↓

temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

↓

Code Optimizer

↓

temp1 := id3 * 60.0
id1 := id2 + temp1

↓

Code Generator

↓

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

procedure names, such things as the no. & types of its arguments, the method of passing each argument & the type returned if any.

When an identifier in the source pgm is detected by the lexical analyzer, the identifier is entered into the symbol table. However the attributes of an identifier cannot normally be determined during lexical analysis.

var position, initial, rate : real ;

the type real is not known when position, initial & rate are seen by the lexical analyzer. The remaining phases enter info. about identifiers into the symbol table & then use this informatn in various ways. For example when doing semantic analysis & intermediate code generation, we need to know what the types of identifiers are, so we can check that the source pgm uses them in valid ways & so that we can generate the proper operations on them. The code generator typically enters & uses detailed info about the storage assigned to identifiers.

(8) Error Detection & Reporting → Each phase can encounter errors. However, after detecting an

error, a phase must somehow deal with that error. ~~However, after detecting~~ so that compilation can proceed, allowing further errors in the source pgm to be detected. The syntax & semantic analysis phases usually handle a large fraction of the errors detected by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. eg. if we try to add two identifiers, one of which is the name of array & the other the name of a procedure.

The Grouping of Phases →

The phases are collected into a front end & back end.

The front end consists of those phases or parts of phases that depend primarily on the source lang. & are largely independent of the target machine. This normally include lexical & syntactic analysis, the creation of the symbol

table, semantic analysis & the generatn of intermediate code. A certain amt of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.

The back end includes those portions of the compiler that depend on the target machine & generally, these portions do not depend on the source language, just the intermediate language. In the back end we find aspects of the code optimization phase & we find code generation along with the necessary error handling & symbol table operations.

It has become fairly routine to take the front end of a compiler & redo its associated back end to produce a compiler for the same source lang. on a diff. machine. If the back end is designed carefully it may not even be necessary to redesign too much of the back end.

Aho Ullman

Passes → In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source pgm or the output of the previous pass, makes the transformations specified by its phases, & writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved with control alternating among several phases.

Aho & Ullman

Bootstrapping & Cross compiler →

A compiler may run on one machine & produce object code for another machine. Such a compiler is often called a Cross-compiler. Many minicomputer & microprocessor compiler are implemented this way. They run on a bigger machine & produce object code for the smaller machine.

A compiler is characterized by three languages: its source lang., its object lang., & the lang. in which it is written. These lang. may all quite different.

C_S^{LA} — L: lang. for which comp. written; A: (object code for which comp produces lang); S: implementation lang. (in which comp written)

Suppose we have a new lang. L which we want to make available on several machine say A & B. As a first step we might write for machine A a small compiler C_A^{SA}, that translates a subset of S of lang. L into machine or assembly code of A. This compiler can be first written in a language that is already available on A.

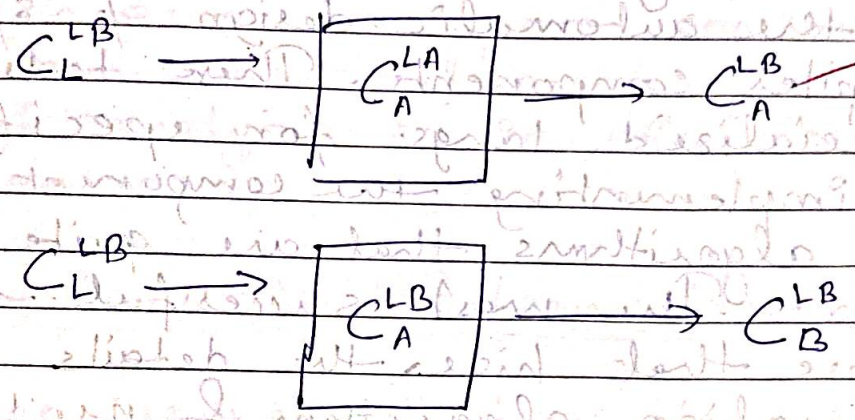
We then write a compiler C_S^{LA} in the simple lang. S. This pgm, when run through C_A^{SA}, becomes C_A^{LA} the compiler for the complete lang. L running on machine A & producing object code for A.

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

Bootstrapping a compiler

Now suppose we want to produce another compiler for L to run on machine B & to produce code for B. If C_S^{LA} has been designed carefully & machine B is not that different from machine A it should be far less work to convert C_S^{LA} into a compiler C_L^{LB} which produces object code for B than it is to write

a new compiler from scratch. Note that we can now use the full language L to implement C_L^{LB}.



Bootstrapping a compiler to a second machine.

Using C_L^{LB} to produce C_B^{LB}, a compiler for L on B, is now a two-step process, as shown in fig. We first run C_L^{LB} through C_A^{LA} to produce C_A^{LB}, a cross-compiler for L which runs on machine A but produces code for machine B. Then we run C_L^{LB} through this cross-compiler to produce the desired compiler for L that runs on machine B & produces object code for B.

If a compiler has been implemented in its own lang., then this arrangement is called a bootstrap arrangement. ~~The~~

Compiler-Construction Tools ⇒

Phasewise tools
① Scanner generator
② Parser generator
③ Syntax-directed translation
④ Data flow Engines
⑤ Automatic code generators.

Some general tools have been created for the automatic design of specific compiler components. These tools use specialized langs. for specifying & implementing the component & many use algorithms that are quite sophisticated. The most successful tools are those that hide the details of the generation algorithm & produce components that can be easily integrated into the remainder of a compiler. The following is a list of some useful compiler-construction tools :-

① **Parser-generators** → These produce syntax analyzers, normally from input that is based on a context-free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler. This phase is now considered one of the easiest to implement. Many parser generators utilize powerful parsing algorithm that are too complex to be carried out by hand.

(2) Scanner generators → These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.

(3) Syntax-directed translation engines →

These produce collections of routines that walk the parse tree, generating intermediate code. The basic idea is that one or more translations are associated with each node of the parse tree & each translatⁿ is defined in terms of translations at its neighbor nodes in the tree.

(4) Automatic code generators → Such a tool takes a collection of rules that define the translation of each operation of the intermediate lang. into the machine lang. for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data.

(5) Data flow Engines → Much of the info. needed to perform good code optimizatⁿ involves "data flow analysis" the gathering of info. about how values are transmitted from one part of a pgm to each other part.

Lexical Analyzer Design

Since the fun° of the lexical analyzer is to scan the source pgm & produce a stream of tokens as output, the issues involved in the design of lexical analyzer are —

1. Identifying the tokens of the lang for which the lexical analyzer is to be built, & to specify these tokens by using suitable notation.

2. Constructing a suitable recognizer for these tokens.

Therefore, the first thing that is required is to identify what the keywords are, what the operators are, & what the delimiters are. These are the tokens of the lang.

After identifying the tokens of the lang, we must use suitable notation to specify these tokens. This notation should be compact, precise & easy to understand. Regular expressions can be used to specify a set of strings, & a set of strings that can be specified by using regular-expression notation is called a regular set.

Tokens of typical programming langs are as follows —

operators = + | - | * | / | mod | div
keywords = if | while | do | then
letter = a | b | c | d | ... | z | A | B | C | ... | Z
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
identifier = letter (letter | digit)*

LEX

It is possible to automate the procedure of obtaining the lexical analyzer from the regular expressions specifying the tokens — and this is what precisely the tool LEX is used to do. LEX is a compiler-writing tool that facilitates writing the lexical analyzer, & hence a compiler.

Structure of LEX →

(1) Definition Section

%{

%}

definition section introduces any initial C pgm code we want copied into the final pgm. Like we have header

We can define tokens as —
%token DIGIT 253

files that must be included for code later in the file to work. Lex copies the material betn "%{" and "%}" directly to the generated c file, so you may write any valid code here. It ends with %% mark.

② Rule Section → It is having two main parts. pattern part & the action part which is separated by the white spaces. Patterns are in UNIX-style regular expressions.

③ User subroutines Section →

It can consists of any legal c code. Lex copies it to the c file after the end of lex generated code.

A simple lex Pgm →

Ex shows a simple lex specification to recognize these verbs.

Soln

```
%{
    /* this sample demonstrates simple
       recognition of a verb/not a verb.
    */

%{
%%

[ /t]+          /* ignore white spaces */;

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go  { printf("%s: is a verb\n", yytext); }
[a-zA-Z]+  { printf("%s: is not a verb \n", yytext); }
. | \n      { ECHO; /* normal default anyway */ }
%%
main()
{ yylex();
}
```

Output did I have fun?

did : is a verb

I : is not a verb.

have : is a verb

fun : is not a verb.

?

Explaining Pgm →

[\t]+

[ ] indicates that any one of the characters within the brackets matches the pattern.

"\t" a tab character

" " a space

+ means that the pattern matches one or more consecutive copies of the subpattern that precedes the plus.

[\t]+ white spaces (any combination of tabs & spaces).

; action is simply a semicolon a do nothing C statements. (it means ignore the input)

| (vertical bar) action that means to use the same action as the next pattern, so all of the verbs use the action specifies for the last one.

The file contains the definition of a function called yylex() that returns '1' whenever an expression you specified is found in input text, '0' when end of file is encountered.

[a-zA-Z]+ {printf("%s: is not a verb\n", yytext);}

The pattern "[a-zA-Z]+" is indicates any alphabetic string with at least one character. The "-" character has a special meaning, it denotes a range of characters beginning with the character left of "-" & ending with the character to its right.

The two that make our lexer work are —

① Lex patterns only match a given i/p char. or string once.

② Lex executes the action for the ~~toge~~ longest possible match for the current input.

eg. "island" longer than "is"

"." (period) matches any single character other than a newline.

"\n" matches a newline character.

ECHO Special action ECHO prints the matched pattern on the o/p (copying any punctuation or other characters.)

We explicitly list this case although it is the default behaviour.

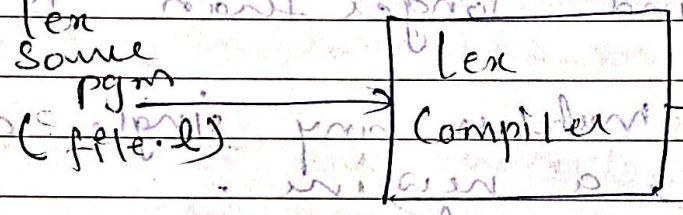
%% end rule section:

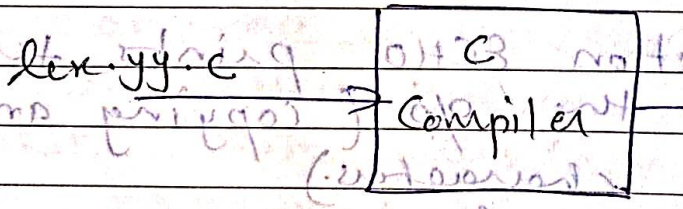
```
main()
{
    yylex();
}
```

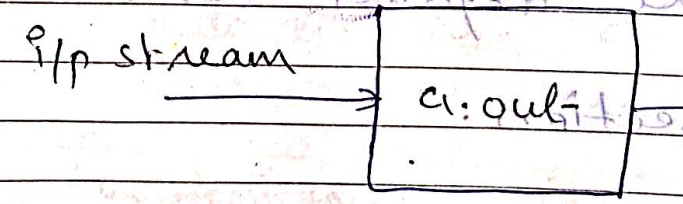
lexer produced by lex is a C routine called yylex(); so we call it.
Unless the actions contain explicit return stmts, yylex() won't return until it has processed the entire i/p.

```
$ lex prch1-02.l
$ cc lex.yy.c -ll
```

lex.yy.c → lex translates the lex specification into the C source file called lex.yy.c which we compile.



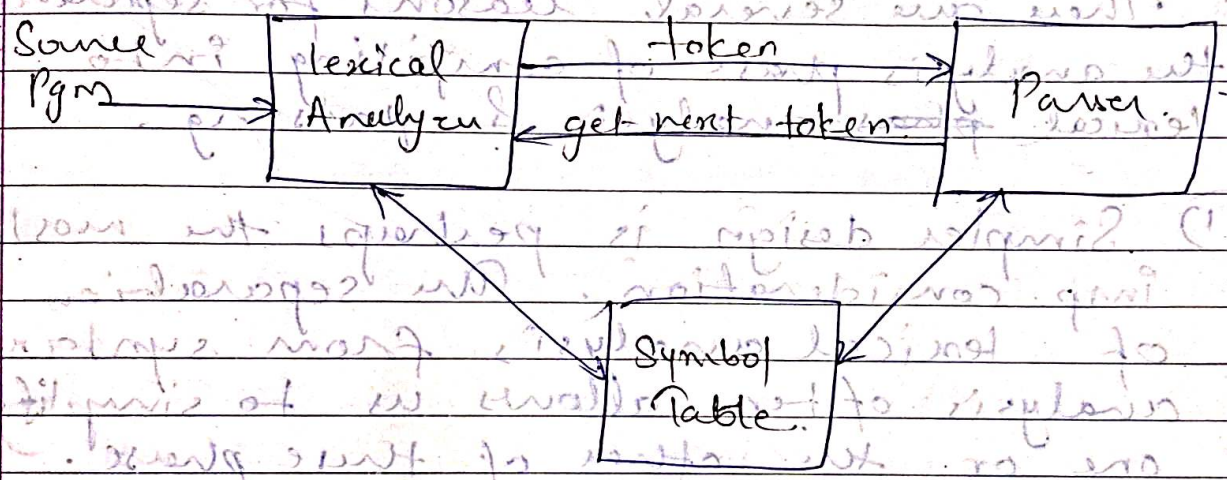




creating a lexical Analyzer with lex

The Role of lexical Analyzer →

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters & produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction, summarized schematically in fig. is commonly implemented by making the lexical analyzer be a subroutine or a coroutine of the parser.



Interaction of lexical Analyzer with Parser.

Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Since the lexical analyzer is the part of the compiler that reads the source text it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source

pgm comments & white spaces in the form of ~~blank~~ blank, tab & newline chars. Another is correlating error msgs from the compiler with the source pgm. for ~~example~~ eg. the lexical analyzer may keep track of the no. of newline characters seen, so that a line no. can be associated with an error msg.

Issues in Lexical Analysis →

There are several reasons for separating the analysis phase of compiling into lexical ~~phas~~ analysis & parsing.

(1) Simpler design is perhaps the most imp consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phase.

(2) Compiler efficiency is improved. A separate lexical analyzer allows us to construct a specialized & potentially more efficient processor for the task. A large amt of ~~the time~~ is spent reading the source pgm & partitioning it into tokens. Specialized buffering techniques for reading input chars & processing tokens can significantly speed up the performance of a compiler.

(3) Compiler portability is enhanced. Input alphabet peculiarities & other device-specific anomalies can be restricted to the lexical analyzer.

Specialized tools have been designed to help automate the construction of lexical analyzer & parsers when they are separated.

UNIT - II

Syntax Analysis

Content free Grammar → CFG

notation specifies a content free lang. that consists of terminals, nonterminals, a start symbol & productions. The terminals are nothing more than tokens of the lang. used to form the lang. constructs. Nonterminals are the variables that denote a set of strings. For eg. S & E are nonterminals that denote statements strings & expression strings, respectively. Therefore content-free grammer is a fom-tuple denoted as -

$$G = (V, T, P, S)$$

where :

V is a finite set of symbols called as nonterminals or Variables.

T is a set a symbols that are called as terminals.

P is a set of productions.

S is a member of V called as start symbol.

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

$$P = \{ S \rightarrow aSa, \quad S \rightarrow bSb, \quad S \rightarrow \epsilon \}$$

Standard Notation:

1. The capital letters toward the start of the alphabet are used to denote nonterminals (eg. A, B, C, etc).

2. Lowercase letters toward the start of the alphabet are used to denote terminals (eg. a, b, c, etc).

3. S is used to denote the start symbol.

4. Lower letters toward the end of the alphabet (eg. u, v, w etc) are used to denote strings of terminals.

Parse Tree → When deriving a string w from S, if every derivation is considered to be a step in the tree construction, then we get the graphical display of the derivation of string w as a tree. This is called a "derivation tree" or a "parse tree" of string w. Therefore, a derivation tree or parse tree is the display of the derivations as a tree.

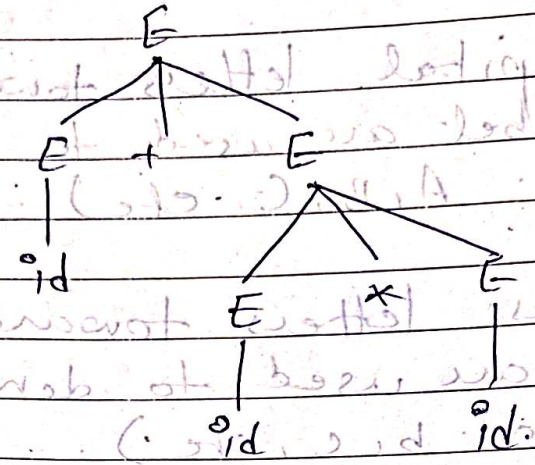
Ex

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$

Draw the parse tree.

string $w = id + id * id$.

Sol^n Parse tree –



id + id * id.

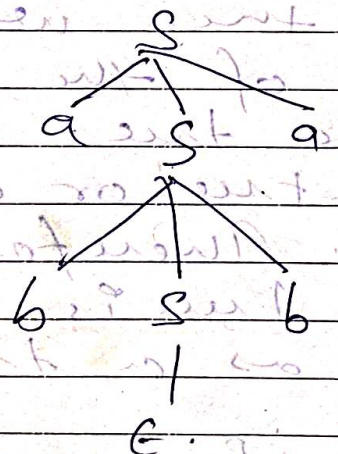
Ex

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow \epsilon$

Draw the parse tree.

string w = abba.

Sol^n Parse Tree –



abba.

Derivation →

1. Left most Derivation (LMD)
2. Right most Derivation (RMD).

1. Left most Derivation (LMD) → The Left most Derivation is that order of derivation in which a left most non terminal is considered first for derivation at every stage in the derivation process.

Ex. $E \to E+E \mid E*E \mid id$ find the LMD & RMD.

string = id + id * id

Soln

LMD

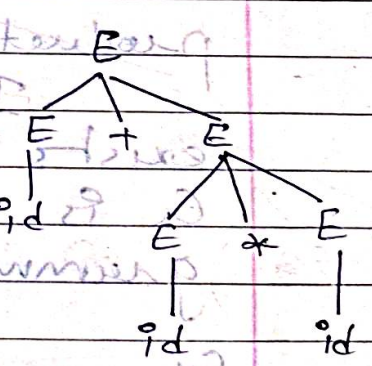
$E \to E+E$

$\to id + E$

$\to id + E*E$

$\to id + id * E$

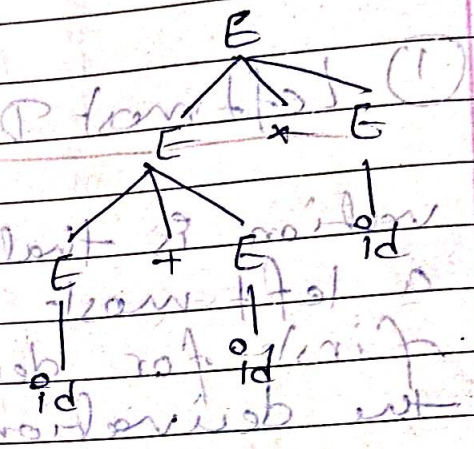
$\to id + id * id$



(2) Right Most Derivation (RMD) → In a right-most derivation, the right most nonterminal is considered first.

RMD

$E \rightarrow E * E$
$\rightarrow E * id$
$\rightarrow E + E * id$
$\rightarrow E + id * id$
$\rightarrow id + id * id$



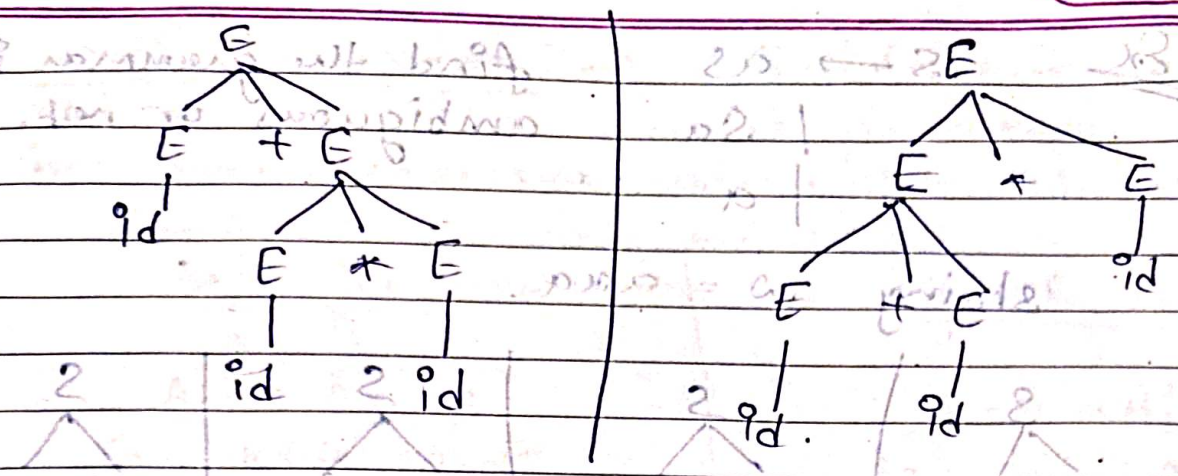
Ambiguous Grammar → There may exist more than one parse tree. This means that more than one way may exist to derive w from S, using the productions of the grammar.

If more than one parse tree exists for some w in L(G), then G is said to be an ambiguous grammar.

Ex. $E \rightarrow E + E * E$
$| \; E * E$
$| \; id.$

find the grammar is ambiguous or not.

string. w = id + id * id.

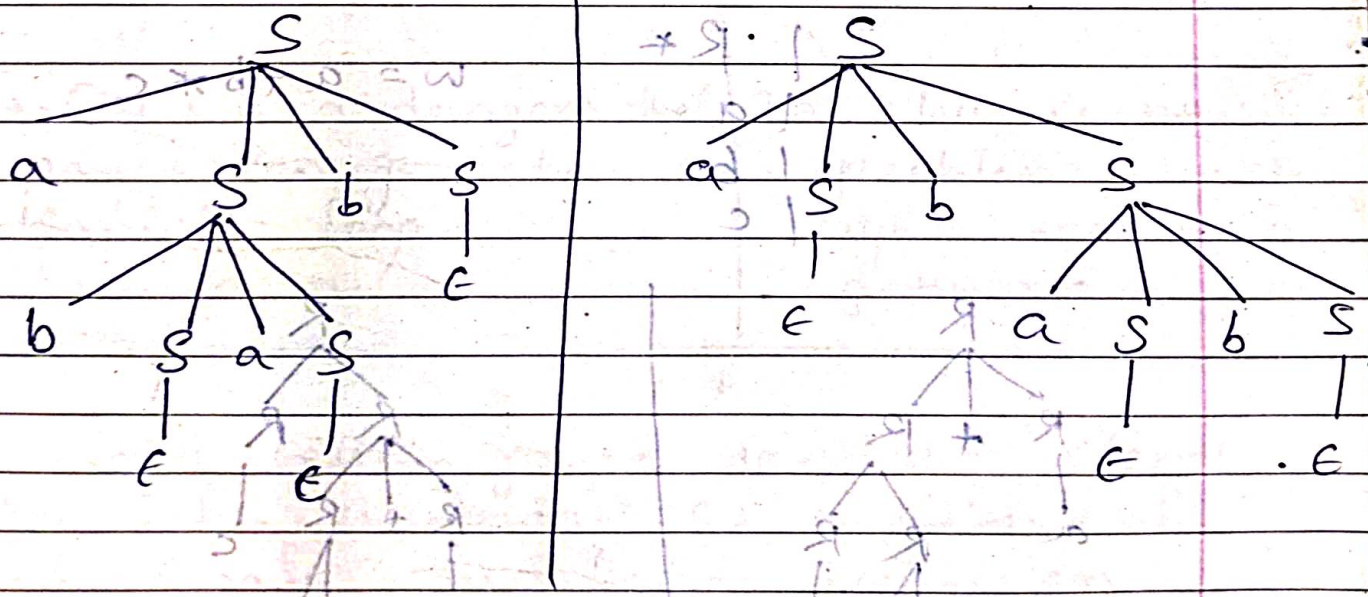


So the given grammar is ambiguous grammar b'coz there exists more than one parse tree for the string id+id*id.

Ex

$S \rightarrow aSbS \mid bSaS \mid \epsilon$ find the grammar is ambiguous or not.

string w = abab.



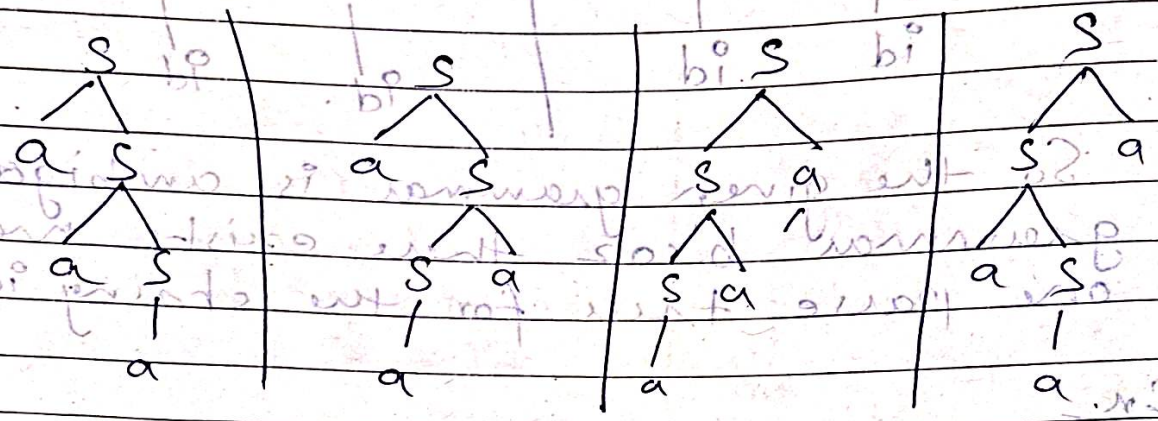
This grammar is also an ambiguous grammar b'coz more than one parse tree exists for a string abab.

Ex

$S \rightarrow aS$
$\quad | \; Sa$
$\quad | \; a$

find the grammar is ambiguous or not.

string $w = aaa$.



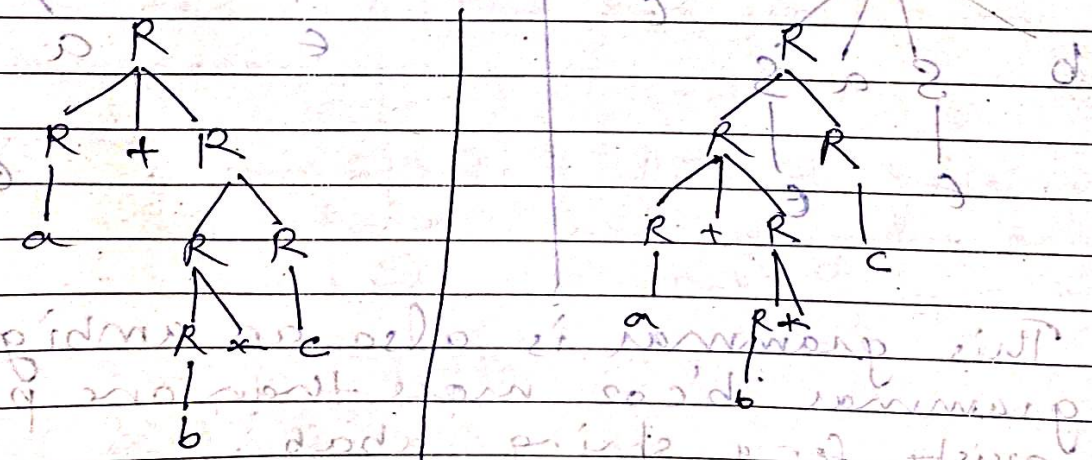
This grammar is also ambiguous

Ex

$R \rightarrow R + R$
$\quad | \; RR$
$\quad | \; R*$
$\quad | \; a$
$\quad | \; b$
$\quad | \; c$

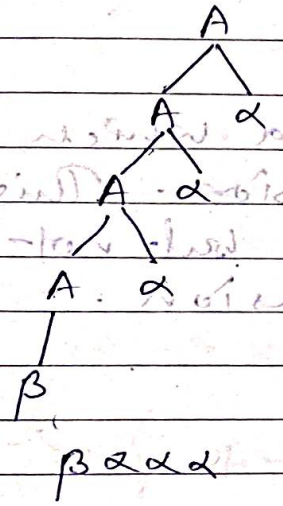
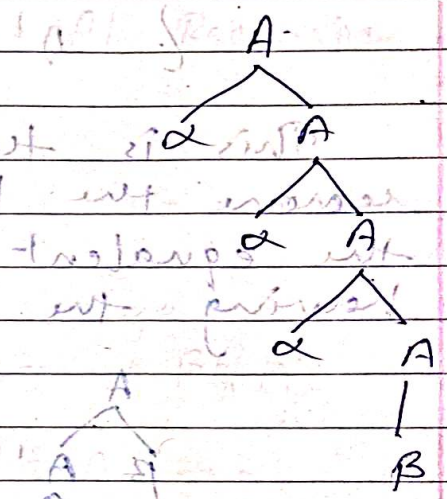
find the grammar is ambiguous or not.

$w = a+b*c$



This grammar is also ambiguous.

Left Recursion → Based on the product rules grammar can be left recursive or right recursive.

Left Recursion	Right Recursion
$A \rightarrow A\alpha / \beta$	$A \rightarrow \alpha A / \beta$
So, $L(G) = \beta\alpha^*$	So, $L(G) = \alpha^*\beta$
$w = \beta\alpha\alpha$	$w = \alpha\alpha\alpha\beta$
	
↳ There is a danger that parser can go into infinite loop.	↳ There is no such problem in the right recursive grammar.

Some certain parser does not accept the left recursion so we need to eliminate the left recursion.

Elimination of Left Recursion ⇒

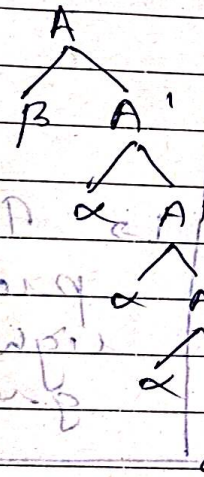
Left recursion can be eliminated from the grammar by replacing $A \rightarrow A\alpha \mid \beta$ with the productions $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$.

Rule

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

$$\begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{cases}$$

This is the formula which will remove the left recursion. This is the equalent grammar but not having the left recursion.



Ex $S \rightarrow a \mid (L)$ Eliminate
$L \rightarrow \underbrace{L,}_{A\ \alpha}\ \underbrace{S}_{\beta} \mid S$ Left Recursion.

Solⁿ

$S \rightarrow a \mid (L)$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \epsilon$

Ex

$S \rightarrow aABe$

$A_A \rightarrow \underbrace{Abc}_{\alpha} / \underbrace{b}_{\beta}$ Eliminate the

$B \rightarrow d$ Left Recursion.

Sol^n:

$S \rightarrow aABe$

$A \rightarrow bA'$

$A' \rightarrow bcA' / \epsilon$

$B \rightarrow d$

Ex

$\underset{A}{S} \rightarrow \underbrace{\underset{A}{S}\underbrace{Sa}_{\alpha}}_{} / \underbrace{b}_{\beta}$ Eliminate the Left Recursion.

Sol^n

$S \rightarrow bS'$

$S' \rightarrow SaS' / \epsilon$

Rule

$A \rightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 \cdots / \beta_1 / \beta_2 / \beta_3 \cdots$

$A \rightarrow \beta_1 A' / \beta_2 A' / \beta_3 A' \cdots$

$A' \rightarrow \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \cdots / \epsilon$

Ex

$S \rightarrow A$

$A \rightarrow \underbrace{Ad}_{\alpha_1} / \underbrace{Ae}_{\alpha_2} / \underbrace{aB}_{\beta_1} / \underbrace{ac}_{\beta_2}$ Eliminate the Left Recursion

$B \rightarrow bBC / f$

$C \rightarrow g$

Sol^n

$S \rightarrow A$

$A \rightarrow aBA' / acA'$

$A' \rightarrow dA' / eA' / \epsilon$

$B \rightarrow bBC / f$

$C \rightarrow g$

Ex.

$S \rightarrow aBDh$
$B \rightarrow \underbrace{Bb}_{\alpha} / \underbrace{c}_{\beta}$
$D \rightarrow EF$
$E \rightarrow g / \epsilon$
$F \rightarrow f / \epsilon$

Eliminate the Left Recursion.

Solⁿ

$S \rightarrow aBDh$
$B \rightarrow cB'$
$B' \rightarrow bB' / \epsilon$
$D \rightarrow EF$
$E \rightarrow g / \epsilon$
$F \rightarrow f / \epsilon$

Ex

$E \rightarrow \underbrace{E+T}_{\alpha} / \underbrace{T}_{\beta}$
$T \rightarrow \underbrace{T*F}_{\alpha} / \underbrace{E}_{\beta}$
$F \rightarrow (E) / id$

Eliminate the Left Recursion.

Solⁿ

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

Ex

$S \rightarrow Aa / b$
$A \rightarrow \underbrace{Ac}_{\alpha_1} / \underbrace{Sd}_{\beta_1} / \underbrace{\epsilon}_{\beta_2}$

Eliminate the Left Recursion.

Solⁿ

$S \rightarrow Aa / b$
$A \rightarrow \cancel{SdA' / A'} \rightarrow Aa / b$
$A' \rightarrow cA' / \epsilon$

$S \rightarrow Aa / b$
$A \rightarrow AadA' / A'$
$A' \rightarrow cA' / \epsilon$

Again Left Recursion

But this is not completely remove the left recursion, B'coz of S. When we ~~exp~~ expand S we again find the left recursion. So this is not eliminating Left Recursion.

So first we expand the prod^n then eliminate Left Recursion.

Rule To eliminate immediate left recursion. ~~the formula~~

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

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

But this formula only eliminates immediate left recursion but not eliminate left recursion when it is in more than one level.

When we have the left recursion in ~~more than one level~~ then first expand that expression & then apply the rule of immediate left Recursion.

So again we solve the example -

Ex) $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

Sol^n $S \rightarrow Aa \mid b$
$A \rightarrow \underbrace{Ac}_{\alpha_1} \mid \underbrace{Aad}_{\alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{\epsilon}_{\beta_2}$

(replacing S with the given prod^n)
(Expanding S first)

now applying the rule it

$S \rightarrow Aa / b$

$A \rightarrow bdA' / A'$

$A' \rightarrow cA' / adA' / \epsilon$

Non deterministic Grammar →

Consider the grammar

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 \text{ ---- } / \gamma$

Here the parser go through one by one from first to last options.

So the parser first see the α but there is more than one option for the α such that $\alpha\beta_1$ or $\alpha\beta_2$ or $\alpha\beta_3$. So it is called Non-deterministic Grammar.

Here the problem of common prefix is occured. To eliminate this problem we have to perform the backtracking but the backtracking is very expensive & difficult. So we have to convert the Non deterministic Grammar (NDG) to the deterministic Grammar (DG) by performing the Left factoring.

Left Factoring

Rule

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \ldots \quad \text{NDG}$$

$$\left.\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \end{aligned}\right\} \text{DG.}$$

For each nonterminal A find the longest prefix α common to two or more of its alternative. If $\alpha \neq \epsilon$, i.e. there is a nontrivial common prefix replace all the A productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Ex

$$\begin{aligned} S &\rightarrow \underline{iEtS} \\ &\mid \underline{iEtSeS} \\ &\mid a \\ E &\rightarrow b \end{aligned}$$

Remove common prefix problem means Do the Left factoring.

Sol^n

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Que

$S \rightarrow AaS \mid AaA$
$A \rightarrow a \mid ab$

left factoring the grammar.

Sol^n

$S \rightarrow AaS'$
$S' \rightarrow S \mid A$
$A \rightarrow aA'$
$A' \rightarrow \epsilon \mid b$

Que

$S \rightarrow abc \mid abd \mid ae \mid f$

Sol^n

$S \rightarrow abS' \mid ae \mid f$
$S' \rightarrow c \mid d$

but here again we have the common prefix problem so that we again do the left factoring

$S \rightarrow aS'' \mid f$
$S'' \rightarrow bS' \mid e$
$S' \rightarrow c \mid d$