CD1

Induction variable

Variable in a loop updated in a regular pattern of each iteration.

	Static Runtime Environment	Stack Based Runtime Environment
Memory allocation	Compile time	Runtime
Memory lifespan	Entire program execution	Limited to the duration of function calls.
memory organization	Fixed segments	Runtime Stack.
Recur -sion	Not suitable less flexible.	Ideal for recursion. More flexible.

CD1

Introduction: The structure of a compiler, Phases of Compilation, The translation process, Major Data Structures in a compiler, Bootstrapping and Porting.
Lexical Analysis (Scanner): The Role of the Lexical Analyzer, Input Buffering, Specification of Tokens, Recognition of Tokens, The lexical Analyzer Generator Lex.

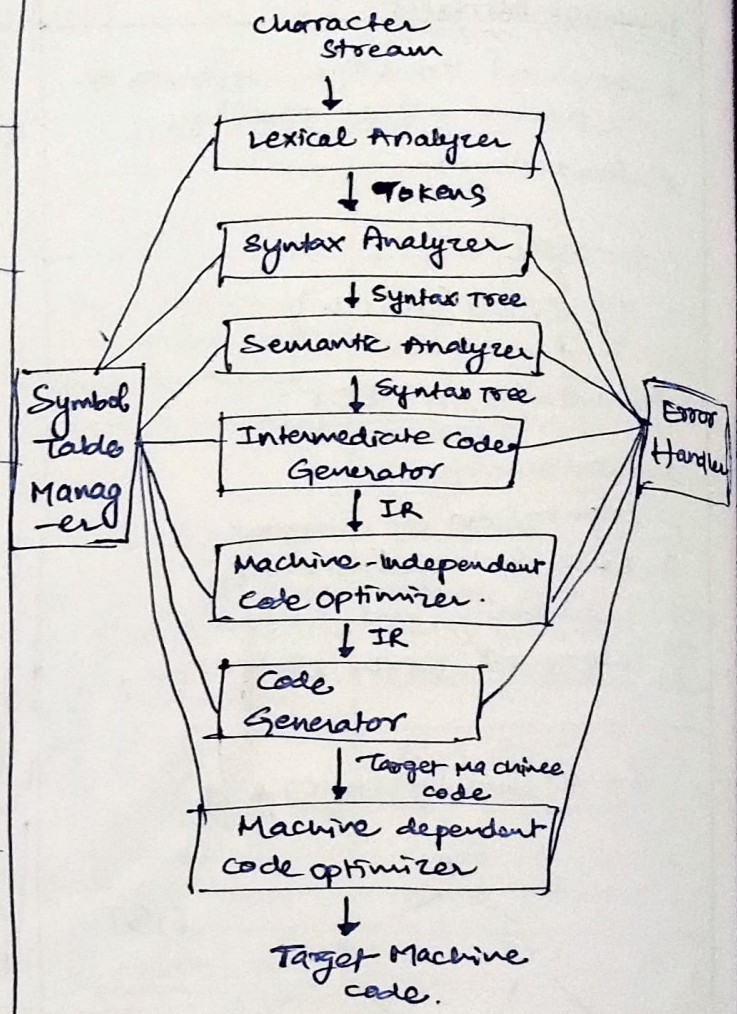
Compiler: A system software which is used to translate the program written in one programming language into machine understandable language to be executed by a computer. It should make the target code efficient and optimized in terms of time & space.

Source Language → translates → Target Language.

↓ reports syntactical errors.

Structure of Compiler

* Compiler operates as a sequence of phases.
* each phase transforms one representation of the source program to another.
* Decomposition of a Compiler.



PHASES OF COMPILER ....

Symbol Table: Which stores information about the entire source program and is used by all phases of the compiler.

Phases of Compilation:

* Lexical Analysis.
* Syntax Analysis.
* Semantic Analysis.
* Intermediate code generation.
* Code optimization.
* Code Generation.
* Symbol Table Management
* Error Handling.

Expt. No.
Date

Lexical Analysis (Scanning):

* First phase of compiler.
* reads the stream of characters that make up the source program starting from left to right
* And groups the characters into meaningful sequences called lexemes.
* For each lexeme, it produces a token of the form:

<token-name, attribute-value>

- token-name: abstract symbol (used during syntax analysis)
- attribute-value: points to an entry in the symbol table.

* These tokens are passed to syntax analyzer.

TOKEN: It represents a logical cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc ...

Ex: p = i + r * 60

p, i, =, +, r, *, 60 are all seperate lexemes.

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

↓ Tokens generated...

Syntax Analysis:

* Second phase.
* parser uses the tokens produced by the lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of the token stream.
* Syntax tree in which each interior node represents an operation and the children of the node represent the arguments (operands) of the operation.

Ex:- p = i + r * 60

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

Semantic Analysis:

* third phase
* uses syntax tree and the information in the symbol table to check for consistency with the language definition.
* It performs type conversion of all the data types into real data types.
* imp part is type checking.
* Some language specification may permit some type conversions called coercions.
  Ex:- int to float.

Intermediate Code Generation

* fourth phase
* In process of translating a source program into target code, a compiler may construct one or more intermediate representations.
* After Semantic Analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation of the source program.
* three-address code consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

Ex:- p = i + r * 60

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

Three-address code :

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 * t2
id1 = t3.
```

Code Optimization :

* fifth phase.
* It gets the intermediate code as input and produces optimized intermediate code as output.
* attempts to improve the intermediate code so that better target code is generated.
* faster, shorter code, or target code that consumes less power.

Ex :- p = i + r * 60

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

Optimized code is :

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation :

* Sixth-phase.

input		output
Intermediate representation/ Optimized machine independent representation.	maps it into →	Target Language.

* If target language is machine code, registers or memory locations are selected for each of the variables used in the program.
* Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Ex :- p = i + r * 60

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

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

Where R1, R2 are registers.

Symbol Table Manager

* It is used to store all the information about identifiers used in the program.
* It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
* allows to find the record for each identifier quickly and to store or retrieve data from that record.

Error Handling :

* Imp function is the detection and reporting of errors in the source program.
* The error message should allow the programmer to determine exactly where the errors have occurred.
* Each phase can encounter errors. After detecting an errors, a phase must handle the error so that compilation can proceed.

Classification of Compilers :

* are classified as single-pass, multi pass, load-and-go, debugging, or optimizing.

⇒ The processes of translation of program to machine understandable code is divided into 2 parts.

1) Lexical Part → Lexical Analysis, Syntax Analysis, Semantic Analysis.
2) Synthesis part → Intermediate code generator, code optimizer, target code generator.

The Translation Process

Ex :- position = initial + rate * 60

↓

Lexical Analyzer.

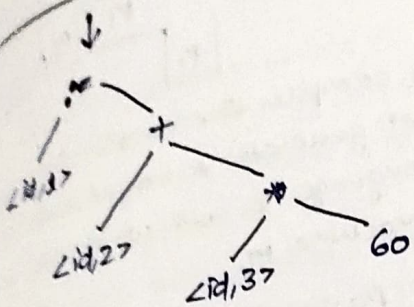
Tokens <id,1> = <id,2> + <id,3> * 60

↓

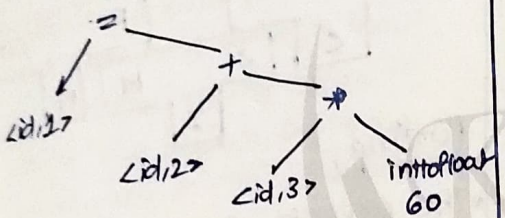
Syntax Analyzer.

↓

Expt. No.
Date



Semantic Analyzer



Intermediate Code Generator

```
temp1 := inttofloat (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, r2
ADDF   r2, r1
MOVF   r1, id1
```

Major Data Structures in a Compiler:

Tokens:

⇒ Scanner collects characters / lexeme into a token.

⇒ It represents the token symbolically as a value of an enumerated data type representing a set of tokens of the source language.

⇒ Sometimes, a single global variable can be used to hold the token information.

⇒ In other cases, an array is required to hold the lexeme (or) token.

Syntax Tree:

⇒ Parser generates syntax Tree.

⇒ It is constructed as a standard pointer-based structure that is dynamically allocated.

⇒ Entire tree can be kept as a single variable pointing to the root.

⇒ Each node is a record.

Symbol Table:

⇒ created and maintained by compilers in order to store information about the occurances of various entities such as variable names, function names, objects, classes, interfaces etc..

⇒ To store the names of all entities in a structured form at one place.

⇒ To verify if a variable has been declared.

⇒ To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

⇒ To determine the scope of a name. (scope resolution).

Page No.

Literal Table:

⇒ Stores constant and strings used in the program.

⇒ One literal table applies globally to the entire program.

Used by code generator to:

* Assign addresses for literals.
* Enter data definitions in the target code file.

⇒ Avoids the repication of constants and strings.

⇒ Quick insertion and lookup are essential.

⇒ Deletion is not allowed.

Temporary Files:

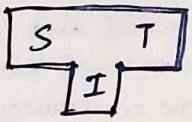
* Some memory during compilation is used to store in temporary files to hold the products of intermediate steps.

Bootstraping:

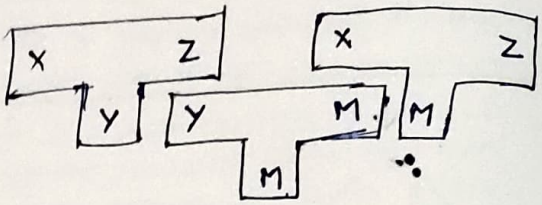
* A compiler is characterized by 3 languages:
  1. Source Language (S).
  2. Target Language (T).
  3. Implementation Language (I).

S → Translated to → T
with the help of (I).

* represented in the form of a T-diagram as S^{T}_{I}

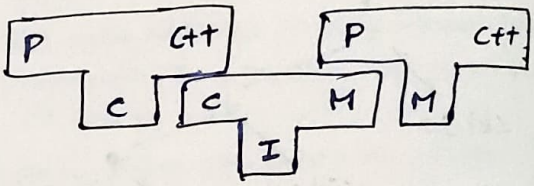


* When a translator is required for an old language on a new machine, or a new language on an old machine, use of existing compilers on either machine is the best choice for developing
* Developing a compiler for a new language by using an existing compiler is called bootstrapping.
* A process by which a simple language is used to translate more complicated programs, which intern may handle an even more complicated program.
* Bootstrapping is used to create compilers and to move them from one machine to another by modifying the backend.



⇒ A compiler for source language X that generates a target code in language Z and which runs on machine M.

Ex:- Pascal translator for C++ by bootstrapping with C for M.



Porting:

* The process of modifying an existing compiler to work on a new machine is often known as porting the compiler.
* To develop a compiler for new hardware machine from an existing compiler, change the synthesis part of the compiler because, synthesis part is machine dependent part. This is called Porting.

Quick and Dirty Compiler:

Native Compiler: Compilers that generates code for the same platform on which it runs.

High Language → computer's native language.

Cross Compiler: Compiler capable of creating executable code for a platform other than the one which the compiler is running.

Expt. No.
Date

Lexical Analysis [Scanner]

Tokens:
- Keywords / Reserve words / predefined words [Lexemes]
- Identifiers: User defined strings (ID)
- Special Symbols.
- Any value associated to a token is called as an attribute of the token. (string or numerical)

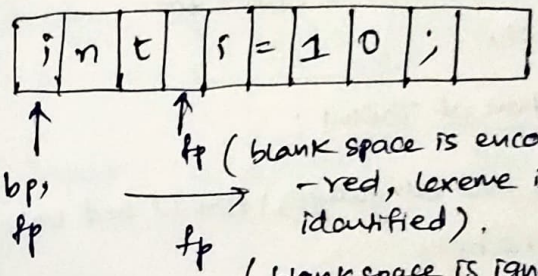
* getNextToken() command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

The Role of Lexical Analyzer:

* To read the source program character by character and form them into logical units called as tokens.
* These tokens are given as inputs to next phase of compiler.
* Scanner rarely converts the entire program into tokens at once, but conversion always depends on the parser.
* lexical analyzer interacts with the symbol table when it discovers a lexeme constituting an identifier and enters that lexeme into the symbol table.
* Identification of lexemes.
* Stripping out comments and white space.
* correlating error messages generated by the compiler with the source program.

Input Buffering

* Scanner scans the input from left to right one character at a time.
  - ⇒ begin ptr (bp)
  - ⇒ forward ptr (fp).



* Input characters are always read from secondary storage, but this reading is costly.
* To speed up the scanning process, buffering technique is used.
* A block of data is first read into a buffer and lexical analysis process is continued on buffer.

Buffering techniques:

1. Buffer (one / two buffers are used).
2. Sentinels.

Buffering:

One buffer scheme:

* only one buffer is used to store the input string.
* Problem: if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled.

Two-buffer scheme: overcomes one buffer scheme problem.

* two buffers are used to store the input string.
* first and second buffer are scanned alternately.

Sentinel:

* Special character introduced at the end of the buffer is called as Sentinel which is not a part of program.
* 'eof' is the natural choice for sentinel.

Specification of Tokens:

1. Strings and Languages: ASCII and Unicode
   * Alphabets
   * Strings.
   * Special Symbols
   * Language
   * Longest match rule.
   * operations. (union, concat, Kleene closure)
   * Notations '(L(r) U L(s))
   * Regular Expression. (x*, x+)
   * Finite automata. ($\{Q, \Sigma, q_0, \delta, F\}$).

Recognition of Tokens:

* Patterns are created by using regular expression.
* These patterns are used to build a piece of code that examines the input strings to find a prefix that matches the required lexemes.

Ex:-

```
Stmt → if expr then stmt
     | if expr then else stmt
     | ε
expr → term relop term
     | term
term → id
     | number.
```

```
digit → [0-9]
digits → digit+
number → digits(.digits)?
         (E[+-]?digits)?
letter → [A-Za-z]
id → letter (letter | digit)
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>
```

with the help of these definitions, lexical analyzer will recognize tokens.

For white space;

ws → (blank/tab/newline)+

The Lexical Analyzer Generator Lex

* we can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical analyzer generator and compiling those patterns into code that functions as a lexical analyzer.
* Also called Lex (Flex).

Lex input file consists of three parts:

1. A collection of definitions.
2. A collection of rules.
3. A collection of auxiliary routines (or) user routines.

* All the sections are seperated by double percent signs, default layout of a lex file is:

```
{ definitions }
%%
{ rules }
%%
{ auxiliary routines }.
```

KESHAV MEMORIAL ENGINEERING COLLEGE

Expt. No._____
Date_____

Declaration section:

* declarations of variables, identifiers and regular definitions.

```
%{
    Declarations
%}
```

Rules section:

* The translation rules each have the form `Rule; {Action;}`.
* Each rule is a regular expression.
* The actions are fragments of code, typically written in C.

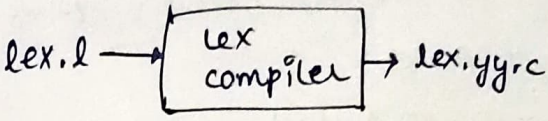
```
%%
    Rule1 {Action1}
    Rule2 {Action2}
%%
```

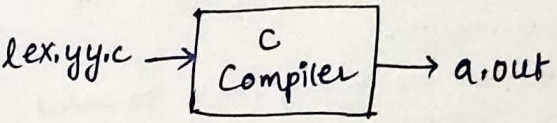
Auxiliary Routines:

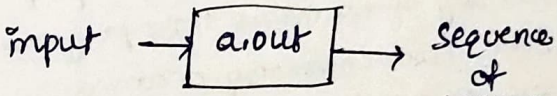
* contains additional functions that are required.
* these functions may be compiled seperately and loaded with the lexical analyzer.

yylex() } are predefined
yywrap() } procedures of LEX.

* Lex program or Lex files are saved with '.l' extension (dot l).







Lex predefined functions and variables

int yylex() → call to invoke lexer, returns token.

char *yytext → pointer to matched string.

yyleng → Length of matched string.

ECHO → write matched string.

CD2:

Syntax Analysis (Parser): The role of the parser, Syntax Error handling and Recovery, Top-Down Parsing, Bottom-Up parsing, Simple LR parsing, More powerful LR parsing, using Ambiguous Grammars, parser Generator Yacc.

Syntax Analysis (Parser)

* Second phase.
* It gets the input from the tokens and generates a syntax tree or parse tree.

Parsing: Determining the syntax or the structure of a program. Parse tree specifies the statements execution sequence.

The Role of Parser:

* The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer.
* Verifies that the string can be generated by the grammar for the source language.
* reports any syntax errors in the program.
* Recovers from commonly occurring errors so that it can continue processing its input.

Functions of the parser:

1. Verifies the structure generated by the tokens.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues:

Parser cannot detect errors such as:-

1. Variable re-declaration.
2. Variable initialization before use.
3. Data type mismatch for an operation.

These can be handled by Semantic Analysis phase.

⇒ There are 3 general types of parsers for grammars:

I. Universal	II. top-down	III. bottom-up

Universal parsing methods can parse any grammars.

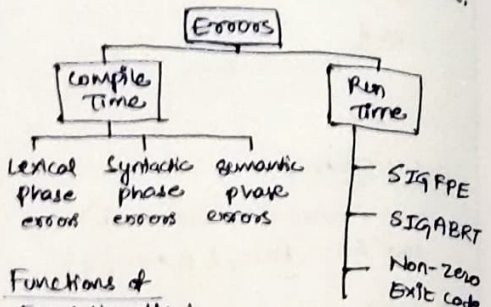
i. the Cocke-Younger-Kasami algorithm
ii. Earley's algorithm.

Commonly used methods in compilers are either top-down or bottom-up

Syntax Error handling and Recovery:

Errors at many different levels. Ex:-

1. Lexical → misspelling a keyword.
2. Syntactic → arithmetic expression with unbalanced parantheses.
3. Semantic → an operator applied to an incompatible operand.
4. Logical → infinitely recursive call.



Functions of Error handler:

* Should report the presence of errors clearly and accurately.
* Should recover from each error quickly.
* Should not significantly slow down the processing of correct programs.

Error recovery Strategies:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction.

Synchronizing token, delimiter. → augmented grammars

Local correction. → no. of insertions, deletions and changes of tokens is as small as possible.

PARSING:

* It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse Tree:

⇒ Graphical representation of a derivation or deduction is called a parse tree.

⇒ interior node → non-terminal.
child → terminals or non-terminals.

Top-Down Parsing

* A parser can start with the start symbol and try to transform it to input string.

Ex:- LL Parser.

* It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from root to the leaves.

Two Types:

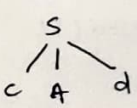
1. Recursive Descent Parsing.
2. Predictive parsing.

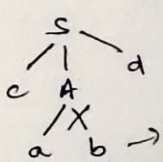
1. Recursive Descent Parsing:

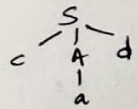
* uses a set of recursive procedures to scan its input.
* may involve back tracking, making repeated scans of the input.

Ex:- G: S → cAd.
A → ab | a.

input string is w = cad.

① 

② 

③ 

Expt. No.
Date

* If input is non-terminal
  - ↳ call corresponding function.
* If input is terminal
  - ↳ compare terminal with corresponding input symbol.
  - ↳ If same ⇒ inpptr++.
* If a non-terminal produces 'n' prods they we have to write corresponding in non-terminal.
* There is no need to declare main function and variables

Ex1:-

$E \rightarrow iE'$

$E' \rightarrow +iE' \mid \in$

↑ non-terminal.

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

i+i$

Ex2:-

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \in$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \in$

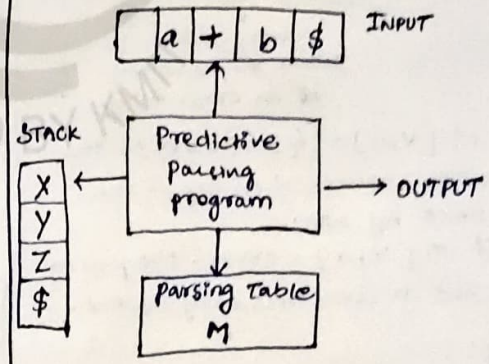
$F \rightarrow (E) \mid id$

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

id+id*id.

2. Predictive Parsing:

* It is a special case of recursive descent parsing where no back tracking is required.
* The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.



Input buffer: It consist of strings to be parsed, followed by $ to indicate the end of the input string.

Stack: It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table: It is 2-D Array

M[A, a], A → non-terminal.
a → terminal.

Predictive parsing program:

* The parser is controlled by a program that considers
  - X → the symbol on top of stack
  - a → current input symbol.
  - ↳ these two determine the parser action.

There are three possibilities;

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the program consults entry M[X, a] of the parsing table M.
   ↓
   entry with either X-production or an error entry.

⇒ If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU.

⇒ If M[X, a] = error, the parser calls an error recovery routine.

FIRST:

rules:-

* X → terminal ? FIRST(X) is {X}.
* X → ε is a production ? then add ε to FIRST(X).
* X → non-terminal & X → aα is a production ? then add a to FIRST(X).
* X → non-terminal & X → $Y_1 Y_2 \dots Y_k$ is a production ? then place 'a' in FIRST(X) if for some i, a is in FIRST(Yi) and ε is in all of FIRST(Y1), FIRST(Y2) ... FIRST(Yi-1).

Ex: A → BC | EFGH | H

B → b
C → c | ε
E → e | ε
F → CE
G → g
H → h | ε

first(H) = first(h) ∪ first(ε)
= {h, ε}

first(G) = {g}

first(C) = first(c) ∪ first(ε)
= {c, ε}.

first(E) = {e, ε}.

first(F) = first(CE)
= (first(C) − {ε}) ∪ first(E)
= ({c, ε} − {ε}) ∪ {e, ε}
= {c, e, ε}.

first(B) = {b}.

first(A) = first(BC) ∪ first(EFGH) ∪ first(H).
= first(B) ∪ (first(E) − {ε}) ∪ first(FGH) ∪ {h, ε}.
= {b} ∪ {e} ∪ (first(F) − {ε}) ∪ first(GH) ∪ {h, ε}
⋮
= {b, c, e, g, h, ε}.

Expt. No.
Date

FOLLOW:

RULES:

* If S is a start symbol, then FOLLOW(S) contains $.
* $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is placed in follow(B).
* $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Ex:- $A \rightarrow BC \mid EFGH \mid H$
$B \rightarrow b$
$C \rightarrow c \mid \epsilon$
$E \rightarrow e \mid \epsilon$
$F \rightarrow CE$
$G \rightarrow g$
$H \rightarrow h \mid \epsilon$.

follow(A) = {$}.

follow(B) =

FIRST:-

FOLLOWS:-

⇒ FOLLOW(A) contains set of all terminals present immediate in right of 'A'

→ FO(A) = {$}

IF $S \rightarrow AaCD$, FO(A) = {a}.

* $S \rightarrow ACD$
$C \rightarrow a \mid b$

FO(A) → FI(C) = {a, b}

FO(D) → FO(S) = {$}.

* $S \rightarrow aSbS \mid bSaS \mid \epsilon$

FO(S) = {$, b, a}.

↓ start symbol

'Follow never contain ϵ'

* $S \rightarrow AaAb \mid BbBa$.
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

FO(A) = {a, b}

FO(B) = {b, a}.

* $S \rightarrow ABC$
$A \rightarrow DEF$
$B \rightarrow \epsilon$
$C \rightarrow \epsilon$
$D \rightarrow \epsilon$
$E \rightarrow \epsilon$
$F \rightarrow \epsilon$

FO(A) = FIRST(B) (ε) = FIRST(C) (ε)

⇒ {$}

Q) $S \to (L) / a$

$L \to SL'$

$L' \to , SL' / \epsilon$

Find First & Follow.

→ FIRST(S) = { (, a }

FIRST(L) = { (, a }

FIRST(L') = { , , ∈ }.

→ FOLLOW(S) = { $ } ∪ FIRST(L') ∪ FOLLOW(L)

= { $, ,) }

FOLLOW(L) = {) }

FOLLOW(L') = FOLLOW(L)

= {) }.

(LL(1)) Non-recursive predictive Parser:

$S \to (L) / a$

$L \to SL'$

$L' \to \epsilon / , SL'$

Step 1:- Find first and follow of above.

Parse table:

	(	)	a	,	$
S	S	1		2		
L	L	3		3		
L'	L'		4		5	

① $S \to (L)$ $S \to a$

↓ First

② $L \to SL'$

③ $L' \to \epsilon$ ⇒ FOLLOW(L')

④ $L' \to , SL'$

* If is LL(1) Grammar because in every cell there is only one entry of production.

* $S \to asbs / bsas / \epsilon$

first(S) ⇒ { a, b, ∈ }

follow(S) ⇒ { b, a, $ }.

	a	b	$
S	1/3	2/3	3

① $S \to asbs$

② $S \to bsas$

③ $S \to \epsilon$ FOLLOW(S) { a, b, $ }

⇒ Not LL(1) grammar because there are multiple cells having more than 1 entry of production.

Algorithm:

repeat

X → top stack symbol
a → symbol pointed to by ip.

if X = terminal (or) $ then

if X = a then

pop X and advance ip

else error()

else

if M[X, a] = $X \to Y_1 Y_2 \dots Y_k$ then

pop X

push $Y_k, Y_{k-1} \dots Y_1$

output $X \to Y_1 Y_2 \dots Y_k$

else error().

until X = $

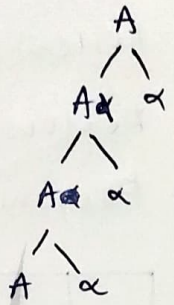
Expt. No.
Date

Problem :-

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Elimination of Left Recursion :

(i) $A \to A\alpha \mid \beta$



① write all the productions with non-left recursive

② introduce new non-terminal related to A'

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

except first non-terminal symbol a write remaining.

Correction of problem :-

$E \to E+T \mid T$

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

$T \to T*F \mid F$

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

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

After removing left recursion

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

$\Rightarrow FI(E) \to FI(T) \to FI(F)$
$\to \{(, id\}.$

$FI(E') \to \{+, \epsilon\}.$

$FI(T) \to FI(F) \to \{(, id\}$

$FI(T') \to \{*, \epsilon\}.$

$FI(F) \to \{(, id\}.$

$\Rightarrow FO(E) \to \{\$,)\}.$

$FO(E') \to \{\$,)\}.$

$FO(T) \to \{+, \$,)\}.$

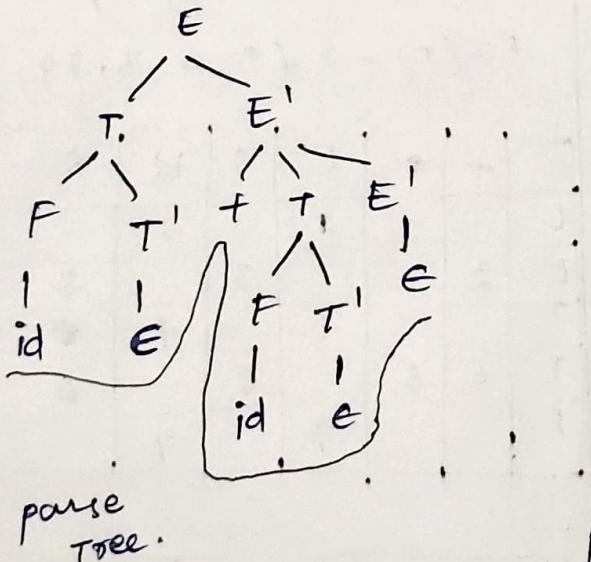
$FO(T') \to \{+, \$,), \}$

$FO(F) \to \{*, +, \$,)\}$

	+	*	(	)	id	$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			7		7	

⇒ id+id$ Stack.

Stack	Input string	Action
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	pop
$E'T'	id+id$	T'→ϵ
$E'	id+id$	E'→+TE'
$E'T+	id+id$	pop
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id.
$E'T'id	id+id$	pop
$E'T'	id+id$	T'→ϵ
$E'	id+id$	E'→ϵ
$	id+id$	Accepted.



parse Tree.

ii) S→(L)/a
L→L,S/S.

S→(L)/a
L→SL'
L'→,S/ϵ

FIRST(S) → {(, a}
FIRST(L) → {(, a}
FIRST(L') → {,, ϵ}.
FOLLOW(S) → {,,), $}
FOLLOW(L) → {)}
FOLLOW(L') → {)}

	(	)	,	a	$
S	S→(L)			S→a	
L	L→SL'			L→SL'	
L'	L'→,S	L'→ϵ			

iii) S→aAB/bA/ϵ
A→aAb/ϵ
B→bB/ϵ.

FIRST(S) ⇒ {a, b, ϵ}
FIRST(A) ⇒ {a, ϵ}
FIRST(B) ⇒ {b, ϵ}
FOLLOW(S) ⇒ {$}.
FOLLOW(A) ⇒ {b, $}.
FOLLOW(B) ⇒ {$}.

	a	b	$
S	S→aAB	S→bA	S→ϵ
A	A→aAb	A→ϵ	A→ϵ
B		B→bB	B→ϵ

Expt. No.
Date

BOTTOM UP PARSING

* constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

Shift Reduce Parser

* A parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Ex:- $S \to aABe$
$A \to Abc \mid b$
$B \to d$.

input: abbcde.

$S \to aABe$
$S \to aAbcBe$ $(A \to Abc)$
$S \to abbcBe$ $(A \to b)$
$S \to abbcde$ $(B \to d)$.

Handles:- A handle of a string is a substring that matches the right side of a production, and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a right most derivation.

Ex:- $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$.

input: $id_1 + id_2 * id_3$.

$E \to E+E$
$E \to E+E*E$ $(E \to E*E)$
$E \to E+E*id_3$ $(E \to id)$
$E \to E+id_2*id_3$ $(E \to id)$
$E \to id_1+id_2*id_3$ $(E \to id)$.

The underlined substrings are called Handles.

Actions in Shift-reduce parser:

* shift: next i/p symbol is shifted onto the top of the stack. (push)
* reduce: parser replaces the handle within a stack with a non-terminal.
* accept: announces successful completion of parsing.
* error: discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

Ex:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

id*id

Stack	Input Buffer	Action
$	id*id$	shift
$id (Handle)	*id$	Reduce by F→i
$F	*id$	Reduce T→F
$T (R)	*id$ (S)	shift (SR)
$T*	id$	shift
$T*id	$	reduce F→id
$T*F (R R R)	$	reduce T→T*F (RR)
$T	$	reduce E→T
$E (↑ start symbol.)	$	ACCEPTED

LR Parsers

* used to parse a large class of CFG is called LR(k) parsing.
* 'L' ⇒ left to right scanning of input.
  'R' → for constructing rightmost derivation in reverse.
  'k' → number of input symbols. (k=1) default if k is omitted.

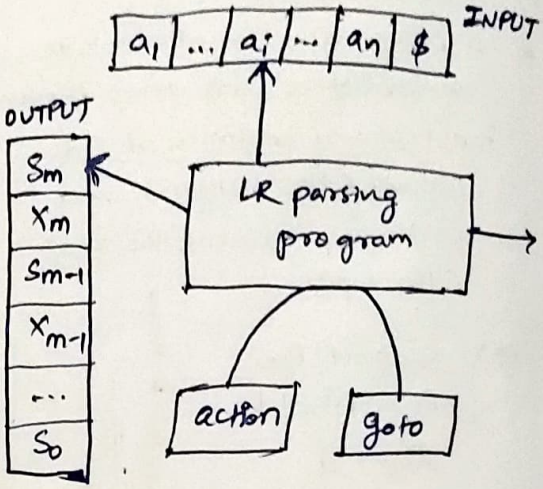
Advantages:

* efficient non-backtracking shift reduce parsing method.
* A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
* It detects a syntactic error as soon as possible.

Drawbacks

* too much of work to construct.
* LR parser generator is needed Ex:- YACC.

LR-parsing Algorithm;



Stack: consists of an input, an output, a stack, a driver program and a parsing table that has two parts, action & goto.

* The driver program is the same for all LR parser.
* The parsing program reads characters from an input buffer one at a time.
* program uses a stack to store a string

  "$S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m$"

  where, S_m is on top,
  X_i is grammar symbol
  S_i is a state.

Action :- shift, reduce, accept & error.

Expt. No.
Date

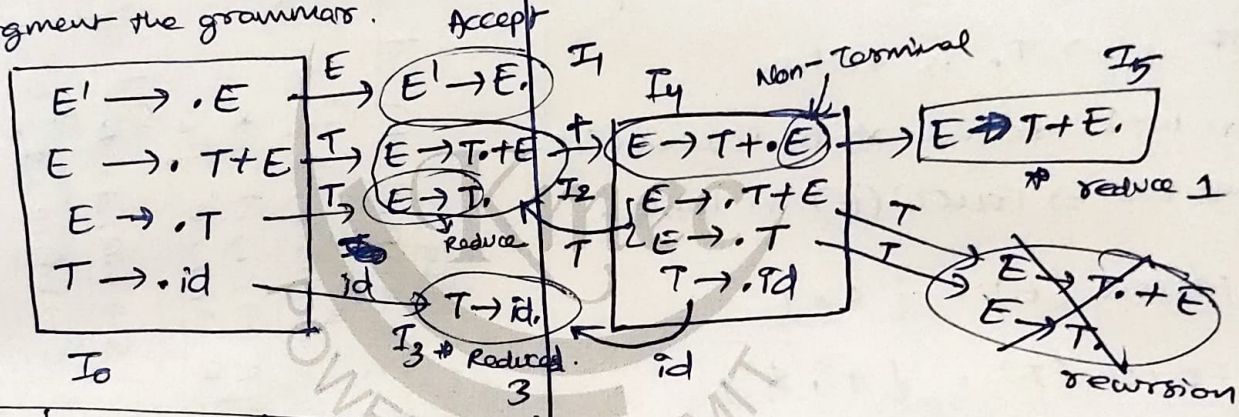
Goto: The function goto takes a state and grammar symbol as arguments and produces a state.

LR(0):

* To construct table we use LR(0) canonical items

* Ex:- $E \to T+E / T$
$T \to id$.

① Augment the grammar.



State	id	+	$	E	T
0	S_3			1	2
1	—	—	Accept		
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		
	Action			Goto.	

* If there is more than one entry in the cell i.e., Shift-reduce (or) reduce-reduce

* Given grammar is not LR(0).

NOTE:- Write reduction in all of the action columns of that row if there is a reduce in any of column of that row.

SLR(1):

* From the LR(0) example.
* Difference is from parsing table

State	id	+	$	E	T
0	S_3			1	2
1			Accept		
2		S_4	r_2		
3		r_3	r_3		
4	S_3			5	2
5		r_1			

* If $E \to T.$ is reduce then write reduce in Follow(Left Hand side) ⇒ Follow(E) in table.
* Follow(E) = { $,) }

  Follow(T) = { +, $ }

  $T \to id. \Rightarrow r_3 \Rightarrow (+, \$)$.

* There is no shift reduce or reduce-reduce conflict in this grammar so it is SLR(1).

CLR (uses LR1 Canonical Items).

$S \to aAd$ ① / bBd ② / aBe ③ / bAe ④

$A \to c$ ⑤

$B \to c$ ⑥

① ⇒ Augment

$S' \to .S, \$$ (look ahead.)

$S \to .aAd, \$$

$S \to .bBd, \$$

$S \to .aBe, \$$

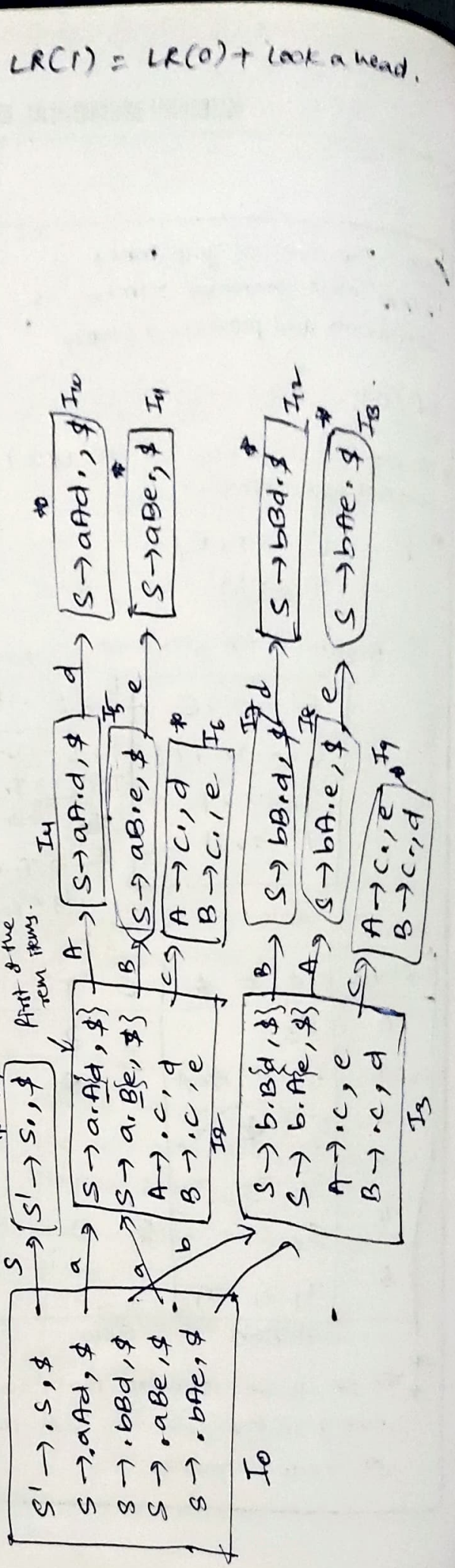
$S \to .bAe, \$$

$S \to .bAe, \$$

$A \to .c$

$B \to .c$

LR(1) = LR(0) + look ahead.



Expt. No.
Date

State	a	b	c	d	e	$	S	A	B
0	S_2	S_3				Acc	1		
1								4	5
2			S_6	S_9				8	7
3				S_{10}					
4					S_{11}				
5				r_5	r_6				
6				S_{12}					
7				S_{13}					
8				r_6	r_5				
9						r_1			
10	r_4								
11						r_3			
12						r_2			
13						r_4			

CLR (canonical LR)

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

① Augmented grammar.

$S' \to S$ ⓪
$S \to CC$ ①
$C \to cC$ ②
$C \to d$ ③

② calculating LR(1) items.

I_0:
$S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

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

goto(I_0, C) :
I_2 : $S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

goto(I_0, C) :
I_3 : $C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

goto(I_0, d) :
I_4 : $C \to d., c/d$

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

goto(I_2, c) :
I_6 : $C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

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

goto(I_3, C) :
I_8 : $C \to cC., c/d$

goto(I_3, c) :
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

goto (I_3, d) :-

$C \to d\cdot, c/d \} I_4$

goto (I_6, C) :-

$I_9 : C \to CC\cdot, \$$ *

goto (I_6, C) :-

$C \to c\cdot C, \$$
$C \to \cdot cC, \$$ $\} I_6$
$C \to \cdot d, \$$

goto (I_6, d) :-

$C \to d\cdot, \$ \} I_7$.

Table :-

State	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Stack	Input string	Action
$0	dd$	shift 4
$0d4	d$	reduce 3 $C \to d$
$0C2	d$	shift 4
~~$0C2d4~~	~~$~~	
$0C2d7	$	reduce 3 $C \to d$
$0C2C5 (↑ 2C)	$	reduce 1 $S \to CC$ ← Pop 4
$0S1	$	ACCEPT.

If the RHS contains 1 symbol then we have to pop 2 symbols from the stack.

LALR (Lookahead LR)

$S \to L = R / R$

$L \to *R / id$

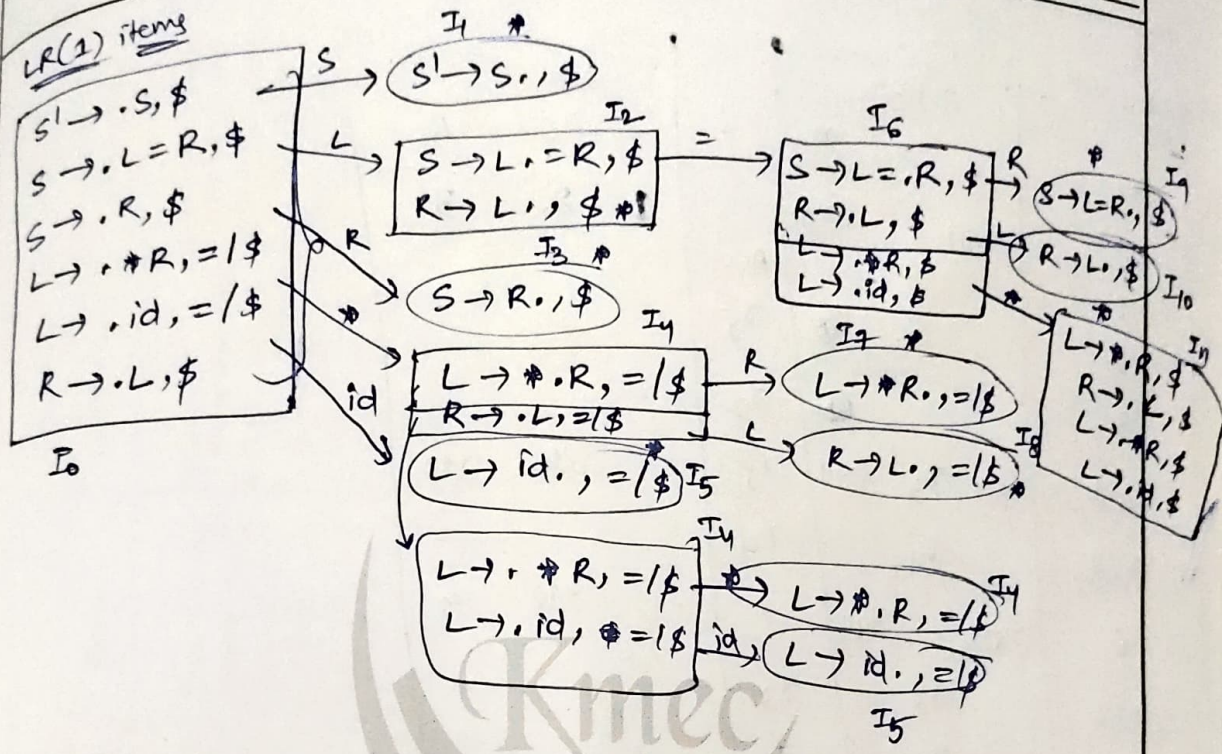
$R \to L$

also parse $id = id$.

Augmented Grammar :

0. $S' \to S$
1. $S \to L = R$
2. $L \to *R$
3. $R \to L$
4. $S \to R$
5. $L \to id$.

Expt. No.
Date



$I_4 \xrightarrow{R} L \to *R., =/\$$ } I_{13}

$I_4 \xrightarrow{L} R \to L., =/\$$ } I_{10}

$I_4 \xrightarrow{*} L \to *.R, =/\$$ } I_4

$I_4 \xrightarrow{id} L \to id., =/\$$ } I_{12}

$I_6 \xrightarrow{id} L \to id., \$$ I_{12}

Combine I_4 & I_{11} into single state

where $I_4 = L \to *.R, =/\$$ $I_{11} \Rightarrow L \to *.R, \$$ $\Rightarrow I_{411}$

$R \to .L, =/\$$ $R \to .L, \$$

$L \to .*R, =/\$$ $L \to .*R, \$$

$L \to .id, =/\$$ $L \to .id, \$$

I_5 & I_{12}

$I_5 = L \to id., =/\$$ $I_{12} = L \to id., \$$ $\Rightarrow I_{512}$

I_7 & I_{13}

$I_7 = L \to *R., =/\$$ $I_{13} = L \to *R., \$$ $\Rightarrow I_{713}$

Similarly :- I_{810}

In LALR, If two states contains same productions but different Look Ahead symbols then we combine the states.

Table:-

States	Action id	=	*	$	Goto S	L	R
0	S_{512}		S_{411}		1	2	3
1				Acc			
2		S_6		r_5			
3				r_2			
411	S_{512}	r_5	S_{411}	r_5		810	713
512		r_4		r_4			
6	S_{512}		S_{411}			810	9
713		r_3		r_3			
810		r_5		r_5			
9				r_1			

id = id .

Stack	input string	Action
$0	id = id $	S512
$0 id 512 (pop)	= id $	reduce 4 L → id
$0 L 2	= id $	S6
$0 L 2 = 6	id $	S512
$0 L 2 = 6 id 512	$	reduce 4 L → id.
$0 L 2 = 6 L 810	$	reduce 5 R → L
$0 L 2 = 6 R 9	$	reduce S → L = R
$0 S 1	$	Accepted.

Differences between LL(1), LR(0), SLR, CLR, LALR.

Type	Direction	LAH	Mechanism	Power	Advantages	Limitations
LL(1)	Top-Down	1	* Predictive parsing using FIRST and FOLLOW sets; * requires left-factored, non-left recursive grammars.	Limited-only works for a subset of CFGs	* Simple implementation, * easy to understand, * implement-	* cannot handle left recursion (or) ambiguous grammars, * requires grammar refact - oring.
LR(0)	Bottom-UP	0	* Uses dot positions * no look ahead used for decision.	Very limited	* Simple state construction	* High chance of conflicts * insufficient-
SLR	Bottom-UP	0 of Follow sets (implicit lookahead)	* Enhances LR(0) items by incorporating FOLLOW sets for reduce decisions.	* More powerful than LR(0), but still limited	* Simpler than full LR(1) * modest table sizes	* May produce conflicts due to coarse LAH.
CLR	Bottom-UP	1 (per item)	* constructs LR(1) + explicit LAH sets for precise parsing decisions	* Maximum power * nearly handles all deterministic CFGs	* Highly accurate * discriminative parsing * minimal conflict	* parsing tables can be extremely large (state explosion)
LALR	Bottom-UP	1 (merged)	* Merges LR(1) states with identical LR(0) cores to reduce table size	* as powerful as CLR	* compact table size * widely used in yacc / bison	* state merging can introduce conflicts not seen in full LR

KESHAV MEMORIAL ENGINEERING COLLEGE

Expt. No.

Date

Page No.

Ambiguity: A grammar that produces more than one parse for some sentence is said to be ambiguous grammars.

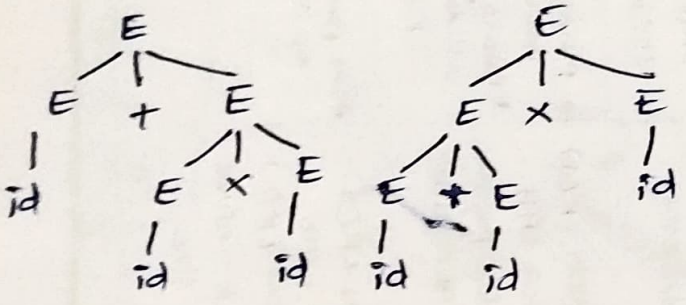
(or)

More than one LMD or more than one RMD for some string is also called ambiguous grammars.

Ex:- $E \to E+E \mid E*E \mid (E) \mid -E \mid id$

for id+id*id

$E \to E+E$	$E \to E*E$
$E \to id+E$	$E \to E+E*E$
$E \to id+E*E$	$E \to id+E*E$
$E \to id+id*E$	$E \to id+id*E$
$E \to id+id*id$	$E \to id+id*id$.



* Rewriting the grammar can eliminate the ambiguity in grammars.

Eliminating Left-Recursion:

* A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α.
* Top-down grammars cannot handle left-recursive grammars.

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

↓ After removing left recursion.

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

Left factoring:

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

↓

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

Handling Ambiguous Grammar:

1. Shift-Reduce conflicts.
2. Reduce-Reduce conflicts.
3. Use of Ambiguity-Resolving techniques.
4. preprocessing
5. Grammar modification.

Parser Generator YACC:

* To automate the process of parsing an input string by parser.
* YACC (Yet Another Compiler Compiler) is one such automatic tool for parser generation.
* It is an UNIX based utility tool for LALR parser generator.
* LEX and YACC work together to analyze the program syntactically, can report conflicts or ambiguities in the form of error messages.

How YACC works?:

1. Grammar Specification. (imp → wBNF)
2. Parsing Table Generation.
3. Code Generation.
4. Integration with LEX
5. Error Handling.

YACC specification

```
%{ & declarations %}
%% token
%%
translation rules
%%
supporting C routines, (optional)
```