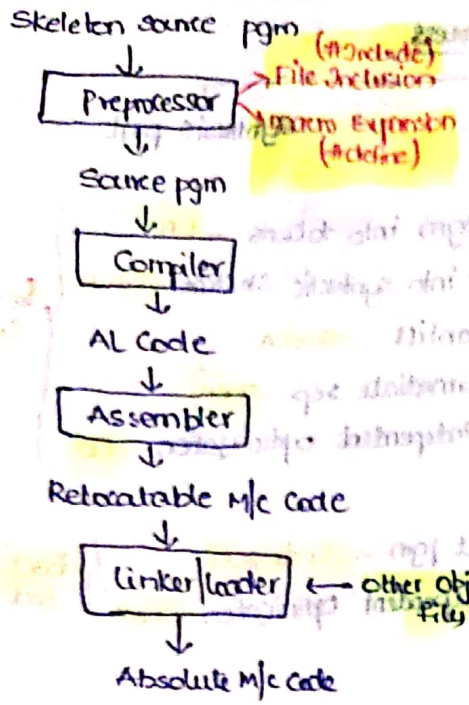
Compiler design :

→ Compiler is a language processor which translates source program into an object program

Language processing system :



*) The preprocessor may expand the shorthands called MACROS into source lang stmts and collects all modules of source pgm

*) Linker allows us to make a single pgm from several files of relocatable M/c code

*) Loader : The process of loading consists of taking relocatable M/c code, altering Relocatable M/c code and placing the altered instructions and data in memory at the proper locations.

Interpretor :

It is interprets each instruction and execute it immediately

- → less memory
- → slower
- → Consider each instruction at a time
- → Debugging is different

→ for (i=0 ; i<10 ; i++)
  PF ("Hello");

Compiler : Translate 1 time
Interpretr : Translate 10 times.
→ It goes as per logical flow of pgm

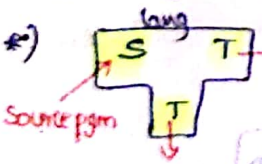
Ex:- LISP, APL, SNOBOL, ADA, PHP, phyton ... etc

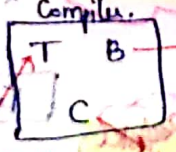
Bootstrapping :

It is a concept of designing a compiler pgm for a pgmming language which is a subset of source language (ie),

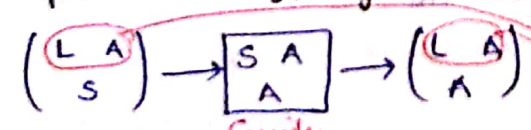
Compiling a compiler pgm in its own language to generate a compiler for new language

*)





Design Compiler for 'L' lang, targeted for M/c A using 'S' lang



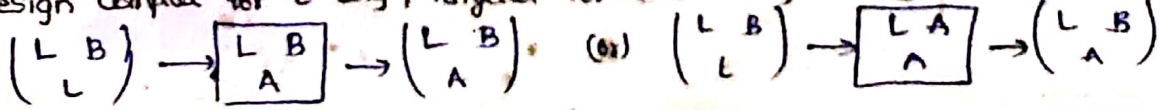
*) when the platform is changed then Compiler also changes for other M/c.

Logic never changes.

Cross Compiler :

A Compiler may run on one M/c and produce target code for another M/c such a Compiler is often called cross Compiler. and the process is called cross Compilation

Design Compiler for 'L' lang, targeted for M/c B using 'L' lang



Compilation process : (LANCE)

Source lang → Front end → Back end → Target lang

*) HLL
- lexical Analysis
- stream of tokens
- Syntax Analysis
- Parse tree
- Semantic Analysis
- Parse tree (Semantically verified)
- Intermediate Code generation
- 3addr Code
- Code optimization
- Target Code generation
- Assembly

Symbol table

Error handler

Ex: z = a + b * c HLL

(lex) lexical Analysis id → a(a|d)*

id = id + id * id

(yacc) Syntax Analyzer

G
S → id = E;
E → E+T | T
T → T * F | F
F → id

get another Compiler Compiler

S
- id
- =
- E
- ;
- id
- +
- T
- T
- *
- F
- F
- id

Semantic Analysis

(Semantically verified parse tree)

like LHS should be var.

Compilation process

Analysis part — Synthesis part

Analysis part :
- Divide the pgm into tokens - LA
- Tokens grouped into syntactic structure - Syn A
- Verify the semantics - Sem A
- Generate intermediate rep - ICG
- Optional M/c independent optimization - CO

(Front End)

Synthesis part :
- Generates target pgm - Code gen
- Optional M/c dependent optimization - CO

(Back End)

ICG

$t_1 = b * c$

$t_2 = a + t_1$ → 3addr code

$x = t_2$

Code optimization

$t_1 = b * c$

$x = a + t_1$

Code generator

Mul R1, R2
Add R0, R1
Mov x, R0

a → R0
b → R1
c → R2

Asm Code:

Data flow Analysis

Register Allocation

Lexical Analysis :

- Stream of characters making up the source pgm is read from left to right and grouped into tokens (Identifiers, Keywords, Constants, Special char)

Ex:- Total = sub1 * sub2 + 2 ;

Buffer
Total
sub1
sub2

Identifier : Total, sub1, sub2
operator : = + *
Constant : 2
Special char : ;

Symbol table

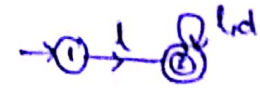
Name	Type	default value
Total	float	0.000
sub1	float	garbage
sub1	float	0.000

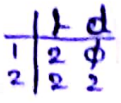
Functions of lexical Analyzer :

- It removes Comments and white spaces.
- Divides the pgm into tokens
- create a record for each identifier in symbol table

Design of lexical Analyzer :

manually :

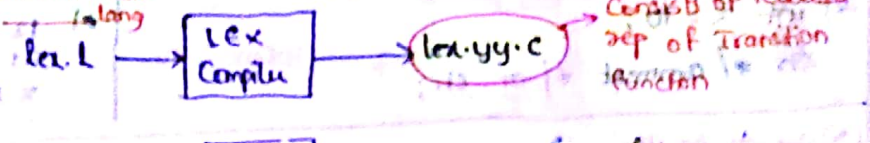
Token - identifier
|
Pattern - letter (letter/digits)
|
RE - l (l/d)*
|
FA - 

|
Transition - 

	l	d
1	2	ϕ
2	2	2

|
Transition fn → Implemented in any pgming lang

Tool (Lex)



lexical Errors :

① unterminated comments :

/* : ... */

② Nested Comments :

/* ... /* ... */ ... */

ignored — Considered as Token

③ length of longest identifiers :

Identifier length should be lesser or equal to 31

④ Invalid identifier :

int 1stmonthsalary ;

⑤ Invalid Constant :

int sal = 50,000 ;

⑥ Illegal symbols :

sal = ₹ 60,000 ;

⑦ 'abc
⑧ abc'
⑨ 'A
⑩ A'
⑪ 7.00.04
⑫ 2.3ab

Error recovery in lexical Analysis :

* Compiler spend most of its time in lexical Analysis phase

① Insert :

/* ... */ → inserted

② Delete :

int 1stmonthsalary → int stmonthsalary

③ replace :

int 1stpay → (F) → int fstpay

④ Transpose :

/* ... */ → /* ... */

Lexeme :

- Smallest unit of logic program
- Each lexeme should have unique id
- Each token represents unique lexeme.

Ex:-
```
main()
{
int xy;
xy=xy+10;
}
```

Lexeme → main | (|) | { | int | xy | = | xy | + | 10 | ; | }

Lexemes : 14

Tokens : 14

Tokens :

- Group of characters will be considered as a token (It follows **max munch rule**)

Types of Tokens

Valid strings in TOC = Tokens in lexical Analysis.

Identifiers
→ It contains letters, digit, _ but shouldn't start with digit

Keywords
int + float + char + short + long + auto + static + case + void + sizeof + break + continue

Operators:
operators are fns but fn need not be operator

+ - * / % = == <= >= != ++ -- ! & | && ||

Literals & constants
integer constant : d+
float : d+.d+
char : ' '
string : " "

Special symbols
; ( ) { }

*) The message inside the printf is not counted as token.

*) `int x=10; ch/*Comment*/ar ch='A';` → Considered as → `int x = 10; ch ar ch = 'A';` Tokens : 11

*) `int x = 10; ch /* comment */ ar ch;` Tokens : 14

*) 1x → No token
x1 → 1 token
x 1 → 2 token

② Syntax Analysis :

CFG : G = (V T P S)
A → α where A ∈ V, α ∈ (V ∪ T)*

Ex:- S → Ax, A → Bbc, C → d | ε

Symbol table :

→ It is an datastructure created and maintained by compilers
→ It stores info about various Entities like var name, fn name, classes, Intitalizes - etc
→ Info is collected at front End (Analysis phase) and used in back End (synthesis phase)

Phase	Usage
Lexical Analysis	create new Entries for Each new identifier
Syntax Analysis	Adds info regarding attr like type, scope dimension, line of reference, line of use - etc
Semantic Analysis	Uses the available info to check for semantics and is updated
ICG	Info in Symbol table helps to add temporary variables information
Code optimization	used in m/c dependent optimization by considering address and aliased var information
Code generator	Generates the Code by using address information of identifiers.

Operations on Symbol table :

① **Non Blocked Structure :**
→ Scope is throughout the pgm
→ Contains only one instance of var
operations : Insert, lookup

② **Block structured :**
→ Scope is within that block
operations : Insert, lookup, set, Reset

```
int i;
{
int j;
{
int i;
}
}
```
→ Block structured

Symbol table Entries:

- Each Entry in the symbol table is associated with attributes that support Compiler in different phases.

The Attributes are Name, Size, Dimension, Type, Line of declaration, Line of usage, Address

Ex:-

Name	Type	Size	Dimension	Line of dec-	Line of usage	Addr
sai	char	4	1	- - -	- -	- -
Age	int	2	0	- - -	- -	- -

*) All Attributes are not of fixed size

- If Chosen small → Can't store more var info → sol → size of symbol table should be dynamic
- If Chosen large → Memory wasted

Symbol table Implementation:

Implementation	Insertion	Lookup	Disadvantage
1. linear list			- lookup time is directly proportional to table size. - Every insertion preceded with lookup operation
a) ordered list → Array (Size should be static)	O(n)	O(logn)	
a) ordered list → linked list	O(n)	O(n)	
b) unordered list (Array, LL) → Place at End	O(1)	O(n)	
2. self organizing list (Most recently accessed data will be moved to head node)	O(1)	O(n)	- poor performance when less frequent items are searched
3. Search Tree	O(logn)	O(logn)	- we have to always keep it balanced
4. Hash table	O(1)	O(1)	- when there are too many collisions then the time comp increases to O(n)

(2) Syntax Analysis:

Context Free grammar:

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

$A \rightarrow \alpha$ where $A \in V$, $\alpha \in (V \cup T)^*$

Ex:- $S \rightarrow Aa$, $A \rightarrow Bb$, $B \rightarrow d \mid \epsilon$

Classification

- Based of num of derivations
  - Ambiguous grammar
  - Unambiguous grammar
- Based on num of strings
  - Recursive grammar
    - left recursive
    - right recursive
    - General recursive
  - Non-recursive grammar

) All top down parsers uses leftmost derivation

) All Bottom up parsers uses Rightmost derivation in reverse

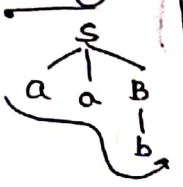
Ambiguous grammar:

An Ambiguous grammar is one that produces more than one left most or more than one right most derivation for some sentence.

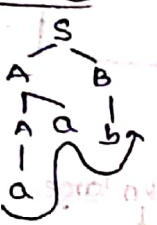
Ex:- $S \to AB \mid aaB$
$A \to a \mid Aa$
$B \to b$
I/P : aab

$\Rightarrow$

LMD-①



LMD-②



∴ Grammar is ambiguous.

Imp

*) I/p is not necessary to check ambiguity

*) Finding ambiguity of CFG is undecidable problem

*) Ambiguous productions are those which has more than one occurence of a given nonterminal on their RHS

$E \to (E)+(E) \mid E \to (E)*(E)$

Note

*) No Algo Exists to Convert an Ambiguous grammar into unambiguous

*) No Algo Exists to verify the ambiguity of CFG

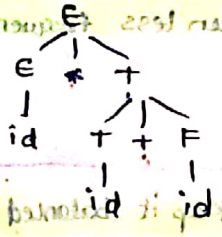
*) using trail and Error method we can Convert Ambiguous Gramar to unambiguous

∴ By redefing Grammar using precedence and Associativity, some of the ambiguous grammar can be Converted to unambiguous grammar

- Some of the ambiguous grammar which cannot be Converted into unambiguous such grammar is called Inherant ambiguous grammar

Converting AG to UAG using precedence and Associativity:

For Ex: id * id + id

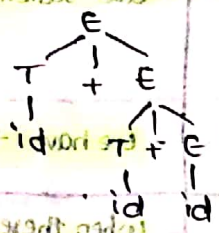


$+ > *$

(Bottom) (Top)

(High precedence) (Low precedence)

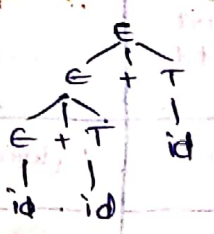
$E \to T + E$ (Right recursive)



id + id + id

'+' is right Associative.

$E \to E + T$ (left recursive)



id + id + id

+ is left Associative

① Convert AG to UAG

$E \to E+E \mid E-E \mid E*E \mid E \div E \mid E \uparrow E \mid id$

(Ass: $\underbrace{(+ = -)}_{\text{left}} < \underbrace{(* = \div)}_{\text{left}} < \underbrace{\uparrow}_{\text{right}} < id$

Write '+' wrt to E & F : $E \to E+F \mid F$

Write '-' wrt to E & F : $E \to E-F \mid F$

Write '*' wrt to F & G : $F \to F*G \mid G$

Write '÷' wrt to F & G : $F \to F \div G \mid G$

Write '↑' wrt to G & H : $G \to H \uparrow G \mid H$

Write id wrt to H : $H \to id$

$E \to E+F \mid E-F \mid F$
$F \to F*G \mid F \div G \mid G$
$G \to H \uparrow G \mid H$
$H \to id$

*) $E \to E*F \mid F$
$F \to F \uparrow T \mid T$
$T \to P+T \mid P$
$P \to id$

Precedence : $* < \uparrow < + < id$

Associativity : left right

Note :

If a grammar is both left and right associative then it is ambiguous grammar

Ex:- $S \to S+S \mid S*S \mid id$. $(+ = *)$ - precedence

Non-Recursive Grammar :

Ex:- $S \to Aa \mid Bb$
$A \to c \mid d$
$B \to e \mid \epsilon$

String : ca, cd, eb, b

Recursive grammar :

$S \to Sa \mid b$ → left recursive
$S \to aS \mid b$ → right recursive
$S \to aSa \mid b$ → General recursive
$S \to SaS \mid b$ → Ambiguos

Modifying the Grammar (Recursive) :

① left recursion to right recursion :

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

Ex:- $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

(i) $E \to \underset{\alpha}{E+T} \mid \underset{\beta}{T}$ → $E \to TE'$, $E' \to +TE' \mid \epsilon$

(ii) $T \to \underset{\alpha}{T*F} \mid \underset{\beta}{F}$ → $T \to FT'$, $T' \to *FT' \mid \epsilon$

G: $E \to TE'$, $E' \to +TE' \mid \epsilon$, $T \to FT'$, $T' \to FT' \mid \epsilon$, $F \to id$.

Note :

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

$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \alpha_n A' \mid \epsilon$

② Eliminating Indirect left recursion :

Even if there may be no immediate left recursion, some num of production rules may act together to give a left recursion

Ex:- $S \to Aa \mid b$, $A \to Ac \mid Sd \mid \epsilon$

$S \to Aa \to Sda$
$A \to Sd \to Aad$

$\Rightarrow$ $S \to Aa \mid b$, $A \to \underset{\alpha_1}{Ac} \mid \underset{\alpha_2}{Aad} \mid \underset{\beta_1}{bd} \mid \underset{\beta_2}{\epsilon}$ → Elimination → $S \to Aa \mid b$, $A \to bdA' \mid A'$, $A' \to cA' \mid adA' \mid \epsilon$

Left factoring :

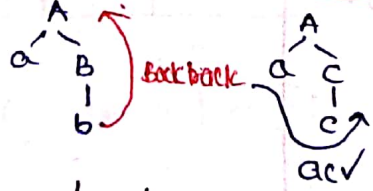
It is a process which isolates common parts of two productions into a single prod

ie $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \alpha\beta_n$ → left factoring → $A \to \alpha A'$, $A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

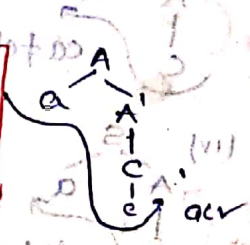
∵ Back-tracking is avoided.

Ex:- $A \to aB \mid ac$, $B \to b$, $c \to c$

I/p : ac



left factoring → $A \to aA'$, $A' \to B \mid c$, $B \to b$, $c \to c$

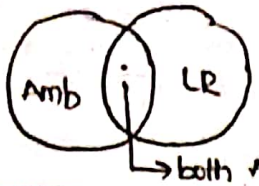


Q] left factoring of $A \to abA \mid abB \mid ac$

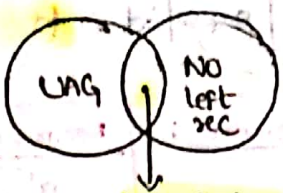
$A \to \underset{\alpha}{a}\underset{\beta_1}{bA} \mid \underset{\alpha}{a}\underset{\beta_2}{bB} \mid \underset{\beta_3}{ac}$ $\Rightarrow$ $A \to aA'$, $A' \to \underset{\alpha}{bA} \mid \underset{\beta_1}{bB} \mid c$ $\Rightarrow$ $A \to aA'$, $A' \to bA'' \mid c$, $A'' \to B \mid A$

Note:

Relation b/w Ambiguous & left rec grammar

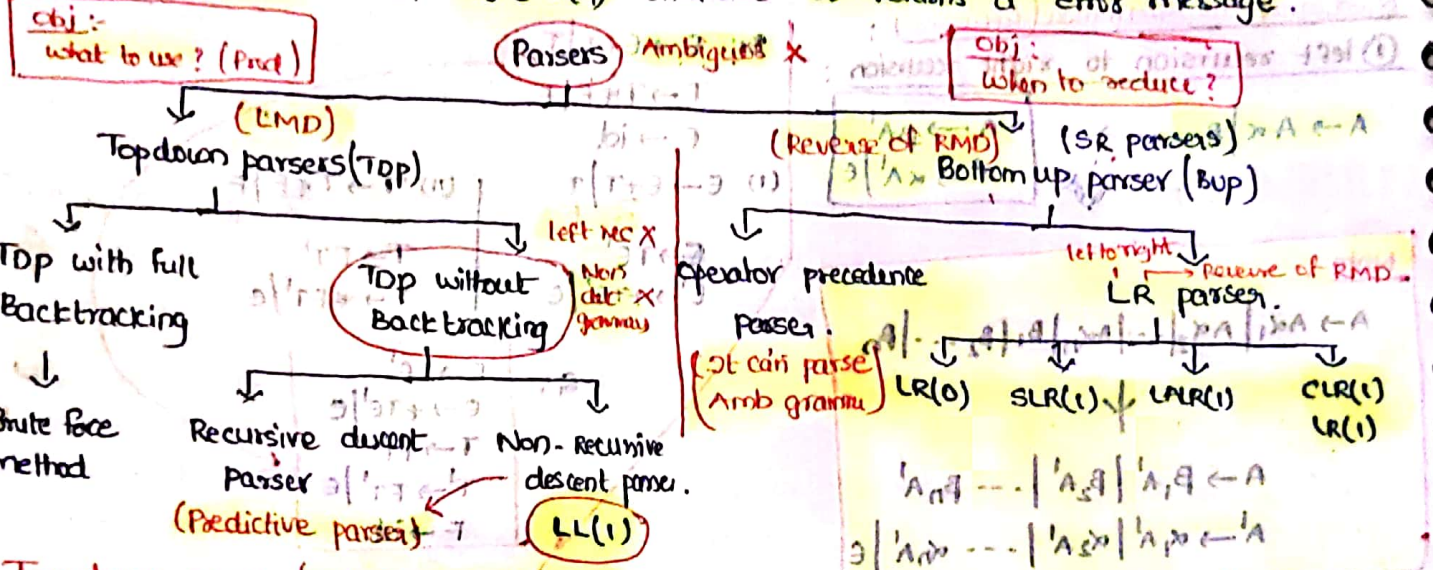


Relation b/w Unambiguous & no left rec



Parsing:

We define parser as a program for a CFG which accepts an input string w and constructs a parse tree of $w \in L(G)$ otherwise it returns a Error message.



Top down parser (with full Backtracking)

① Brute force method:

Ex: $S \to Aa \mid Bb$, $A \to c \mid d$, $B \to e \mid f$

Sol) (I) S

(II) S → A a

(III) S → A a, A → c, ca ≠ eb

(IV) S → A a, A → d, da ≠ eb

(V) S → A a (full Backtrack)

(VI) S → B b

(VII) S → B b, B → e, eb = eb ✓

Parse tree: S → B b, B → e

Note:

Main problem is Backtracking.

Due to Backtracking performance decreases so, out dated method.

Top down parser (without Back tracking)

② Recursive descent parser :

- Backtracking is avoided by realizing each non-terminal as one function call

Ex:- $S \to aA \mid bB$, $A \to c$, $B \to d$

I/p : bd

Note : ∵ Recursion May fall into Infinite loop

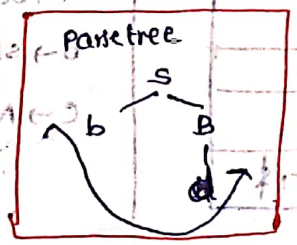
sol)

```
S()
{
  If(input[i]=='a')
  {
    i++;
    A();
  }
  If(input[i]=='b')
  {
    i++;
    B();
  }
  Else
    Error();
}
```

```
A()
{
  If(input[i]=='c')
    Accept();
  Else
    Error();
}
```

```
B()
{
  If(input[i]=='d')
    Accept();
  Else
    Error();
}
```

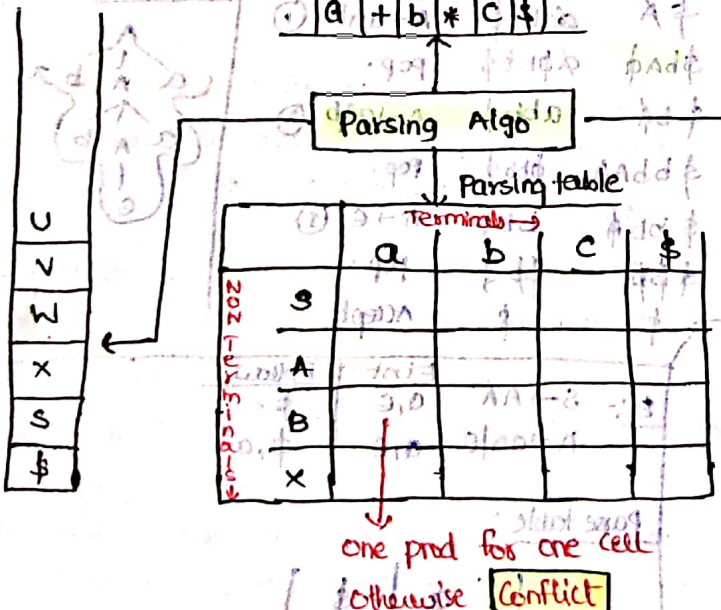
```
main()
{
  S();
}
```



Predictive parser : / LL(1) parser

- Tabular representation of recursive decento parser

Structure :-



Initial Configuration

stack	Input
$S	w $

stack	I/p	Actions
$	$	Accept
(Term) a	a	pop
(NT) X	a	pop X (Push RHS in Reverse)
	Otherwise	Error

First and Follow set :

- used for Construction of predective parse table

First set : First(α) be the set of terminals that begin the strings derivable from α

(i) $A \to a \Rightarrow First(A) = \{a\}$

(ii) $A \to \epsilon \Rightarrow First(A) = \{\epsilon\}$

(iii) $A \to X_1 X_2 \cdots X_n$

a) First(A) = First(X_1)

b) If First(X_1) contains ϵ then add First(X_2) to First(A)

c) If First(X_i) contains ϵ $\forall$ i = 1 to n then add ϵ to First(A)

Follow set: Follow(A) for non terminal A to be set of terminals that can appear immediately to the right of 'A' in some sentencial form

(i) Follow(S) = {$}

(ii) $A \to \alpha B \beta$

a) Follow(B) = First(β)

b) If First(β) contains ϵ then Follow(B) = Follow(α)

Follow set never contain 'ϵ'

Ex:- $E \to TE'$, $E' \to +TE' \mid \epsilon$, $T \to FT'$, $T' \to *FT' \mid \epsilon$, $F \to id$.

	First	Follow
E	id	$
E'	+, ε	$
T	id	$, +
T'	*, ε	$, +
F	id	*, +, $

	First	Follow
S → aA	a	$, b
A → BC | d	a, d, ε	a, $, b
B → sb | ε	a, ε	d, a, $, b
C → Aa | ε	d, ε, a	a, b, $

Predictive parser Construction:

Rule 1: If $A \to \alpha$ is a production then add $A \to \alpha$ to M[A, First(α)]

Rule 2: If First(α) contains 'ϵ' then add $A \to \alpha$ to M[A, Follow(A)]

Ex:- $A \to aAb \mid \epsilon$

	First	Follow
A	a, ε	b, $

Predictive parse table:

	a	b	$
A	A → aAb	A → ε	A → ε

Stack	I/p	Action
$A	abb$	A → aAb (1)
$bAa	abb$	pop.
$bA	bb$	A → aAb (2)
$bbAa	bb$	pop.
$bbA	b$	A → ε (3)
$bb	b$	pop.
$b	$	pop.
$	$	Accept

Parse tree:

Ex:- $S \to aA \mid AB$, $A \to b \mid \epsilon$, $B \to c \mid \epsilon$

	First	Follow
S	a, b, c, ε	$
A	b, ε	$, c
B	c, ε	$

Parse table:

	a	b	c	$
S	S → aA	S → AB	S → AB	S → AB
A		A → b	A → ε	A → ε
B			B → c	B → ε

Grammar is suitable

Ex:- $S \to AA$, $A \to aA \mid \epsilon$

	First	Follow
S	a, ε	$
A	a, ε	$, a

Parse table:

	a	$
S	S → AA	S → AA
A	A → aA, A → ε	A → ε

Conflict

Grammar is not suitable

Back.

Note:
In the predictive parse table multiple defined entries may possible. such grammars are not suitable for parsing

LL(1) Grammar :

A Grammar whose parsing table donot have multiple defined entries?

- LL(1)
  - L → Num of lookahead symbols
  - L → LMD
  - L → left to right scanning

Ex:- $S \to aA \mid bB$ → LL(1)
$S \to aaA \mid baB$ → LL(2)
$S \to abc\cdots kA \mid abc\cdots tA$ → LL(k)

Properties of LL(1) :

- Grammar should not be Ambiguous.
- Grammar should not be left recursive
- It should be left factored
- All Entries in the parsing table are unique for LL(1) Grammar
- A Grammar G is LL(1) iff the following conditions hold for distinct production $(A \to \alpha \mid \beta)$

(i) $First(\alpha) \cap First(\beta) = \phi$
(ii) If $First(\beta)$ contains ϵ then $First(\alpha) \cap Follow(A) = \phi$

Ex:- $E \to EaE \mid b$
Ambiguous
Not LL(1)

$A \to bAa \mid Ad \mid \epsilon$
left recursive
Not LL(1)

$S \to AA$
$A \to aA \mid \epsilon$
(i) $\{a\} \cap \{\epsilon\} = \phi$
(ii) $\{a\} \cap \{a, \$\} \neq \phi$
Not LL(1)

$S \to aA \mid AB$ ($a \cap b = \phi$, $a \cap \$ = \phi$ ✓)
$A \to b \mid \epsilon$ ($b \cap \epsilon = \phi$, $b \cap \$ = \phi$ ✓)
$B \to c \mid \epsilon$ ($c \cap \epsilon = \phi$, $c \cap \$ = \phi$ ✓)
LL(1)

Bottom up parsers :

- It constructs parse tree from leaf (input sentence) and reduce it towards the root
- It uses RMD in Reverse.

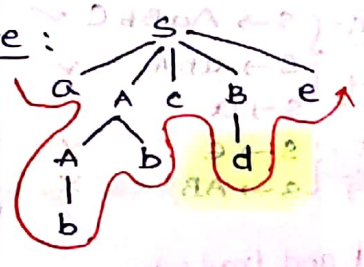
Ex:- $S \to aACBe$
$A \to Ab \mid b$
$B \to d$
I/p : abbcde

⇒

$abbcde$		$A \to b$
$aAbcde$		$A \to Ab$
$aAcde$		$B \to d$
$aACBe$		$S \to aACBe$
S		Accept

4 → d > b ①
3 → d = b ②
2 → d < b ③
1

Parse tree :



*) A Handle of a string is a substring that matches the right side of the production and whose reduction to the terminal on the left side of the production is said to be Handle pruning

- Each replacement of the RHS of the production by LHS is called Reduction process

Shift reduce parser :

- A convinient way to implement a shift reduce parser is to use a stack to hold the grammar symbols and an input buffer to hold the string to be parsed.

Initial Configuration

Stack	Input
$	w$

Actions
- shift – push i/p
- Reduce($A \to \alpha$) – pop α, push A
- Accept – If stack contains 'S' and Input contain '$'
- Error – unsuccesful parse

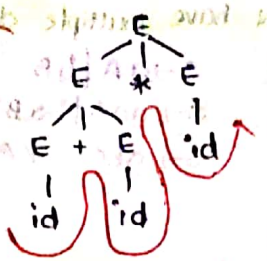
Note:
Shift reduce parser is also applicable to Ambiguous Grammar

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

I/P : id + id * id

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	E→id 5
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	E→id 4
$E+E	*id$	E→E+E 1
$E	*id$	shift
$E*	id$	shift
$E*id	$	E→id 2
$E*E	$	E→E*E 3
$E	$	Accept

Parse tree :



- id+id*id

Parsed Amb Grammar

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

Ans : * > + (left)

I/P : id+id*id

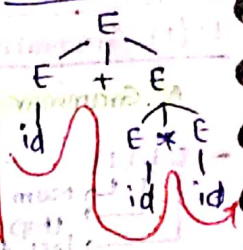
Precedence table :

①	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accept

Now, Input : id+id*id

②		Action
$	<id> + <id> * <id> $	E→id
$	E + <id> * <id> $	E→id
$	E + E * <id> $	E→id
$	E + E * E $	E→E*E
$	E + E $	E→E+E
$	E $	Accept

③ Parse tree



Parsed Amb Grammar

Operator precedence parser :

- Efficient shift reduce parser
- It delimits the handles in a right sentencial form by giving precedence relation b/w certain pair of terminals
- It accepts only a specific class of grammars called operator grammar which is a ε free grammar and no two non terminals are adjacent (unit prod Allowed)

Ex:-
- OG: $S \to AaBbC$ ✓, $S \to abA$ ✓, $S \to a$ ✓
- Not OG: $S \to \epsilon$ ✗, $S \to AB$ ✗

Operator precedence grammar :

- It is an operator grammar in which three disjoint precedence relations are present :
  1. $a < b$ - a gives precedence to b
  2. $a = b$ - a has same precedence as b
  3. $a > b$ - a has precedence over b

- helps to delimit the handles

Lead and trail set :

- Computation of precedence is aided by two functions called lead and trail set

Lead set :
- If $A \to \alpha a \beta$ then Lead(A) = {a} iff α is ε or single non terminal
- If $A \to B\alpha$ is a production then Lead(A) = Lead(B)

Trail set :
- If $A \to \alpha a \beta$ is a prod then trail(A) = {a} iff β is ε or single non terminal
- If $A \to \alpha B$ is a production then trail(A) = trail(B)

Ex:- $S \to aAbB$, $A \to Bc$, $B \to dC$, $C \to e$

	lead	trail
S	a	b, d, e
A	c, d	c
B	d	d, e
C	e	e

Ex:- $S \to AaB$, $A \to bS$, $B \to Ad$

	lead	trail
S	a, b	a, d
A	b	b, a, d
B	d, b	d

Computation of precedence relations using lead and trail set :

- Consider a production is of the form

$A \to X_1 X_2 \cdots X_n$

① If X_i and X_{i+1} are terminals then $X_i = X_{i+1}$

② If X_i and X_{i+2} are terminals and X_{i+1} is non terminal then $X_i = X_{i+2}$

③ If X_i is terminal and X_{i+1} Non terminal then $X_i < lead(X_{i+1})$

④ If X_i is nonterminal and X_{i+1} is terminal then $Trail(X_i) > X_{i+1}$

⑤ $\$ < lead(S)$

⑥ $Trail(S) > \$$

Ex:- Construct operator precedence parse table for $S \to aAd$, $A \to bc$

	lead	trail
$S \to aAd$	a	d
$A \to bc$	b	c

=	>	<
$b = c$, $a \doteq d$	$Trail(A) > d$, $c > d$	$a < lead(A)$, $a < b$

$\$ < a$, $d > \$$

Parse table :

	a	b	c	d	$
a		<		=	
b			=		
c			.	>	
d				>	
$	<				Accept

delimited the handles

Operator precedence parsing Algorithm :

- Consider If 'a' is on the top of the stack and 'b' is a symbol pointed by i/p pointer.

① If $a = b = \$$ then
- parser announces succesfull completion

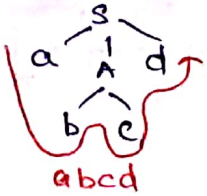
② If $a < b$ or $a = b$ then
- push b on to the stack
- Advance i/p pointer to next symbol

③ If $a > b$ then
- Pop the stack repeatedly until the top of the stack is related by less than or equal to to the terminal most recently popped

Ex:- Construct parse tree for i/p "abcd" using operator precedence parse table above.

Stack	Input	Action
$	abcd$	shift
$a	bcd$	shift
$ab	cd$	shift
$abc	d$	$A \to bc$ (2)
$aA	d$	shift
$aAd	$	$S \to aAd$ (1)
$S	$	Accept

Parse tree :



Q) Consider the grammar

$E \to E*F$
$E \to F+E$
$E \to F$
$E \to F-E$
$E \to id$

wof is true

a) $* > +$, $Trail(F) > +$, $id > +$

b) $\to *$, $Trail(E) > *$, $- > *$

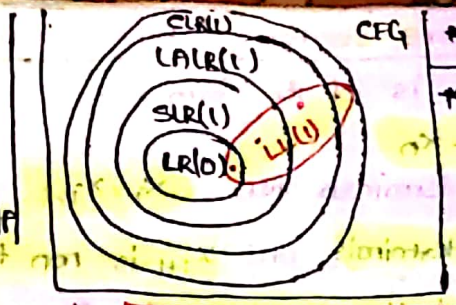
c) $+ \doteq -$, Not possible

d) $* < -$, $* < lead(F)$, $* < id$

A	lead	trail
$E \to E*F$ | $E+E$ | F | $F-E$	$*, +, -$, id	$*, +, -$, id
$F \to id$	id	id

LR(k) parser.

LR(k)
- → Num of look ahead symbols
- → RMD in reverse
- → left to right scanning of i/p

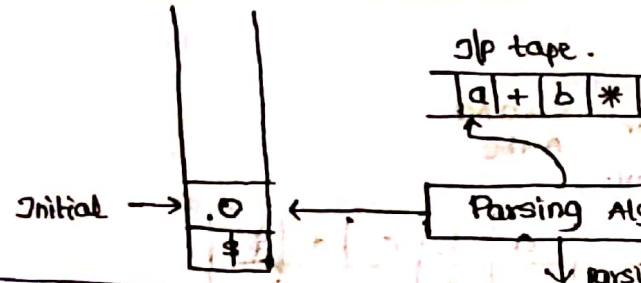


CFG
*) No Amb Grammar in LR(k)
*) Every LL(1) Grammar is CLR(1) but Every CLR(1) need not be LL(1).

ie $LR(0) \subseteq SLR(1) \subseteq LALR(1) \subseteq CLR(1)$

LR(k) structure :

- Common for all LR(0), SLR(1), LALR(1) & CLR(1)



Initial configuration

stack	Input
$0	w$

4 Actions
1) Shift J - Push i/p & state J.
2) Reduce A → α
3) Accept - Succesful parse
4) Error - unsuccesful parse

Imp
(i) pop 2*|α| symbols
(ii) push A
(iii) push [Goto(current state, A)]

parsing table

	Action							
	a	+	b	*	c	$	S	A
0								
1								
2								
3								
4								

Imp *)
LR(k)
- LR(0) / SLR(1) - Uses LR(0) Items
- LALR(1) / CLR(1) - Uses LR(1) Items

LR(0) Items :

An LR(0) Item of a grammar G to be a production of G with a dot at some position on the RHS.

Ex:- A → xyz

4 Items : A → .xyz
A → x.yz
A → xy.z
A → xyz.

Note :
A → ∈ Generates only one item ie A → .

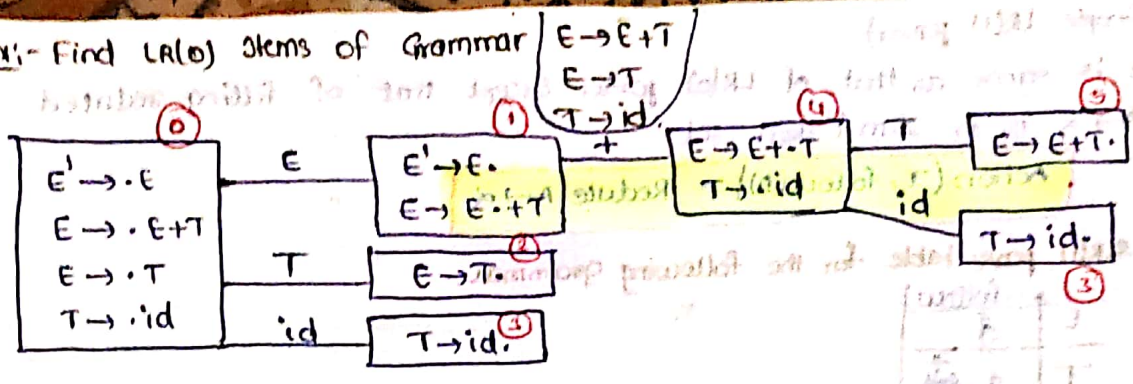
Procedure to Find LR(0) Items :

Step ① : Augment the grammar with production ie S' → S where S is start
S' → .S → start
S' → S. → End.

Step ② : find the closure of i/p (closure(I))
If A → α.Bβ is a production in item 'i'
B → γ is a production then add B → .γ to Item i If it is not there

Step ③ : find Goto(I,X)
If A → α.Xβ is in Item i, we define Goto(I,X) is the closure set of all items.
Contains A → αX.β

Ex:- Find LR(0) items of Grammar $E \to E+T$, $E \to T$, $T \to id$



Construction of LR(0) parse table :

- If $S' \to S.$ is in item i then set Action (I, $) = Accept
- If $A \to \alpha.a\beta$ is a production then set Action (I, a) = shift J
- If $A \to \alpha.B\beta$ is in item i then set Goto (I, B) = J
- If $A \to \alpha.$ is in item i then set Action (I, All terminals) = reduce $A \to \alpha$.

Construct parse table for above Grammar :

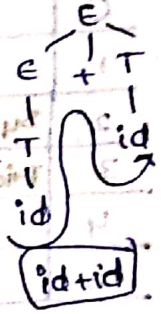
	Action			Goto	
	id	+	$	E	T
0	S_3		-	1	2
1		S_4	Accept		
2	Reduce E→T				
3	Reduce T→id				
4	S_3				5
5	Reduce E→E+T				

Error Fields

Construct parse tree using LR(0) parse table

I/p : id+id

Stack	Input	Action
$0	id+id$	S3
$0 id 3	+id$	T→id ④
$0 T 2	+id$	E→T ③
$0 E 1	+id$	S4
$0 E 1 + 4	id$	S3
$0 E 1 + 4 id 3	$	T→id ②
$0 E 1 + 4 T 5	$	E→E+T ①
$0 E 1	$	Accept ✓



Parser Conflicts in LR(0) :

① Shift reduce Conflict :

- It occurs in an LR(0) parser state if the LR(0) item set contains items of the form

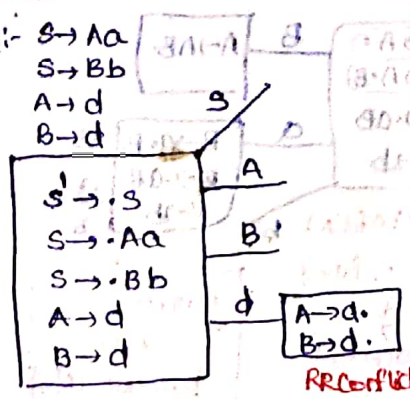
$A \to \alpha.a\beta$	shift
$B \to \alpha.$	reduce

② Reduce Reduce Conflict :

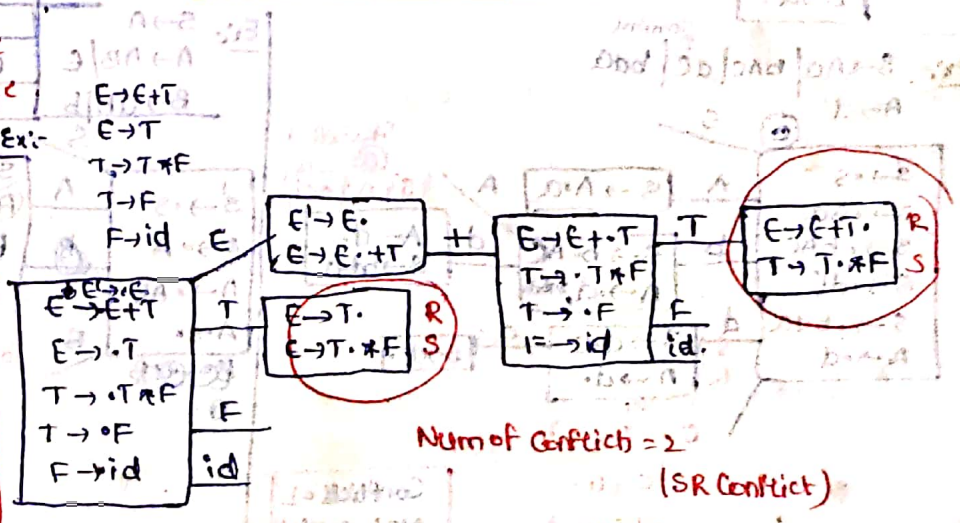
- It occurs in LR(0) parser state if the LR(0) items set contains items of the form

$A \to \alpha.$	reduce
$B \to \alpha.$	reduce

Ex:- $S \to Aa$, $S \to Bb$, $A \to d$, $B \to d$



Ex:- $E \to E+T$, $E \to T$, $T \to T*F$, $T \to F$, $F \to id$



SLR(1) Parser : (Simple LR(1) parser)

Constructing SLR(1) is same as that of LR(0) parser Except that of filling reduced Entries ie of $A \to \alpha$ is in Item i then set

Action (I, follow(A)) = Reduce $A \to \alpha$

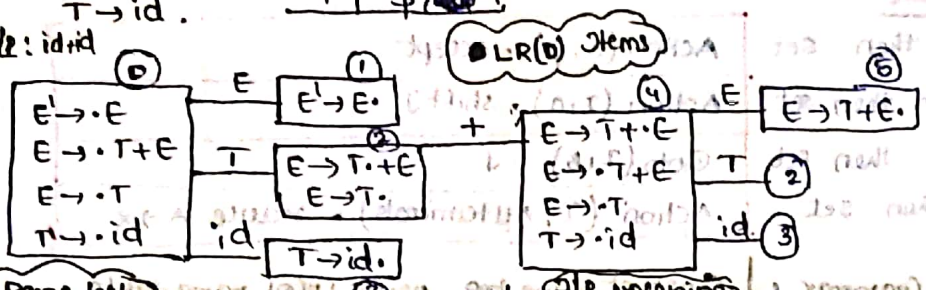
Ex:- Construct an SLR(1) parse table for the following grammar

$E \to T + E$
$E \to T$
$T \to id$.

	follow
E	$
T	$, +

Sol: id+id

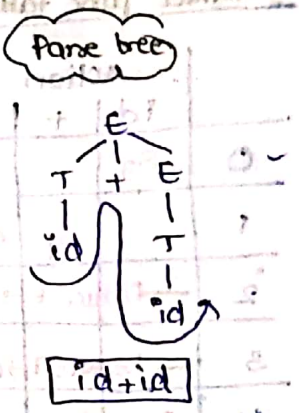
LR(0) Items



Parse table

	Action			Goto	
	+	id	$	E	T
0		S3		1	2
1			Accept		
2	S4		E→T		
3	T→id		T→id		
4		S3		5	2
5			E→T+E		

Stack	Input	Action
$0	id+id$	S3
$0 id 3	+id$	T→id
$0 T 2	+id$	S4
$0 T 2 + 4	id$	S3
$0 T 2 + 4 id 3	$	T→id
$0 T 2 + 4 T 2	$	E→T
$0 T 2 + 4 E 5	$	E→T+E
$0 E 1	$	Accept



Parser Conflicts in SLR(1):

① shift reduce conflict:

It occurs in SLR(1) parser state of the LR(0) item set contains items of the form

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

ie $\{a\} \cap Follow(B) \neq \phi$

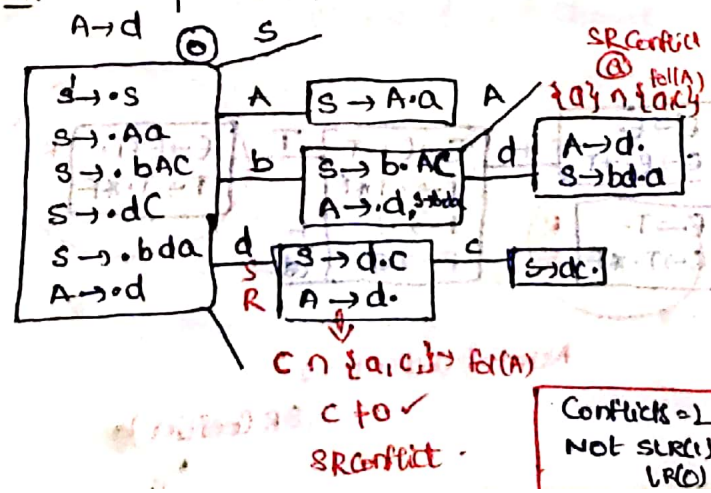
② Reduce Reduce Conflict:

It occurs in SLR(1) parser state of the LR(0) item set contains items of the form:

$A \to \alpha \cdot$
$B \to \alpha \cdot$

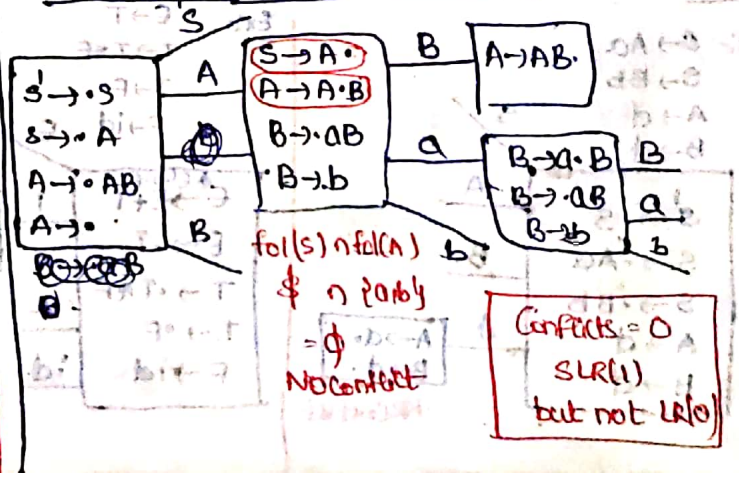
ie $Follow(A) \cap Follow(B) \neq \phi$

Ex:- $S \to Aa \mid bAc \mid dc \mid bda$ (Terminal)
$A \to d$



Ex:-

	follow
S→A	$
A→AB/ϵ	a, b
B→aB/b	a, b



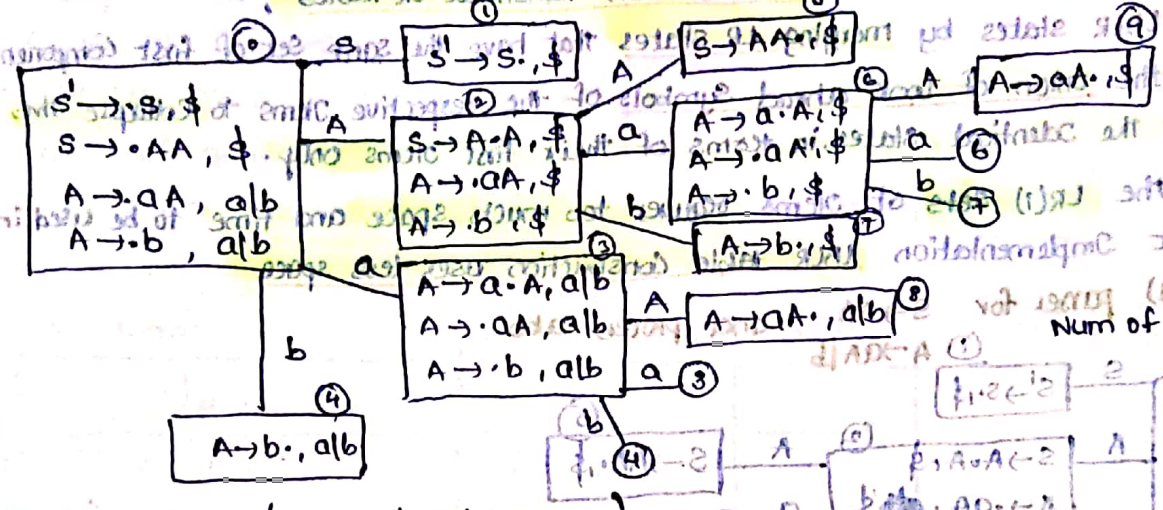
LR(1) Items :

- CLR(1) and LALR(1) are based on LR(1) item
- An LR(1) item is of the form $A \to \alpha \cdot \beta, a$ where
  - $A \to \alpha \cdot \beta$ → LR(0) item
  - a → lookahead symbol

LR(1) item = LR(0) item + lookahead symbol

Note: If $A \to \alpha \cdot B\beta, a$ is in I, item i and $B \to \gamma$ is a prod then add $B \to \cdot\gamma, First(\beta a)$ to item i if it is not already there.

Ex:- Find LR(1) items of $S \to AA$ and construct CLR(1) parse table and process abb



Num of LR(1) items = 10

CLR(1) parser :- (Canonical LR(1) parser)

- Same as that of LR(1) parser except that of filling reduced entries.

If $A \to \alpha \cdot, a$ is in item i then set Action (I_i, a) = reduce $A \to \alpha$

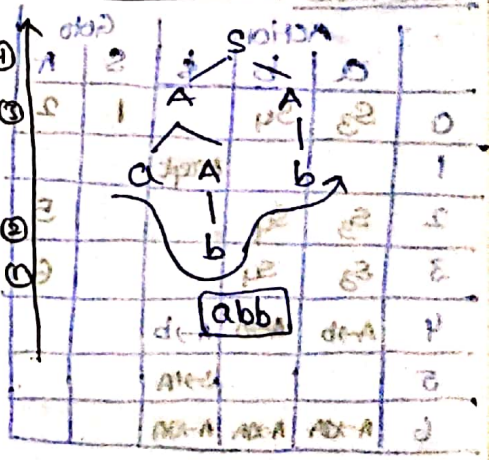
Parse table :

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	A→b	A→b			
5			S→AA		
6	S_6	S_7			9
7			A→b		
8	A→aA	A→aA			
9			A→aA		

Input processing :

Stack	Input	Action
$0	abb$	S_3
$0a3	bb$	S_4
$0a3b4	b$	A→b
$0a3A8	b$	A→aA
$0A2	b$	S_7
$0A2b7	$	A→b
$0A2A6	$	S→AA
$0S1	$	Accept

Parse tree :



Parser Conflicts in CLR(1) :

(1) shift reduce Conflict :

It occurs in CLR(1) parser state if LR(1) item set contains items of the form

$A \to \alpha \cdot a\beta$
$A \to \alpha \cdot, a$

Reduce - Reduce Conflict :

It occurs in CLR(1) parser state if the LR(1) item rep is of the form

$A \to \alpha \cdot , a$
$B \to \alpha \cdot , a$

Note:
Every LL(1) is LR(1) but Every LR(1) need not to be LL(1)

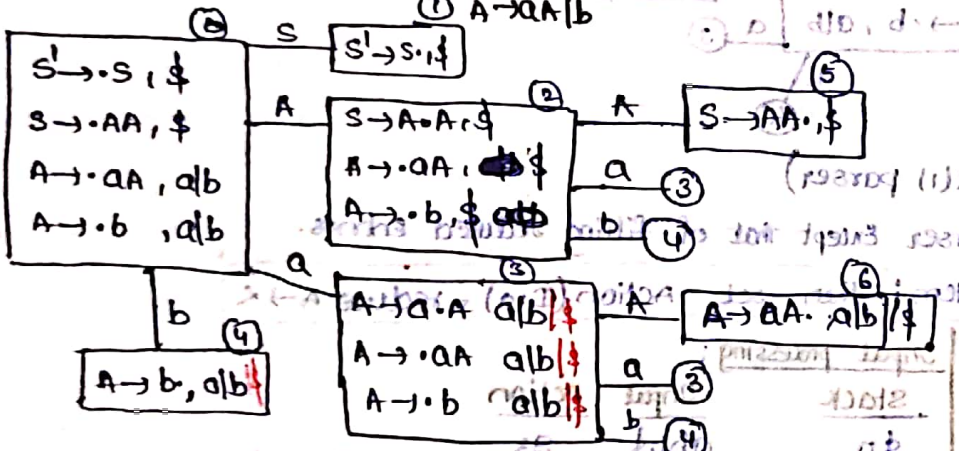
LALR(1) parser : (look ahead LR(1) parser)

- The table update by it considered smaller than canonical LR tables.
- We can find LALR states by merging LR states that have the same set of first components.
- We can take the union of look ahead symbols of the respective items to collapse the overhead of the identical states in terms of their first items only.

*) Constructing the LR(1) sets of items requires too much space and time to be used in practice but implementation LALR table construction uses less space

Construct LALR(1) parser for $S \to AA$ and process abb

(1) $A \to aA | b$



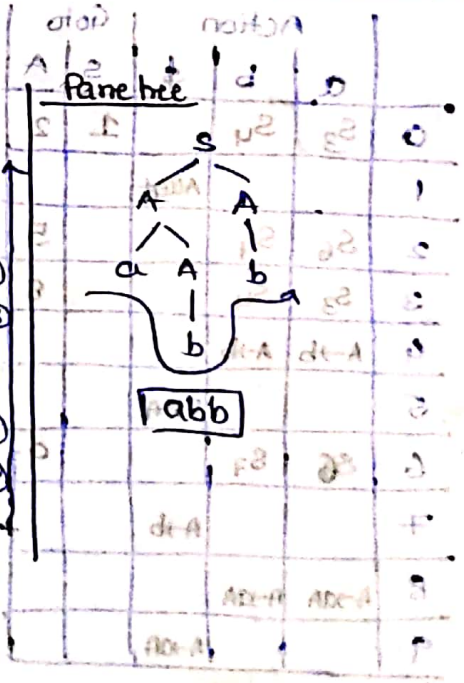
Parse table :

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	A→b	A→b	A→b		
5			S→AA		
6	A→aA	A→aA	A→aA		

Input processing :

Stack	Input	Action
$0	abb$	S_3
$0a3	bb$	S_4
$0a3b4	b$	A→b (4)
$0a3A5	b$	A→aA (3)
$0A2	b$	S_4
$0A2b4	$	A→b (2)
$0A2A5	$	S→AA (1)
$0S1	$	Accept

Parse tree



Parser Conflicts in LALR(1) :

- shift reduce
- reduce reduce

⇒ may obtain after merging

Note :
If there is no SR conflict in CLR(1) parser state then it will never be reflected in the LALR(1) states obtained by combining / merging LR(1) states

Note:
The reduction process to LALR(1) states may introduce RR conflicts in the parser. so, that grammar is said to be not LALR(1) grammar.

Note:
If there are n_1, n_2, n_3 states in SLR(1), LALR(1) CLR(1) parser respectively then

$n_1 = n_2 \leq n_3$ ie SLR(1) = LALR(1) ≤ CLR(1)

Note:
- Among All the LR parsers, CLR(1) is more powerful because it accepts more Class of grammar
- LALR(1) parser is more Efficient as it takes less space and accept more Class of Grammar than LR(0), SLR(1) but not CLR(1)
- The parser generator tool (yacc) uses LALR(1) parsing Algorithm

Points to remember:

Grammar	LL(1)	LR(0)	SLR(1)	CLR(1)	LALR(1)
① S → a | b	✓	✓	✓	✓	✓
② E → E+T | T T → id	✗	✓	✓	✓	✓
③ E → T+E | T T → id	✗	✗	✗	✓	✓
④ S → Aa | bAc | dc | bda A → d	✗	✗	✗	✓	✓
⑤ S → Aa | bAc | Bc | bBa A → d B → d	✗	✗	✗	✗	✓
⑥ S → A | a A → a	✗	✗	✗	✗	✗

Syntax directed translation: (SDT)

CFG + Semantic rules
Translation Rules
Actions
⇒ SDT

- More powerful than CFG
- less powerful than CSG

Attributes

Inherited Attribute
- Attribute at any node which will be computed by depending on parents & siblings
- Ex:- S → ∝Aβ
  {A(x) = f(S|∝|β)}
  Translation
  Inherited Attr

Synthesized Attribute (Bottom up)
- Attribute at any node which will be computed by depending on children
- Ex:- S → (S)
  {S(c) = S1.c + 1}
  S → SS
  {S.c = S1.c + S2.c}
  Synthesized Attr

$E \to E+T \quad \{T.x = E_1.x + E.x\}$ → Inherited

$T \to T*F \quad \{T_1.x = F_1.x\}$ → Inherited

$E \to a \quad \{E.x = a.c\}$

$T \to b \quad \{T.x = b.c\}$ → synthesized

$F \to c \quad \{F.x = c.c\}$

'E' is neither inherited nor synthesized

$S \to SS \quad \{S.x = 100\}$

$S \to \varepsilon \quad \{\}$

Attr not depending on any one

Both inherited & synthesized

$S \to SS \quad \{S_1.x = S_2.x\}$ → Inherited

$S \to \varepsilon \quad \{\}$ → Inherited & syn

∴ x is inherited

Ex:- $S \to SS \quad \{S_1.x = S_2.x + 1, \; S.y = S_2.y\}$ (1) (5)

$S \to a \quad \{S.y = a.c\}$ (3)

$S \to Sb \quad \{S_1.x = S.x, \; S.y = S.x\}$ (2) (4)

$S \to \varepsilon \quad \{S.y = 0\}$ (1) (5)

'x' is synthesized & inheritance is minimized

Syntax directed definition :

L-Attribute grammar

- Attribute value is computed by depending on parent or left sibling or children
- Translation can be placed anywhere in RHS of production
- Evaluation depends on topological order

S-Attribute Grammar

- Attribute value is computed by depending on children
- Translation can be placed at the end of production
- Evaluation depends on Bottom up parsing ie (Reverse of RMD)

Note :
- Every S-Attributed grammar is L-Attributed grammar

L-Attr / S-Attr

Evaluation of SDT :

Attribute present

L-Attributed

Topological ordering (Inorder)

Evaluate

$E \to E \; \{Print('+')\} \; +T$ (1)

$E \to T \; \{\}$ (2)

$T \to T \; \{Print('*')\} \; *F$ (3)

$T \to F \; \{\}$ (4)

$F \to id \; \{print\; id\}$ (5)

L-Attr Approach

Top down: (LMD)

S-Attributed

Topological order.
Bottom up parsing
(Reverse of RMD)

No Attribute

Top down parsing

LMD Evaluation
(Prefix evaluation)

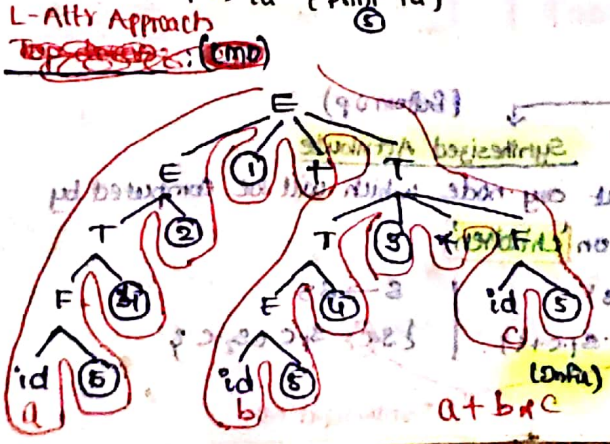
Bottom up parsing

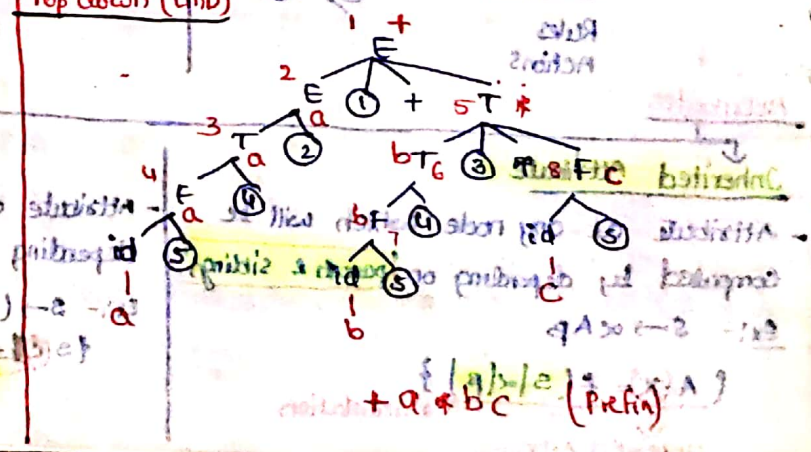
Reverse of RMD Evaluation

Ex: a+b*c

- Top down
- Bottom up / S-Attr.
- L-Attr

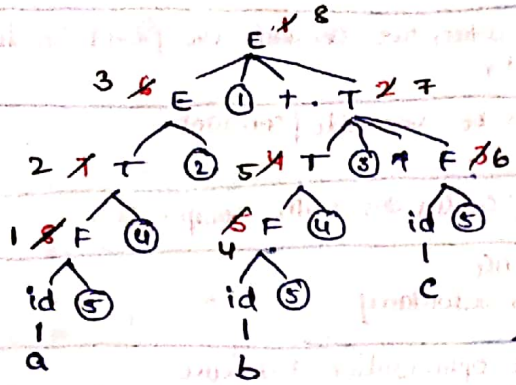
Top down (LMD)





Bottom up approach / SAttr approach :

- Reverse of RMD



A → RMD
B → Reverse of RMD

o/p : a b c * +

Applications of SDT

- perform semantic analysis
- Generate parse tree.
- To Evaluate an Expression
- Convert decimal to binary
- To generate Intermediate Code

Imp – Used to create any datastructure
Imp Conversion from Infix to prefix
prefix to postfix
postfix to infix.

Intermediate Code generation :

- It is standard form that can be easily Convertible to Assembly Code.
- M/c Independent Code
- Implemented using SDT

Why ICG ?
→ Compiler depends on Architecture
→ o/p of the Compiler is Assembly Code. so, it requires knowledge of registers, address, opcodes & Addressing modes.

Representation

Linear form

Postfix
- a = b * c
- a b c * =

Three Address Code
- Every stmt has max 3 address.
- Ex:- x = y + 2 * 3 + a →

```
t1 = 2 * 3
t2 = y + t1
t3 = t2 + a
x  = t3
```

Triple Notation

	ope rator	ope rand 1	ope rand 2
(1000)	*	2	3
(1001)	+	y	(1000)
(1002)	+	(1001)	a
x	=		(1002)

Quadruple Notation

	ope rator	ope rand 1	ope rand 2	result
(1000)	*	2	3	t_1
(1001)	+	y	t_1	t_2
(1002)	+	t_2	a	t_3
x	=	t_3		x

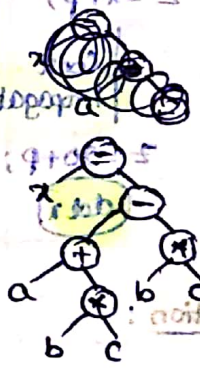
Indirect
- Indirect Addressing Of Triple Notation

1000	[1000]
1001	[1001]
1002	[1002]

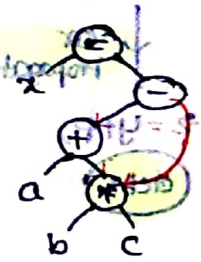
Non-Linear form

x = a + (b * c) – (b * c)

Syntax tree



Directed Acyclic graph



→ Common sub expr can be Eliminated using DAG
↓ optimization
redundancy ↓

Code optimization (CO)

M/c Independent CO
- High level Code
- Intermediate Code (3-Addr)

M/c dependent CO
→ Assembly code

Why Code optimization ?
→ may reduce time
space

Optimization Techniques :

Technique	Type: Independent	Type: dependent	Remarks
1. Constant folding	✓	×	- Applied when two constants are present in the an operator
2. Copy propagation	✓	✓	- It can be variable / constant
3. Strength reduction	✓	✓	- Replacing costlier ins with cheaper ins
4. Common sub expr Elimination	✓	×	- Uses DAG - Reduces redundancy
5. Dead Code Elimination	✓	✓	- Hybrid optimization technique - unreachable code deletion
6. loop jamming / merging / Combining	✓		- reduces total num of Executions by merging
7. loop unrolling	✓	✓	- Loop unrolling is not possible when loop var is present in the loop body - Applicable when loop iterates Even num times
8. Code motion	✓	✓	- moving the code to reduce the num of instructions

Constant folding :

x = 10 * 2 + y → optimize → x = 20 + y

MOV X, 10
MUL Y, 2
ADD X, Y
3 Ins → 2 Ins
ADD Y, 20
MOV X, Y

A[10 * 2] = 5 → optimize → A[20] = 5

x = y + 1 * 2
x = 10 * y + d
} Can't be Applied

Copy propagation

x = y;
z = x + P;
→ var propagation → z = y + P;

x = 20
z = x + P;
→ Const Propagation → z = 20 + P;

Strength reduction

x = x * 2 → Costlier
x = x << 1 / x = x + x → Cheaper

Common Sub Expr Elimination

x = a+b * a+b + a+b * b
x = a+b * a+b + 3

Dead Code Elimination :

```
main()
{
  int x = 10;
  if(x)
    pf("Gate2019");
  else
    pf("Cat2019");
}
```

x deleted → Const Copy propagation →

```
main()
{
  if(10)
    pf("Gate2019");
  else
    pf("Cat2019");
} Dead code (unreachable)
```

Dead code Elimination →

```
main()
{
  pf("Gate2019");
}
```

loop Jamming / Merging / Combining

```
for(i=0; i<9; i++)
  x = x+i;
for(j=0; j<9; j++)
  y = y+2i;
```

Init Comp Term
1 + 27 + 1 ⇒ 29 Execution
1 + 27 + 1 ⇒ 29
58 Exe

→ loop merging →

optimized code

```
for(i=0; i<9; i++)
{
  x = x+i;
  y = y+2i;
}
```

1 + 36 + 1 ⇒ 38 Exe

loop unrolling :

```
for(i=1; i<=10; i++)
    pf("gate");
```

1+30+1 → (31) Exe

loop unrolling →

```
for (i=1; i<=5; i++)
{
   pf ("gate");
   pf ("gate");
}
```

1+20+1 → (22) Exe.

```
for(i=1; i<=10; i++)
{
   x=x+i;
}
```

→ No loop rolling bcoz loop var is present in loop body

Code Motion :

```
for(i=0; i<q; i++)
{
   x=x+i;
   y=100;
}
```

→ 1+36+1 → (38) Exe

Code motion →

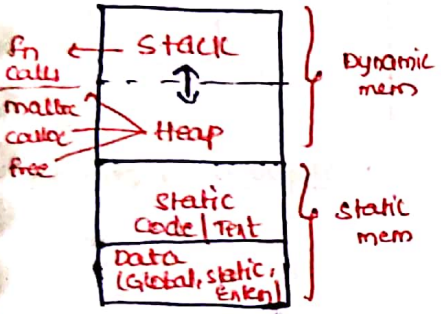
```
for(i=0; i<q; i++)
   x=x+i;
y=100;
```

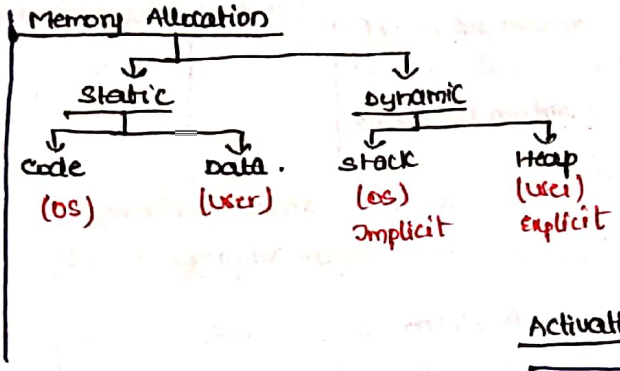
1+27+1 → 29 Exe

, 1 En

(30) Exe.

Runtime Environment :





Activation record :

→ Every function (call) requires one Activation record
→ It is a datastructure
→ It contains info about the function

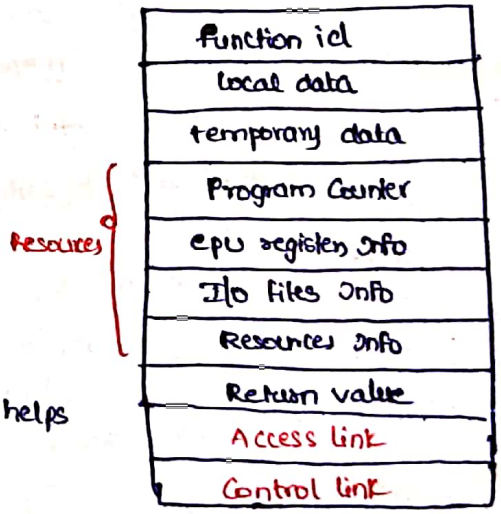
Access link :

- Used to Access non local data

Control link

- It links to prev fn (caller)
- It is used to understand calling sequence which helps to return value from Current fn (Callee) to prev fn (Caller)

Activation record for function :



Ada, Haskell
Python, Fortran
C, C++, Java – static scope
Lisp – dynamic scope
Fortran – Type checking

x—The End—x