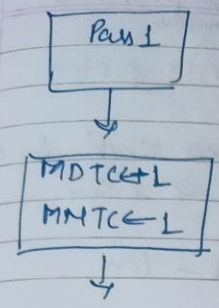
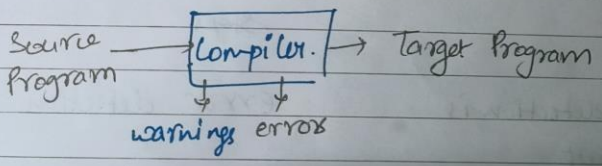


Compilers:

It is a software / program that converts a program written in HLL (source language) to a low level language (object/ target language)

① It also reports errors present in source programs



Types of compilers

Simple Pass compilers → Is a type of compiler that process the source code only once.

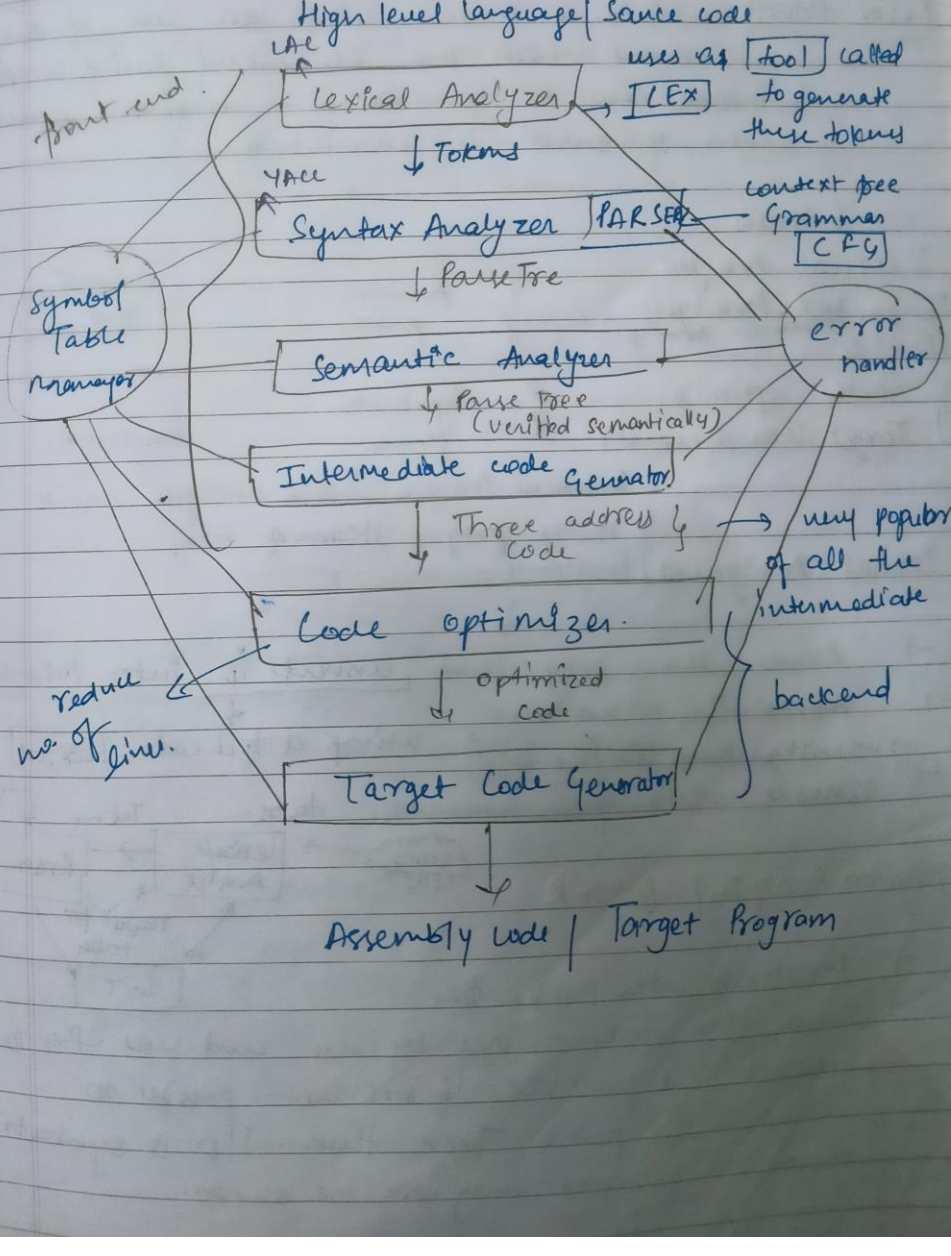
Multi Pass Compiler- It is a type of compiler that processes the source code multiple times (to convert HLL → low level lang) i.e to convert source code to target / object code.

Interpreter - Is also a prog. that scans the pure high level code line by line and convert it to executable code line by line.

Interpreted prog. is usually slower than compiled ones
eg → Perl, Python, Matlab

	COMPILER	Interpreter
1)	Takes entire prog. at once as input	It takes single line of code at a time.
2)	Speed - high	Speed - low
3)	generates intermediate object code. So, memory requirement is more	Memory requirement is less b/c no intermediate code created.
4)	error	
	all the errors are displayed together	Continues translating the prog. until the 1st error is met, in which case it stops.
5)	error detection is difficult	error detection is easy
6)	Compilers are larger in size	Smaller in size
7)	C, C++, Scala,	7) Perl, Python, Ruby

Phases of a Compiler.



Symbol table → Data structure being used by the compiler to store all the information related to identifiers
eg → its types, scope, size, location

→ all phases are interact with the symbol table manager (SMB)

Error Handler → It is a module which take care of the event which are encountered during compilation
↳ it takes care to continue the compilation process even if error is encountered.

→ 6 phases

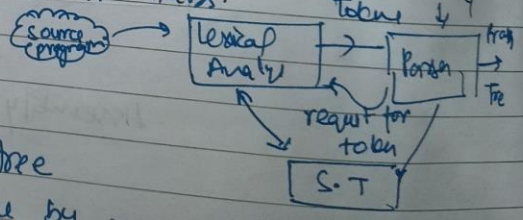
1) Lexical Analysis } Analysis phase — Front end
2) Syntax Analysis (semantic analy..)
3) ICG
4) Code optimizer } Synthesis phase — Backend.
5) Target Code Generator

Lexical Analyzer → (In lexical Analysis, lexical analyzer converts source prog → steam of tokens function →)

↓

→ Reads the prog and converts it into tokens
→ suppose if we pass a=b+c;
→ we will get token like [id=id+id] using a fod call [LEX tool]
→ remove white spaces, comments, tabs.

2) Syntax Analyzer / PARSER



→ constructs the parse tree
→ Takes the tokens one by one and use cfg to construct the parse tree. If it is not possible to construct the parse tree then. i/p is syntatically incorrect & error msg will be shown.

Tokens → A token is a sequence of characters that can be treated as unit / single logical entity.
keyword int a = 15 ; , typical tokens are keywords, identifiers, operators (for, if, while) / variables / name + - *

3) Semantic Analyzer

→ Verifies the parse tree, whether its meaningful or not.
→ It uses the Parse tree & info in the symbol table to check the source program for semantic consistency with the language definition.

4) Intermediate code Generation.

Generates the intermediate code (we have many popular code)
eg Three address code.

5) Code Optimizer.

removing unnecessary code lines.

→ We get optimized code (ie our no. of lines are reduced
→ C.O phase attempts to improve the intermediate code so that it run faster & consumes less resource

6) Target Code Generator.

→ final phase of the compiler which gives the target code / assembly code.

Symbol table

Lexical Analyzer generator

↑

Case Study : LEX and YACC

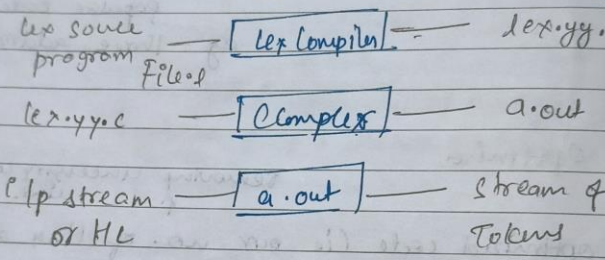
Lex is a tool / computer program which generates Lexical Analyzer. It is used with YACC parser / generator

↓

(LEX takes regular expression as an specification)

1st of phase of compiler which converts source code in HLL → stream of tokens

Lex Compiler : { It will transform the i/p pattern into a transition diagram and generate code in a file called lex.yy.c



1. Source code is in lex language with filename.l extension.
2. It is given to lex compiler when is the lex tool, and produces lex.yy.c → C program as output
3. C compiler runs this code ie lex.yy.c program and produces an output [a.out]

↓

Lexical Analyzer

4. [a.out] transforms an i/p stream into a sequence stream of Tokens

%{ #include<stdio.h>
int c=;
%}
%%
pattern {action}
%%
main()
{ }

eg1 %{
#include <stdio.h>
%}
%%
"hi" {printf("By");}
action (will be in C lang. syntax)
.* {printf("Wrong");}
%%
main()
{ printf("enter i/p");
yylex(); // Take ip & generate the tokens a/c to pattern rule in section 2
}
int yywrap() // to identify end of ip
{ return 1; }

Parser generator

↑ → It generates a look LALR parser.

YACC : (Yet - Another - Compiler - Compiler)

- It stands for Yet Another Compiler - Compiler.
- It is a tool for generating look a head left - to - Right (LALR(1)) parser.
- It takes I/p from the lexical Analyzer & generates parse tree.
- Syntax Analyzer / Parser is the 2nd Phase of the compiler which takes I/p as tokens and generate a parse tree.

working (3 steps)

Step 1: Yacc specification file.y → [yacc compiler] → y.tab.c

Step 2: y.tab.c → [Compiler] → a.out {syntax analyzer}

Step 3: I/p (Tokens from lexical Analyzer) → [a.out] → O/P {parse}

1) IP to the yacc compiler will be a file with .y extension, it will contains desired grammer in Yacc format. Yacc will convert it into C code in the form of y.tab.c file.

1) This y.tab.c file will be given as input to C compiler and output will be the LALR(1) parser.

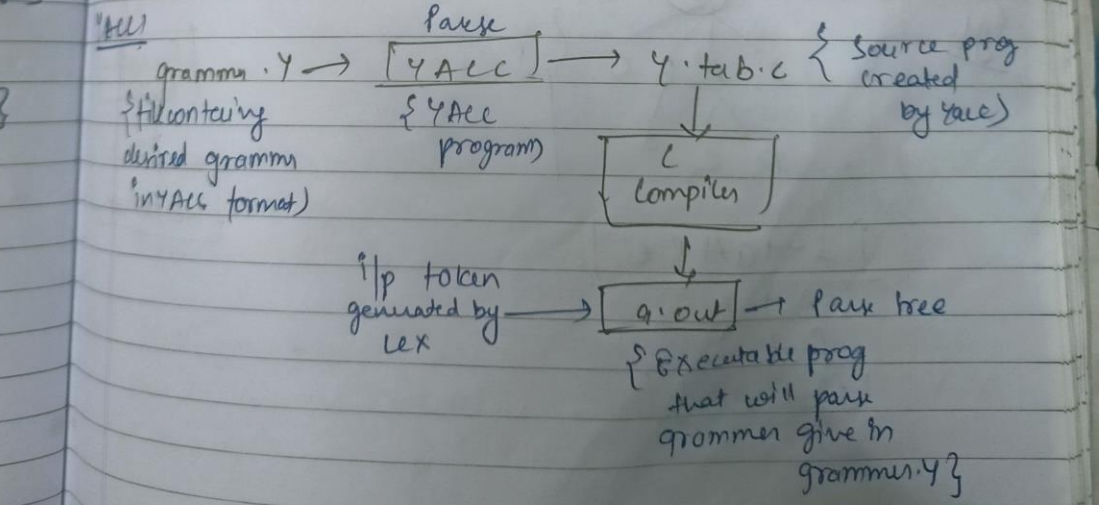
2) Tokens generated by the lexical analyzer (using the lex tool) will be given as a input to [a.out] i.e our LALR parser and we will get the parse tree as output.

Syntax:-

definitions | declaration
%% to by
%% %%
Rules head : body1 {action1} | body2 {action2}
%% %% C code.
Auxiliary Routines | Supplementary code

* YACC takes Grammar as specification

"All"



Symbol table

Types of Parser:-

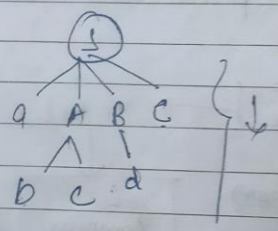
- Top Down Parser
  - Backtracking
  - Predictive Parser (LL(1))
- Bottom Up Parser (Shift Reduce Parser)
  - Operator precedence parser
  - LR Parser
    - LR(0)
    - SLR (1)
    - CLR (1)
    - LALR (1)

Top Down Parser :- (TDP)

The process of construction of parse tree starting from root & proceed to children is called TDP. i.e. starting from start symbol of grammer and reaching the i/p stream

Top down
- Recursive Descent
  - Backtracking
  - Non Backtracking
    - Predictive Parser
      - LL Parser

Eg:

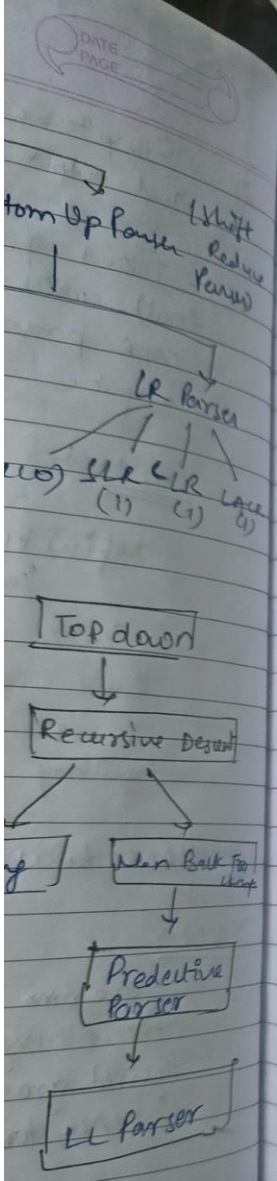


S → aABe

A → bc

B → d

w → abcde



⇒ Back Tracking :-

NOTE :- - TDP internally uses leftmost derivation
- TDP constructed for the grammer if it is free from ambiguity & left recursion.

Classification of TDP :-

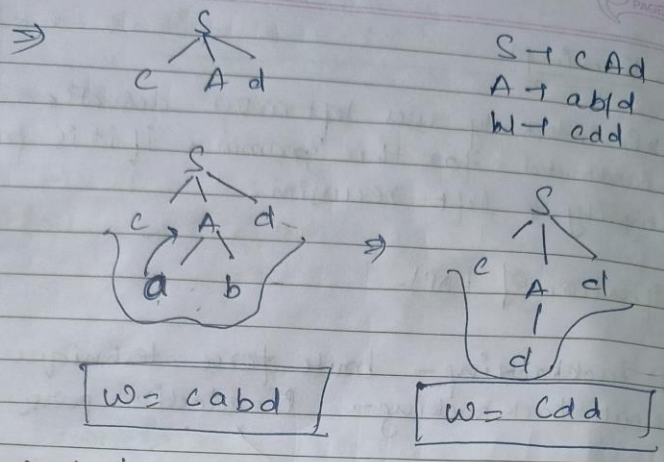
With backtracking – brute force technique
Without backtracking – Predective parse
- LL(1)
- recursive descent parser.

⇒ Recursive Descent :
parsing ⊤ parse constructed from top 2
i/p is read from left to right
↓ The i/p is recursively parsed for preparing a parse tree with or without backtracking.

Backtracking :-

The parse tree is started from root node & i/p string is matched against the production rules for replacing them.

Eg Let grammer G be,
S → cAd
A → ab|d
w = cdd

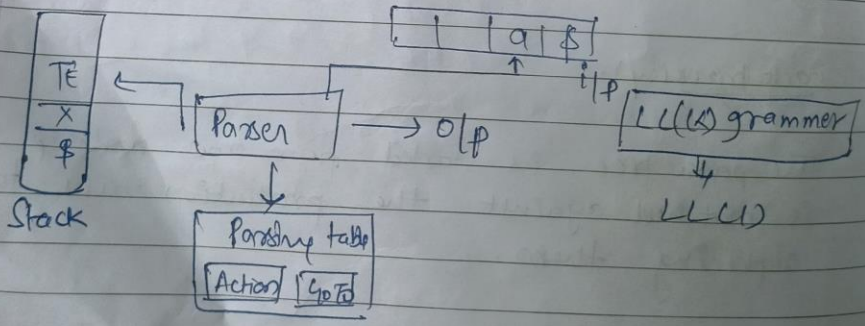


Limitation!-

→ If the given grammer has more no. of alternatives then the cost of backtracking is high.

Predective Parser!-

→ It is a type of Recursive Descent with no backtracking



→ Prededivite Parser us a stack and a prasing table to pase the input and genrate a pase tree

→ Both the Stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed.

→ The parse refers to the parsing table to take any decision on the input and stack element combination

LL Parser

It accepts LL grammars

It is denoted LL(K)

- i/p from left to right
- left most derivation
- no. of look a heads generally K≥1 ∴ LL(K) = LL(1)

A grammar G is LL(1) if there are two distinct productions

$A \longrightarrow \alpha | \beta$

1. for no terminal $\alpha < \beta$ derive string beginning with a
2. At most one of α & β can derive empty string
3. If $\beta \rightarrow \varepsilon$ then 'a' does not derive any string begning with a terminal is Follow(A)

LL(1) uses data structure

- i/p buffer
- Stack
- Parsing table

alternatives

back tracking

rammer

rocessing

Ex Parsing:-

"Shift Reduce Parsing":-

→ Shift Reduce Parsing is a process of reducing a string to the start symbol of a grammar

→ It uses a stack to hold the grammar and an input to hold the string

A string ⟶ (reduce to) the starting symbol

→ Shift reduce parsing performs two actions Shift and Reduce
(1) (2)

→ At shift action, the current symbol in the i/p string is pushed to a stack.

→ At each reduction, the symbol will replaced by the non-terminals.

→ The symbol is the right-side of the production & non-terminal is the left side of the production

Example

Grammar

S → S+S
S → S-S
S → (S)
S → a

Input string
a-(a2+a3)

Example

Stack Contents	Input String	Actions
$ – Start	$a_1 - (a_2 + a_3)$ $	Shift a_1
$ a_1	$- (a_2 + a_3)$ $	Reduce by S → a
$ S	$- (a_2 + a_3)$ $	Shift –
$ S –	$(a_2 + a_3)$ $	Shift (
$ S – (	$a_2 + a_3)$ $	Shift a_2
$ S – (a_2	$+ a_3)$ $	Reduce by S → a
$ S – (S	$+ a_3)$ $	Shift +
$ S – (S +	$a_3)$ $	Shift a_3
$ S – (S + a_3	$)$ $	Reduce by S → a
$ S – (S + S	$)$ $	Shift)
$ S – (S + S)	$	reduce S → S+S
$ S – (S)	$	reduce S → (S)
$ S – S	$	reduce S → S–S
$ S	$	Accepted

Example Consider the grammar

E → 2E2
E → 3E3
E → 4

i/p string 32423

Stack	i/p buffer	Action
$	32423$	shift 3
$3	2423$	Shift 2
$32	423$	Shift 4
$324	23$	~~Shift~~ reduce E→4
$32E	23$	Shift 2
$32E2	3$	Reduce E → 2E2
$3E	3$	Shift 3
$3E3	$	Reduce E → 3E3
$E	$	Accept

Operator Precedence Parsing:-

1) Any grammar (G) is called an operator P-G by two conditions
   a) There exist no production rule which contain ∈ (epsilon) on it's right side.
   b) There exist no production rule which contains two non-terminal adjacent to each other on it's right hand side.

→ A parser that reads and understand an operator precedence grammar is called as operator precende parser.

→ Example- Which is not an operator precendence grammar

$$E \rightarrow EAE \mid (E) \mid -E \mid id \quad (X)$$
$$A \rightarrow + \mid - \mid \times \mid / \mid \wedge$$

↓ Substituting

Example- Which is an operator precendence Grammar

$$E \rightarrow E+E \mid E-E \mid E \times E \mid E/E \mid (E) \mid -E \mid id$$

(terminal) (non terminals)

Operator precendence can only established b/w the terminal of the grammar. It ignores the non-terminal.

Parsing Action :

→ Both end of the given input string, add the $ symbol.
→ Now scan the I/p string from left to right until the > is encountered.
→ Scan towards left over all the equal precedence until the first left most < is encountered.
→ Everything b/w left most < and right most > is handle.
→ $ on $ means parsing is successful.

There are three operator precedence relation
a > b = Terminal a has higher precedence than b
a < b = " " " lower " " "
a = b = " " same " " "

Precedence table :-

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

Rules
id, a, b, c, high
$ = low
+ > +
* > *
id * id
$ A $

example

Step 3

Bottom ↑ up appr

Example: Consider the following grammar and construct the operator precedence parser.

$E \rightarrow EAE \mid id$

$A \rightarrow + \mid *$

Then parse the following string: id + id * id

Step 1: Convert the given grammar to operator precedence grammar.

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

Step 2: Construct the operator precedence table:-

terminal symbols are:-

$\{id, +, *, \$\}$

table:-

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	A

id → highest

$ → lowest

+ > +

* > * -

$ A $

Step 3: Parse the given string:- id + id * id

$E \rightarrow E+E$

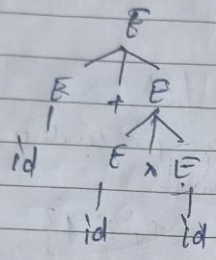
$E \rightarrow E*E$

$E \rightarrow id$

Stack	Relation	Input	Action
$	<	id + id * id $	Shift id
$ id	>	+ id * id $	reduce E → id
$ E	<	+ id * id $	shift +
$ E +	<	id * id $	shift id
$ E + id	>	* id $	reduce E → id
$ E + E	<	* id $	shift *
$ E + E *	<	id $	shift id
$ E + E * id	>	$	reduce E → id.

$ E+E*id	>	↑	Reduce E → id
$ E+E*E	>	↑	reduce E → E*E
$ E+E	>	↑	reduce E → E+E
$ E	A	↑	Accept

Step 4: Generate Parse tree (follow bottom up approach using table)



LR-Parser:

least powerful — SLR (Simple LR) → LALR (Look Ahead LR) → CLR(1) Canonical LR — most powerful

LR parser ⇒ Non recursive shift reduce bottom up parser LR(K) → look ahead symbol

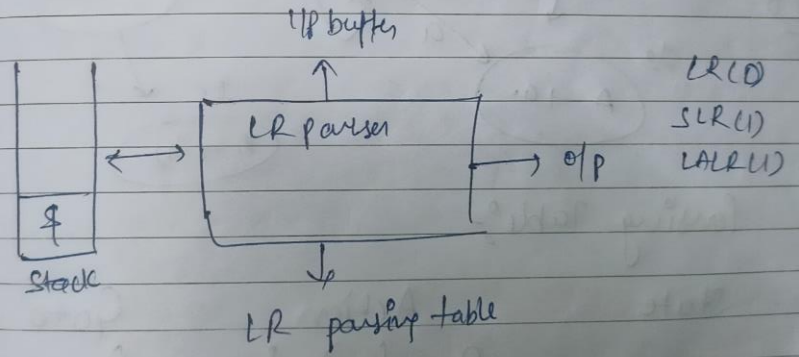
- left to right scanning i/p stream
- Construction of right most derivation in reverse

① SLR → Simple LR parser
→ Works on smallest class of grammers
→ Few no of states
→ Simple and fast construction

② LR(1) :- LR Parser
→ Works on complete set of LR(1) grammer.
→ Large no. of states
→ Slow construction

③ LALR(1) :- Look A head LR parser
→ Works on intermediate size of grammer
→ No. of states are same as SLR(1)

Structure of LR Parser



→ To construct LR(0) & SLR(1) tables we use canonical collection of LR(0) items.
→ To construct LALR(1) & CLR(1) table we use canonical collection of LR(1) items.

SLR(1) parsing :-

↓

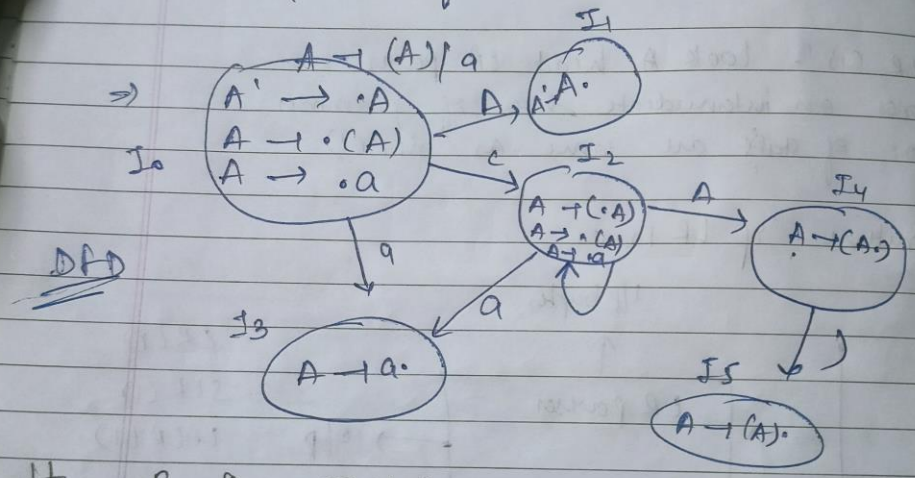
Simple LR

→ Smallest class of grammar

→ Few no: of state

→ Simple & fast to construct

☆ In SLR we place the reduce move only in the follow of left hand side not to entire row.



Parsing Table :-

State	Action				Goto
	a	(	)	$	A
0	S_3	S_2			1
1				Accept	
2	S_3	S_2			4
3					
4			S_5		
5					