

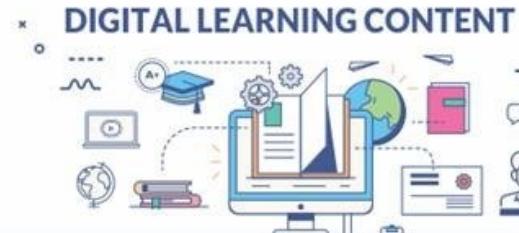


COMPILER DESIGN

SUBJECT CODE: 303105349

Vishal Singh , Assistant Professor
Computer Science & Engineering





CHAPTER-3

Top-down parsing





What is Parsing

Parsing: is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree.

-> Parser is also known as Syntax Analyzer

➤ Parsing technique is divided in two Types.

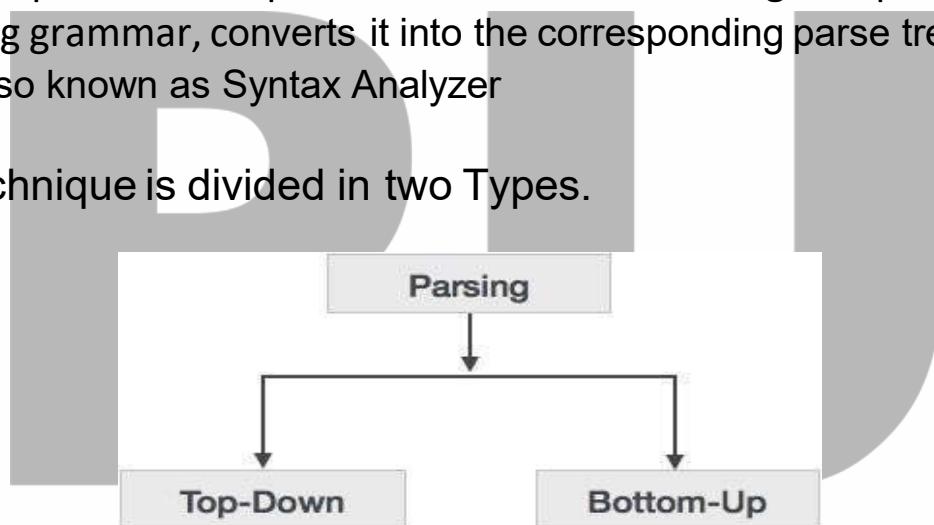


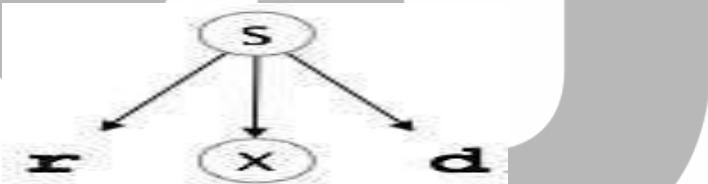
Image source : Google



Top - Down Parsing:

Top-down parser: is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals.

- It uses left most derivation



Bottom-Up/Shift Reduce parser :Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the stat symbol

- It uses reverse of the right most derivation.



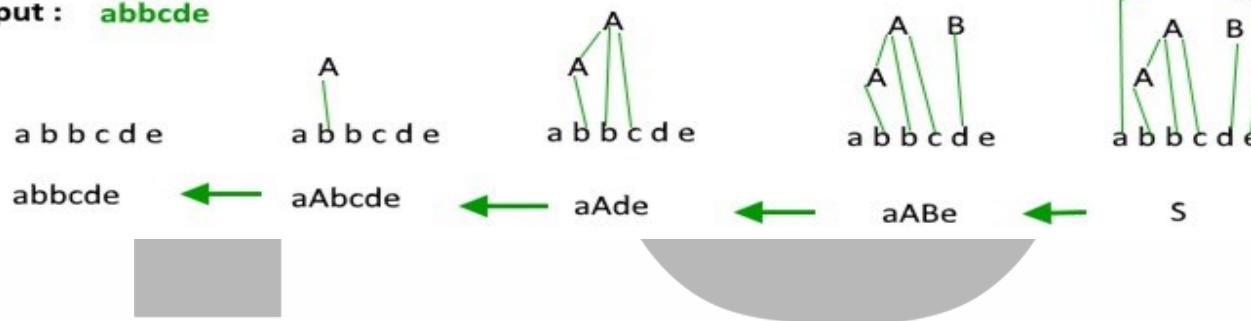
Bottom-Up parser

$S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

Input : abbcde





Top-Down and Bottom-Up parser are further divided in following types

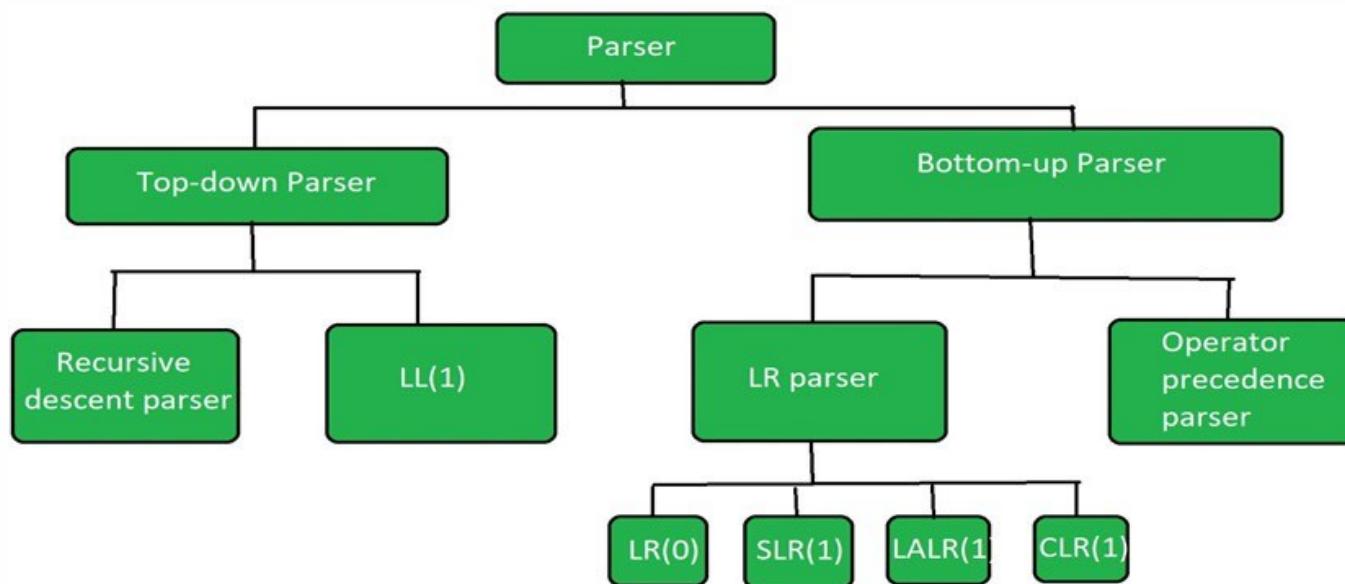


Image source : Google



Recursive Descent parser:

It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non Recursive Descent Parser OR LL (1):

It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

LR parser:

LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar

Operator Precedence Parser:

It generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears in the right-hand side of any production.



FIRST() and FOLLOW() Function

FIRST() : FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

Rules to compute FIRST(X):

1. If x is a terminal, then $\text{FIRST}(x) = \{ 'x' \}$
2. If $x \rightarrow \epsilon$, is a production rule, then add ϵ to $\text{FIRST}(x)$.
3. If $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$ is a production,
 - a) $\text{FIRST}(X) = \text{FIRST}(Y_1)$
 - b) If $\text{FIRST}(Y_1)$ contains ϵ then $\text{FIRST}(X) = \{ \text{FIRST}(Y_1) - \epsilon \} \cup \{ \text{FIRST}(Y_2) \}$
 - c) If $\text{FIRST}(Y_i)$ contains ϵ for all $i = 1$ to n , then add ϵ to $\text{FIRST}(X)$.



Example of FIRST ()

Consider the following Production Rules of Grammar

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

Find out the FIRST() Function of all Production



Solution of Given Grammar

FIRST sets

$\text{FIRST}(E) = \text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \epsilon \}$

$\text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(T') = \{ *, \epsilon \}$

$\text{FIRST}(F) = \{ (, \text{id} \}$

Example:2

$S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

Find out the FIRST() of all production



FOLLOW ():

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

Rules to compute Follow(X):

- 1) $\text{FOLLOW}(S) = \{ \$ \}$ // where S is the starting Non-Terminal
- 2) If $A \rightarrow pBq$ is a production, where p, B and q are any grammar symbols, then everything in $\text{FIRST}(q)$ except ϵ is in $\text{FOLLOW}(B)$.
- 3) If $A \rightarrow pB$ is a production, then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.
- 4) If $A \rightarrow pBq$ is a production and $\text{FIRST}(q)$ contains ϵ , then $\text{FOLLOW}(B)$ contains $\{ \text{FIRST}(q) - \epsilon \} \cup \text{FOLLOW}(A)$



Example of FOLLOW()

Consider the following Production Rules:

$E \rightarrow TE'$

$E' \rightarrow +T E' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | id$

Find out the FOLLOW() of every production

P U



Solution

FIRST set

$\text{FIRST}(E) = \text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \epsilon \}$

$\text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(T') = \{ *, \epsilon \}$

$\text{FIRST}(F) = \{ (, \text{id} \}$

FOLLOW Set

$\text{FOLLOW}(E) = \{ \$,) \} \text{ // Note ')' is there because of 5th rule}$

$\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{ \$,) \} \text{ // See 1st production rule}$

$\text{FOLLOW}(T) = \{ \text{FIRST}(E') - \epsilon \} \cup \text{FOLLOW}(E') \cup \text{FOLLOW}(E) = \{ +, \$,) \}$

$\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{ +, \$,) \}$

$\text{FOLLOW}(F) = \{ \text{FIRST}(T') - \epsilon \} \cup \text{FOLLOW}(T') \cup \text{FOLLOW}(T) = \{ *, +, \$,) \}$



LL(1) Parsing

Construction of LL(1) Parsing Table:

To construct the Parsing table, we have two functions:

First() : If there is a variable, and from that variable if we try to drive all the strings then the beginning *Terminal Symbol* is called the first.

Follow() : What is the *Terminal Symbol* which follow a variable in the process of derivation.



Example of LL(1) parsing

Example-1:

Consider the Grammar:

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid e$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid e$
 $F \rightarrow id \mid (E)$

➤ Calculate the FIRST() and FOLLOW() for the above grammar then construct Parsing Table



FIRST() and FOLLOW ()

	FIRST	FOLLOW
$E \rightarrow TE'$	$\{ \text{id}, (\}$	$\{ \$,) \}$
$E' \rightarrow +TE'/e$	$\{ +, e \}$	$\{ \$,) \}$
$T \rightarrow FT'$	$\{ \text{id}, (\}$	$\{ +, \$,) \}$
$T' \rightarrow *FT'/e$	$\{ *, e \}$	$\{ +, \$,) \}$
$F \rightarrow id/(E)$	$\{ \text{id}, (\}$	$\{ *, +, \$,) \}$



LL(1) Parsing Table

	ID	+	*	()	\$
E →				E →		
E	TE'			TE'		
	E' →			E' -	E' -	
E'	+TE'			> e	> e	
T →				T →		
T	FT'			FT'		
	T' →			T' -	T' -	
T'	T' → e	*FT		> e	> e	
			F →			



How can check LL(1) Grammar

Note: If LL(1) Parsing Table contain single entry in every row and column so this grammar will pass LL(1) parser. If any row or column contains multiple entry then the given grammar will not be pass LL(1) parser.

- Means Given Grammar is LL(1) Grammar



LR-Parser:

- A general shift reduce parsing is LR parsing.
- The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse

LR –parser are of following types:

- (a) LR(0)
- (b) SLR(1)
- (c) LALR(1)
- (d) CLR(1)

To construct this parser we need Two function

- a) Closure()
- b) Go To()



Augmented Grammar & LR(0) Items

Let's consider the following grammar

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

The augmented grammar for the above grammar will be

$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

LR(0) Items: An LR(0) is the item of a grammar G is a production of G with a dot at some position in the right side.

$S \rightarrow ABC$ -> Given production

LR(0) Item of given production

$$S \rightarrow .ABC$$

$$S \rightarrow A.BC$$

$$S \rightarrow AB.C$$

$$S \rightarrow ABC.$$



Closure() & Goto()

Closure():

If I is a set of items for a grammar G , then $\text{closure}(I)$ is the set of items constructed from I by the two rules:

1. Initially every item in I is added to $\text{closure}(I)$.
2. If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production then add the item $B \rightarrow .\gamma$ to I , if it is not already there. We apply this rule until no more items can be added to $\text{closure}(I)$.

Goto () :

$\text{Goto}(I, X) =$

1. Add I by moving dot after X .
2. Apply closure to first step.



Example of Closure

 $S' \rightarrow S$
 $S \rightarrow AA$
 $A \rightarrow aA/b$

$\text{Closure}(S' \rightarrow S) = S' \rightarrow .S$ non terminal. So add production of S

$S \rightarrow .AA$ non terminal after dot. So add production of A

 $A \rightarrow .aA/ .b$
 $\text{Closure}(S \rightarrow A.A) = S \rightarrow A.A$
 $A \rightarrow .aA/ .b$
 $\text{Closure}(A \rightarrow aA) = A \rightarrow a.A$

Image source : Google



Example of GO to()

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA/b$

$\text{Goto}(S' \rightarrow .S, S) = S' \rightarrow S . \square \longrightarrow$ nothing is present after dot, no closure

$S \rightarrow .AA \longrightarrow$ non terminal after dot. So add production of A

$A \rightarrow .aA / .b$

$\text{Goto}(S \rightarrow .AA, A) = S \rightarrow A . A \longrightarrow$ apply closure

$A \rightarrow .aA / .b$

$\text{Goto}(A \rightarrow aA, a) = A \rightarrow a . A \longrightarrow$ apply closure

$A \rightarrow .aA / .b$

Image source : Google



LR(0) Parser

Consider the following grammar

G:
 $S \rightarrow AA \text{ ----- (1)}$
 $A \rightarrow aA \text{ ----- (2)}$
 $A \rightarrow b \text{ ----- (3)}$

Step: 1- Convert this grammar in Augmented grammar G':

G':
 $S' \rightarrow \bullet S$
 $S \rightarrow \bullet AA$
 $A \rightarrow \bullet aA$
 $A \rightarrow \bullet b$

Step-2: Find out the Closure() and Goto() for the given grammar

Step-3: Based up closure and Goto() construct the DFA of Augmented production



DFA Of LR(0) Parser

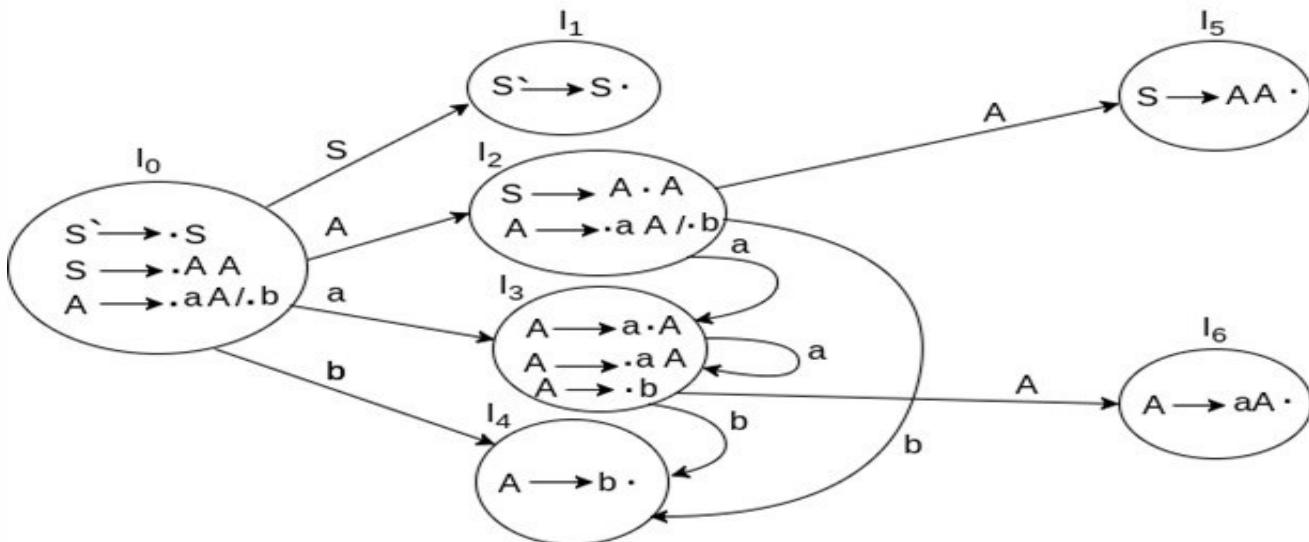


Image source : Google



LR(0) Parsing Table:

States	Action			Go to	
	a	b	\$	A	S
I ₀	S ₃	S ₄		2	1
I ₁			accept		
I ₂	S ₃	S ₄		5	
I ₃	S ₃	S ₄		6	
I ₄	r ₃	r ₃	r ₃		
I ₅	r ₁	r ₁	r ₁		
I ₆	r ₂	r ₂	r ₂		

Note: LR (0) table contain single entry in every row and column so this grammar will pass LR(0) parser. If any row or column contains multiple entry then the given grammar will not be pass LR(0) parser.



SLR(1) Parser

SLR(1) Parser:

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

Example of SLR (1) Parsing :

Consider the following grammar G:

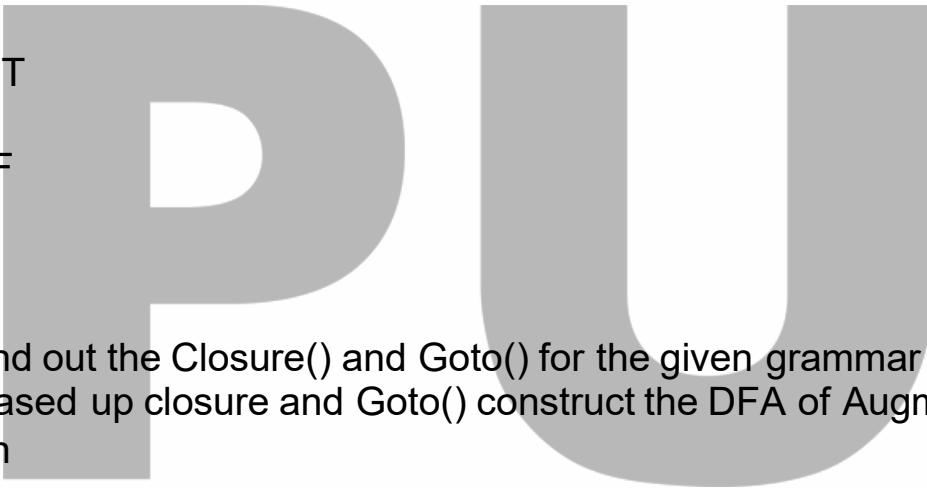
$S \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow id$



SLR(1) Parser

Step-1: Convert this Grammar in to Augmented Grammar G'

$S' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T * F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet id$



Step-2: Find out the Closure() and Goto() for the given grammar

Step-3: Based up closure and Goto() construct the DFA of Augmented production



DFA for SLR(1)

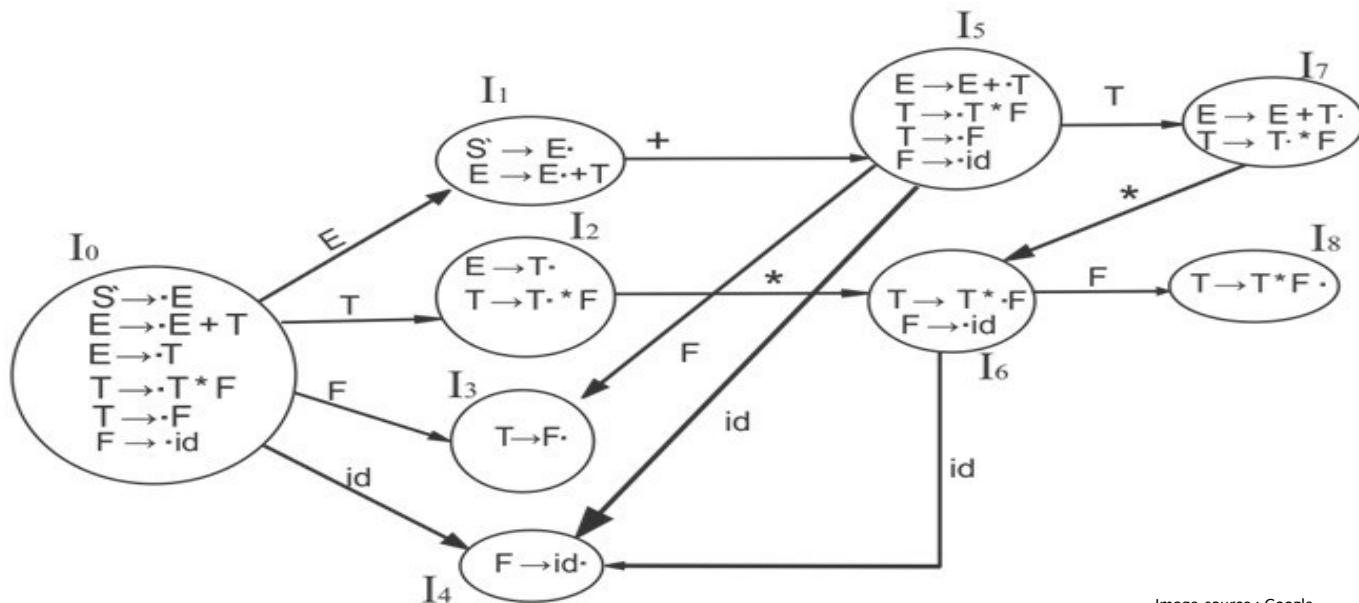


Image source : Google



SLR(1) Parsing Table:

1. If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the FOLLOW() of that variable.

States	Action				Go to		
	id	+	*	\$	E	T	F
I ₀	S ₄				1	2	3
I ₁		S ₅		Accept			
I ₂		R ₂	S ₆	R ₂			
I ₃		R ₄	R ₄	R ₄			
I ₄		R ₅	R ₅	R ₅			
I ₅	S ₄				7	3	
I ₆	S ₄					8	
I ₇		R ₁	S ₆	R ₁			
I ₈		R ₃	R ₃	R ₃			



CLR(1)/LR(1) Parser

CLR(1) /LR(1) Parsing:

CLR refers to canonical look ahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

LR(1) Item= LR(0) Item + Look ahead Symbol



Example of CLR(1) Parser

Consider the following grammar to construct to LR(1) item

G:

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

Augmented grammar with LR(1) Item

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet A A, \$$

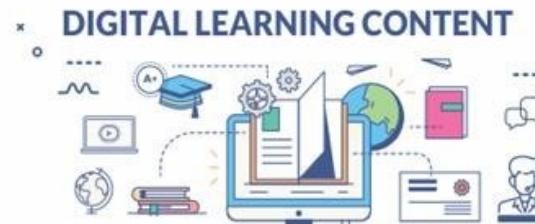
$A \rightarrow \bullet a A, a/b$

$A \rightarrow \bullet b, a/b$

To construct the CLR(1) Parser

Step-1: construct Augmented grammar with LR(1) item

Step-2: Find out the Closure() and Goto() for the given grammar with Look ahead symbol



DFA for CLR(1)

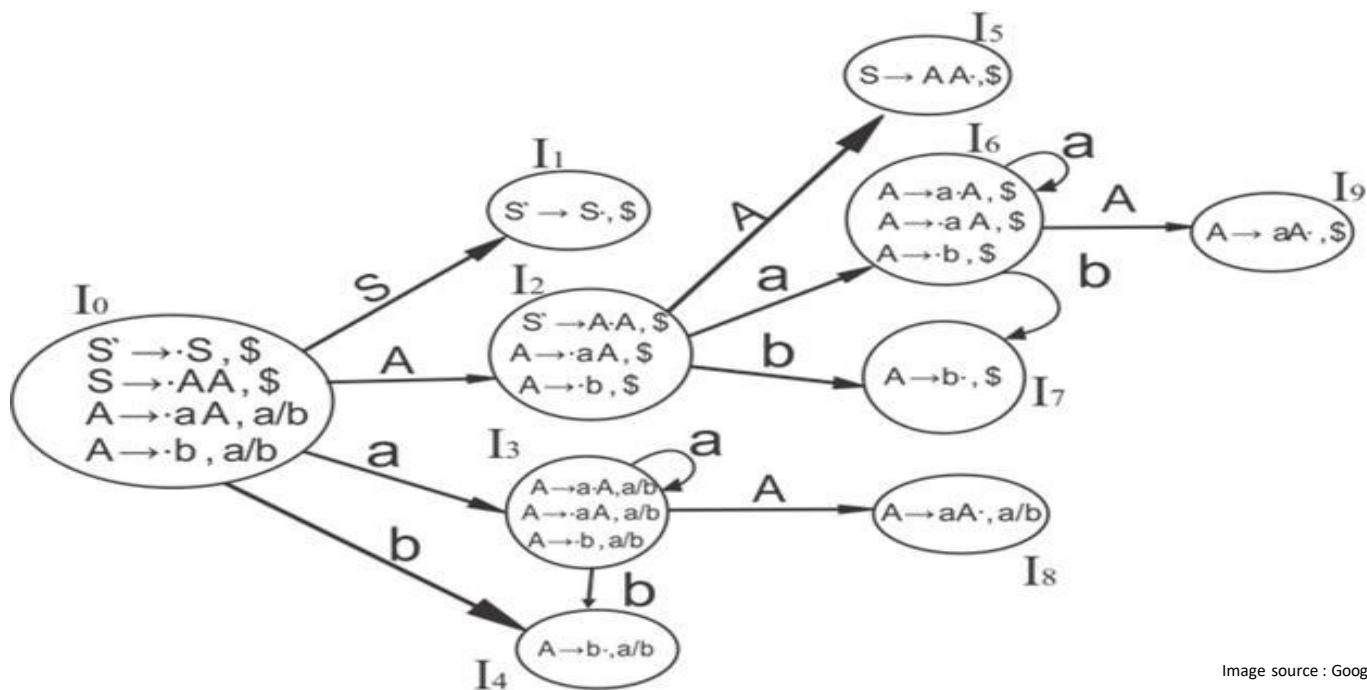


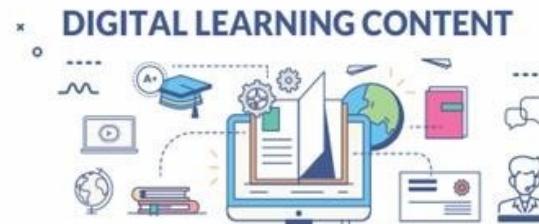
Image source : Google



CLR(1) parsing Table

Construction of CLR(1) Parsing Table:

1. If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the look ahead symbol of that the production



CLR(1) Parsing Table

States	a	b	\$	S	A
I ₀	S ₃	S ₄			2
I ₁			Accept		
I ₂	S ₆	S ₇			5
I ₃	S ₃	S ₄			8
I ₄	R ₃	R ₃			
I ₅			R ₁		
I ₆	S ₆	S ₇			9
I ₇			R ₃		
I ₈	R ₂	R ₂			
I ₉			R ₂		



Example of LALR(1)

Consider the following grammar to construct to LR(1) item

G:

$S \rightarrow AA$, $A \rightarrow aA$, $A \rightarrow b$

Augmented grammar with LR(1) Item

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet A A, \$$

$A \rightarrow \bullet a A, a/b$

$A \rightarrow \bullet b, a/b$

To construct the LALR(1) Parser

Step-1: construct Augmented grammar with LR(1) item

Step-2: Find out the Closure() and Goto() for the given grammar with Look ahead symbol

Step-3: Based up closure and Goto() construct the DFA of Augmented production

Step-4: If two state of DAF is having same production but different look ahead symbol then merge these two step in single state.



LALR(1) Parsing

LALR(1) Parsing

LALR refers to the look ahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.



DFA for LALR(1)

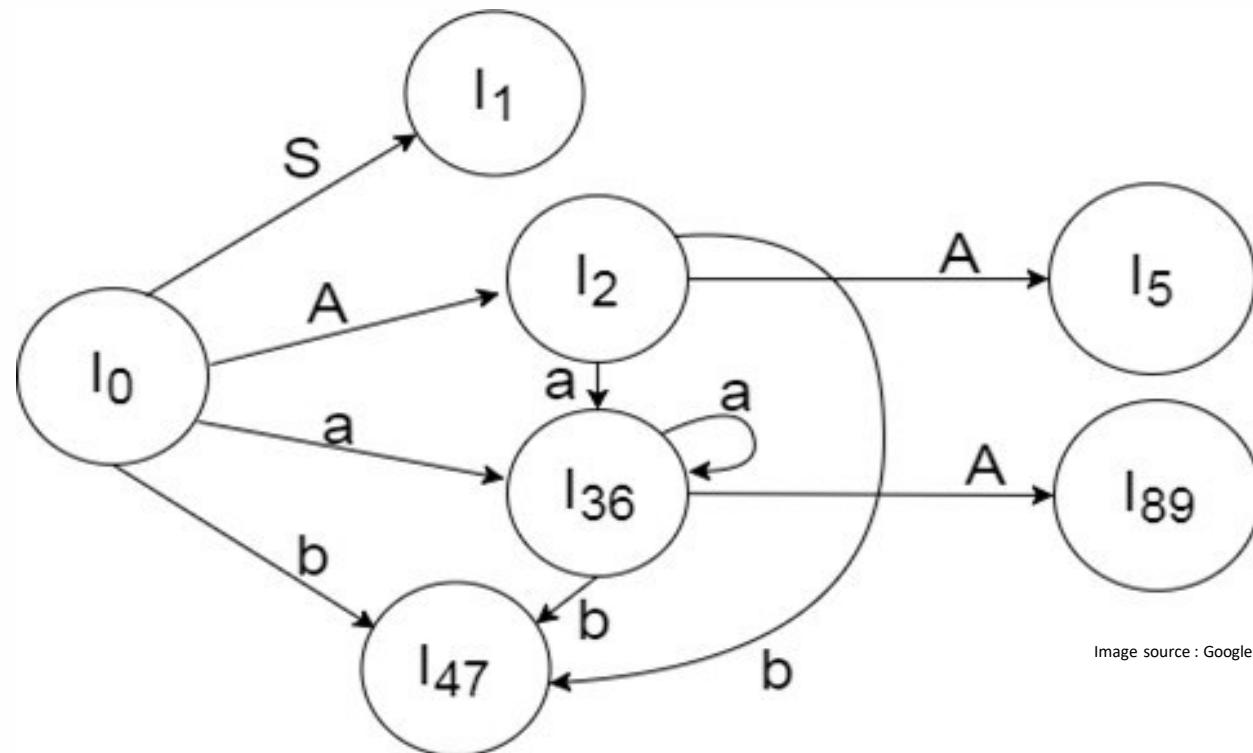


Image source : Google



LALR(1) Parsing Table

LALR(1) Parsing Table:

1. If a state is going to some other state on a terminal then it correspond to a shift move.
2. If a state is going to some other state on a variable then it correspond to go to move.
3. Final entry of reduction can be done based on the look ahead symbol of that the production



LALR(1) Parsing Table

States	a	b	\$	S	A
I ₀	S ₃₆	S ₄₇		12	
I ₁		accept			
I ₂	S ₃₆	S ₄₇			5
I ₃₆	S ₃₆ S ₄₇				89
I ₄₇	R ₃ R ₃	R ₃			
I ₅			R ₁		
I ₈₉	R ₂	R ₂	R ₂		



Operator Precedence parsing:

- Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.
- A grammar is said to be operator precedence grammar if it has two properties:
- No R.H.S. of any production has a ϵ .
- No two non-terminals are adjacent.
- Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.



Operator Precedence parsing

There are three operator precedence relations:

$a > b$ means that terminal "a" has the higher precedence than terminal "b".

$a < b$ means that terminal "a" has the lower precedence than terminal "b".

$a \doteq b$ means that the terminal "a" and "b" both have same precedence.

Example of Operator precedence:

Grammar

$E \rightarrow E+T/T$

$T \rightarrow T^*F/F$

$F \rightarrow id$



Operator Precedence Table

	E	T	F	id	+	*	\$
E	X	X	X	X	÷	X	➢
T	X	X	X	X	➢	÷	➢
F	X	X	X	X	➢	➢	➢
id	X	X	X	X	➢	➢	➢
+	X	÷	≤	≤	X	X	X
*	X	X	÷	≤	X	X	X
\$	≤	≤	≤	≤	X	X	X

Image source : Google

For the given string W= id + id * id check whether this parser will parsed or not



For the given string W= id + id * id check whether this parser will parsed or not

$\$ \ll \text{id1} \gg + \text{id2} * \text{id3} \$$

$\$ \ll F \gg + \text{id2} * \text{id3} \$$

$\$ \ll T \gg + \text{id2} * \text{id3} \$$

$\$ \ll E \doteq + \ll \text{id2} \gg * \text{id3} \$$

$\$ \ll E \doteq + \ll F \gg * \text{id3} \$$

$\$ \ll E \doteq + \ll T \doteq * \ll \text{id3} \gg \$$

$\$ \ll E \doteq + \ll T \doteq * \doteq F \gg \$$

$\$ \ll E \doteq + \doteq T \gg \$$

$\$ \ll E \doteq + \doteq T \gg \$$

$\$ \ll E \gg \$$



Image source : Google

Accept.

- ✖ **DIGITAL LEARNING CONTENT**



Parul® University



www.paruluniversity.ac.in





COMPILER DESIGN

SUBJECT CODE: 203105351

Shyambabu Pandey, Assistant Professor
Computer Science & Engineering





CHAPTER-3

Top Down Parsing

DIGITAL LEARNING CONTENT



Content

- Introduction to YACC
- Error recover by YACC
- Example of YACC
Specific





Introduction to

- A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. They are also known as compiler-compilers.
- YACC (Yet another compiler-compiler) is a LALR(1) parser generator.
- It provides a tool to produce a parser for a given grammar LALR (1) grammar.
- It is used to produce source code of syntactic analyzer of the language ~, LALR(1) grammar.



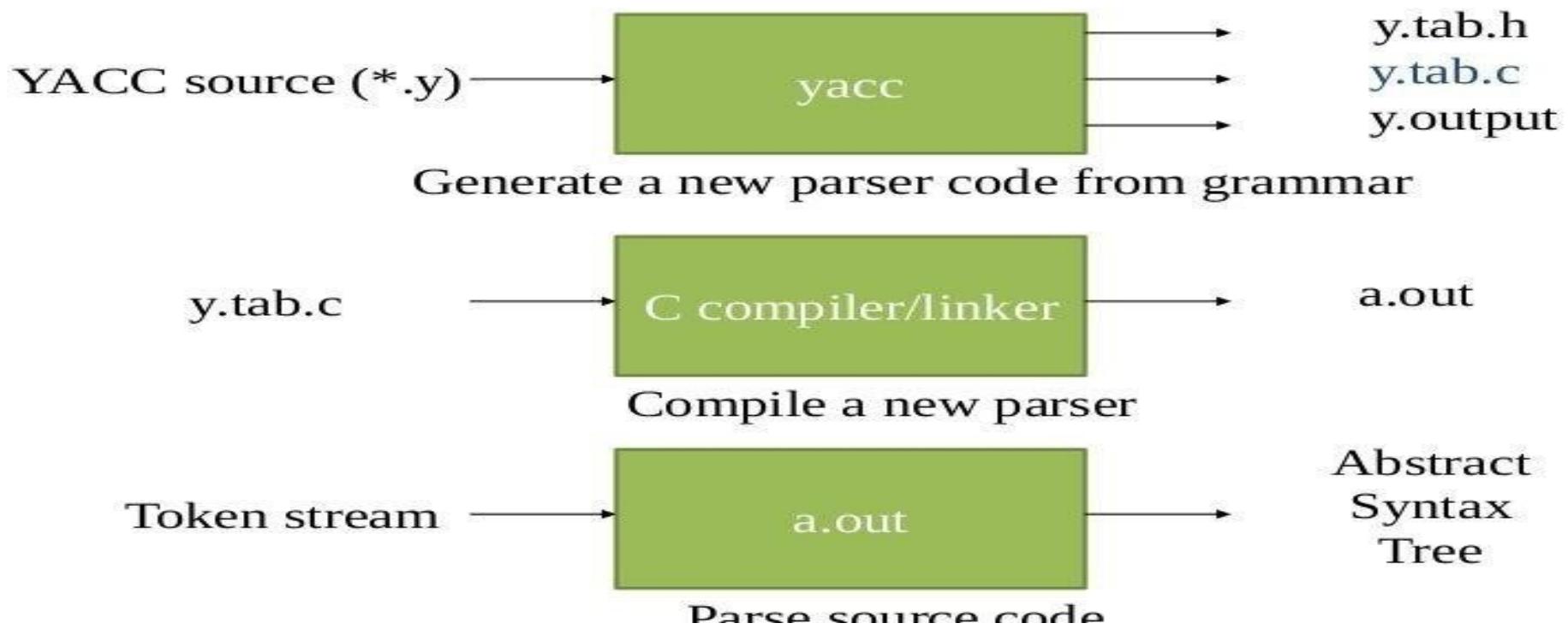
Introduction to

- It generates c code of syntax analyzer of parser.
- YACC (Yet another compiler-compiler) is a LALR(1) parser generator.
- It provides a tool to produce a parser for a given grammar LALR (1) grammar.
- It is used to produce source code of syntactic analyzer of the language by LALR(1) grammar.





How does YACC





How does YACC

- The tool yacc can be used to generate automatically an LALR parser.
- Input of yaac is divided into three sections:
 1. Definition
 1. Rules
 1. Subroutines





- The def section consists of token declarations & C code bracketed by % { and % }.
- The grammar is placed in the rules section.
- User subroutines are added in subroutines section.





YAAC

%{

C declarations

%}

yaac declarations

%%

Grammar rules

%%

Additional c code





Example of

C declarations

yacc declarations

Grammar rules

Additional C code

```
%{  
#include <stdio.h>  
%}  
  
%token NAME NUMBER  
%%  
  
statement: NAME '=' expression  
          | expression  
          ;  
  
expression: expression '+' NUMBER { $$ = $1 + $3; }  
          | expression '-' NUMBER { $$ = $1 - $3; }  
          | NUMBER  
          ;  
%%  
int yyerror(char *s)  
{  
    fprintf(stderr, "%s\n", s);  
    return 0;  
}  
  
int main(void)  
{  
    yyparse();  
    return 0;  
}
```