Parul® University

COMPILER DESIGN
SUBJECT CODE: 203105351

Prof. Praveen Kumar, Assistant Professor
Computer Science & Engineering

Parul® University

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.

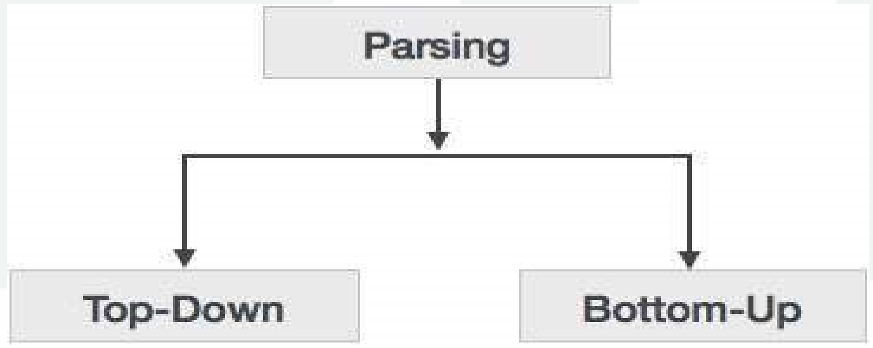
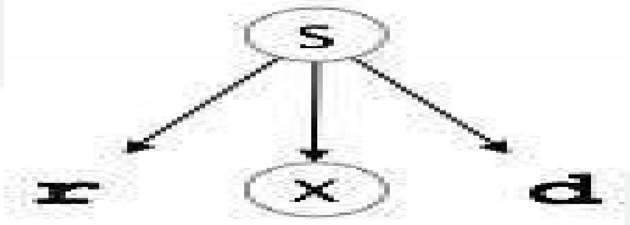


Image source : Google

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.

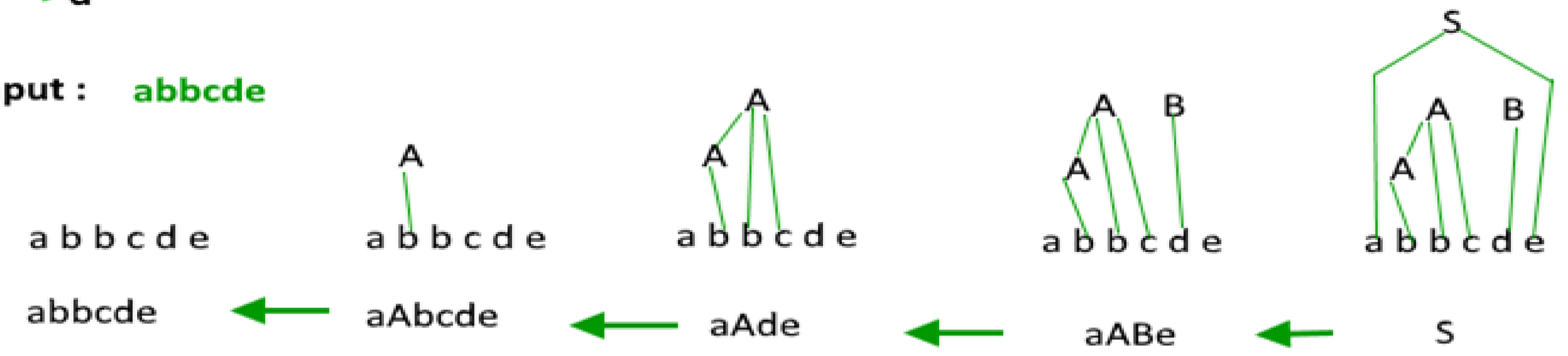
Bottom-Up parser

$S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

Input : abbcde



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

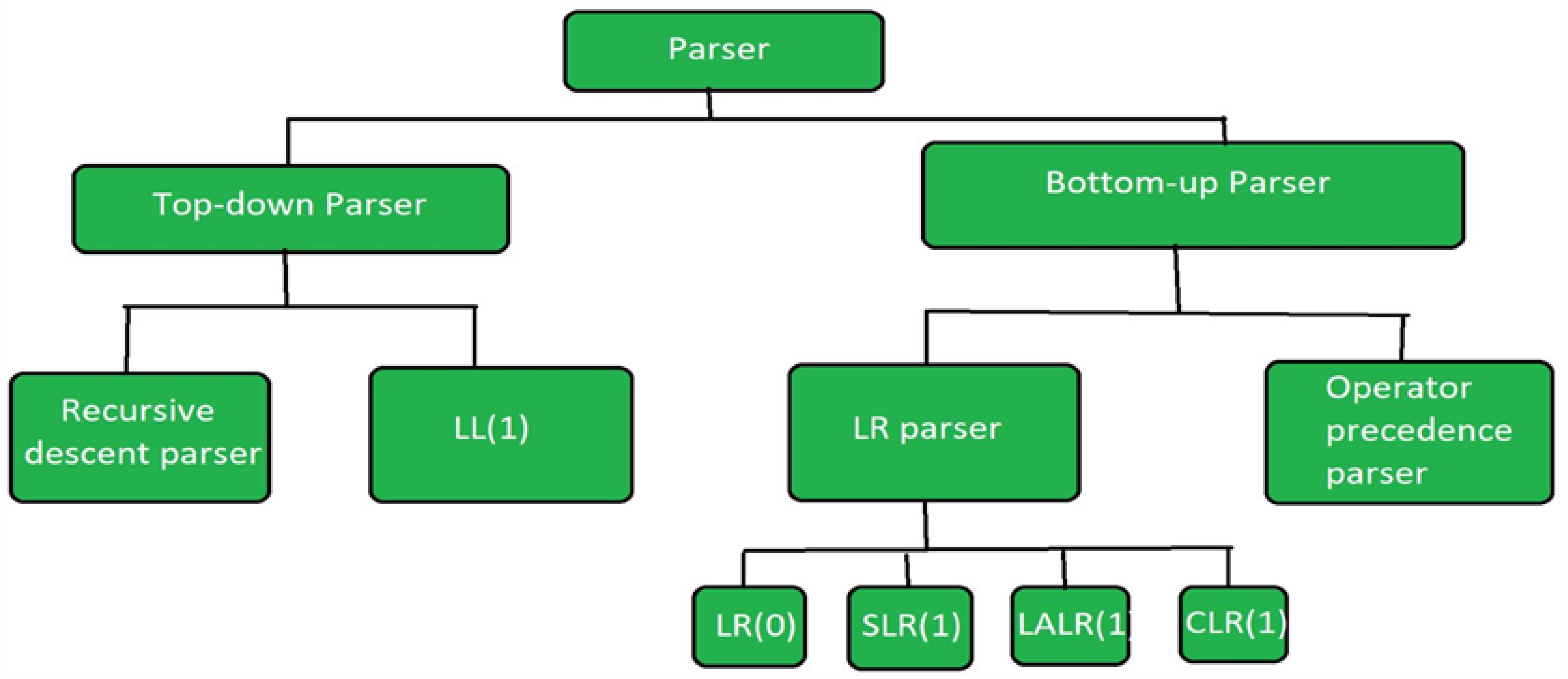


Image source : Google

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 FIRST(x) = { ‘x’ }
2. If x-> Є, is a production rule, then add Є to FIRST(x).
3. If X->Y1 Y2 Y3….Yn is a production,
   a.) FIRST(X) = FIRST(Y1)
   b.) If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }
   c.) If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Example of FIRST ()

Consider the following Production Rules of Grammar

E -> TE’

E’ -> +T E’|Є

T -> F T’

T’ -> *F T’ | Є

F -> (E) | id

Find out the FIRST() Function of all Production

Solution of Given Grammar

FIRST sets

FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, Є }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, Є }

FIRST(F) = { (, id }

Example:2

S -> ACB | Cbb | Ba

A -> da | BC

B -> g | Є

C -> h | Є

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) FOLLOW(S) = { $ }   // where S is the starting Non-Terminal

2) If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).

3) If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).

4) If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

Parul® University

DIGITAL LEARNING CONTENT

Example of FOLLOW()

Consider the following Production Rules:

E -> TE’

E’ -> +T E’|Є

T -> F T’

T’ -> *F T’ | Є

F -> (E) | id

Find out the FOLLOW() of every production

Solution

FIRST set

FIRST(E) = FIRST(T) = { (, id }
FIRST(E’) = { +, Є }
FIRST(T) = FIRST(F) = { (, id }
FIRST(T’) = { *, Є }
FIRST(F) = { (, id }

FOLLOW Set

FOLLOW(E) = { $,) } // Note ')' is there because of 5th rule
FOLLOW(E’) = FOLLOW(E) = { $,) } // See 1st production rule
FOLLOW(T) = { FIRST(E’) – Є } U FOLLOW(E’) U FOLLOW(E) = { + , $,) }
FOLLOW(T’) = FOLLOW(T) = { + , $,) }
FOLLOW(F) = { FIRST(T’) – Є } U FOLLOW(T’) U 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 --> TE'

E' --> +TE' | e

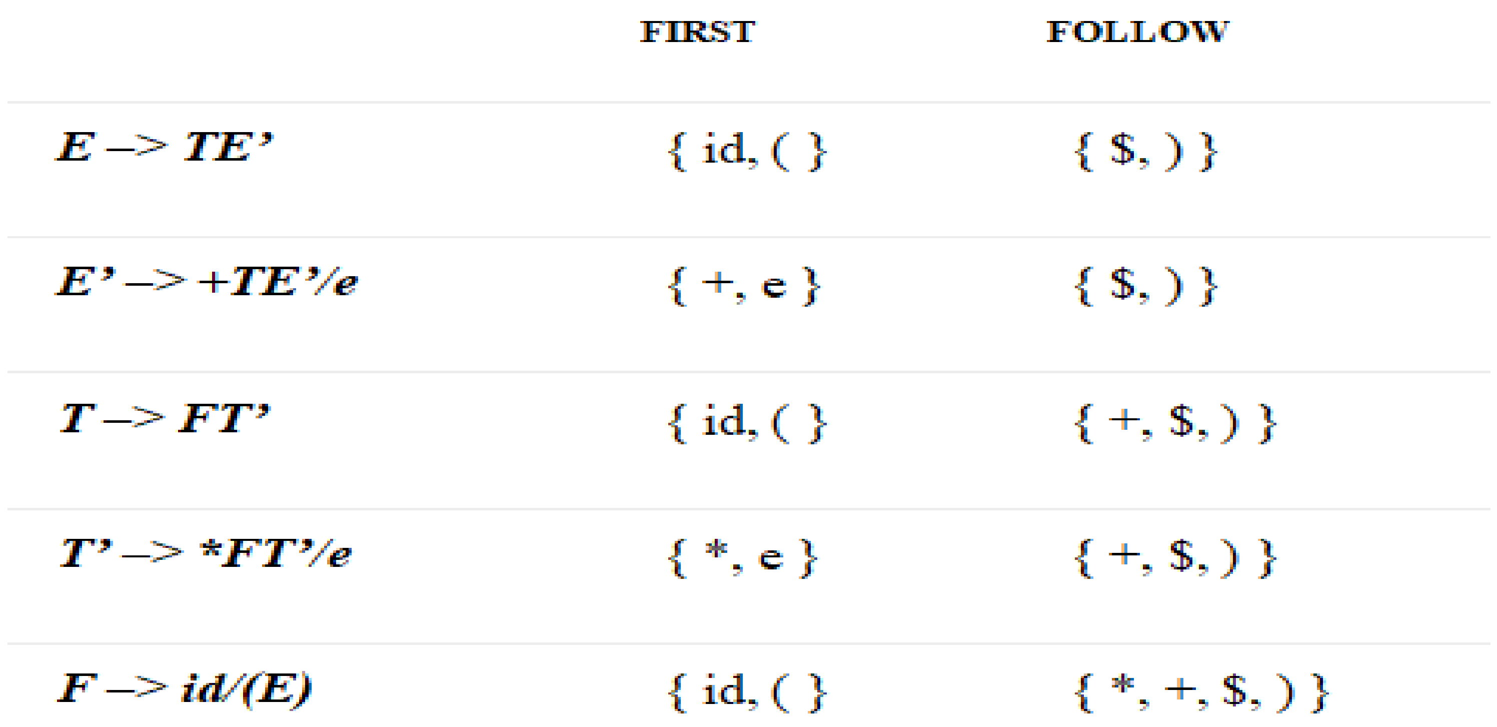
T --> FT'

T' --> *FT' | e

F --> id | (E)

➢Calculate the FIRST() and FOLLPW() for the above grammar then construct Parsing Table

FIRST() and FOLLOW ()



	FIRST	FOLLOW
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/e	{ +, e }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
*T’ –> *FT’/e*	{ *, e }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

LL(1) Parsing Table

	ID	+	*	(	)	$
E	E –> TE'			E –> TE'		
E'		E' –> +TE'			E' –> e	E' –> e
T	T –> FT'			T –> FT'		
T'		T' –> e	T' –> *FT		T' –> e	T' –> 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()

Parul® University

DIGITAL LEARNING CONTENT

Augmented Grammar &LR(0) Items

Let’s consider the following grammar

S -> AA

A -> aA | b

The augmented grammar for the above grammar will be

S’ -> S

S -> AA

A -> aA | 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 -> ABC -> Given production

LR(0) Item of given production

S -> .ABC

S -> A.BC

S -> AB.C

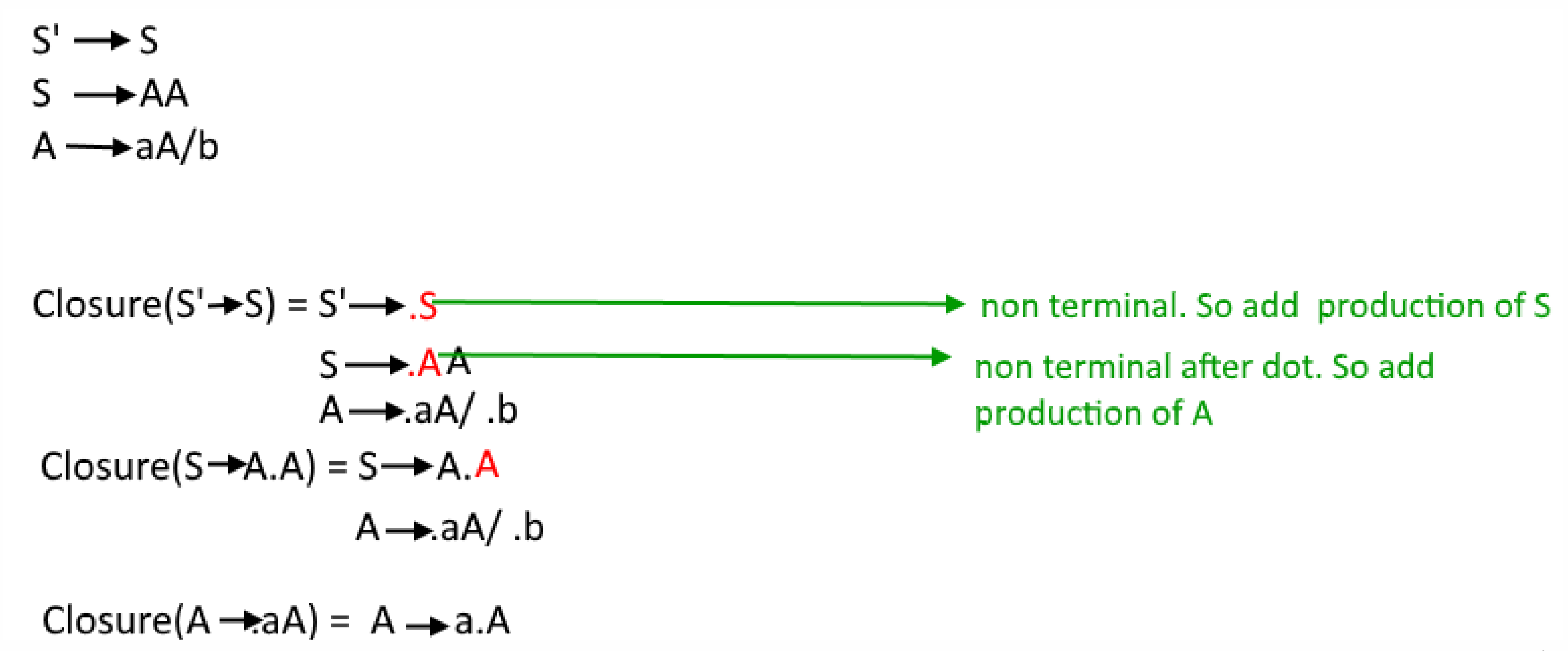
S -> ABC.

Closure() & Goto()

Closure():
If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:
1. Initially every item in I is added to closure(I).
2. If A -> α.Bβ is in closure(I) and B -> γ is a production then add the item B -> .γ to I, If it is not already there. We apply this rule until no more items can be added to closure(I).

Goto () :
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

Parul® University

Example of GO to()

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA/b$

$\text{Goto}(S' \rightarrow .S, S) = S' \rightarrow S.\boxed{}$ $\longrightarrow$ 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 → AA -----------(1)
A → aA -----------(2)
A-> b -------------(3)

Step: 1- Covert this grammar in Augmented grammar G':
G':
S` → •S
S → •AA
A → •aA
A → •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

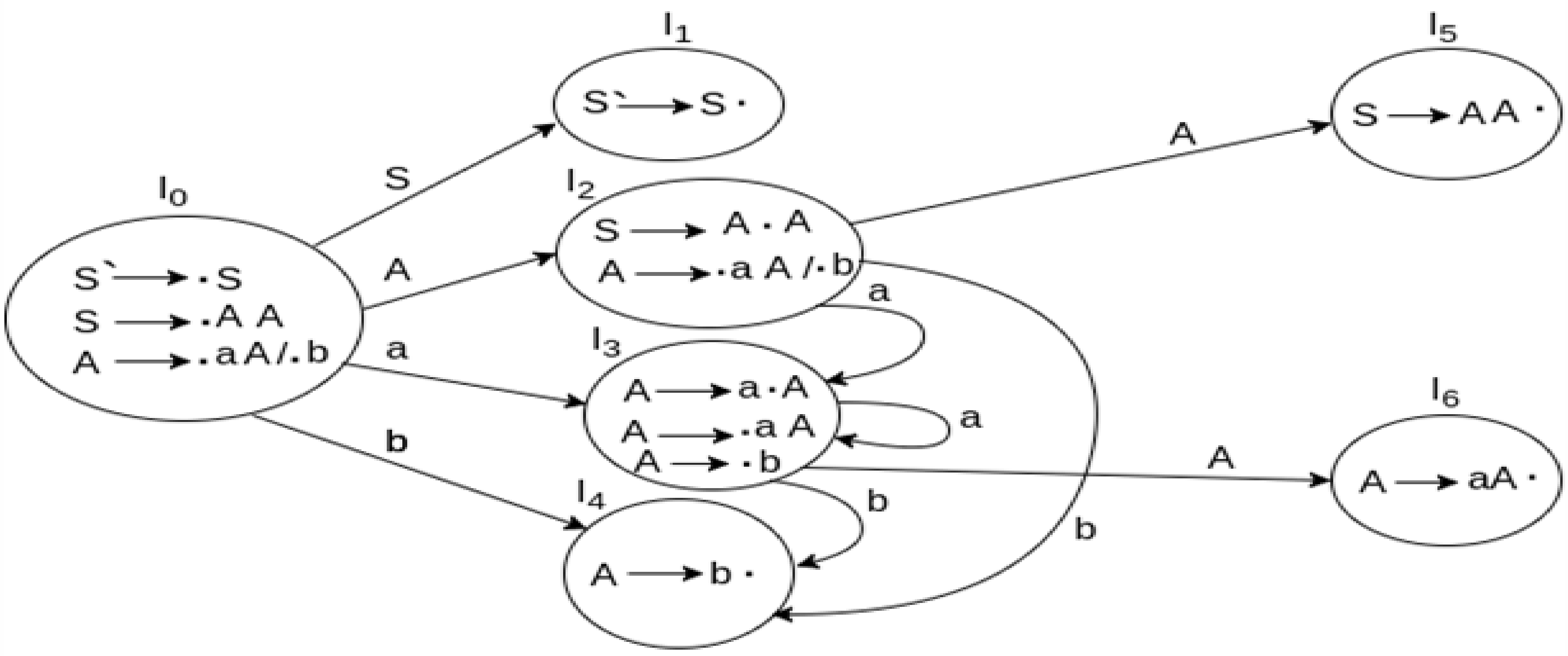
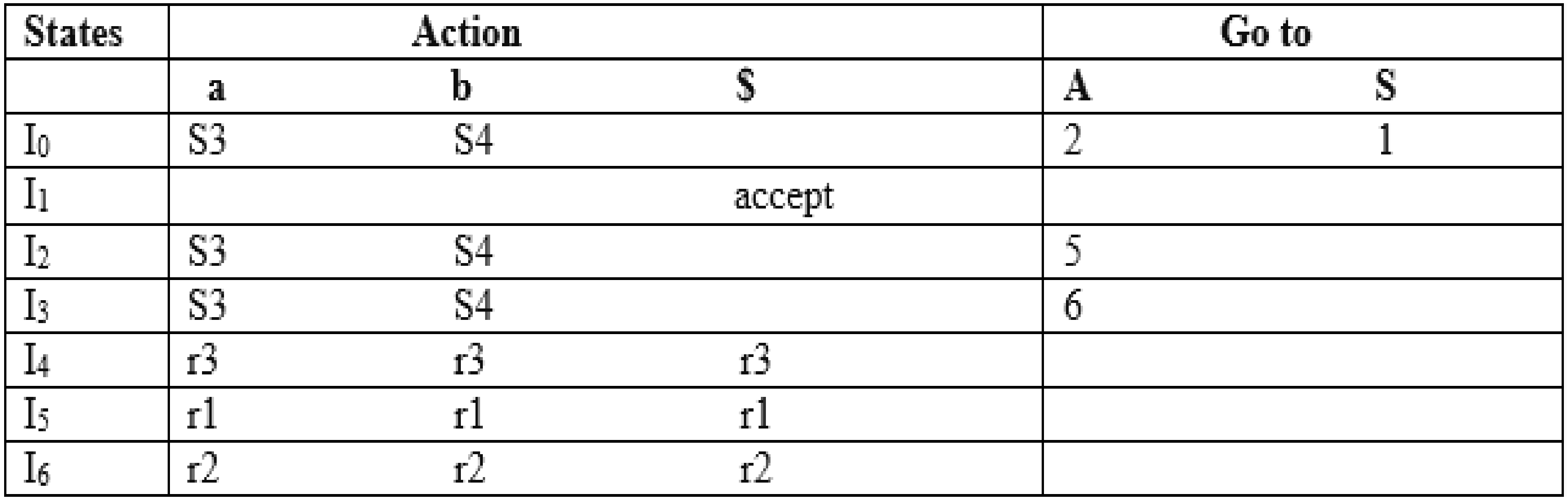


Image source : Google

Parul® University

LR(0) Parsing Table:



States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

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` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •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

Parul® University

DIGITAL LEARNING CONTENT

DFA for SLR(1)

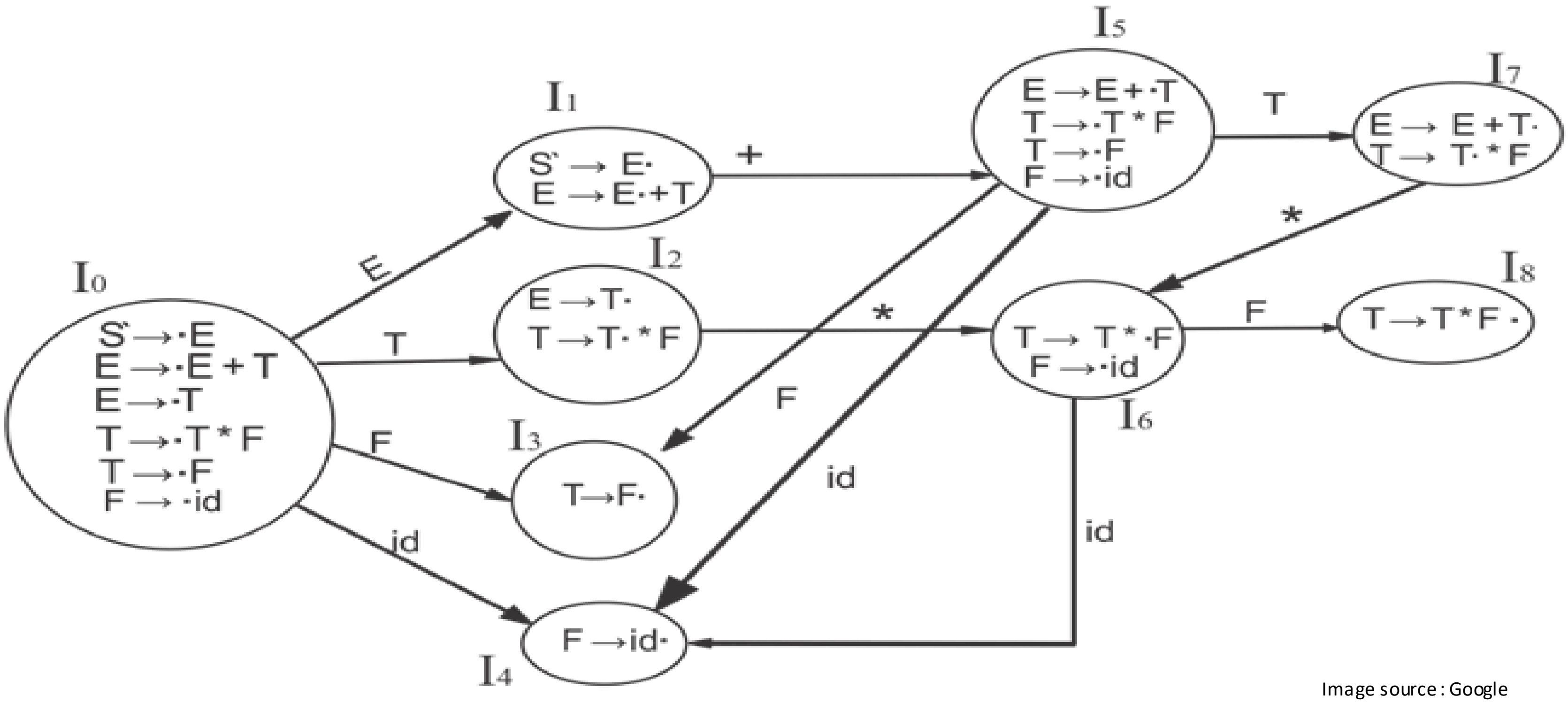
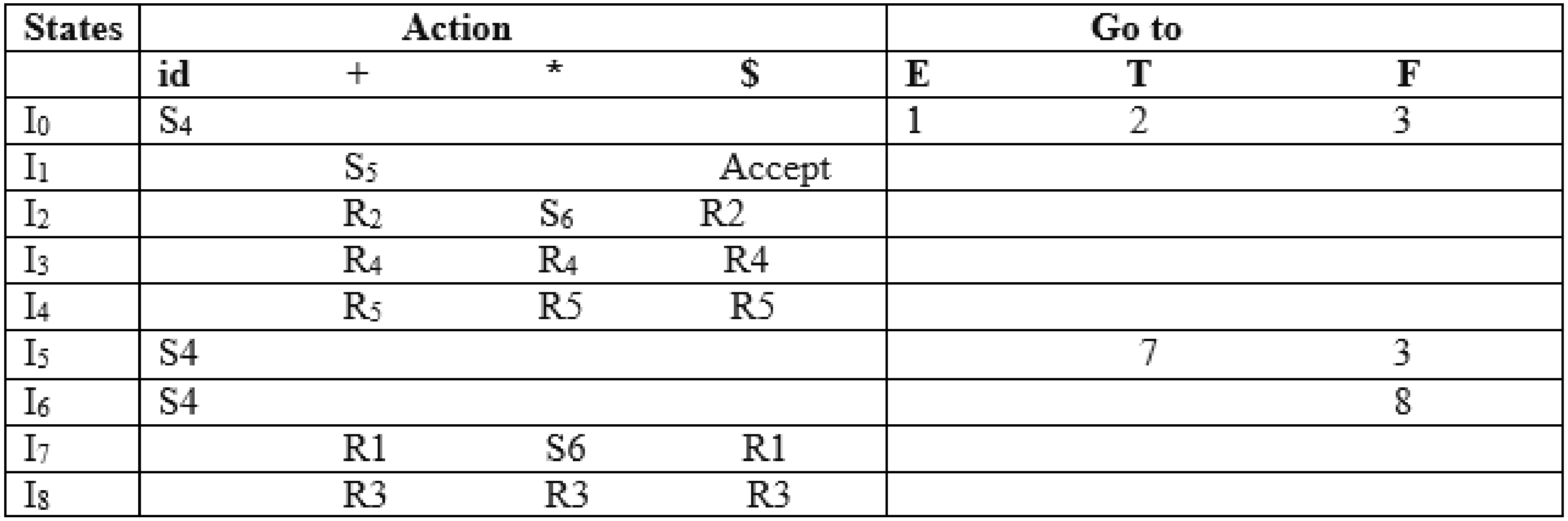


Image source : Google

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	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

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 → AA

A → aA

A → b

Augmented grammar with LR(1) Item

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •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

Parul® University

DIGITAL LEARNING CONTENT

DFA for CLR(1)

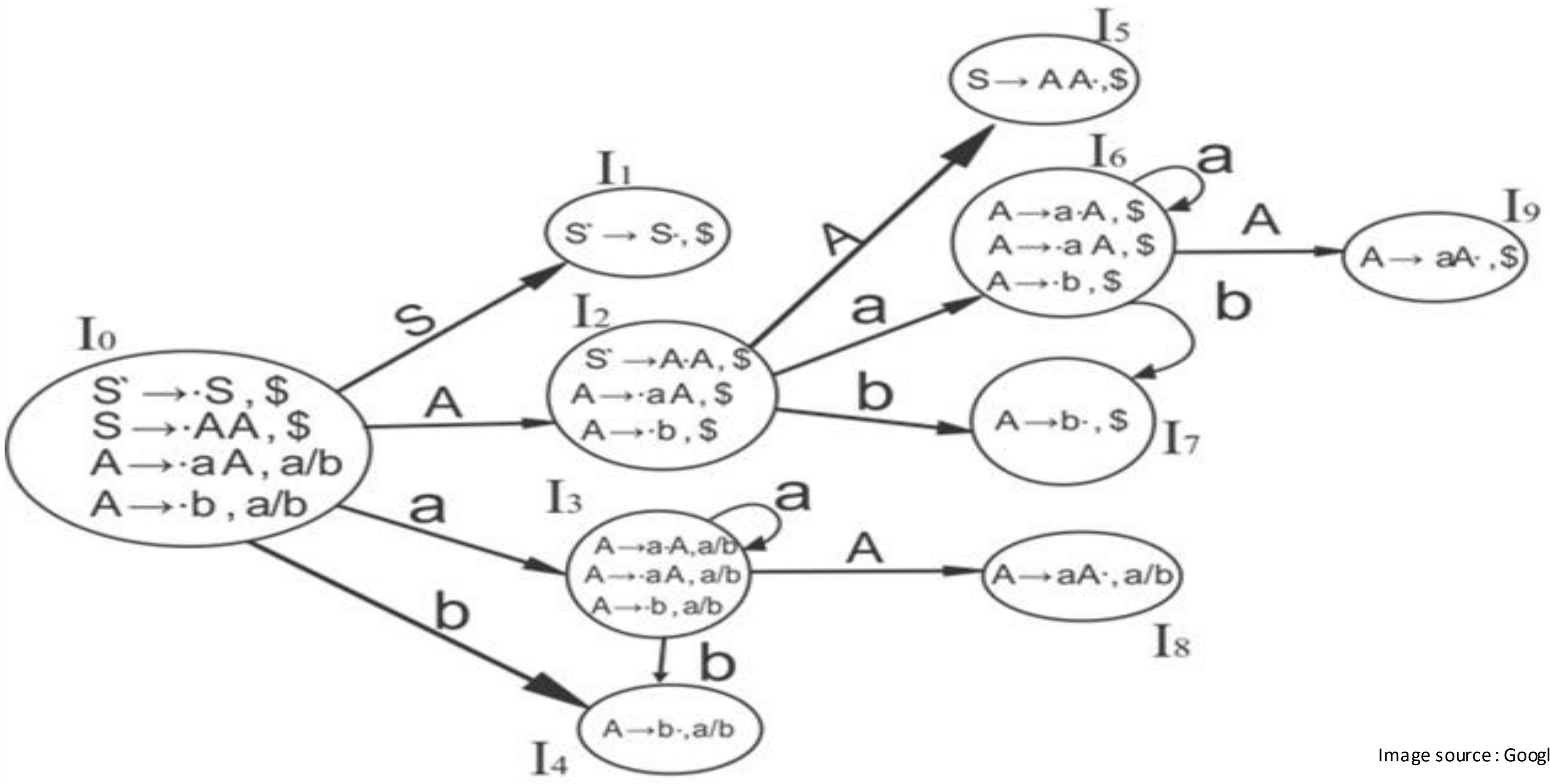


Image source : Google

CLR(1) parsing Table

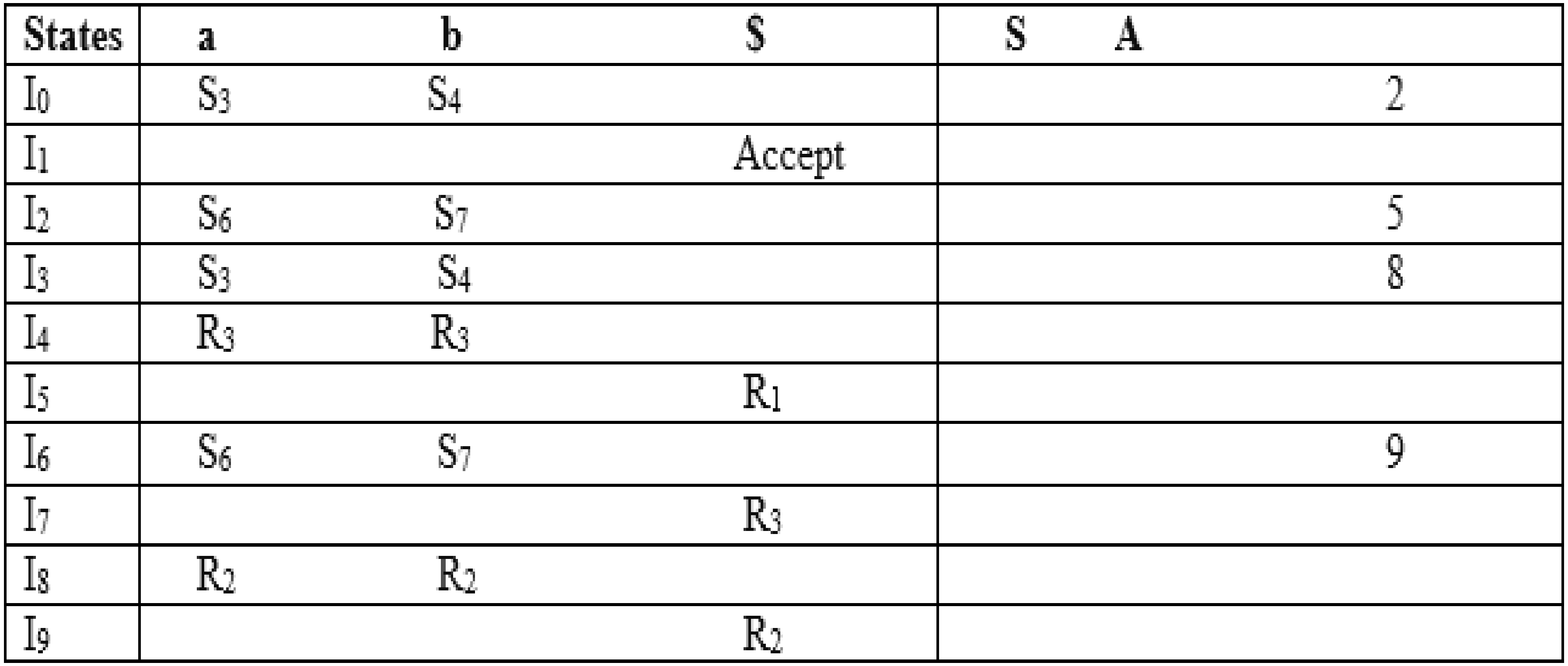
Construction of CLR(1) Parsing Table:

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

Parul® University

DIGITAL LEARNING CONTENT

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 → AA ,A → aA ,A → b

Augmented grammar with LR(1) Item

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •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.

Parul® University

DIGITAL LEARNING CONTENT

DFA for LALR(1)

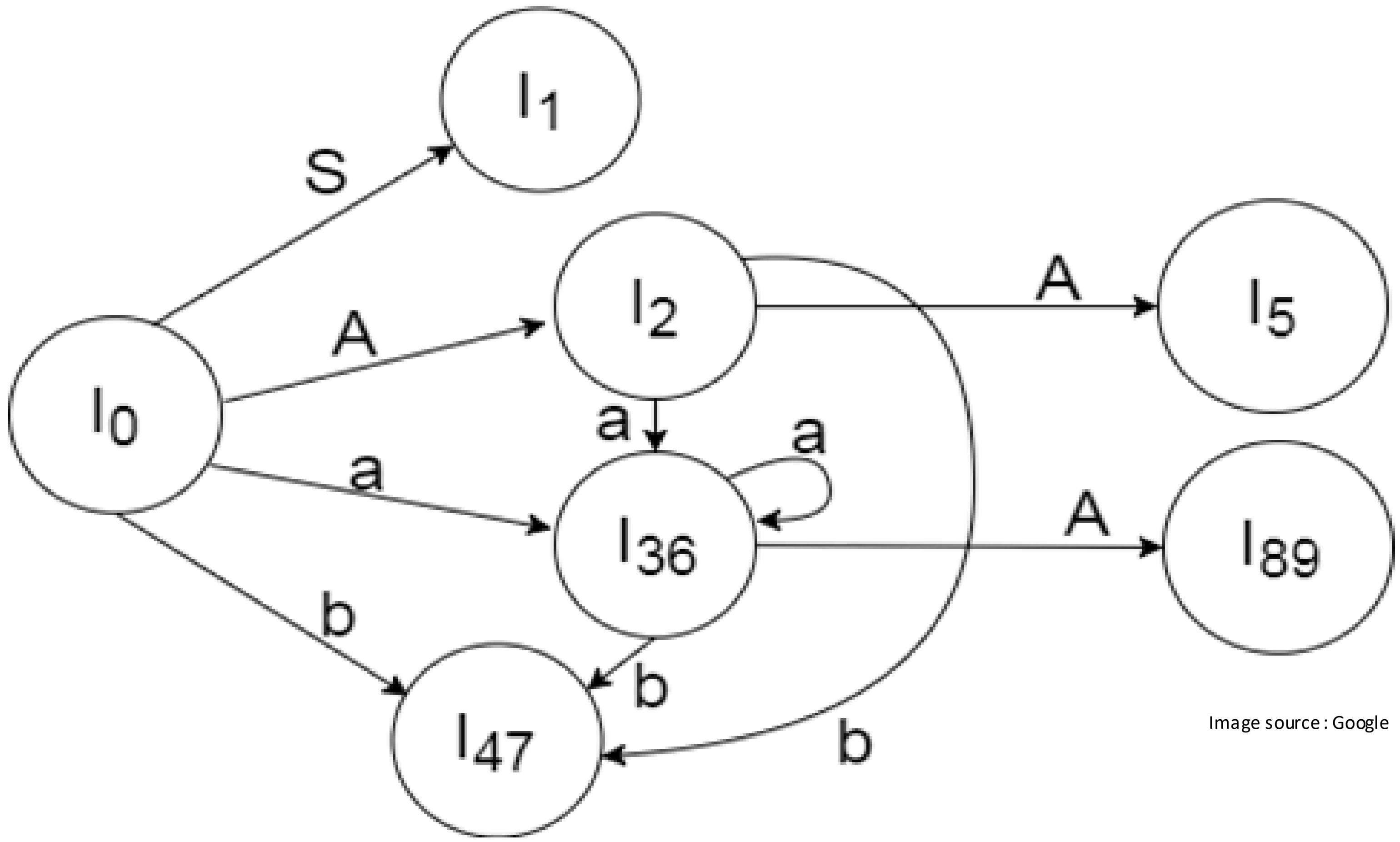


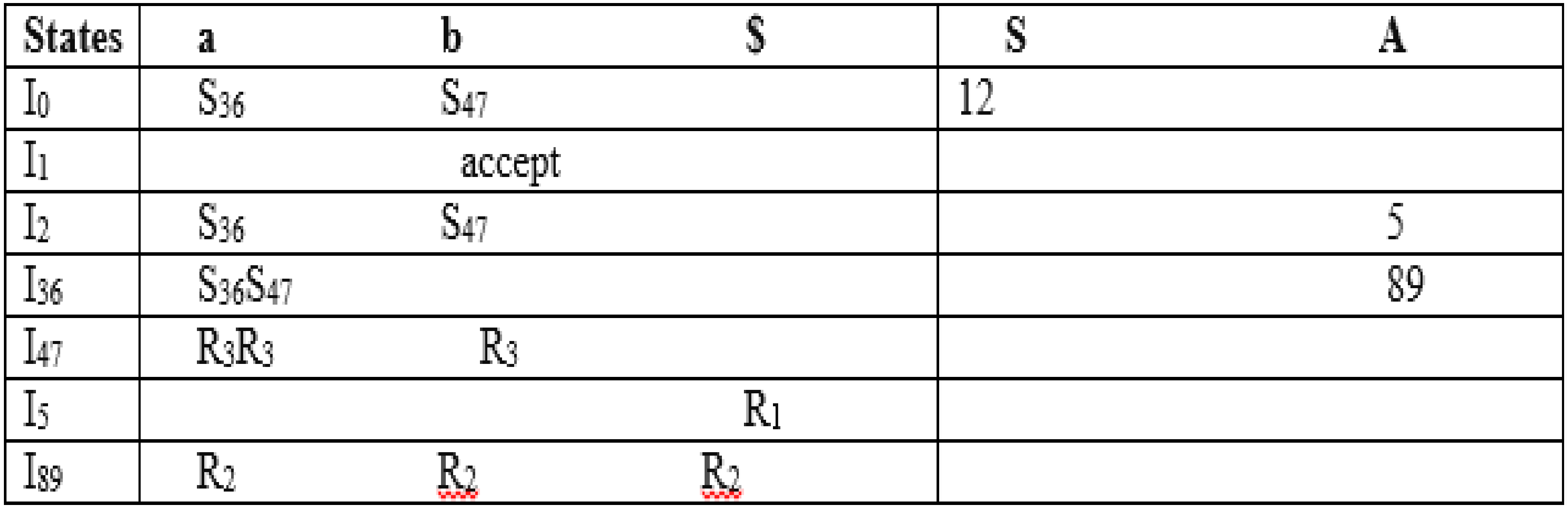
Image source : Google

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_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Image source : Google

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 ∈.
- 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:

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

$a \lessdot b$ 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:

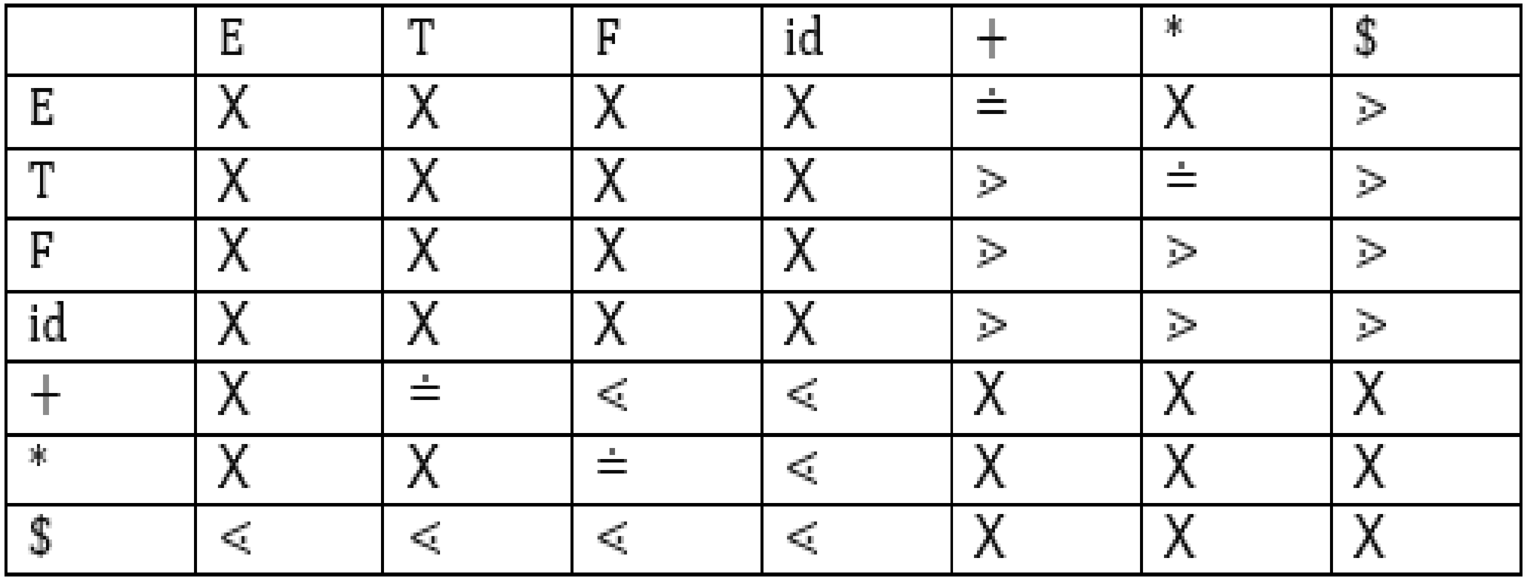
Grammar

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow id$

Operator Precedence Table

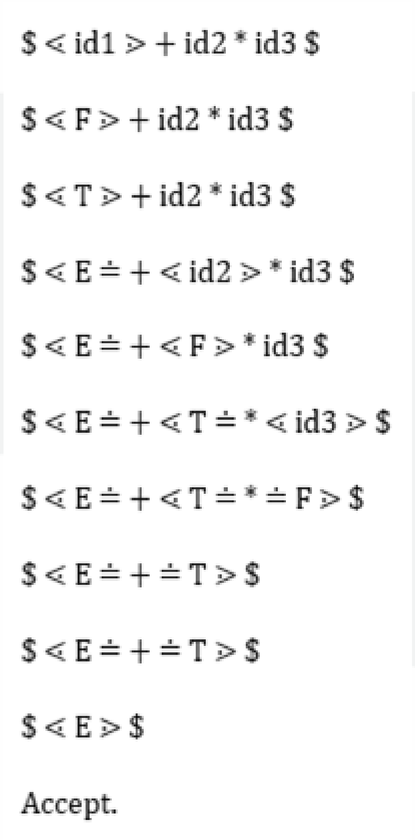


	E	T	F	id	+	*	$
E	X	X	X	X	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

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



$\$ \lessdot id1 \gtrdot + id2 * id3 \$$

$\$ \lessdot F \gtrdot + id2 * id3 \$$

$\$ \lessdot T \gtrdot + id2 * id3 \$$

$\$ \lessdot E \doteq + \lessdot id2 \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot id3 \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

Image source : Google

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

Conten

ts

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

Introduction to YACC

- 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 by LALR(1) grammar.

Parul® University

DIGITAL LEARNING CONTENT

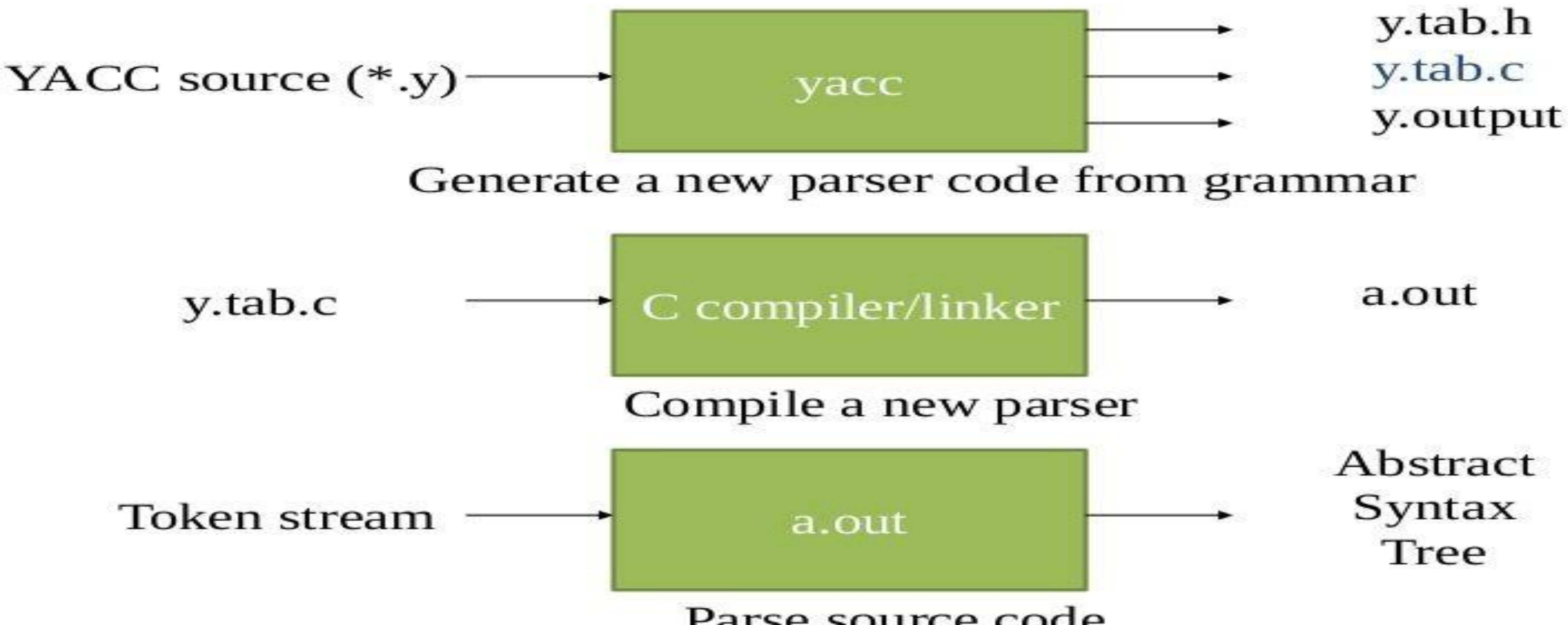
Introduction to YACC

- It generates c code of syntax analyzer of parserr.
- 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.

Parul® University

DIGITAL LEARNING CONTENT

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. Defintion

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.

Parul® University

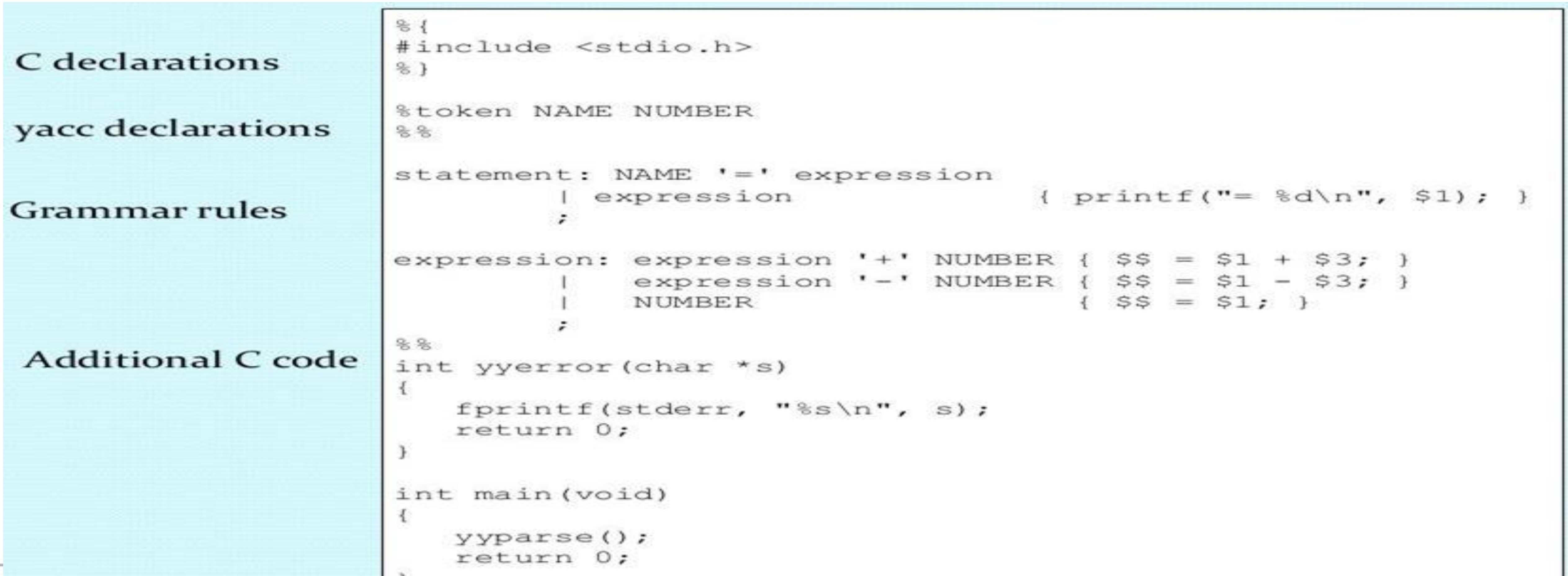
YAAC

```
%{
        C declarations
%}

        yaac declarations
%%

        Grammar rules
%%
         Additional c code
```

Example of



```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
         | expression                { printf("= %d\n", $1); }
         ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
          |  expression '-' NUMBER { $$ = $1 - $3; }
          |  NUMBER                { $$ = $1; }
          ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

int main(void)
{
   yyparse();
   return 0;
```