TOP DOWN PARSERS

C.Naga Raju

B.Tech(CSE),M.Tech(CSE),PhD(CSE),MIEEE,MCSI,MISTE

Professor

Department of CSE

YSR Engineering College of YVU

Proddatur

- Contents
- Role of the Parser
- Types of parsers
- Recursive descent parser
- Recursive Descent Predictive Parser
- Non-Recursive Predictive Parser
- GATE Problems and Solutions

The Role of the Parser

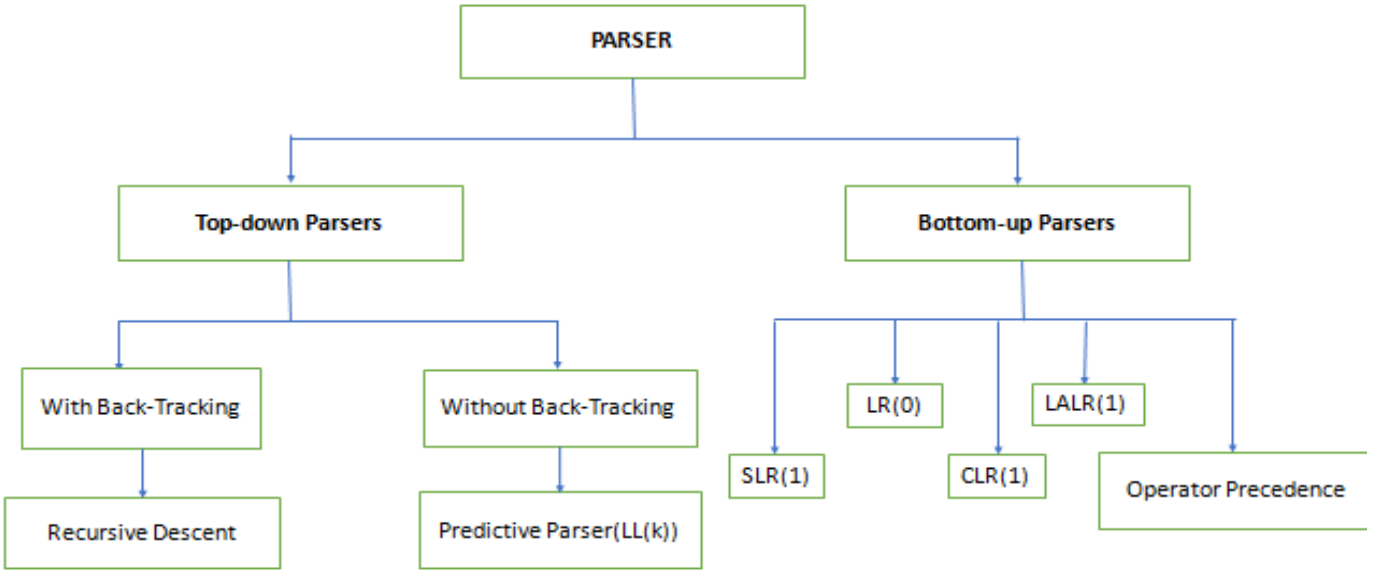
- Parsing is the process of determining, a stream of tokens generated by a grammar.
- The Main Tasks of the Parsers are to
  - ✓ Obtain a String of Tokens from the Lexical Analyzer
  - ✓ Verify the syntactic structure of the string, related to the Programming Language
  - ✓ If it finds any Syntax Errors ,send to error handling

Types of Parsers

Parsing Techniques are divided into Two general classes

Top-Down Parsing

Bottom-Up Parsing



Top-Down Parser

- Top-Down Parser parses an input string of tokens with Left-Most Derivation.
- Top-Down Parser traverse the parse tree from the Root to the Leaves with left most derivation

Ex: for the given grammar and input string

E→E + E

E→E * E

E→ id

Id * Id + Id

construction of Top-Down Parsing

E

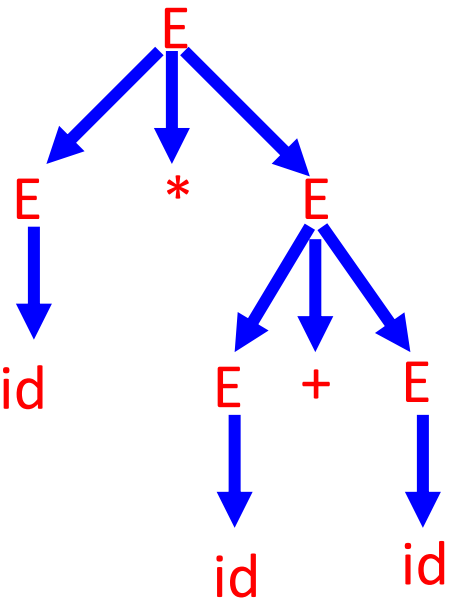
→E * E E → E * E

→Id * E E → id

→Id * E + E E → E + E

→Id * Id + E E → id

→Id * Id + Id E → id



Top-Down Parsing

- Top-Down Parsing is divided into Two Techniques
  - Recursive Descent Parser (TDP with Backtracking)
  - Predictive Parser (TDP without Backtracking)

Recursive Descent Parser

- Top-Down Parsing with Backtracking is called as Recursive Descent Parser
- It requires backtracking to find the correct production.
- It tries to find different possibilities to construct the parse tree for the given input string.
- if one possibility fails, it applies the back tracking procedure for all the given productions until to find.
- If no matches are found it sends the error message to error handling table

Example of Recursive Descent Parser (Backtracking)

Context Free Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Input String

Id – Id * Id

Example of Recursive Descent Parser (Backtracking)

E

Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

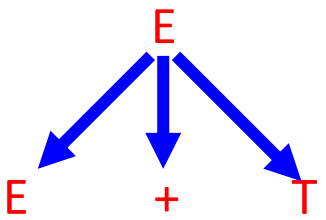
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

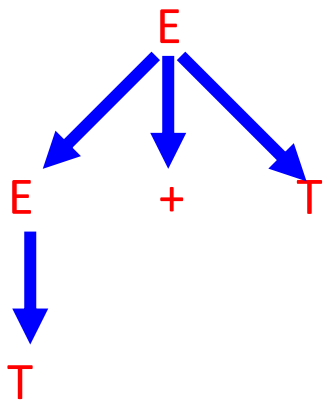
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

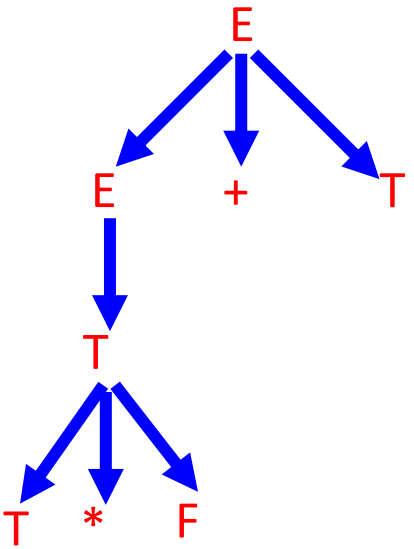
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

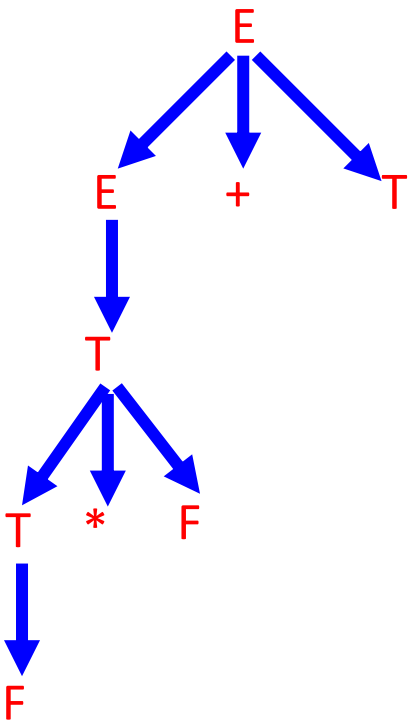
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

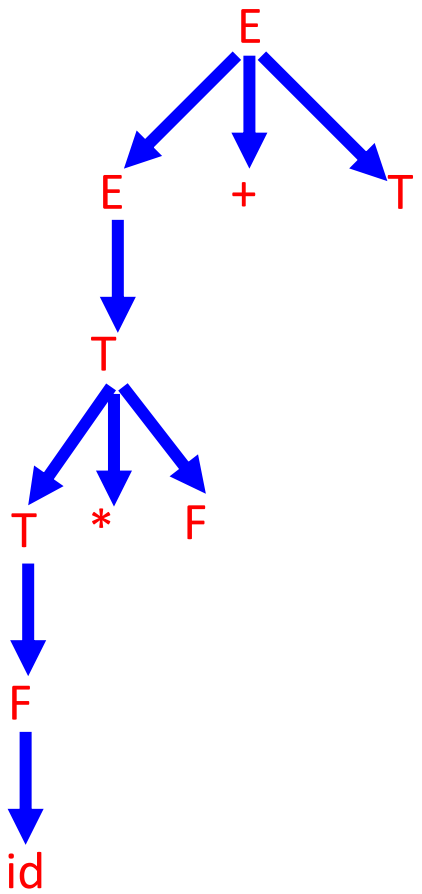
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

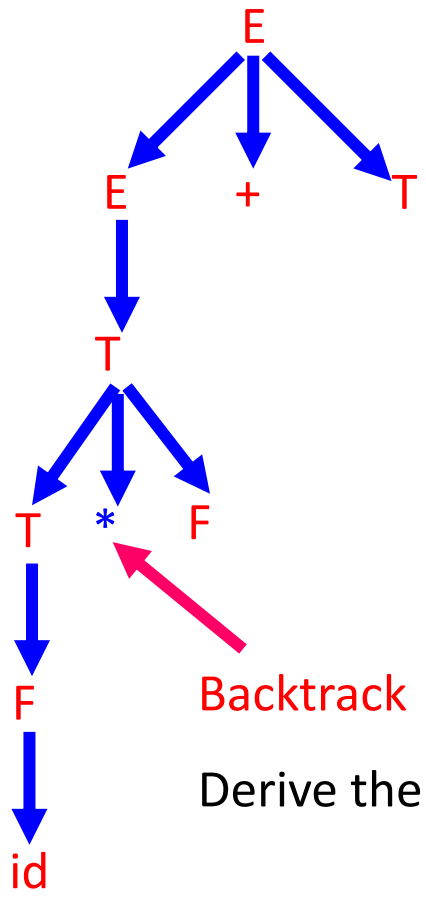
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Backtrack it ,Doesn't

Derive the input string

Example of Recursive Descent Parser (Backtracking)

E

Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

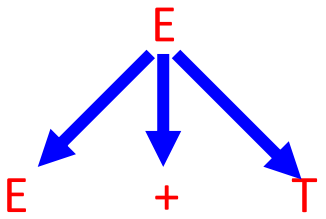
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

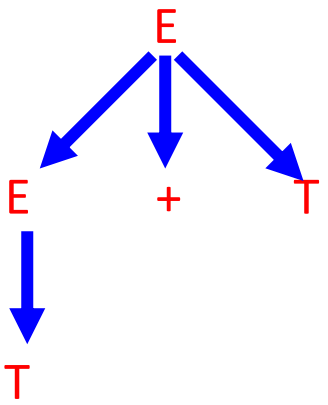
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

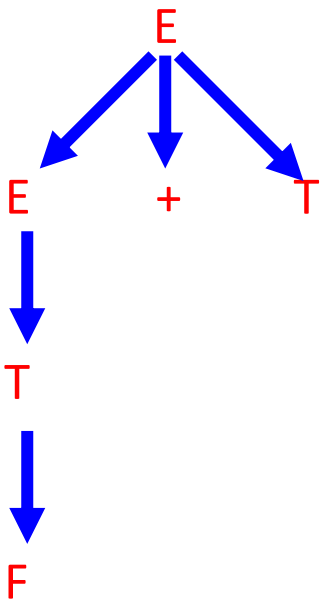
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

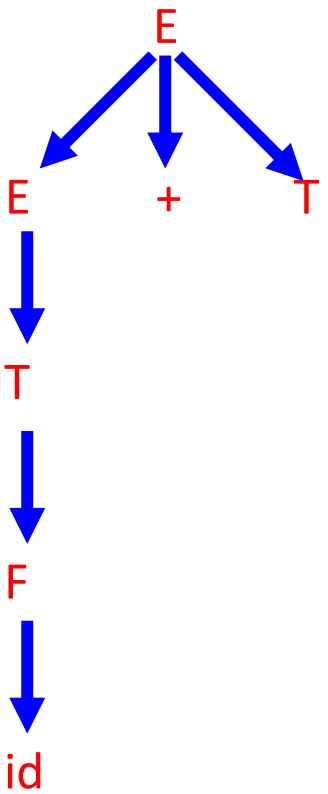
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

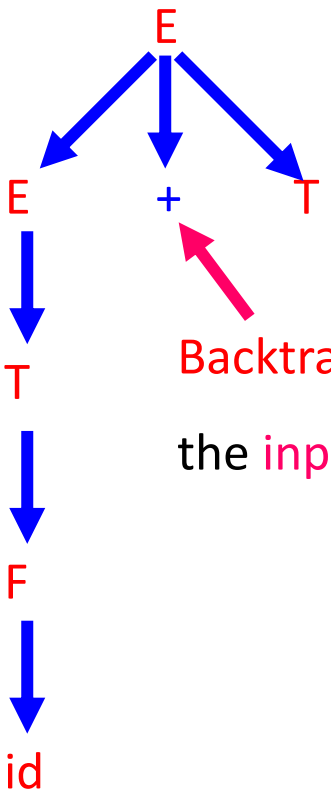
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Backtrack it ,

the input string

Input String

Id – Id * Id

Doesn't Derive

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)

E

Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

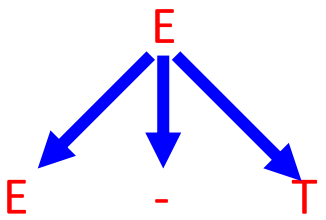
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

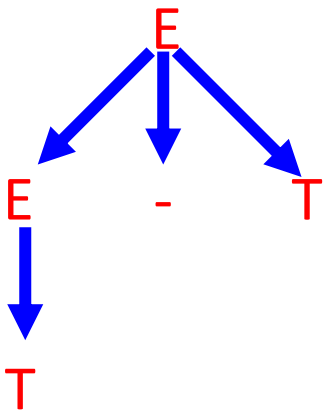
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

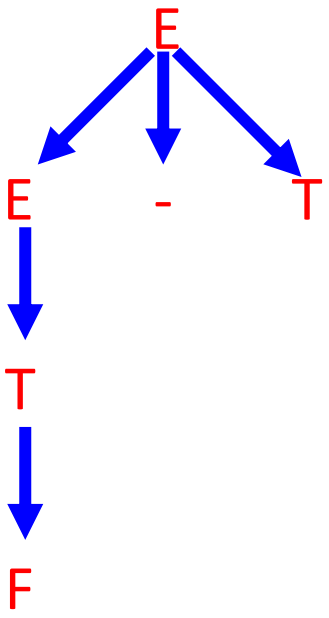
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

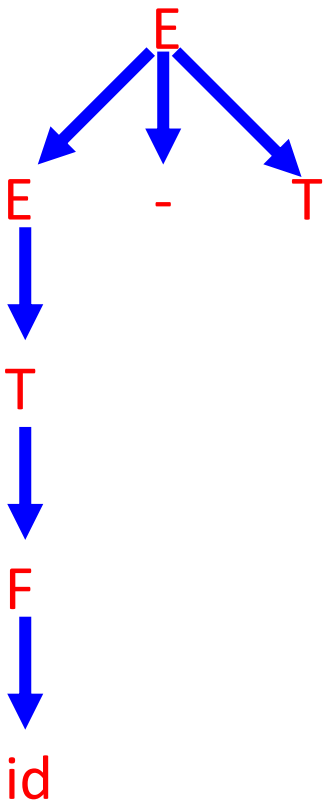
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

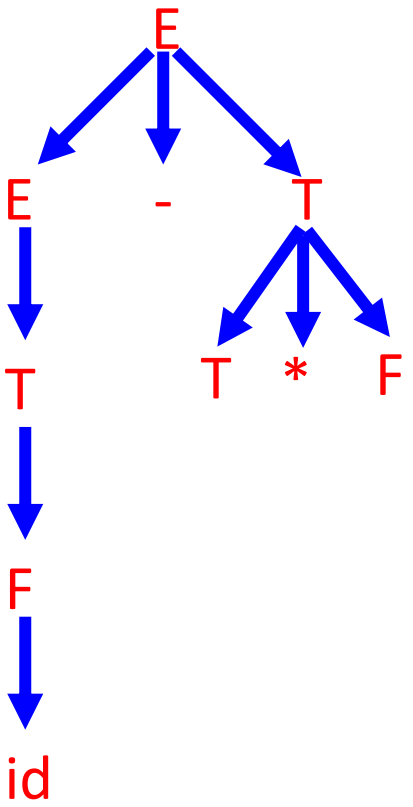
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

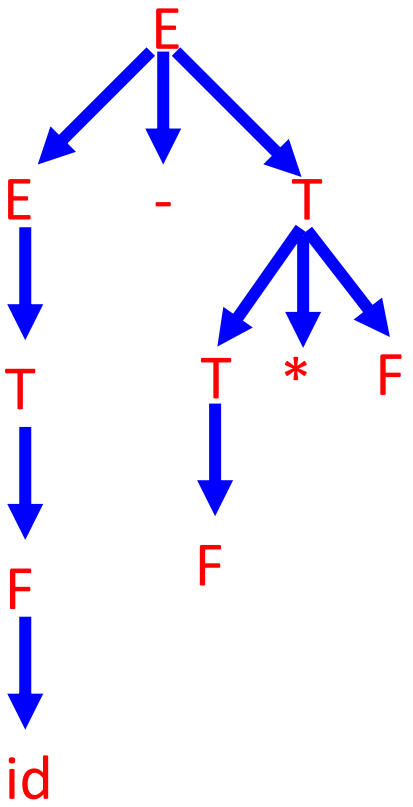
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

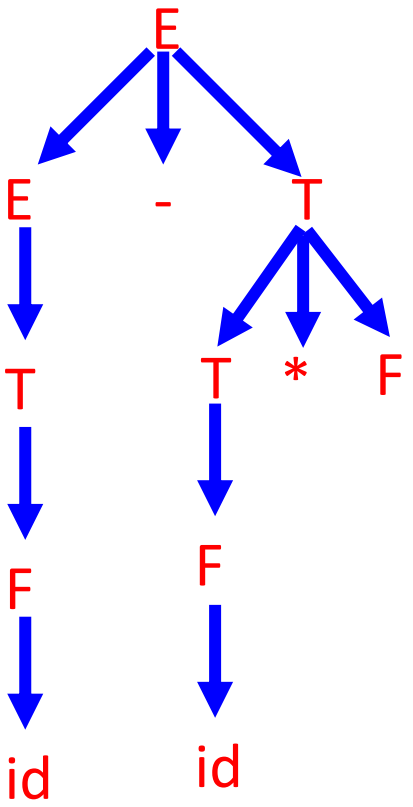
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

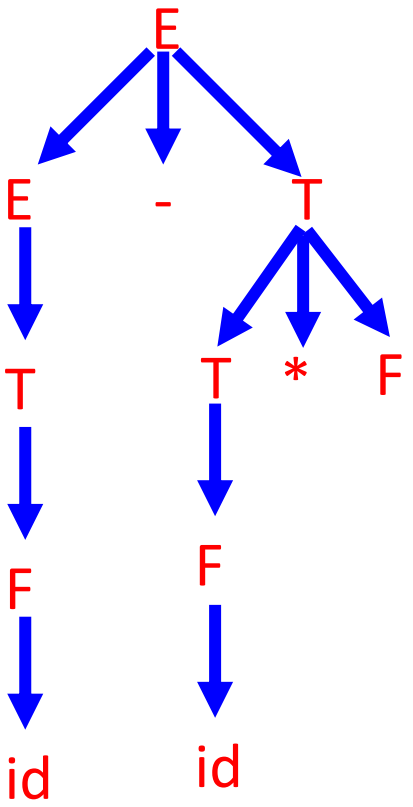
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

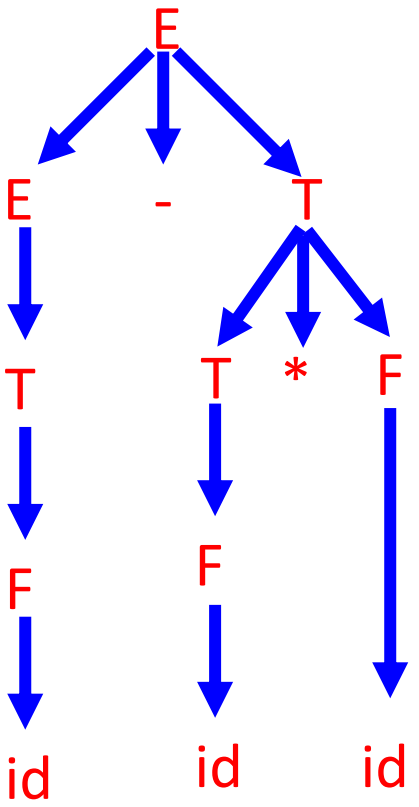
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Example of Recursive Descent Parser (Backtracking)



Input String

Id – Id * Id

Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Draw backs of Back Tracking

- It is more powerful but much slower,
- It is not suitable for practical compilers.
- Backtracking Parsers are not frequently used, because, Backtracking is not very efficient.
- An Ambiguity Grammar can cause backtracking
- Left Factoring can also cause a backtracking
- A **Left-recursive Grammar** can cause a backtracking.

Predictive Parser

- **Top-Down Parsing without Backtracking** is called as **Predictive Parser**
- It attempts to predict the next construction in the input string using one or more look-ahead tokens

Types of Predictive Parsers

There are two types of predictive parsers

- Recursive Descent Predictive Parser
- Non-Recursive Predictive Parser

Recursive Descent Predictive Parser

- Recursive-Descent Predictive Parsing is a top-down method of syntax analysis
- It executes a set of recursive procedures to process the input.
- A Procedure is associated with each non-terminal of a grammar.
- In Predictive parsing the look-ahead symbol unambiguously determines the procedure selected for each non-terminal.

- The Predictive Parsers are constructed based on Transition diagrams
- The Transition Diagram is used by Lexical Analyzer.
- Predictive Parser (TDP) also uses the Transition diagram

- Predictive Parser will take one diagram for each Non-Terminal
- The Labels of Edge are Tokens and Non-Terminals
- Predictive Parser makes Recursive procedure call for non-terminals.

Observation

- If a Predictive Parser creates Transition diagram for a Grammar, such that the Grammar is Non Left-Recursive and Non-Left Factoring

Example

Context Free Grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)$

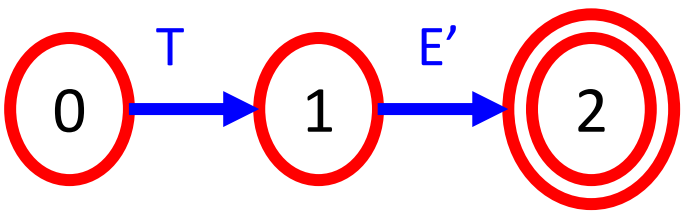
$F \rightarrow id$

Example: Transition Diagram

Context Free Grammar

$E \rightarrow TE'$

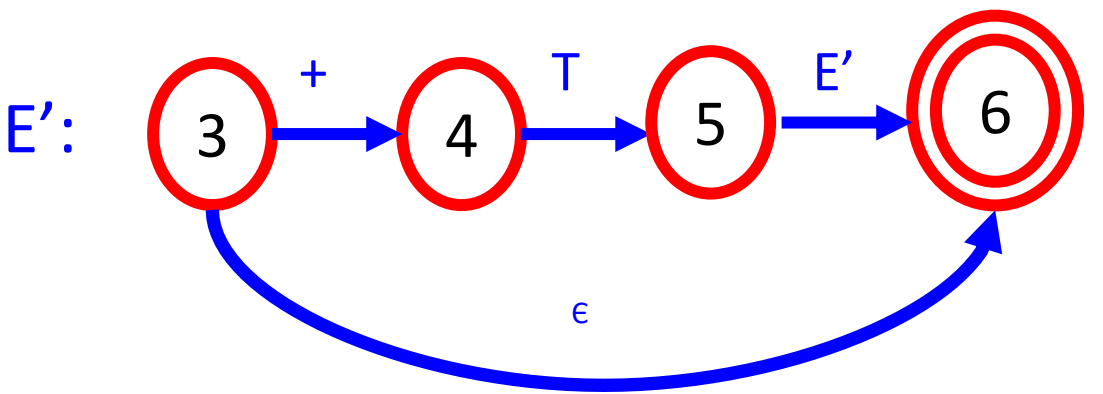
E:



Example: Transition Diagram

Context Free Grammar

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

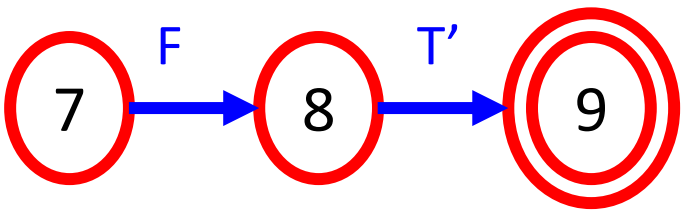


Example: Transition Diagram

Context Free Grammar

$T \rightarrow FT'$

T:

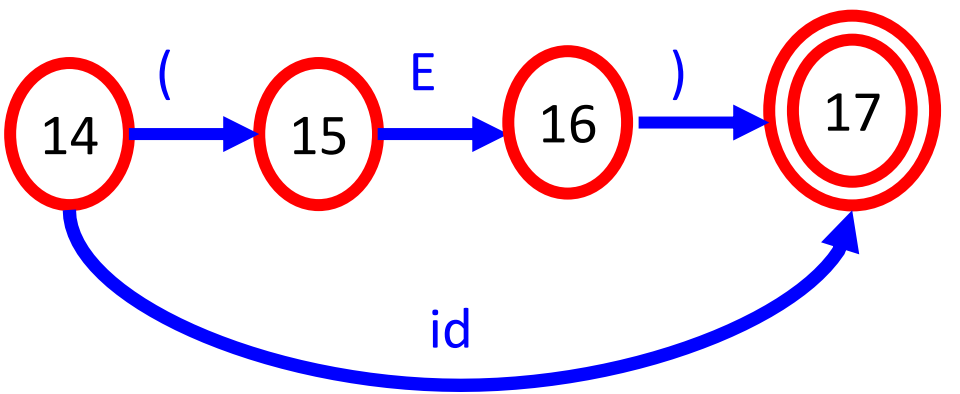


Example: Transition Diagram

Context Free Grammar

$F \rightarrow (E)$ $F \rightarrow id$

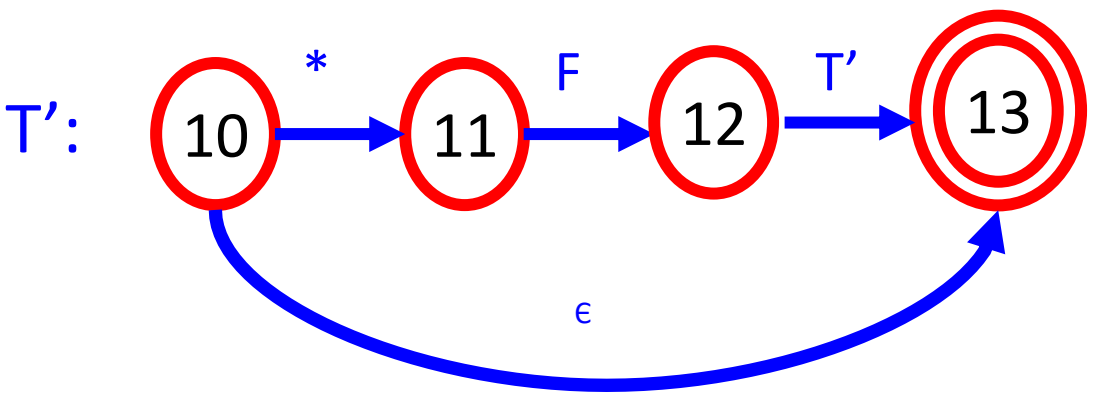
F:



Example: Transition Diagram

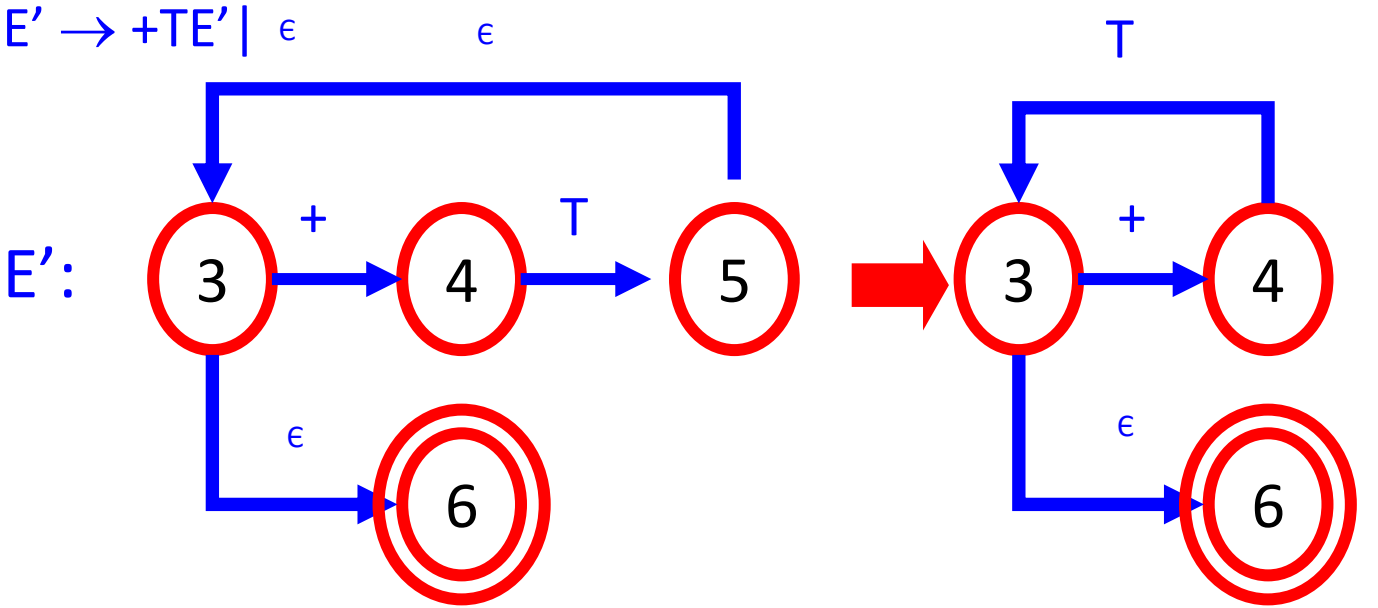
Context Free Grammar

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



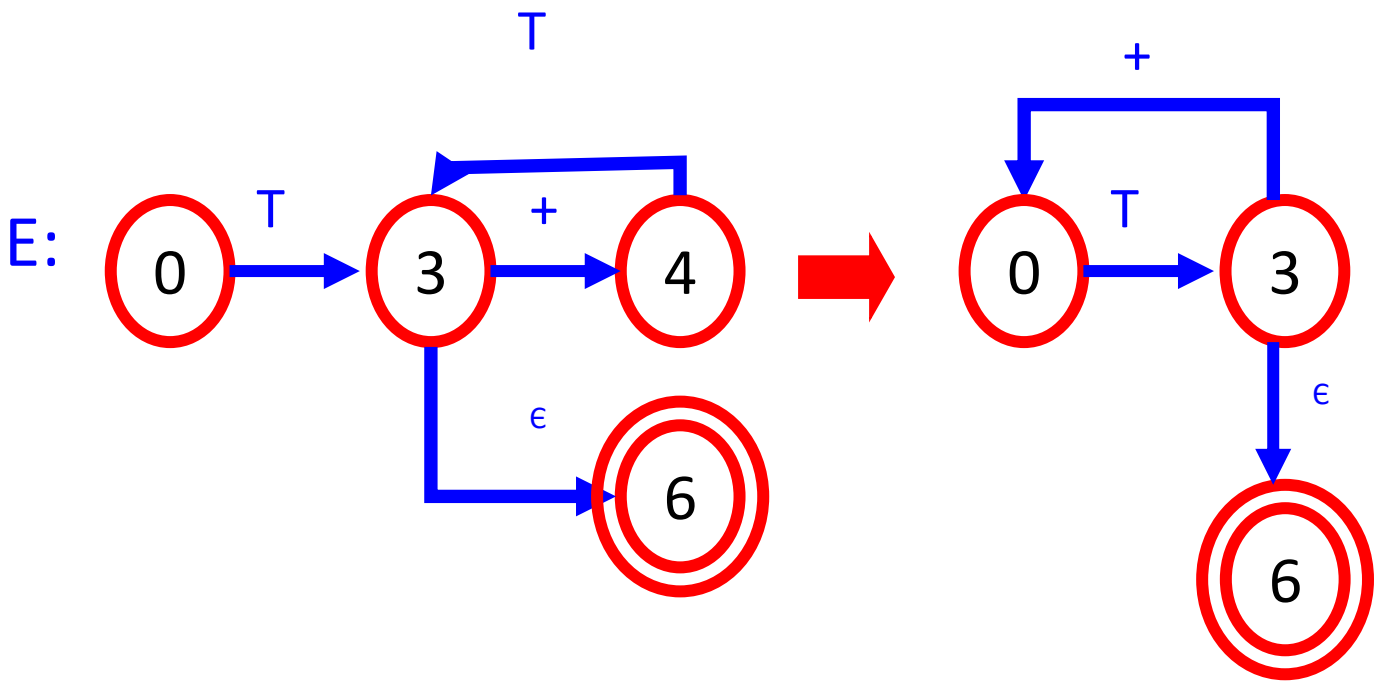
Example: Simplified Transition Diagram

Context Free Grammar



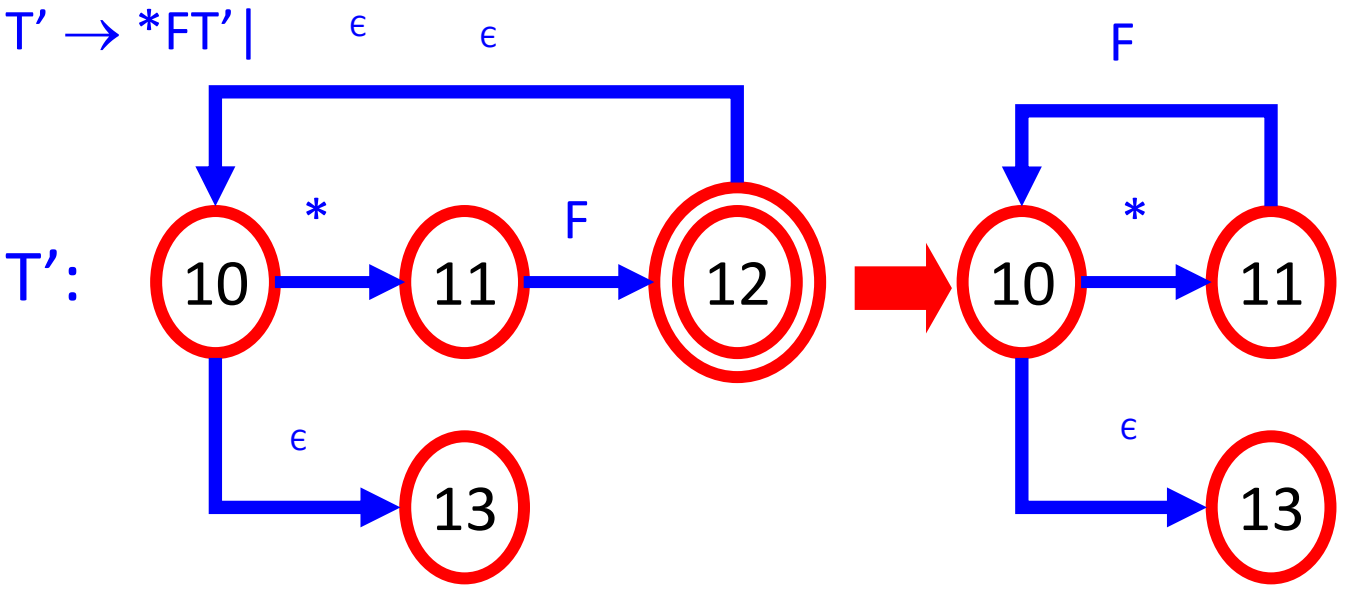
Example: Simplified Transition Diagram

E → TE’



Example: Simplified Transition Diagram

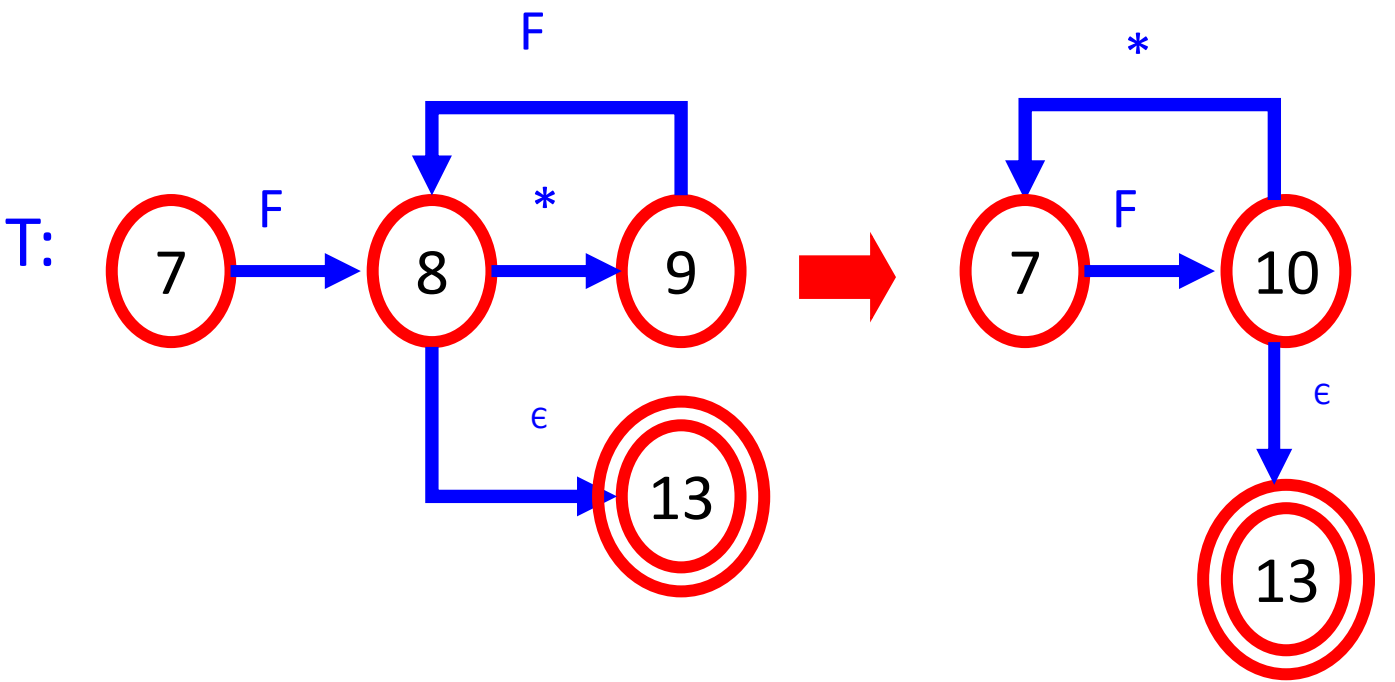
Context Free Grammar



Example: Simplified Transition Diagram

Context Free Grammar

T → FT’

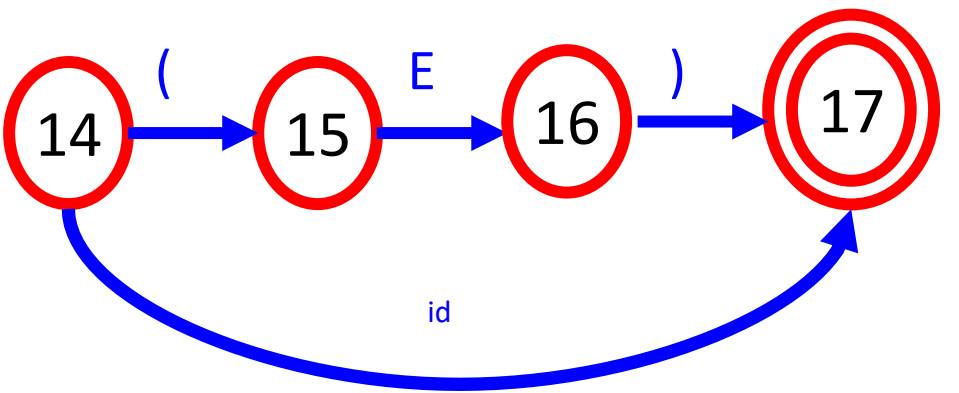


Example: Simplified Transition Diagram

Context Free Grammar

$F \rightarrow (E)$ $F \rightarrow id$

F:



Example

Context Free Grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Suffers from Ambiguity

Example

Context Free Grammar

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Ambiguity is Eliminated

Suffers from Left Recursion

Example

Context Free Grammar

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow F * T$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Left Recursion is Eliminated

Suffers from Left Factoring

Example

Context Free Grammar

$E \rightarrow T\ T'$

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

$T \rightarrow F F'$

$F \rightarrow *T \mid \epsilon$

$F \rightarrow (E)$

$F \rightarrow id$

Left Factoring is Eliminated

Example

Context Free Grammar

$E \rightarrow T \{ +T \}$

$T \rightarrow F \{ *F)$

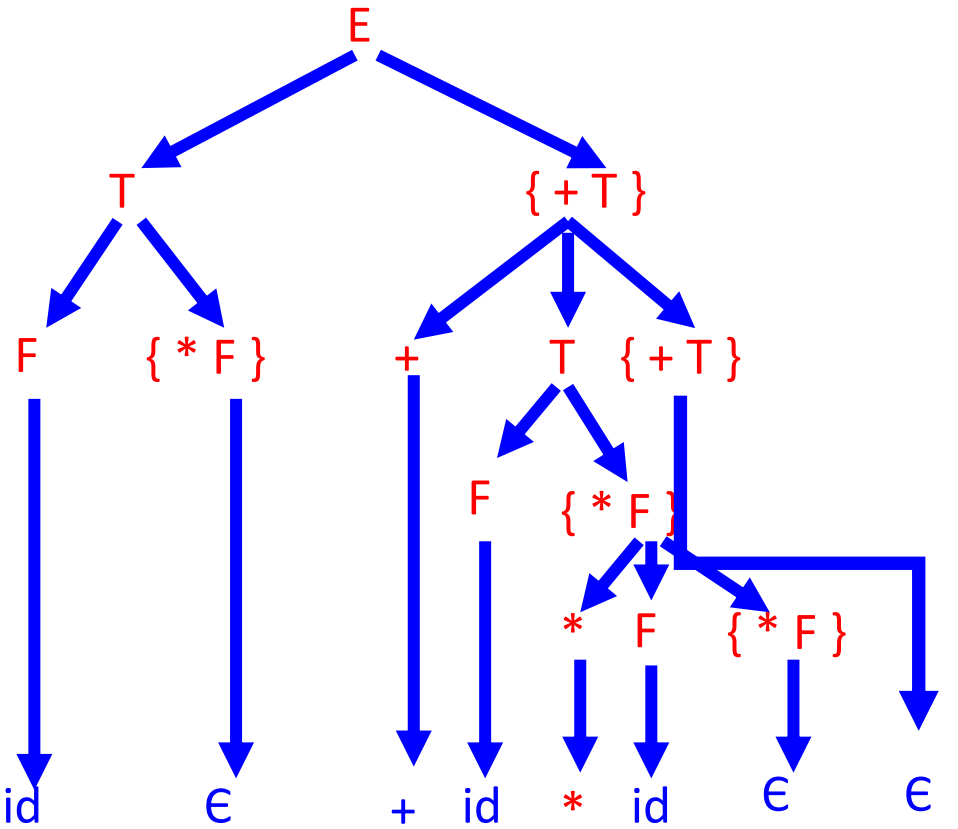
$F \rightarrow (E)$

$F \rightarrow id$

Input String

Id + Id * Id

Example: Parse Tree



Advantages of Recursive Descent Predictive Parser

- The Grammar is Un-Ambiguous Grammar
- The Grammar Free from Left Recursion
- Less Choose of Right Part
- The Grammar is Deterministic

Dis-Advantages of Recursive Descent Predictive Parser

- The Grammar is not the Standard Form of Context Free Grammar

Non-Recursive Descent Predictive Parser or LL(k)

- A Non-Recursive Predictive Parser can be built by using

  1.stack explicitly,

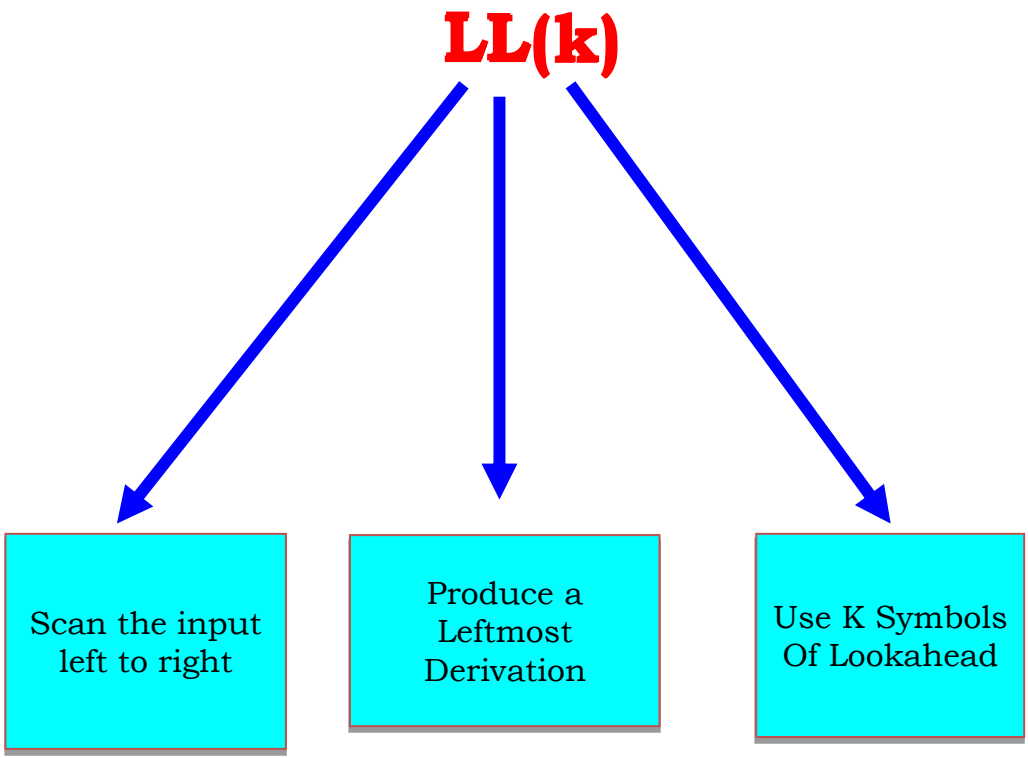
  2. but not implicitly via recursive calls.

- Non-Recursive Predictive Parser derives a leftmost derivation
- A Non-Recursive predictive parser is mainly depends on

  1.the next input symbol

  2. and the current non terminal for processing.

LL(k) Parse Tree



LL(1) Grammars

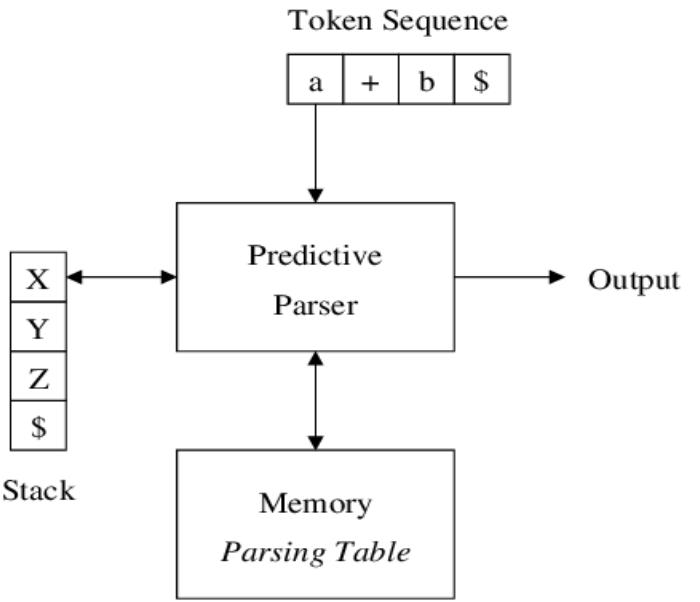
- A Grammar whose parsing table has no multiple-defined entries is said to be LL(1) Grammar.
- **First L:** scanning from left to right
- **Second L:** producing a leftmost derivation
- **1:** using one input symbol of lookahead at each step to make parsing action decision.

LL(1) Parser

- LL(1) Parsing uses an explicit stack rather than recursive calls to perform a parse
- **The predictive(LL(1)) parsing is used to** determine the productions to be applied for a non-terminal.
- This process is done by using a **Parsing Table.**

LL(1) Parser

- The Basic idea of LL(1) Parser is Table-driven and use stack



Construction of LL(1) Parsing:

- After computing the First and Follow sets for each Non-Terminal symbol we have to construct the Parsing table.
- the table Rows contain the Non-Terminals and the columns contain the Terminal Symbols.
- All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

First Function-

- First(α) is a set of terminal symbols that begin in strings derived from α.
- Rules For Calculating First Function-

Rule-01:

- For a production rule X → ∈,
- First(X) = { ∈ }

Rule-02:

- For any terminal symbol ‘a’,
- First(a) = { a }
- for a production rule X → aBAb
- first(X)={a}

Rule-03:

- For a production rule $X \rightarrow Y1Y2Y3$

Calculating First(X)

- If $\in \notin$ First(Y1), then First(X) = First(Y1)
- If $\in \in$ First(Y1), then First(X) = { First(Y1) – $\in$ } $\cup$ First(Y2Y3)

Calculating First(Y2Y3)

- If $\in \notin$ First(Y2), then First(Y2Y3) = First(Y2)
- If $\in \in$ First(Y2), then First(Y2Y3) = { First(Y2) – $\in$ } $\cup$ First(Y3)
- Similarly, we can make expansion for any production rule

$X \rightarrow Y_1Y_2Y_3.....Y_n$.

Tips:

- $\in$ may appear in the first function of a non-terminal.
- $\in$ will never appear in the follow function of a non-terminal.
- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Calculate the first and follow functions for the given grammar-

S → (L) / a

L → SL'

L' → ,SL' / ∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { (, a }

First(L) = First(S) = { (, a }

First(L') = { , , ∈ }

Follow Functions-

Follow(S) = { $ } ∪ { First(L') – ∈ } ∪ Follow(L) ∪ Follow(L') = { $, , ,) }

Follow(L) = {) }

Follow(L') = Follow(L) = {) }

Non Terminals	Input				
	a	(	)	,	$
S	S→a	S→(L)	S→(L)	S→(L)	S→(L)
L	L → SL’	L → SL’	L → SL’		
L’			L’→∈	L’ → ,SL’	

GATE QUESTIONS

1) Which of the following be sufficient to convert an arbitrary CFG to an LL(1) grammar?

A) Removing left recursion alone

B) Factoring the grammar alone

C) Removing left recursion and factoring the grammar

D) None of these

Option D

Explanation

- Removing left recursion and factoring the grammar do not be sufficient to convert an arbitrary CFG to LL(1) grammar

2) Which one of the following is a top-down parser?

A. Recursive descent parser.
B. Operator precedence parser.
C. An LR(k) parser.
D. An LALR(k) parser

Option C

Explanation

- Recursive Descent parsing is LL(1) parsing which is top down parsing.

3)Which of the following derivations does a top-down parser use while parsing an input string? The input is assumed to be scanned in left to right order (GATE CS 2000).

A. Leftmost derivation
B. Leftmost derivation traced out in reverse
C. Rightmost derivation
D. Rightmost derivation traced out in reverse

Option A

Answer (a)

- In top down parsing, we just start with the start symbol and compare the right side of the different productions against the first piece of input to see which of the productions should be used.
- A top down parser is called LL parser because it parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence.

4.For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E1, E2, and E3. |epsilon is the empty string, $ indicates end of input, and, | separates alternate right hand sides of productions.

S → a A b B | b A a B | ε
A → S
B → S

	a	b	$
S	**E1**	**E2**	S → ε
A	A → S	A → S	error
B	B → S	B → S	**E3**

(A) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b, $}

(B) FIRST(A) = {a, b, $}
FIRST(B) = {a, b, ε}
FOLLOW(A) = {a, b}
FOLLOW(B) = {$}

(C) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = ∅

(D) FIRST(A) = {a, b} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

Answer: (A)

$S \rightarrow aAbB \mid bAaB \mid \varepsilon$
$A \rightarrow S$
$B \rightarrow S$

	a	b	$
S	**E1**	**E2**	$S \rightarrow \varepsilon$
A	$A \rightarrow S$	$A \rightarrow S$	error
B	$B \rightarrow S$	$B \rightarrow S$	**E3**

Now in the above question,

FIRST(S) = { a, b, epsilon}

FIRST(A) = FIRST(S) = { a, b, epsilon}

FIRST(B) = FIRST(S) = { a, b, epsilon}

FOLLOW (A) = { b , a }

FOLLOW (S) = { $ } U FOLLOW (A) = { b , a , $ }

FOLLOW (B) = FOLLOW (S) = { b ,a , $ }

epsilon corresponds to empty string.

5.Consider the grammar given below:

S → Aa

A → BD

B → b | ε

D → d | ε

a	b	d	$
3	2	1	0

Let a, b, d, and $ be indexed as follows:

Compute the FOLLOW set of the non-terminal B and write the index values for the symbols in the FOLLOW set in the descending order. (For example, if the FOLLOW set is {a, b, d, $}, then the answer should be 3210)

Your Input _____(GATE CSE 2019)

- Follow(B) = First(D) ∪ Follow(A)

  [When D is ε then Follow(B) = Follow(A)]

  ∴ Follow(B) = {d} ∪ {a} = {a, d}

  As a = 3 and d = 1 then Descending order = 31

6.The grammar A→AA|(A)|εA→AA|(A)|ε
is not suitable for predictive-parsing because the grammar is

A. ambiguous
B. left-recursive
C. right-recursive
D. Both A and B

Option D

1.Construct the LL(1) parsers to balance the strings of patterns

Given grammar is

S->(S)S/ε

Input String:

()

CONTD…

NO	PARSING TABLE	INPUT	ACTION
1	$S	()$	S->(S)S
2	$S)S(	()$	MATCH
3	$S)S	)$	S->ε
4	$S)	)$	MATCH
5	$S	$	S->ε
6	$	$	ACCEPT

2.Construct LL(1) parser to check whether nested if-else is accepted or not

Given grammar is

S->I|E

I->i(E)SL

L->Es

L->E

E->0|1|…|9

Input String:

If(1)

If(0)

others

Else

others

S.N0	PARSING STACK	INPUT	ACTION
1	$S	i(0) i(1) 0e0$	S->I
2	$I	i(0)i(1) 0e0$	I->i(E)SL
3	$LS)E(i	i(0)i(1) 0eo$	Match
4	$LS)E(	(0)i(1) 0eo$	Match
5	$LS)E	0)i(1) 0eo$	E->0
6	$LS)0	0)i(1) 0eo$	match
7	$LS)	)i(1) 0eo$	Match
8	$LS	i(1) 0e0$	S->I
9	$LI	i(1) 0e0$	I->i(E)SL

Contd....

S.N0	PARSING STACK	INPUT	ACTION
10	$LLS)E(i	i(1) 0e0$	Match
11	$LLS)E(	(1) 0e0$	match
12	$LLS)E	1) 0e0$	E->1
13	$LLS)1	1)0e0$	match
14	$LLS)	) 0e0$	Match
15	$LLS	0e0$	S->0
16	$LL0	0e0$	Match
17	$LL	e0$	L->Es
18	$LSe	e0$	match

Contd....

S.N0	PARSING STACK	INPUT	ACTION
19	$LS	0$	S->0
20	$L0	0$	MATCH
21	&L	$	L->E
22	$	$	ACCEPT

Problem-01:

Calculate the first and follow functions for the given grammar-

S → A

A → aB / Ad

B → b

C → g

Solution-

We have-

The given grammar is left recursive.

So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' / \in$

$B \rightarrow b$

$C \rightarrow g$

- Now, the first and follow functions are as follows-

First Functions-

First(S) = First(A) = { a }

First(A) = { a }

First(A') = { d , ∈ }

First(B) = { b }

First(C) = { g }

Follow Functions-

Follow(S) = { $ }

Follow(A) = Follow(S) = { $ }

Follow(A') = Follow(A) = { $ }

Follow(B) = { First(A') – ∈ } ∪ Follow(A) = { d , $ }

Follow(C) = NA

Problem-02:

Calculate the first and follow functions for the given grammar-

S → AaAb / BbBa

A → ∈

B → ∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { First(A) – ∈ } ∪ First(a) ∪ { First(B) – ∈ } ∪ First(b) = { a , b }

First(A) = { ∈ }

First(B) = { ∈ }

Follow Functions-

Follow(S) = { $ }

Follow(A) = First(a) ∪ First(b) = { a , b }

Follow(B) = First(b) ∪ First(a) = { a , b }

Problem-03:

Calculate the first and follow functions for the given grammar-

$E \rightarrow E + T \mid T$

$T \rightarrow T \times F \mid F$

$F \rightarrow (E) \mid id$

Solution-

We have-

The given grammar is left recursive.

So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Now, the first and follow functions are as follows-

First Functions-

First(E) = First(T) = First(F) = { (, id }

First(E') = { + , ∈ }

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

First(T') = { x , ∈ }

First(F) = { (, id }

Follow Functions-

Follow(E) = { $,) }

Follow(E') = Follow(E) = { $,) }

Follow(T) = { First(E') – ∈ } ∪ Follow(E) ∪ Follow(E') = { + , $,) }

Follow(T') = Follow(T) = { + , $,) }

Follow(F) = { First(T') – ∈ } ∪ Follow(T) ∪ Follow(T') = { x , + , $,) }

Problem-04:

Calculate the first and follow functions for the given grammar-

S → ACB |CbB | Ba

A → da |BC

B → g | ∈

C → h |∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { First(A) – ∈ } ∪ { First(C) – ∈ } ∪ First(B) ∪ First(b) ∪ { First(B) – ∈ } ∪ First(a) = { d , g , h , ∈ , b , a }

First(A) = First(d) ∪ { First(B) – ∈ } ∪ First(C) = { d , g , h , ∈ }

First(B) = { g , ∈ }

First(C) = { h , ∈ }

Follow Functions-

Follow(S) = { $ }

Follow(A) = { First(C) – ∈ } ∪ { First(B) – ∈ } ∪ Follow(S) = { h , g , $ }

Follow(B) = Follow(S) ∪ First(a) ∪ { First(C) – ∈ } ∪ Follow(A) = { $, a , h , g }

Follow(C) = { First(B) – ∈ } ∪ Follow(S) ∪ First(b) ∪ Follow(A) = { g , $, b , h }

Example

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

1. It is not in left recursion so we can find FIRST and FOLLOW functions

2. FIRST function

- First(S)=First(aBDh) ➔ {S → aBDh}
  =First(a)----------Rule 2:First(terminal)={terminal}
  ={a}
- First(B)=First(cC) ➔ {B → cC}
  =First(c)----------Rule 2:First(terminal)={terminal}
  ={c}
- First(C) = First(bC) ➔ {C→bC}
  =First(b)----------Rule 2:First(terminal)={terminal}
  ={b}
- First(C) = First(∈) ➔ {C→ ∈}
  =First(∈)----------Rule 1:First(∈)={∈}
  ={∈}

- First(D) =First(EF) ➔ {D→EF}
  ={ First(E) – ∈ } ∪ First(F) = { g , f }
  ={ g , f }
- First(D) =First(∈) ➔ {E→∈} && {F→∈}
  ={∈} ----------Rule 1:First(∈)={∈}
- First(E) =First(g) ➔ {E→g}
  ={g} ----------Rule 2:First(terminal)={terminal}
- First(E) =First(∈) ➔ {E→∈}
  ={∈} ----------Rule 1:First(∈)={∈}
- First(F) =First(f) ➔ {F→f}
  ={f} ----------Rule 2:First(terminal)={terminal}
- First(F) =First(∈) ➔ {F→∈}
  ={∈ } ----------Rule 1:First(∈)={∈}

The First Function for the grammar is

NON-TERMINALS	FIRST
S	{ a }
B	{ c }
C	{ b , ∈ }
D	{ g , f , ∈}
E	{ g , ∈ }
F	{ f , ∈ }

Follow Function-

- Follow(α) is a set of terminal symbols that appear immediately to the right of α.
- Rules For Calculating Follow Function-

Rule-01:

- For the start symbol S, place $ in Follow(S).

Rule-02:

- For any production rule A → αB,
- Follow(B) = Follow(A)

Rule-03:

- For any production rule A → αBβ,
- If ∈ ∉ First(β), then Follow(B) = First(β)
- If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Tips:

- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.
- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

2. FOLLOW function

- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

Follow(S) = {$}

- Follow(S)={$, For the start symbol S, place $ in Follow(S)

S is not present on R.H.S.

S → aBDh

B → cC

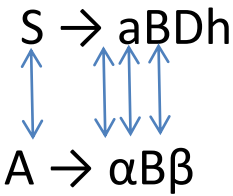
C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

Follow(B)= {g,f,h}



Here β=Dh

If ∈ ∉ First(D), then Follow(B) = First(β)=First(D)

={g,f

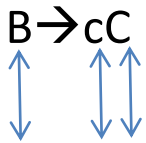
If ∈ ∈ First(D), then Follow(B) = { First(Dh) – ∈ } ∪ Follow(S)

=First{h}

={h

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

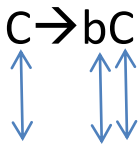
Follow(C)= {g,f,h}



In the form of A → α B

For A → αB, then Follow(B) = Follow(A)

So, Here Follow(C)=Follow(B)
={g,f,h



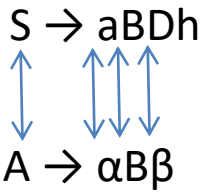
In the form of A → α B

For A → αB, then Follow(B) = Follow(A)

So, Here Follow(C)=Follow(C)
={g,f,h}

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

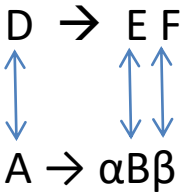
Follow(D)= {h}



- Here β=h
- If ∈ ∉ First(D), then Follow(B) = First(β)=First(h)
  ={h}

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

Follow(E)= {f,h}



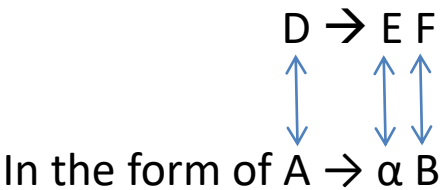
Here β=F

•If ∈ ∉ First(F), then Follow(D) = First(β)=First(F)
= {f

•If ∈ ∈ First(F), then Follow(D) = { First(F) – ∈ } ∪ Follow(D)
={h

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

Follow(F)= {g,f,h}



- For A → αB, then Follow(B) = Follow(A)
- So, Here Follow(F)=Follow(D)
  ={g,f,h

The Follow Function for the grammar is

NON-TERMINALS	FOLLOW
S	{ $ }
B	{ g , f , h }
C	{ g , f , h }
D	{ h }
E	{ f , h }
F	{ h }

First and Follow functions

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

NON-TERMINALS	FIRST	FOLLOW
S	{ a }	{ $ }
B	{ c }	{ g , f , h }
C	{ b , ∈ }	{ g , f , h }
D	{ g , f , ∈ }	{ h }
E	{ g , ∈ }	{ f , h }
F	{ f , ∈ }	{ h }

Non Terminals	Input						
	a	**b**	**c**	**g**	**f**	**h**	**$**
S	S → aBDh						S → aBDh
B			B → cC	B → cC	B → cC	B → cC	
C		C → bC		C →∈	C →∈	C →∈	
D				D→ EF	D → EF	D → EF	
E				E → g	E → ∈	E → ∈	
F					F → f	F → ∈	

THANK YOU