UPES

UNIVERSITY WITH A PURPOSE

Content of this lecture

- Introduction to Parsing Technique
- Top down parser

UPES
UNIVERSITY WITH A PURPOSE

Parsing Techniques

- Parsing technique is used to check tokens generated are satisfying the grammatical rules of the language or not, if they satisfy then corresponding Parse tree is generated otherwise error is reported to error handler.
- Based on the methodology adopted for the generation of parse tree, parsing techniques can be categorized in to two categories:
  - Top down parser
  - Bottom parser

Parsing Techniques

- Top Down Parser
  - Input is scanned from left to right one symbol at a time
  - Build the parse tree from top(root) to bottom(leaves)
  - We have to decide which rule A → β should be applied to a node labelled A.
  - Expanding A results in new nodes (children of A) labelled by symbols in β
  - It can be also viewed as finding a leftmost derivation for an input string
- Bottom up Parser
  - Input is scanned from left to right one symbol at a time
  - Build the parse tree from bottom(leaves) to top(root).
  - Input string will be reduced to starting symbol.
  - Reducing rule A → β results in adding a node A to the tree. A's children are the nodes labelled by the symbols in β

UPES
UNIVERSITY WITH A PURPOSE

Parsing Techniques

- Consider the grammar G1 = ({E, F, T}, {a, (,), ∗,+}, P, E) where the productions are:

  E → E + T

  E → T

  T → T ∗ F

  T → F

  F → (E)

  F → a

and the string x = a + a ∗ a

Parsing Techniques

- Top-down method: rules are considered in the same order as a leftmost derivation

$E \Rightarrow \underline{E} + T$

$E \Rightarrow \underline{T} + T$

$E \Rightarrow \underline{F} + T$

$E \Rightarrow a + \underline{T}$

$E \Rightarrow a + \underline{T} * F$

$E \Rightarrow a + \underline{F} * F$

$E \Rightarrow a + a * \underline{F}$

$E \Rightarrow a + a * a$

UPES
UNIVERSITY WITH A PURPOSE

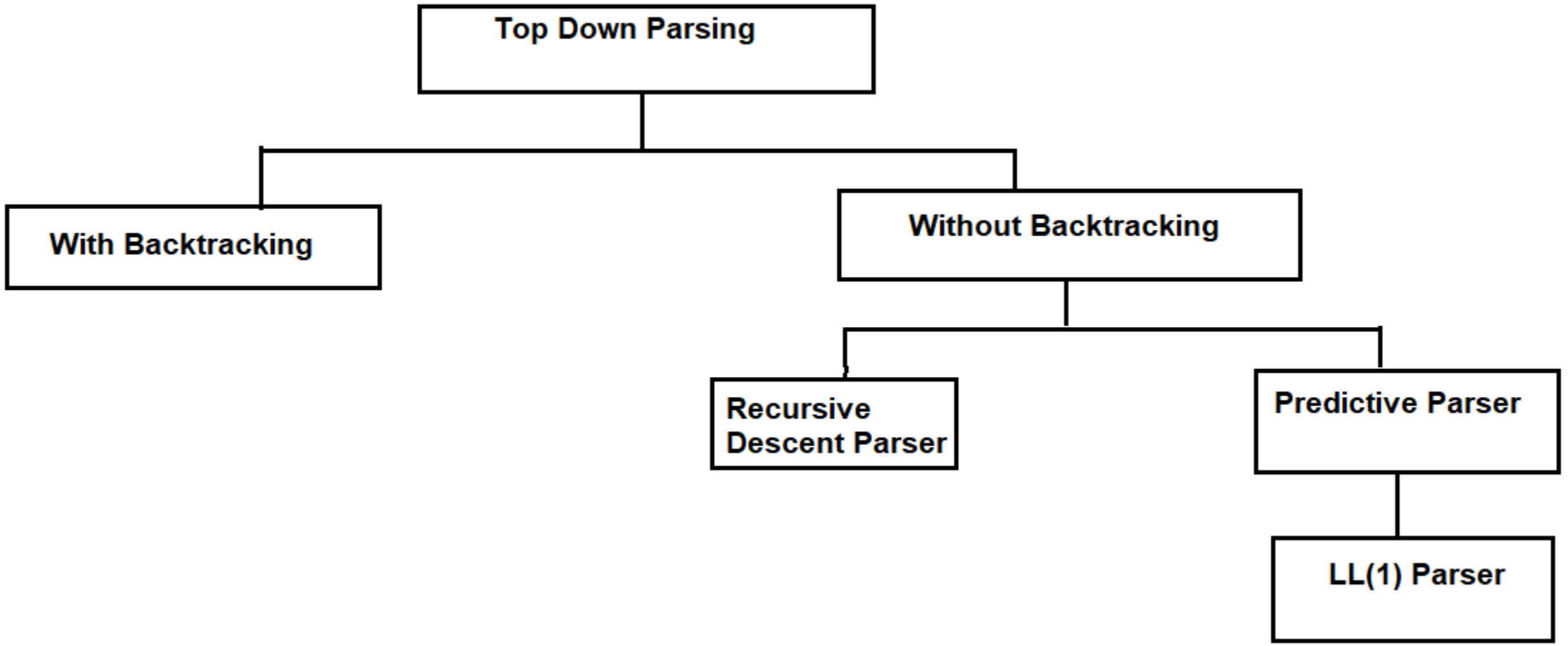
Parsing Techniques

- Bottom-up method: rules are considered in the same order as a reverse rightmost derivation

$$
\begin{aligned}
& a + a * a \\
\Leftarrow\ & \underline{F} + a * a \\
\Leftarrow\ & \underline{T} + a * a \\
\Leftarrow\ & \underline{E} + a * a \\
\Leftarrow\ & E + \underline{F} * a \\
\Leftarrow\ & E + \underline{T} * a \\
\Leftarrow\ & E + T * \underline{F} \\
\Leftarrow\ & E + \underline{T} \\
\Leftarrow\ & \underline{E}
\end{aligned}
$$

UPES
UNIVERSITY WITH A PURPOSE

Top Down Parsing Techniques



UPES
UNIVERSITY WITH A PURPOSE

Top Down Parsing with Backtracking

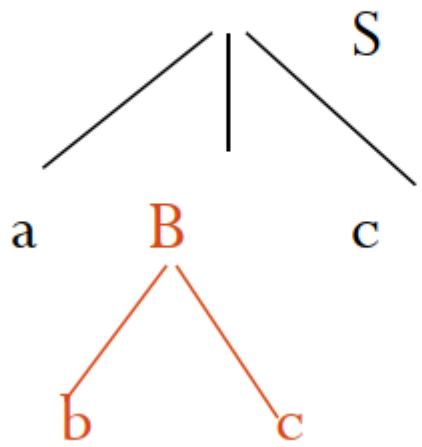
- In this category, parser can make repeated scans of input.
- If required input string is not achieved by applying one production rule then another production rule can be applied at each step to get the required string.

Top Down Parsing with Backtracking

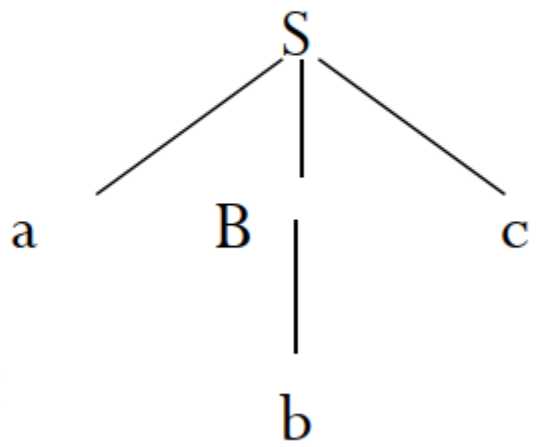
$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



fails, backtrack



Top Down Parsing with Backtracking Algorithm

- S→aAd
- A→bc|b

Top Down Parsing with Backtracking Algorithm

- S→aAd
- A→bc|b

```
Procedure S ()
{
        if (input symbol = 'a') then
        {
                advance();
                if A() then
                {
                        if (input symbol='d') then
                        {
                                advance();
                                declare success;
                        }
                        else error;
                }
                else error;
        }
        else error
}
```

Top Down Parsing with Backtracking Algorithm

- S→aAd
- A→bc|b

```
Procedure S ()
{
        if (input symbol = 'a') then
        {
                advance();
                if A() then
                {
                        if (input symbol='d') then
                        {
                                advance();
                                declare success;
                        }
                        else error;
                }
                else error;
        }
        else error
}
```

```
Procedure A ()
{
        if (input symbol = 'b') then
        {
                advance();
                if (input symbol='c') then
                {
                        advance();
                        declare success;
                }
                else error;
        }
        if (input symbol = 'b') then
        {
                advance();
                declare success;
        }
        else error;
}
```

UPES
UNIVERSITY WITH A PURPOSE

Top Down Parsing with Backtracking- Limitations

- Its not efficient as undoing semantic action requires lot of overhead.
- Entries made in the symbol table during parsing have to be removed while backtracking.
- The order in which the alternative production rules are applied can also affect the efficiency.
- Existence of left recursion will cause parser to enter into infinite loop. This must me removed.
- Its difficult to locate the position of error on occurrence of failure.

Top Down Parsing without Backtracking

- In this category, once the production rule is applied, it can't be undone. The parser which follows top down parsing without backtracking are:
  - Recursive Descent Parser
  - Predictive Parser
  - LL(1) parser

- Need to remove Left recursion and Left factoring from grammar before doing top down parsing without backtracking.

Recursive Descent Parser

- It is defined as a parser that uses various recursive procedures to process the input string with no backtracking.
- It can be easily implemented using a language which is recursive in nature.
- The first symbol of the string on R.H. S. of a production will uniquely determines the correct alternative to choose.
- In this technique, there is no need for the procedures to return an indication of success or failure since the calling procedure has no scope of trying another alternate. Rather on failure, an error correcting routine is called.

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

  E -> TE’

  E’ -> +TE’ | Ɛ

  T -> FT’

  T’ -> *FT’ | Ɛ

  F -> (E) | **id**

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

F -> (E) | **id**

```
Procedure E()
{
        T();
        E’();
}
```

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

F -> (E) | **id**

```
Procedure E()
{
        T();
        E’();
}
Procedure E’ ()
{
        if (input symbol = +) then
        {
                advance();
                T();
                E’();
        }
}
```

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

F -> (E) | **id**

```
Procedure E()
{
    T();
    E’();
}
Procedure E’ ()
{
    if (input symbol = +) then
    {
        advance();
        T();
        E’();
    }
}
```

```
Procedure T()
{
    F();
    T’();
}
```

UPES
UNIVERSITY WITH A PURPOSE

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

E -> TE’
E’ -> +TE’ | Ɛ
T -> FT’
T’ -> *FT’ | Ɛ
F -> (E) | **id**

```
Procedure E()
{
        T();
        E’();
}

Procedure E’ ()
{
        if (input symbol = +) then
        {
                advance();
                T();
                E’();
        }
}
```

```
Procedure T()
{
        F();
        T’();
}

Procedure T’ ()
{
        if (input symbol = *) then
        {
                advance();
                F();
                T’();
        }
}
```

UPES
UNIVERSITY WITH A PURPOSE

Recursive Descent Parser

- Write down the algorithm using recursive procedure to implement following grammar:

E -> TE’
E’ -> +TE’ | Ɛ
T -> FT’
T’ -> *FT’ | Ɛ
F -> (E) | **id**

```
Procedure E()
{
        T();
        E’();
}

Procedure E’ ()
{
        if (input symbol = ‘+’) then
        {
                advance();
                T();
                E’();
        }
}
```

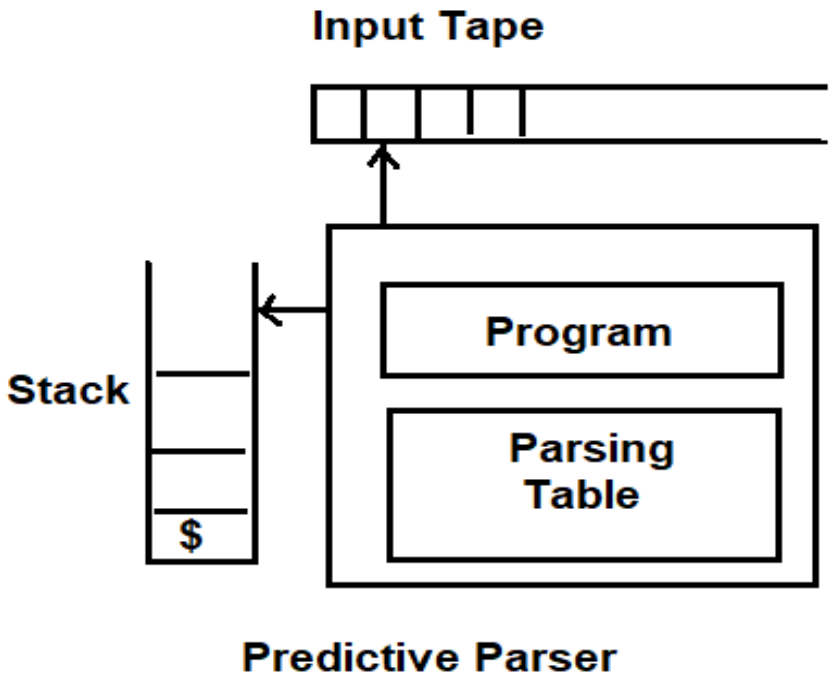
```
Procedure T()
{
        F();
        T’();
}

Procedure T’ ()
{
        if (input symbol = ‘*’) then
        {
                advance();
                F();
                T’();
        }
}
```

```
Procedure F ()
{
        if (input symbol = ‘id’)
        {
                advance();
        }
        else if (input symbol=‘(‘)
        {
                advance();
                E();
                if (input symbol=‘)’)
                        advance();
                else   error();
        }
        else error()
}
```

Predictive Parser

- It is a top down backtracking less parser which generates the parse tree in LL(1) manner.
- There should be no left recursion and left factoring in the grammar.



Components of Predictive Parser

- Input tape: which contains the string to be parsed, only terminals are there in the input tape. $ symbol is there at the end of the string which indicates that string has finished.
- Stack: which contains sequence of grammar symbols (terminals and non terminals both) preceded by $ symbol. $ indicates the bottom of the stack.
- Parsing Table: It is a two dimensional array M[A, a] where A is representing the non terminal and a is representing the terminal or $ symbol.
- Program: Parser is controlled by a program.

How Program controls Parser

- If current input symbol and top stack symbol are $ then parser halts and announce successful completion of parsing.
- If current input symbol and top stack symbol matches but both are not $ then parser pop off the top stack symbol and advances the input pointer to next input symbol.
- If top of the stack is Non terminal (X) and present input symbol is ‘a’ then program consults entry M [X,a] in the parsing table and take the action accordingly.
  - If this entry is an error entry then parser calls the error recovery routine.
  - If M[X,a]={X→UvW} then parser replaces the X on top of the stack by WvU where U will be top.
- Initial configuration of parser is Stack: $S Input: w$
- To fill the entries of parsing table, we use First() and Follow() functions.

UPES
UNIVERSITY WITH A PURPOSE

Steps for finding First

- First(α) represents the set of the terminal symbols which occurs as first symbol in the strings derived from α where α is any string of grammar symbols.
  - If X→a then First(X)=a
  - If X→λ then First(X)= λ
  - First of terminal will be that terminal itself
  - If X→aα then First(X)=a
  - If X→$Y_1Y_2Y_3...Y_k$ then
    - First(X)=First(Y_1) ;if First(Y_1) does not contain λ
    - First(X)=First(Y_1)-{λ} U First(Y_2);if First(Y_1) contain λ and First(Y_2) does not contain λ
    - First(X)=First(Y_1)-{λ} U First(Y_2)-{λ} U First(Y_3);if First(Y_1) and First(Y_2) contain λ and First(Y_3) does not contain λ

Steps for finding Follow

- Follow(A) is the set of the terminals which occur immediately after (follow) the non-terminal A in the strings derived from the starting symbol
  - $ is in Follow(X) if X is start symbol
  - If A→ αBβ and β≠ λ then
    - If first(β) does not contain λ then Follow(B)={First(β)}
    - If first(β) contains λ then Follow(B)={First(β)}-{λ} U Follow(A)
  - If A→ αB then Follow(B)={Follow(A)}

Numerical

- Find first and follow for the following grammar
  - S→AaA|BbB
  - A→bB
  - B→ ε

Numerical

- Find first and follow for the following grammar
  - S→AaA|BbB
  - A→bB
  - B→ ε

  - First(B)=ε
  - First(A)=b
  - First(S)=b

Numerical

- Find first and follow for the following grammar
  - S→AaA|BbB
  - A→bB
  - B→ ε

  - First(B)=ε Follow(B)={$, a, b}
  - First(A)=b Follow(A)={$, a}
  - First(S)=b Follow(S)={$}

Numerical

- Find first and follow for the following grammar
  - E→E+T|T
  - T→T*F|F
  - F→(E)|id

Numerical

- Find first and follow for the following grammar
  - E→E+T|T
  - T→T*F|F
  - F→(E)|id

Numerical

- Find first and follow for the following grammar
  - E→E+T|T
  - T→T*F|F
  - F→(E)|id

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

Numerical

- Find first and follow for the following grammar
  - E→E+T|T
  - T→T*F|F
  - F→(E)|id

  - First(E)={(,id} Follow(E)={$,+,)}
  - First(T)={(,id} Follow(T)={$,+,),*}
  - First(F)={(,id} Follow(F)={$,+,),*}

Numerical

- Find first and follow for the following grammar
  - S→L=R
  - S→R
  - L→*R
  - L→id
  - R→L

Numerical

- Find first and follow for the following grammar
  - S→L=R
  - S→R
  - L→*R
  - L→id
  - R→L

  - First(S)={*,id}
  - First(L)={*,id}
  - First(R)={*,id}

Numerical

- Find first and follow for the following grammar
  - S→L=R
  - S→R
  - L→*R
  - L→id
  - R→L

  - First(S)={*,id} Follow(S)={$}
  - First(L)={*,id} Follow(L)={$,=}
  - First(R)={*,id} Follow(R)={$,=}

Numerical

- Find first and follow for the following grammar
  - E→TE`
  - E`→+TE`|ε
  - T→FT`
  - T`→*FT`|ε
  - F→(E)|id

Numerical

- Find first and follow for the following grammar
  - E→TE`
  - E`→+TE`|ε
  - T→FT`
  - T`→*FT`|ε
  - F→(E)|id

  - First(E)={(,id}
  - First(T)={(,id}
  - First(F)={(,id}
  - First(E`)={+, ε}
  - First(T`)={*, ε}

Numerical

- Find first and follow for the following grammar
  - E→TE`
  - E`→+TE`|ε
  - T→FT`
  - T`→*FT`|ε
  - F→(E)|id

  - First(E)={(,id} Follow(E)={),$}
  - First(T)={(,id} Follow(T)={+,),$}
  - First(F)={(,id} Follow(F)={+,*,),$}
  - First(E`)={+, ε}  Follow(E`)={),$}
  - First(T`)={*, ε}  Follow(T`)={+,),$}

Rules for filling the entries in Parsing Table

- For each production rule A->α in the grammar do the following:
  - For each terminal a in First(α) add A→α in M[A, a]
  - If ε is in First(α), and b is in Follow(A) then add A→α to M[A, b]
  - If ε is in First(α), and $ is in Follow(A) then add A→α to M[A, $]
  - All the remaining entries in the table are error.

Algorithm-Predictive Parser

- Input: String to be parsed and parsing table M for Grammar
- Output: Either stack will become empty or it will give an error
- Steps:
  - While(Stack is not empty)do
  - {
    - Let X be top stack symbol and let a be next input symbol
    - If (X is terminal or $) then
      - If(X=a) then
        - » Pop X from stack and remove a from the input
      - Else
        - » Error()
    - Else
      - If M[X,a]=A→$Y_1Y_2…Y_n$ then
        - » Pop X and Push $Y_nY_{n-1}…Y_1$ onto stack
      - Else
        - » Error()
  - }

Numerical

- Construct predictive parsing table for the following grammar and check whether string id+id*id is accepted or not.

  - E→TE`
  - E`→+TE`|ε
  - T→FT`
  - T`→*FT`|ε
  - F→(E)|id

Numerical

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

- Step 1: Eliminate left recursion and left factoring: Already removed

Numerical

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

- Step 1: Eliminate left recursion and left factoring: Already removed
- Step 2: Compute First of all non terminals: First(E)=First(T) =First(F) {(,id} First(E`)={+, ε}  First(T`)={*, ε}

Numerical

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

- Step 1: Eliminate left recursion and left factoring: Already removed
- Step 2: Compute First of all non terminals: First(E)=First(T) =First(F) {(,id} First(E`)={+, ε}  First(T`)={*, ε}
- Step 3: Compute Follow of all non terminals: Follow(E)=Follow(E`) {),$}  Follow(T)=Follow(T`) {+,),$}
  Follow(F)={+,*,),$}

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

Non Terminal			INPUT SYMBOLS			
	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

- Step 1: Eliminate left recursion and left factoring: Already removed
- Step 2: Compute First of all non terminals: First(E)=First(T) =First(F) {(,id} First(E`)={+, ε} First(T`)={*, ε}
- Step 3: Compute Follow of all non terminals: Follow(E)=Follow(E`) {),$} Follow(T)=Follow(T`) {+,),$} Follow(F)={+,*,),$}
- Step 4: Construct the predictive parsing table

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

Non Terminal			INPUT SYMBOLS			
	id	+	*	(	)	$
E	E→ TE'			E → TE'		
E'		E →+TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

- Step 1: Eliminate left recursion and left factoring: Already removed
- Step 2: Compute First of all non terminals: First(E)=First(T) =First(F) {(,id} First(E`)={+, ε} First(T`)={*, ε}
- Step 3: Compute Follow of all non terminals: Follow(E)=Follow(E`) {),$} Follow(T)=Follow(T`) {+,),$} Follow(F)={+,*,),$}
- Step 4: Construct the predictive parsing table

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E→ TE'			E → TE'		
E'		E' →+TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

- Step 1: Eliminate left recursion and left factoring: Already removed
- Step 2: Compute First of all non terminals: First(E)=First(T) =First(F) {(,id} First(E`)={+, ε} First(T`)={*, ε}
- Step 3: Compute Follow of all non terminals: Follow(E)=Follow(E`) {),$} Follow(T)=Follow(T`) {+,),$} Follow(F)={+,*,),$}
- Step 4: Construct the predictive parsing table.
- Step 5: Check the acceptance of the string id+id*id using predictive parsing program.

- E→TE`
- E`→+TE`|ε
- T→FT`
- T`→*FT`|ε
- F→(E)|id

Non Terminal	INPUT SYMBOLS					
	id	+	*	(	)	$
E	E→ TE'			E → TE'		
E'		E →+TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE'$	id+id * id$	E->TE'
	FT'E'$	id+id * id$	T->FT'
	id T'E'$	id+id * id$	F->id
id	T'E'$	+id * id$	Match id
id	E'$	+id * id$	T'->Є
id	+TE'$	+id * id$	E'-> +TE'
id+	TE'$	id * id$	Match +
id+	FT'E'$	id * id$	T-> FT'
id+	idT'E'$	id * id$	F-> id
id+id	T'E'$	* id$	Match id
id+id	* FT'E'$	* id$	T'-> *FT'
id+id *	FT'E'$	id$	Match *
id+id *	idT'E'$	id$	F-> id
id+id * id	T'E'$	$	Match id
id+id * id	E'$	$	T'-> Є
id+id * id	$	$	E'-> Є

LL(1) Grammar

- The algorithm for constructing a predictive parsing table can be applied to any grammar G to produce a parsing table M.
- For some grammars however, M may have some entries that are multiply-defined.
- If G is left recursive or ambiguous, then M will have at least one multiply-defined entry.
- A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

LL(1) Grammar

- LL(1) grammars have several distinctive properties.
  - Parsing table has no multiply-defined entries is said to be LL(1).
  - No ambiguous or left recursive grammar can be LL(1).
  - A grammar G is LL(1) iff whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following conditions hold :-
    - FIRST(α) , FIRST(β) must be disjoint.
    - At most one of the strings α or β can derive ε i.e. $FIRST(\alpha) \cap FIRST(\beta) \neq \varepsilon$.
    - If $\beta \Rightarrow^* \varepsilon$, then $FIRST(\alpha) \cap Follow(A) = \phi$ i.e. α does not derive any string beginning with the terminal in FOLLOW(A)
- In predictive parser when we restrict grammar to be only LL(1) grammar then it is called LL(1) parser.

UPES
UNIVERSITY WITH A PURPOSE

LL(1) Grammar

G1 = ({E, F, T}, {a, (,), ∗,+}, P, E) where the productions are:

E → E + T|T

T → T ∗ F|F

F → (E)|a

Is this LL(1) grammar

LL(1) Grammar

G1 = ({E, F, T}, {a, (,), ∗,+}, P, E) where the productions are:

E → E + T|T

T → T ∗ F|F

F → (E)|a

It is not LL(1) grammar

LL(1) Grammar

G2 = ({S, A}, {a, b,c,d}, P, S) where the productions are:

S → cAd

A → ab|a

Is this LL(1) grammar?

LL(1) Grammar

G2 = ({S, A}, {a, b,c,d}, P, S) where the productions are:

S → cAd

A → ab|a

It Is not LL(1) grammar?

LL(1) Grammar

G3 = ({E, E′, T, T′, F}, {a, ∗, +,(,)}, P, E) with the set of productions P:

E → TE′ T → F T′ F → (E)|a

E′ → +TE′| ε T′ → ∗F T′| ε

Is this LL(1) grammar?

LL(1) Grammar

G3 = ({E, E′, T, T′, F}, {a, ∗, +,(,)}, P, E) with the set of productions P:

E → TE′ T → F T′ F → (E)|a

E′ → +TE′| ε T′ → ∗F T′| ε

It is LL(1) grammar

LL(1) Grammar

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

Is this LL(1) grammar

LL(1) Grammar

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string abba

LL(1) Parser

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string ‘abba‘

First (S)={a}

First (B)={b, ε}

LL(1) Parser

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string abba

First (S)={a} Follow(S)={$}

First (B)={b, ε} Follow(B)={a}

LL(1) Parser

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string abba

First (S)={a} Follow(S)={$}

First (B)={b, ε} Follow(B)={a}

	a	b	$
S			
B			

LL(1) Parser

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string abba

First (S)={a} Follow(S)={$}

First (B)={b, ε} Follow(B)={a}

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

LL(1) Parser

G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar, now do the parsing of the string abba

stack	input	output
$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	
$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	accept, successful completion

First (S)={a} Follow(S)={$}

First (B)={b, ε} Follow(B)={a}

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

UPES
UNIVERSITY WITH A PURPOSE

LL(1) Parser

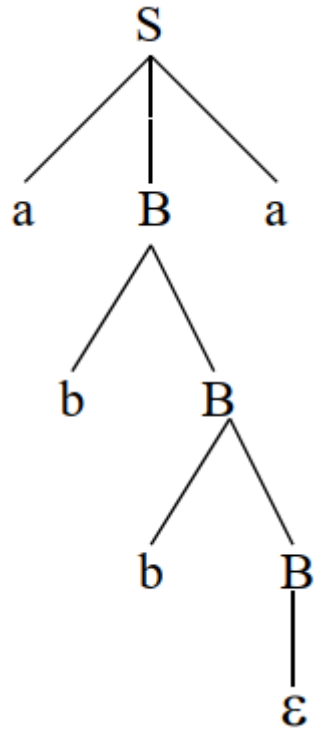
G4 = ({S, B}, {a,b}, P, S) with the set of productions P:

S→aBa

B→bB| ε

It is LL(1) grammar,

now do the parsing of the string abba



stack	input	output
$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	
$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	accept, successful completion

First (S)={a} Follow(S)={$}

First (B)={b, ε} Follow(B)={a}

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

UPES
UNIVERSITY WITH A PURPOSE

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

Step 3: Calculate First and Follow.

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

Step 3: Calculate First and Follow.

First(S)={i,a}

First(S`)={e, ε}

First(C)={b}

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

Step 3: Calculate First and Follow.

First(S)={i, a}	Follow(S)={$,e}
First(S`)={e, ε}	Follow(S`)={$,e}
First(C)={b}	Follow(C)={t}

UPES
UNIVERSITY WITH A PURPOSE

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

Step 3: Calculate First and Follow.

First(S)={i, a} Follow(S)={$,e}

First(S`)={e, ε}  Follow(S`)={$,e}

First(C)={b} Follow(C)={t}

Step 4: Construct parsing table.

	a	b	e	i	t	$
S						
S`						
C						

UPES
UNIVERSITY WITH A PURPOSE

LL(1) Grammar

G1 = ({S, S`,C}, {i, t, a, e, b}, P, S) where the productions are:

S→iCtSS`|a

S`→eS| ε

C → b Construct parsing table for this and check is this grammar LL(1)

Step 1: Remove Left recursion

Step 2: Remove Left Factoring

Step 3: Calculate First and Follow.

First(S)={i, a} Follow(S)={$,e}

First(S`)={e, ε}  Follow(S`)={$,e}

First(C)={b} Follow(C)={t}

Step 4: Construct parsing table.

	a	b	e	i	t	$
S	S-->a			S-->iCtSS`		
S`			S`-->eS S`-->ε			S`-->ε
C		C -->b				

LL(1) Grammar

G1 = ({S, L}, {(,),a, ,},P,S) where the productions are:

S→(L)|a

L→L,S| S

Construct parsing table for this, check is this grammar LL(1), also check string ‘(a,a)’ is accepted:

THANK YOU

UPES
UNIVERSITY WITH A PURPOSE

Dr Anurag Jain
Assistant Professor (SG), Virtualization Department, School of Computer Science,
University of Petroleum & Energy Studies, Dehradun - 248 007 (Uttarakhand)
Email: anurag.jain@ddn.upes.ac.in , dr.anuragjain14@gmail.com
Mobile: (+91) -9729371188