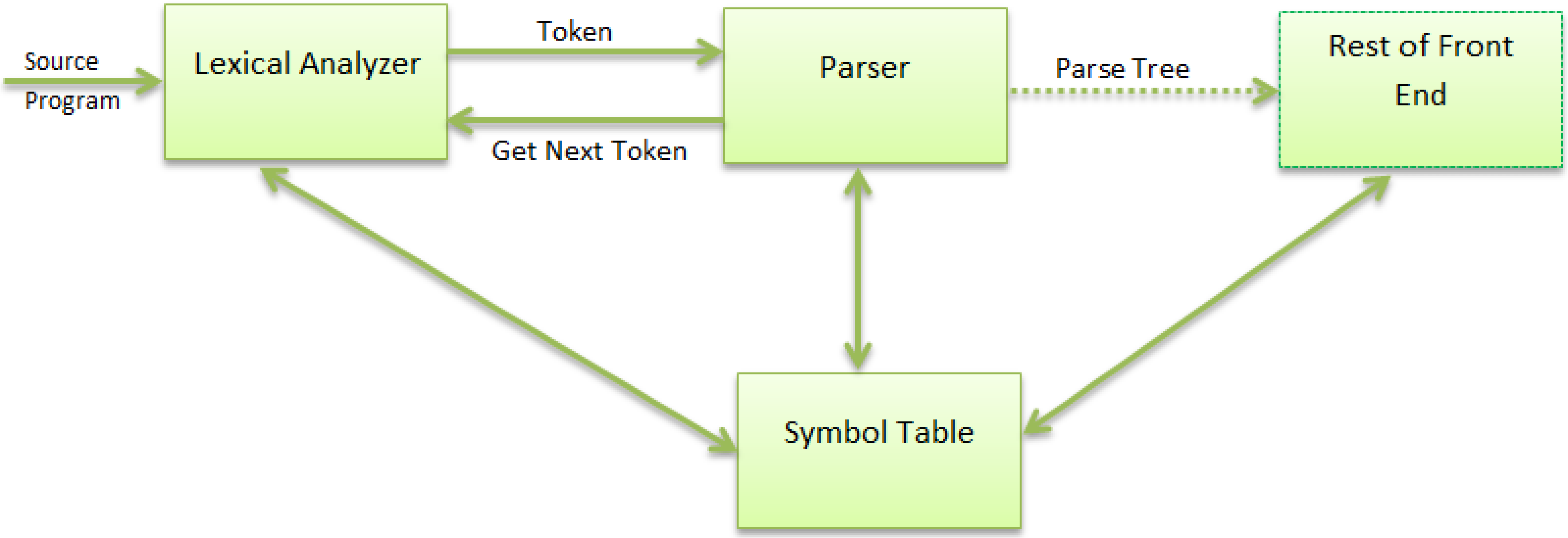
Role of the Parser

Presidency University, Bengaluru

Role of the Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer.
- Syntax Analyzer verifies that the string of token names can be generated by the grammar for the source language.
- We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.



Types of Parsers

There are three general types of parsers for grammars:

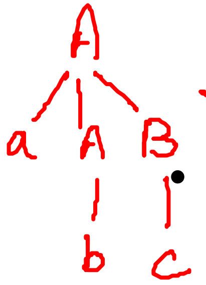
- **Universal**
- **Top-down**
- **Bottom-up**

- Universal parsing methods such as the **Cocke-Younger-Kasami algorithm** and **Earley's algorithm** can parse any grammar. These general methods are, however, too inefficient to use in production compilers.

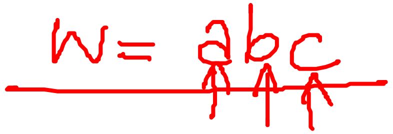
- The methods commonly used in compilers can be classified as being either **top-down** or **bottom-up.**

Types of Parsers

- Parsing can be defined as **top-down** or **bottom-up** based on how the parse-tree is constructed.
- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root.



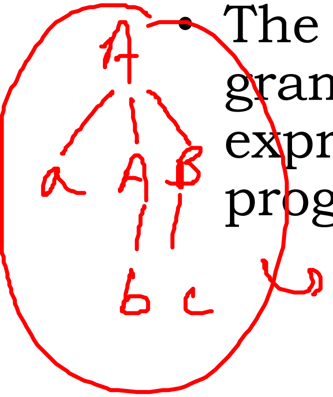
✓ → Top-Down



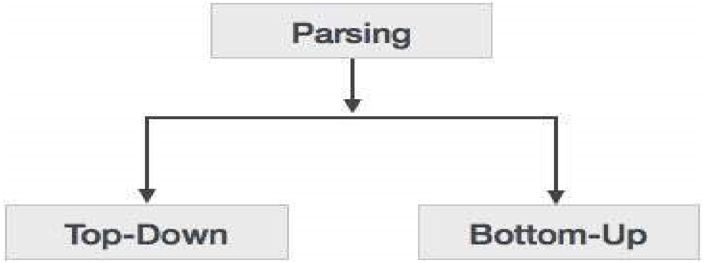
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

✓

- The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, **LL and LR grammars**, are expressive enough to describe most of the syntactic constructs in modern programming languages.



→ Bottom-up



Error Handler with Syntax Analyzer

The **error handler** in a parser has goals that are simple to state but challenging to realize:

- **Report the presence of errors clearly and accurately.**
- **Recover from each error quickly enough to detect subsequent errors.**
- **Add minimal overhead to the processing of correct programs.**

Error-Recovery Strategies

- **Planning the error handling** right from the start can both **simplify the structure of a compiler and improve its handling of errors**.

- Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when **parsing cannot continue.**

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability.

- **Panic-Mode Recovery**
- **Phrase-Level Recovery**
- **Error Productions**
- **Global Correction**

Error-Recovery Strategies

- **Panic-Mode Recovery**
  - With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

- **Phrase-Level Recovery**
  - On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.

- **Error Productions**
  - A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.

- **Global Correction**
  - Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Context Free Grammar

Reg → FA < NFA
DFA
CFG → PDA → Push Down Automata
CSG → LBA

By design, every programming language has **precise rules** that prescribe the **syntactic structure of well-formed programs**.

The syntax of programming language constructs can be specified by **context-free grammars** or **BNF (Backus-Naur Form) notation**.

Grammars offer significant benefits for both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.

int a, b [illegible] ⇒

- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program.
- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

Context Free Grammar (CFG)

Grammars systematically describe the syntax of programming language constructs like expressions and statements.

A CFG can be used to generate strings in its language

– “Given the CFG, construct a string that is in the language”

A CFG can also be used to recognize strings in its language

– “Given a string, decide whether it is in the language”

Context Free Grammar (CFG)

Formal Definition of a CFG

Variables

A context-free grammar (grammar for short) consists of **terminals, nonterminals, a start symbol,** and **productions**.

e.g. G
$S \to SS$ 1)
$S \to AA$ 2)
$A \to a \mid \epsilon$
→ CFG

- **Terminals** are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal“.

Terminals → a, ϵ

- **Nonterminals** are syntactic variables that denote sets of strings.

Variables: — S, A

- In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar.
- The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.
  - Each production consists of: (a) A nonterminal called the head or left side of the production.
  - A body or right side consisting of zero or more terminals and nonterminals.

Parsing

i/p →

g:-

PARSING
It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

1) g: $X \rightarrow X+X \mid X*X \mid a \mid \epsilon$
I/p: $a*a*a+a$

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

✓ 1) Terminals:- ? a, *, +, ε
– 2) Variables:- ? X
– 3) Parse Tree (Left most Derivation)
4) Ambgiuous / Unambiguous

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top-down parsing : A parser can start with the start symbol and try to transform it to the input string. Example : **LL Parsers**. ✓
- Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol. Example : **LR Parsers**. ✓

Top-Down Parser

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.

Equivalently, It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of Top-Down Parsing

1. **Recursive descent parsing**
2. **Predictive parsing**

Recursive Recent Parser

A general form of top-down parsing, called recursive descent parsing, which may require backtracking to find the correct production to be applied.

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead.

This parsing method may involve backtracking.

Example for : Backtracking

Consider the grammar G :

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

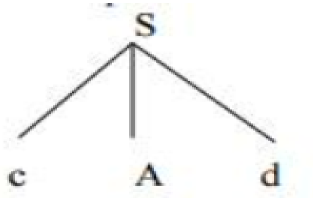
and the input string **w=cad**.

Recursive Recent Parser

The parse tree can be constructed using the following top-down approach :

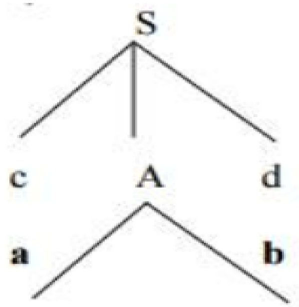
Step1:

Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’. Expand A using the first alternative.



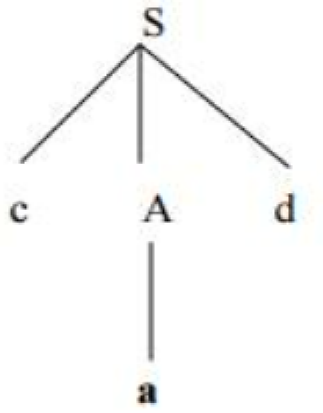
Recursive Recent Parser

Step3:

The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’.But the third leaf of tree is b which does not match with the input symbol **d.**Hence discard the chosen production and reset the pointer to second **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Non Recursive Predictive Parser

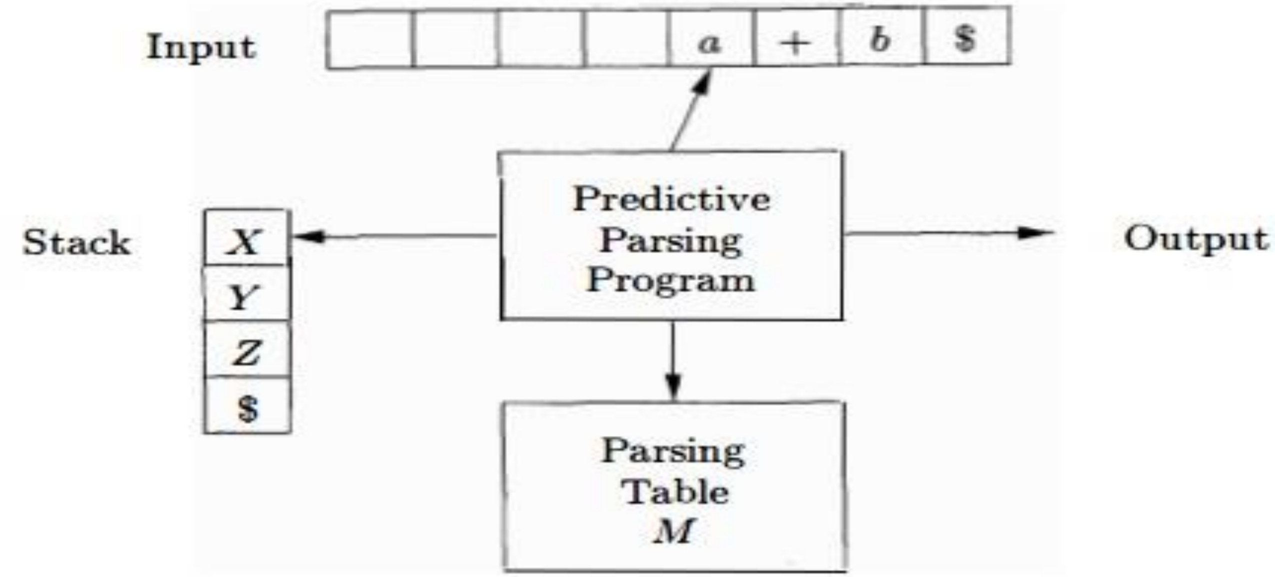
- It is possible to build a **nonrecursive predictive parser** by maintaining a stack explicitly, rather than implicitly via recursive calls. (A parser using the single-symbol look-ahead method and top-down parsing **without backtracking** is called **LL(1) Parser** or **Non-Recursive Parser**)

- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.

- The nonrecursive parser looks up the production to be applied in parsing table.

Non Recursive Predictive Parser

- A table-driven predictive parser has an **input buffer, a stack, a parsing table, and an output stream.**
- The **input buffer** contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string.
- The **stack** contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $.
- The **parsing table** is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $.



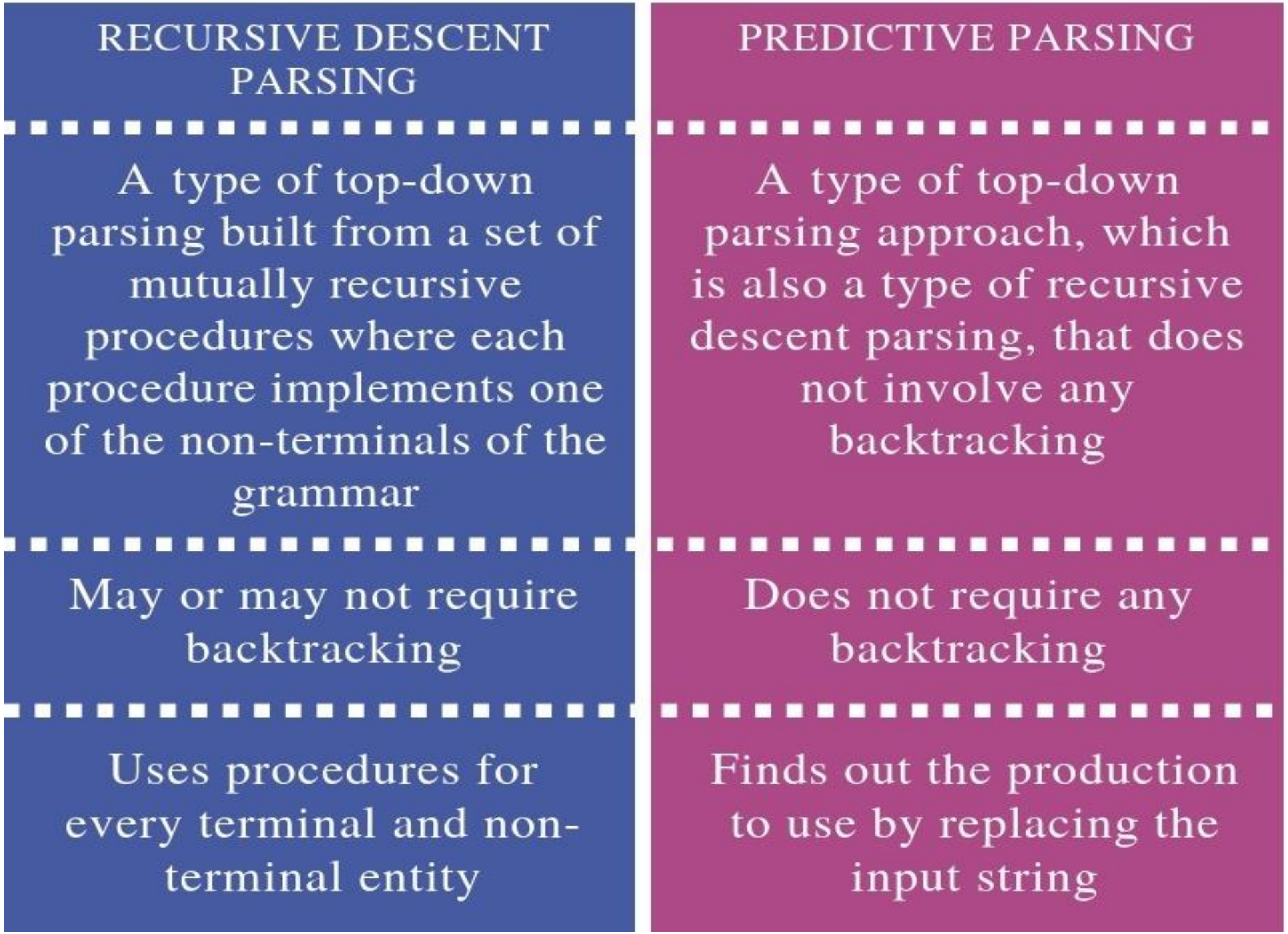
Non Recursive Predictive Parser

The parser is controlled by a program that behaves as follows.

- The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser.
- There are three possibilities.

1. If **X = a = $,** the parser halts and announces successful completion of parsing.
2. If **X = a ≠ $,** the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If **X is a nonterminal**, the program consults entry **M[X,a] of the parsing table M**. This entry will be either an X-production of the grammar or an error entry. If, for example, **M[X,a] = {X→UVW},** the parser replaces X on top of the stack by WVU (with U on top). If **M[X,a]=error**, the parser calls an error recovery routine.

Difference between RCP and NRPR

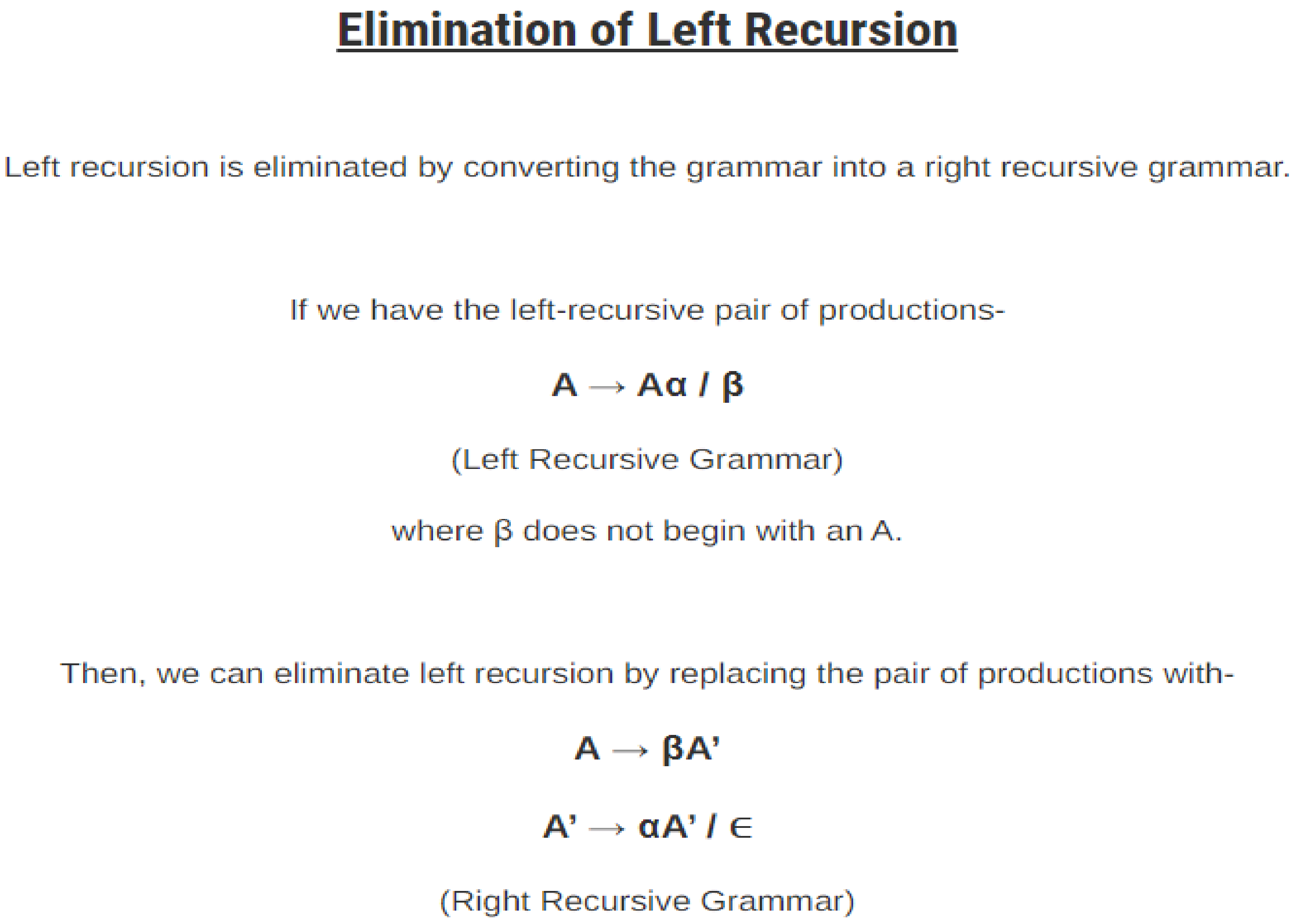


RECURSIVE DESCENT PARSING	PREDICTIVE PARSING
A type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of the non-terminals of the grammar	A type of top-down parsing approach, which is also a type of recursive descent parsing, that does not involve any backtracking
May or may not require backtracking	Does not require any backtracking
Uses procedures for every terminal and non-terminal entity	Finds out the production to use by replacing the input string

Predictive Parsing Table Construction

- Before Table Construction
  - Eliminate Left Recursion
  - Preform Left Factoring
- For Table Construction
  - FIRST
  - FOLLOW
- Table Construction

Before Table Construction



Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha \,/\, \beta$$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \,/\, \in$$

(Right Recursive Grammar)

For Example,

$$E \rightarrow E + T \mid T$$

is replaced by

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

Left Recursion

Consider the following grammar and eliminate left recursion-

A → ABd / Aa / a

B → Be / b

Q 2. Consider the following grammar and eliminate left recursion-

E → E + E / E x E / a

Q 3. Consider the following grammar and eliminate left recursion-

E → E + T / T

T → T x F / F

F → id

Left Factoring

$$A \rightarrow \underline{\alpha}\beta_1 \mid \underline{\alpha}\beta_2 \mid \dots \mid \underline{\alpha}\beta_n$$

↓

$$A \rightarrow \alpha A^1$$
$$A^1 \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$E \rightarrow \underbrace{iEtS}_{\alpha} \mid \underbrace{iEtS}_{\alpha}\underbrace{eS}_{\beta} \mid \underline{a} \mid \underline{Bb}$$

$$E \rightarrow iEtSE' \mid a \mid Bb$$
$$E' \rightarrow \epsilon \mid eS$$

For Example,

E → iEtS | iEtSeS | a | Bb

1) 2) 3] 4)

is replaced by

E → iEtSE' | a | Bb

E' → ε | eS

Left Factoring

Q1 .Do left factoring in the following grammar-

- S → iEtS / iEtSeS / a
- E → b

Q 2. Do left factoring in the following grammar-

- A → aAB / aBc / aAc

Q 3. Do left factoring in the following grammar-

- S → bSSaaS / bSSaSb / bSb / a

For Table Construction

Rules for FIRST Sets

1. If X is a terminal **then** First(X) is just { X }
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production $X \rightarrow Y_1Y_2..Y_k$ **then** add first($Y_1Y_2..Y_k$) to first(X)

Example - First

Consider the production rules-

A → abc / def / ghi

Then, we have-

First(A) = { a , d , g }

2)S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

First Functions-

First(S) = { a }

First(B) = { c }

First(C) = { b , ∈ }

First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }

First(E) = { g , ∈ }

First(F) = { f , ∈ }

For Table Construction

Rules for Follow Sets

1. First put $ (the end of input marker) in Follow(S)(S is the start symbol)

2. If there is a production A → αBβ **then** everything in FIRST(β) except for ε is placed in FOLLOW(B)

3. If there is a production A → αB **then** everything in FOLLOW(A) is in FOLLOW(B)

    - If there is a production A → αBβ where FIRST(β) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Example

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

- **Follow Functions-**
- Follow(S) = { $ }
- Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h }
- Follow(D) = First(h) = { h }
- Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = { h }

Practice Questions on First and Follow

1)Calculate the first and follow functions for the given grammar-

S → A

A → aB / Ad

B → b

C → g

2)Calculate the first and follow functions for the given grammar-

S → (L) / a

L → SL’

L’ → ,SL’ / ∈

Table Construction

INPUT : Grammar G

OUTPUT: Parsing table M

For each production A → α , do the following :

1. For each terminal ‘a’ in FIRST(A), add A→ α to M[A,α].
2. If ε is in FIRST(α) then for each terminal b in FOLLOW(A).
   A → α to M[A,b]. If b is $ then also add A → α to M[A,$].
3. If there is no production in M[A,a], then set M[A,a] to error.

Table Construction

Construct Predictive Parsing Table for the following CFG

S → cAd

A → ab | a

Step 1: Eliminate the Left Recursion and Perform Left Factoring if present

- **There is no Left Recursion in the given CFG.**
- **Left Factoring present in the given CFG (In the Non-Terminal A).**
  Left Factoring can be performed by the following

A → ab | a can be replaced by

A → aA'
A' → b | ε

Table Construction

CFG after Left Factoring:

S → cAd

A → aA'

A' → b | ε

Step 2: Find FIRST set for all the non-terminals

FIRST(S) = { c } - Rule (1)

FIRST(A) = { a } - Rule (1)

FIRST(A') = { b, ε } - Rule (1) & (2)

Table Construction

Step 3: Find FOLLOW set for all the non-terminals

FOLLOW(S)	= { $ }	- Rule (1)
FOLLOW(A)	= { d }	- Rule (2)
FOLLOW(A’)	= { d }	- Rule (3)

Step 4: Construct Predictive Parsing Table

NT/T	a	b	c	d	$
S			S → cAd		
A	A → aA’				
A’		A’ → b		A’ → ε	

S → cAd
A → aA’
A’ → b | ε

Table Construction

Construct Predictive Parsing Table for the following CFG

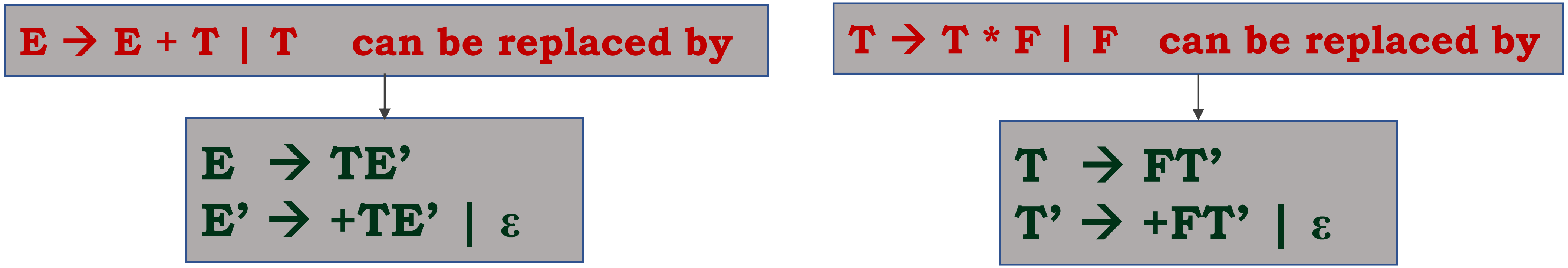
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Step 1: Eliminate the Left Recursion and Perform Left Factoring if present

- **Left Recursion present in the given CFG.** **(In the Non-Terminals E and T)**



$E \rightarrow E + T \mid T$ **can be replaced by**

$E \rightarrow TE'$

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

$T \rightarrow T * F \mid F$ **can be replaced by**

$T \rightarrow FT'$

$T' \rightarrow +FT' \mid \varepsilon$

Table Construction

CFG after Left Recursion:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → +FT’ | ε

F → (E) | id

No Left Factoring present in the given CFG.

Step 2: Find FIRST set for all the non-terminals

FIRST(E) = { (, id } - Rule (3)

FIRST(E’) = { +, ε } - Rule (1, 2)

FIRST(T) = { (, id } - Rule (3)

FIRST(T’) = { *, ε } - Rule (1, 2)

FIRST(F) = { (, id } - Rule (3)

Table Construction

Step 3: Find FOLLOW set for all the non-terminals

FOLLOW(E) = { $,) } - Rule (1,2)

FOLLOW(E’) = { $,) } - Rule (3)

FOLLOW(T) = { +, $,) } - Rule (2,3)

FOLLOW(T’) = { +, $,) } - Rule (3)

FOLLOW(F) = { *, +, $,) } - Rule (2,3)

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → +FT’ | ε
F → (E) | id

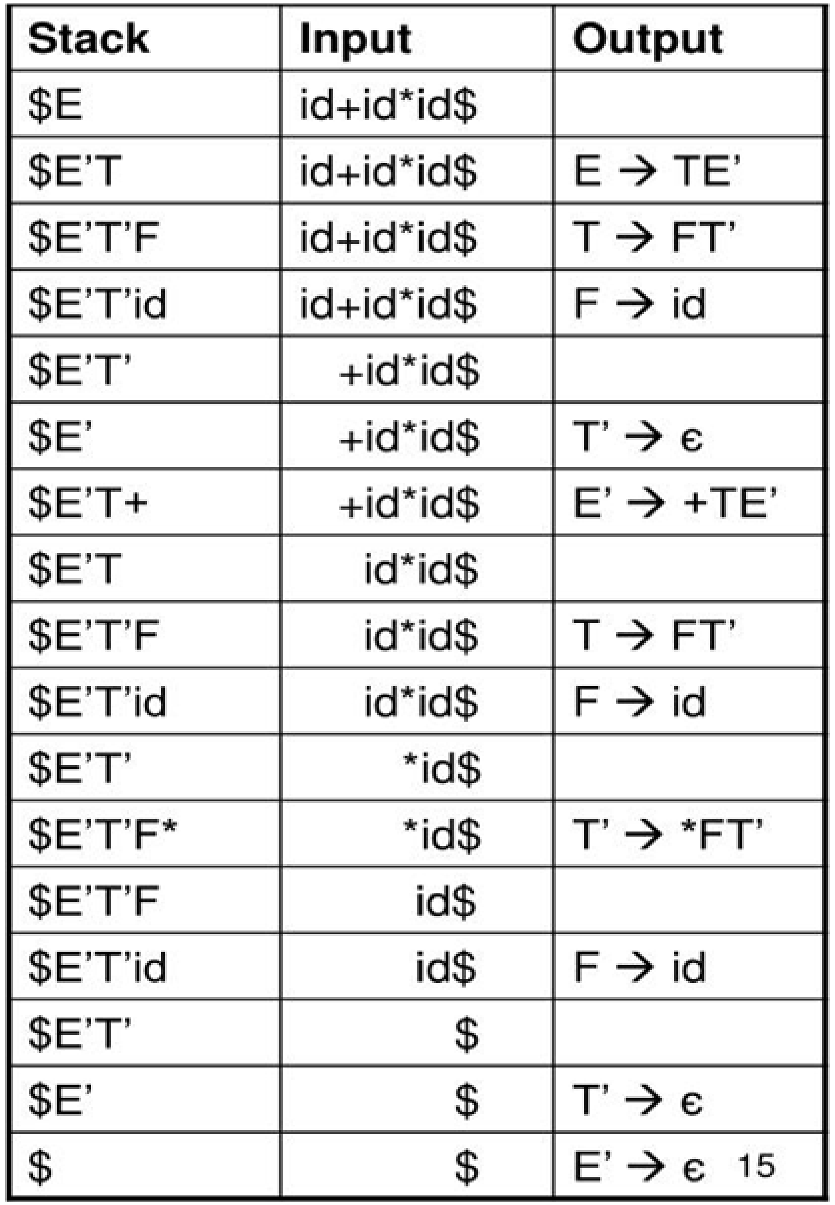
Step 4: Construct Predictive Parsing Table

NT/T	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)		

Input Validation by Predictive Parser

Consider the input String id+id*id$

NT/T	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)		



Stack	Input	Output
$E	id+id*id$	
$E’T	id+id*id$	E → TE’
$E’T’F	id+id*id$	T → FT’
$E’T’id	id+id*id$	F → id
$E’T’	+id*id$	
$E’	+id*id$	T’ → ϵ
$E’T+	+id*id$	E’ → +TE’
$E’T	id*id$	
$E’T’F	id*id$	T → FT’
$E’T’id	id*id$	F → id
$E’T’	*id$	
$E’T’F*	*id$	T’ → *FT’
$E’T’F	id$	
$E’T’id	id$	F → id
$E’T’	$	
$E’	$	T’ → ϵ
$	$	E’ → ϵ 15

LL(1) Grammar

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).

A Grammar whose parsing table has no multiply-defined entries is said to be LL(1).

LL(1) grammar have several distinctive properties.

- No ambiguous of left recursive grammar can be LL(1).
- It can be also be shown that a grammar G is LL(1) if and only if whenever A → α | β are two district productions of G the following conditions hold:
    1. For no terminal a do both α and β derive strings beginning with a.
    2. At most one of α and β can derive the empty string.
    3. If β → ε, then α does not derive any string beginning with a terminal in FOLLOW(A).

Construct Predictitive parser table and perform parsing

S → (L) / a

L → SL’

L’ → ,SL’ / ∈

- Input : (a, a)

LL(1) Grammar

Consider this following grammar:

S → iEtS | iEtSeS| a
E → b

After eliminating left factoring, we have

S → iEtSS'|a
S' → eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }
FIRST(S') = { e, ε }
FIRST(E) = { b }

FOLLOW(S) = { $, e }
FOLLOW(S') = { $, e }
FOLLOW(E) = { t }

LL(1) Grammar

Step 4: Construct Predictive Parsing Table

S → iEtSS’|a
S’ → eS | ε
E → b

NT/T	i	t	a	e	b	$
S	S → iEtSS’		S → a			
S’				S → eS S’ → ε		S’ → ε
E					E → b	

M[S’, e] contains two productions in a single entry (S’→eS and S’→**ε). **Since there are more than one production, the grammar is **not LL(1) grammar**.

Bottom-Up Parsing

Presidency University, Bengaluru

BOTTOM-UP PARSERS

- Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
- A general type of bottom-up parser is a shift-reduce parser.
- A much more general method of shift-reduce parsing, called as LR parsing.

SHIFT-REDUCE PARSING

- Constructing a parse tree for an input string beginning at the leaves (bottom) and going towards the root (top) is called shift-reduce parsing.

Reduction:

- We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

Shift-reduce Parsing

Example:

Consider the grammar:
S → aABe
A → Abc | b
B → d

The sentence to be recognized is **abbcde**.

abbcde	(A → b)
aAbcde	(A → Abc)
aAde	(B → d)
aABe	(S → aABe)
S	

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a **substring that matches the right side of a production**, and whose reduction to the non-terminal on the left side of the production represents **one step along the reverse of a rightmost derivation**.

Handle pruning:

A rightmost derivation in reverse can be obtained by "handle pruning".

Shift-reduce Parsing

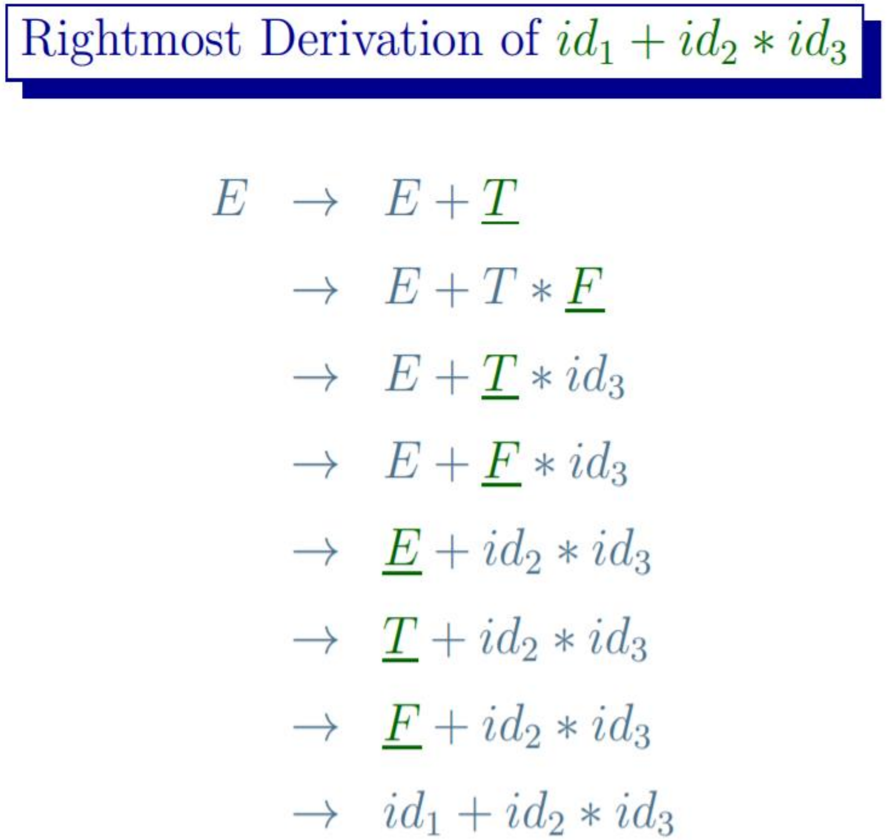
Given CFG

E → E + T | T
T → T * F | F
F → (E) | id

Input String : **id + id * id**

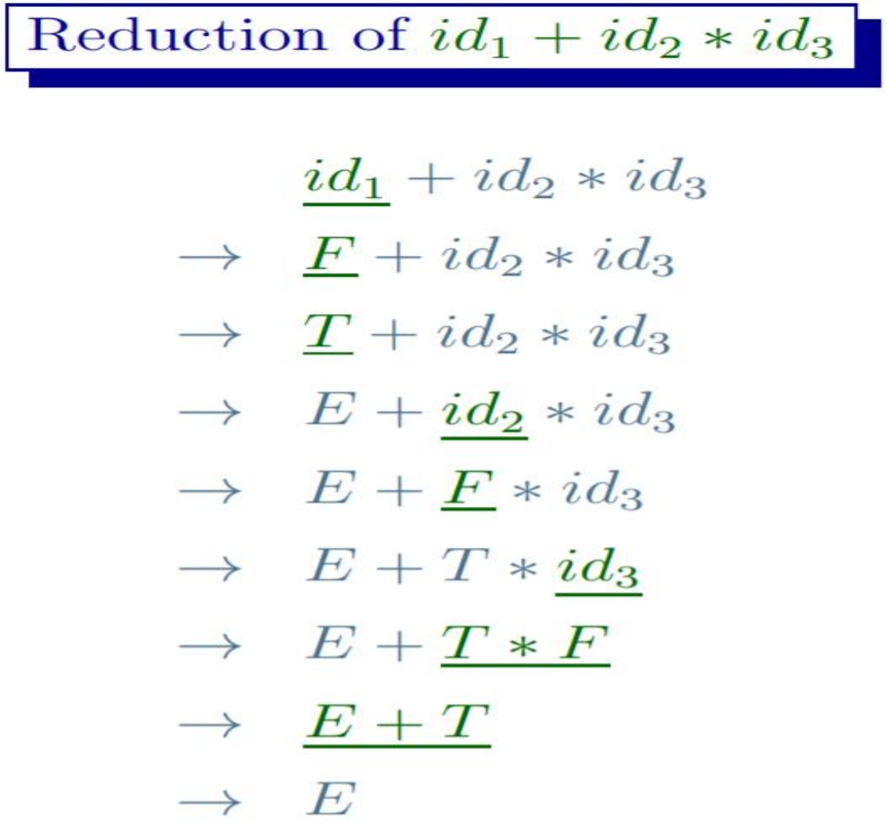
Handle pruning:

A rightmost derivation in reverse can be obtained by “handle pruning”.



Rightmost Derivation of $id_1 + id_2 * id_3$

$$\begin{aligned}
E &\rightarrow E + \underline{T} \\
&\rightarrow E + T * \underline{F} \\
&\rightarrow E + \underline{T} * id_3 \\
&\rightarrow E + \underline{F} * id_3 \\
&\rightarrow \underline{E} + id_2 * id_3 \\
&\rightarrow \underline{T} + id_2 * id_3 \\
&\rightarrow \underline{F} + id_2 * id_3 \\
&\rightarrow id_1 + id_2 * id_3
\end{aligned}$$



Reduction of $id_1 + id_2 * id_3$

$$\begin{aligned}
& \underline{id_1} + id_2 * id_3 \\
\rightarrow\ & \underline{F} + id_2 * id_3 \\
\rightarrow\ & \underline{T} + id_2 * id_3 \\
\rightarrow\ & E + \underline{id_2} * id_3 \\
\rightarrow\ & E + \underline{F} * id_3 \\
\rightarrow\ & E + T * \underline{id_3} \\
\rightarrow\ & E + \underline{T * F} \\
\rightarrow\ & \underline{E + T} \\
\rightarrow\ & E
\end{aligned}$$

Shift-reduce Parsing

Actions in shift-reduce parser:

shift - The next input symbol is shifted onto the top of the stack.
reduce - The parser replaces the handle within a stack with a non-terminal.
accept - The parser announces successful completion of parsing.
error - The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. **Shift-reduce conflict:** The parser cannot decide whether to shift or to reduce.

2. **Reduce-reduce conflict:** The parser cannot decide which of several reductions to make.

Shift-reduce Parsing

Stack Implementation of Shift Reduce Parsing

Initial Configuration

Stack	Input Buffer
$	w$

After N number of Handle Pruning (Rightmost Derivation in Reverse)

Final Configuration

Stack	Input Buffer
$S	$

Where $ used to mark the bottom of the Stack and also right end of the Input String.

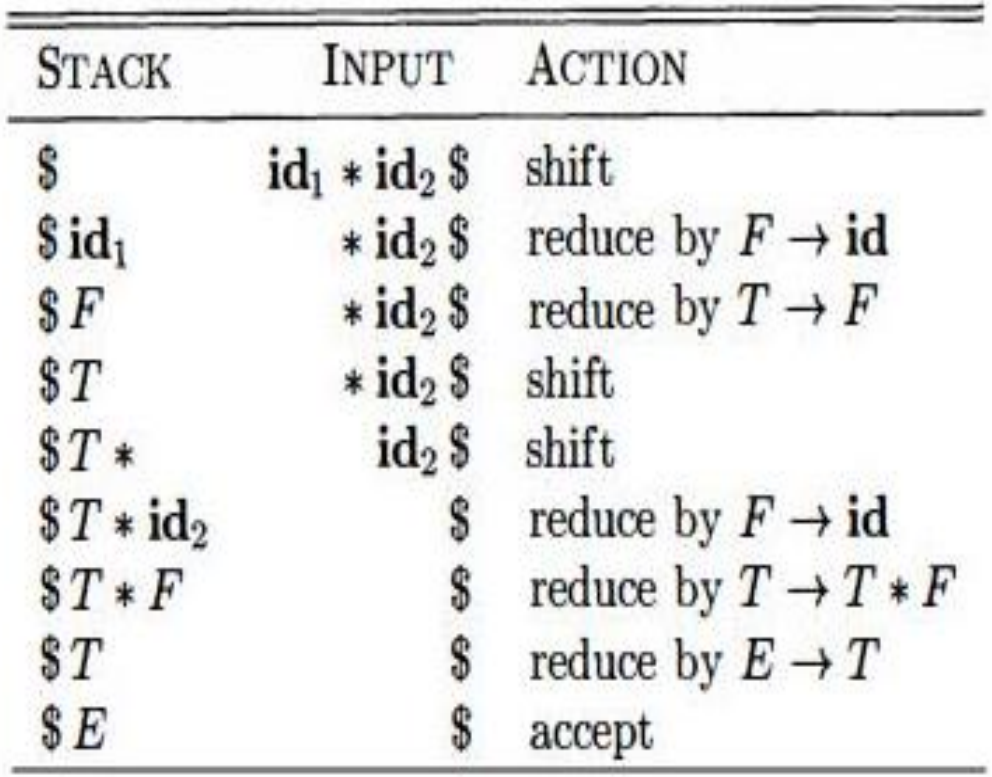
Viable prefixes:

The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes

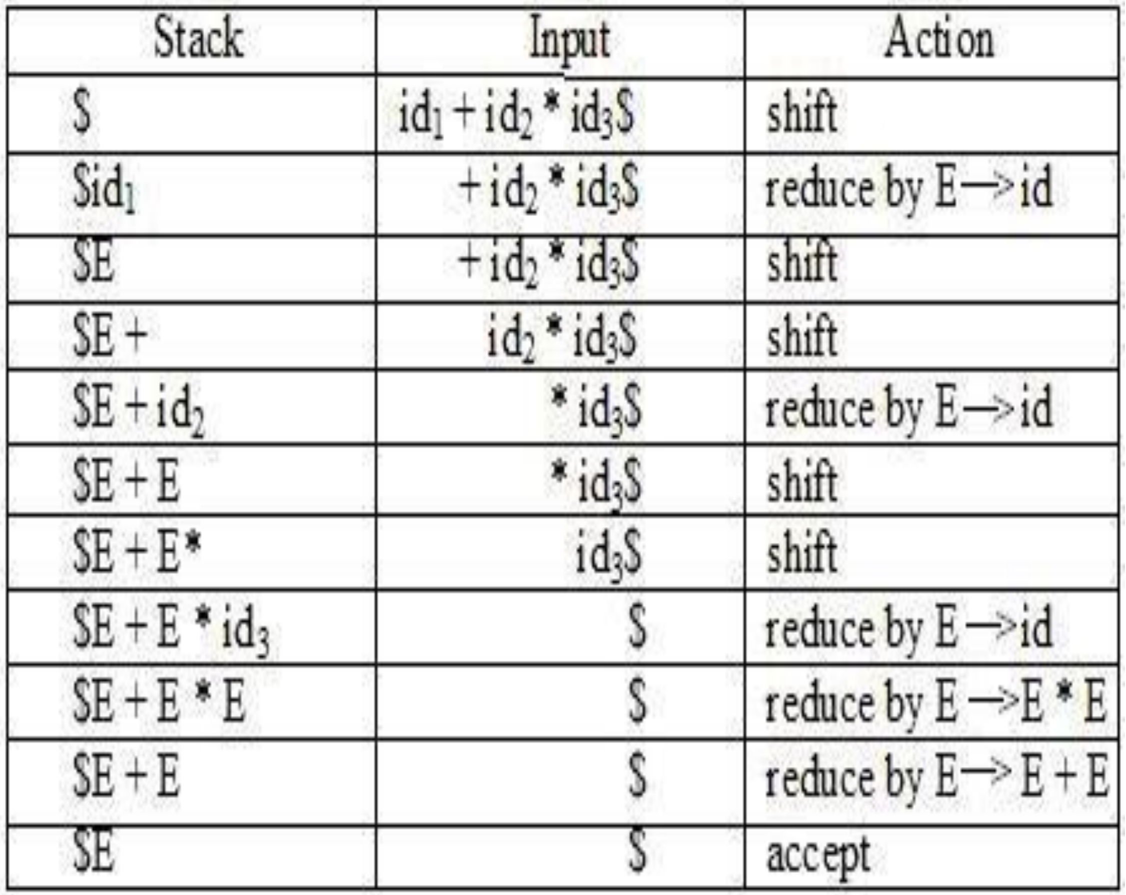
Shift-reduce Parsing

E → E + T | T
T → T * F | F
F → (E) | id

E → E+E
E → E*E
E → (E)
E → id



STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\,T$	$\$$	reduce by $E \rightarrow T$
$\$\,E$	$\$$	accept



Stack	Input	Action
$\$$	$id_1 + id_2 * id_3\$$	shift
$\$id_1$	$+ id_2 * id_3\$$	reduce by E—>id
$\$E$	$+ id_2 * id_3\$$	shift
$\$E +$	$id_2 * id_3\$$	shift
$\$E + id_2$	$* id_3\$$	reduce by E—>id
$\$E + E$	$* id_3\$$	shift
$\$E + E*$	$id_3\$$	shift
$\$E + E * id_3$	$\$$	reduce by E—>id
$\$E + E * E$	$\$$	reduce by E—>E * E
$\$E + E$	$\$$	reduce by E—>E + E
$\$E$	$\$$	accept

Presidency University, Bengaluru

LR Parser

LR PARSERS

- Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol. Example : **LR Parsers**.

An **efficient bottom-up syntax analysis technique** that can be used CFG is called **LR(k) parsing**. The **'L' is for left-to-right scanning** of the input, the **'R' for constructing a rightmost derivation in reverse**, and the 'k' for the number of input symbols.

We shall consider the cases where **k = 0 and k = 1**. **An LR(0) parser does not use look-ahead to decide its shift or reduce actions.**

LR PARSERS

Advantages of LR parsing:

1. It recognizes virtually **all programming language constructs** for which CFG can be written.
2. It is an **efficient non-backtracking shift-reduce parsing method**.
3. A grammar that can be parsed using LR method is a **proper superset of a grammar** that can be parsed with predictive parser
4. It detects a **syntactic error as soon as possible**.

Drawbacks of LR method:

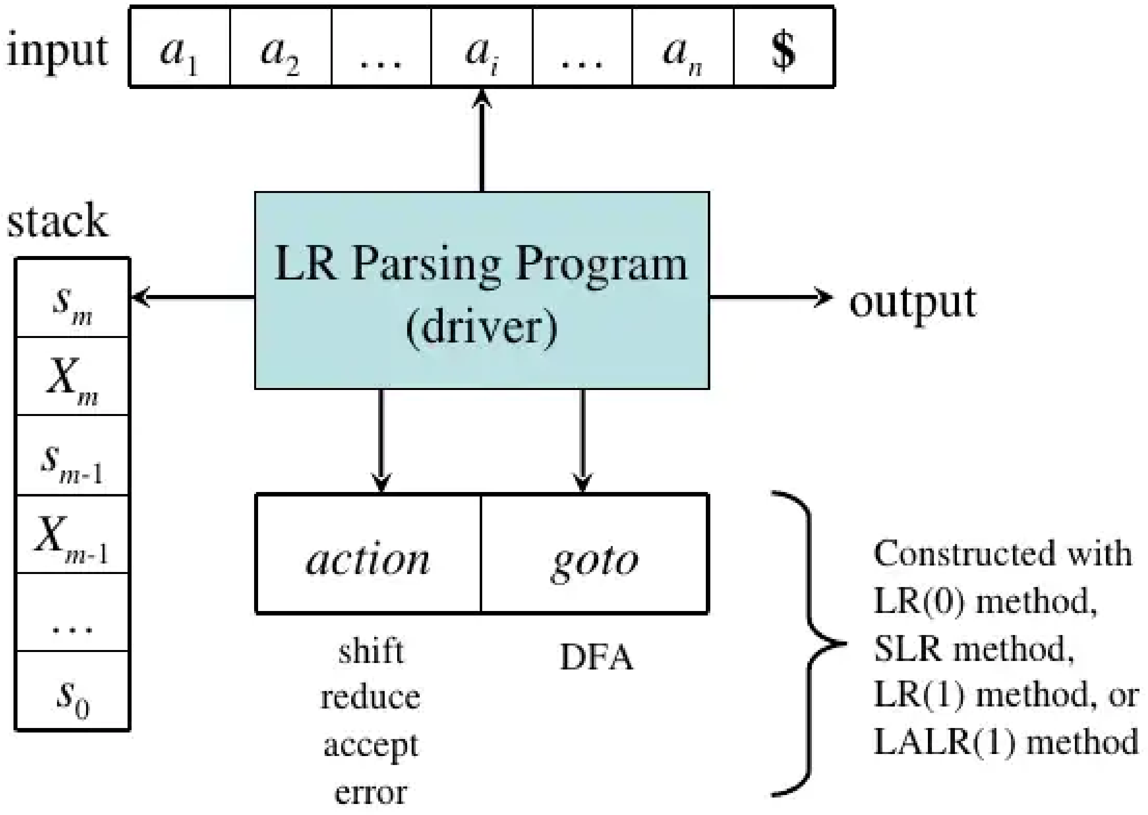
- It is **too much of work to construct** a LR parser.
- A **specialized tool**, an LR parser generator, is needed (**Eg. YACC**)

Types of LR parsing method:

1. **SLR- Simple LR**
   Easiest to implement, least powerful.
2. **CLR- Canonical LR**
   Most powerful, most expensive.
3. **LALR- Look-Ahead LR**
   Intermediate in size and cost between the other two methods.

LR PARSERS

Model of an LR Parser



The LR parsing algorithm:

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

- The **driver program is the same for all LR parser**.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $\mathbf{s_0X_1s_1X_2s_2 \ldots X_ms_m}$, where $\mathbf{s_m}$ **is on top**. Each $\mathbf{X_i}$ **is a grammar symbol** and **each** $\mathbf{s_i}$ **is a state**.
- The parsing table consists of two parts: **action** and **goto** functions.

LR PARSERS

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G. Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer. The parser then executes the following program:

```
set ip to point to the first input symbol of w$; repeat forever begin
let s be the state on top of the stack and a the symbol pointed to by ip;

if action[s, a] = shift s' then begin
        push a then s' on top of the stack; advance ip to the next input symbol end

else if action[s, a] = reduce A→β then begin
        pop 2* | β | symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A→ β
end

else if action[s, a] = accept then
        return

else error( )
end
```

SLR Parser

Presidency University, Bengaluru

Constructing SLR Parsing Table

To perform SLR parsing, take grammar as input and do the following:

1. **Find LR(0) items.**
2. **Completing the closure.**
3. **Compute goto(I,X), where, I is set of items and X is grammar symbol.**

LR(0) items:

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

For example, production A → XYZ yields the four items :

A→ . XYZ
A → X . YZ
A → XY . Z
A → XYZ .

Constructing SLR Parsing Table

Closure operation:

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:

- Initially, every item in I is added to closure(I).
- 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 new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

- Augment G and produce G'
- Construct the canonical collection of set of items C for G'
- Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Constructing SLR Parsing Table

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct C = {I_0, I_1, …. I_n}, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in I_i and goto(I_i,a) = I_j, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in I_i , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).
   (c) If [S'→S.] is in I_i , then set action[i,$] to "accept".

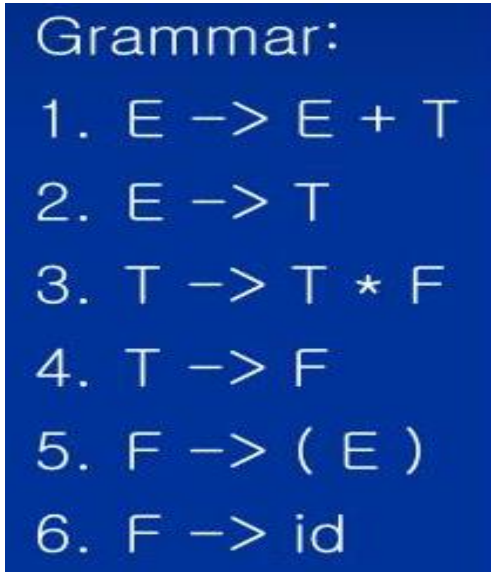
If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all non-term
   If goto(I_i,A) = Ij, then goto[i,A] = j.

4. All entries not defined by rules (2) and (3) are made "error".

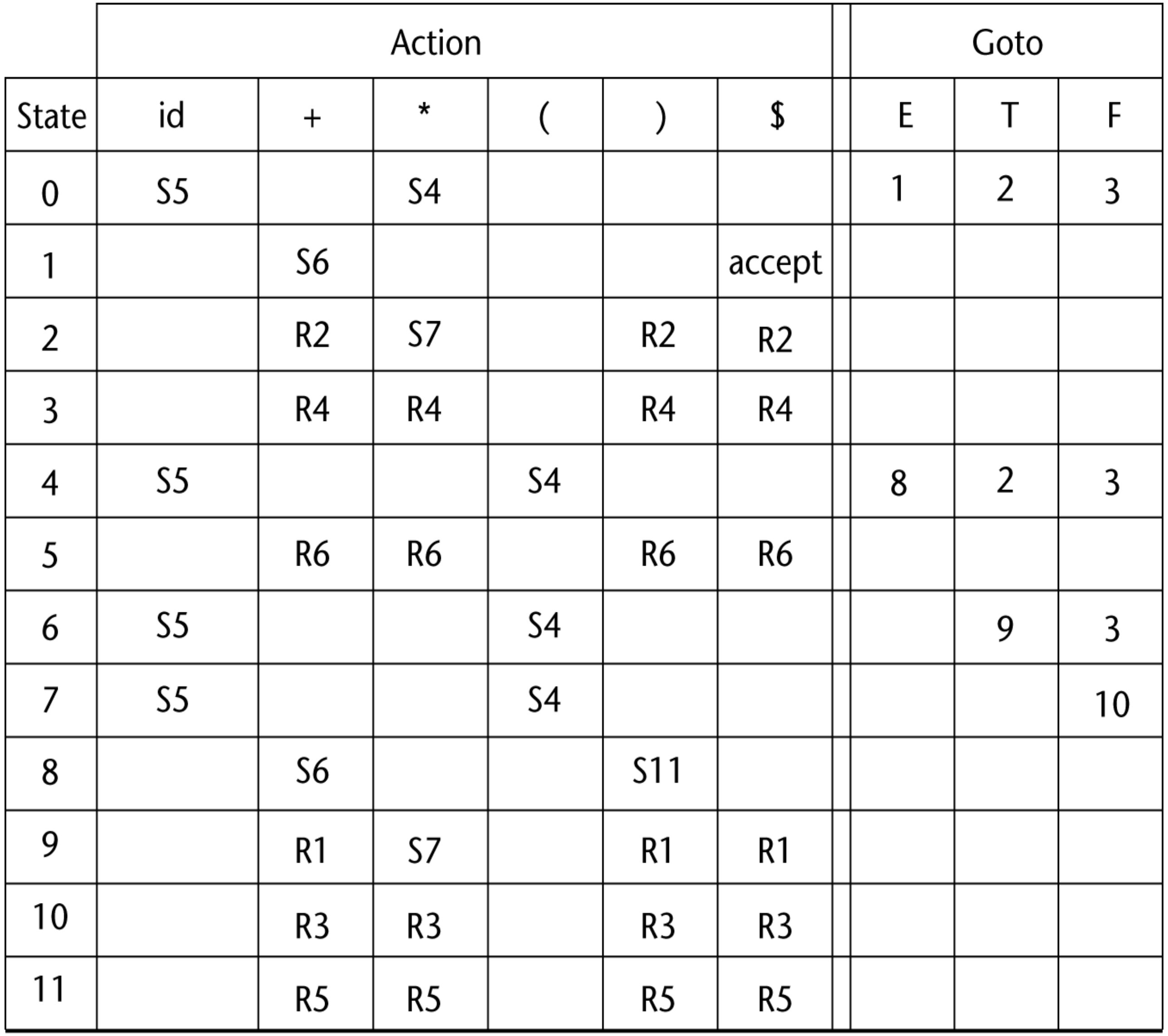
5. The initial state of the parser is the one constructed from the [S'→.S].

Constructing SLR Parsing Table



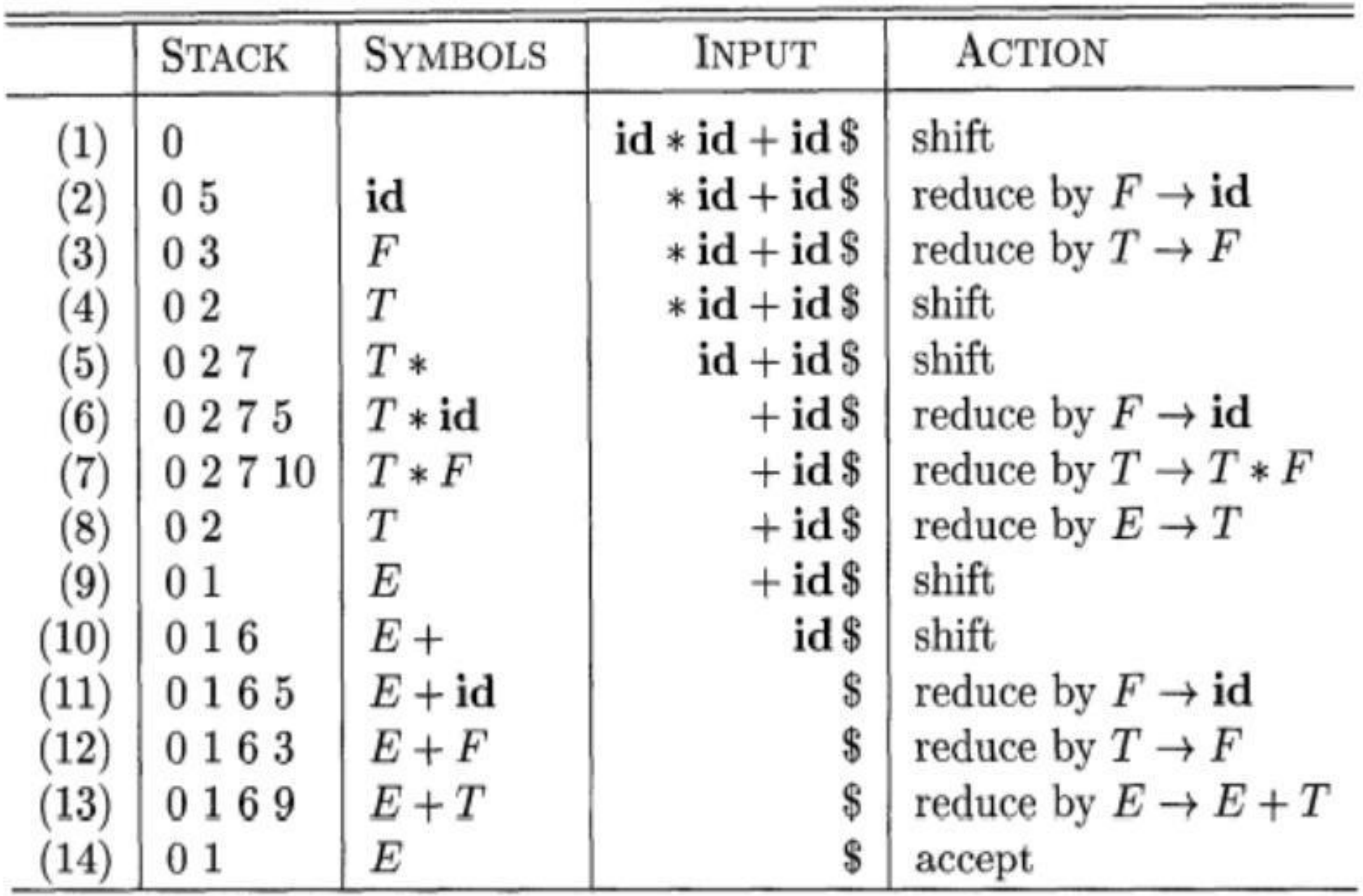
Grammar:

1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Constructing SLR Parsing Table



	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \rightarrow F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \rightarrow T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \rightarrow T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \rightarrow E + T$
(14)	0 1	E	$\$$	accept

Construct SLR Parsing Table for the following CFG

Grammar:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

Construction of SLR Parsing Table

Step1: Construct Augmented Grammar

E' → E
E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

Step2: Find LR(0) Items, Perform closure operation & Compute Goto

I_0
E' → · E
E → · E+T
E → · T
T → · T*F
T → · F
F → · (E)
F → · id

goto(I_0, E) I_1
E' → E·
E → E · +T

goto(I_0, T) I_2
E → T·
T → T· *F

I_3 goto(I_0, F)
T → F·

I_4 goto(I_0, ( )
F → (· E)
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id

I_5 goto(I_0, id)
F → id·

I_6 goto(I_1, +)
E → E+·T
T → · T*F
T → · F
F → ·(E)
F → ·id

I_7 goto(I_2, *)
T → T* · F
F → ·(E)
F → . id

I_8 4, E)
F → (E·)
E → E· +T

goto(I_4, T)
~~E → T.~~ ⇒ I_2
~~T → T. *F~~

~~goto(I_4, F)~~
~~T → F.~~ ⇒ I_3

goto(I_4, ( )
~~F → (·E)~~
~~E → .E+T~~
~~E → · T~~
~~T → .T*F~~
~~T → ·F~~ ⇒ I_4
~~F → ·(E)~~
~~F → · id~~

~~goto(I_4, id)~~
~~F → id.~~ ⇒ I_5

I_9 goto(I_6, T)
E → E+T·
T → T·*F

goto(I_6, F)
~~T → F·~~
⇒ I_3

goto(I_6, ( )
~~F → (· E)~~
~~E → · E+T~~
~~E → · T~~ ⇒ I_4
~~T → · T*F~~
~~T → ·F~~
~~F → ·(E)~~
~~F → · id~~

goto(I_6, id
~~F → id.~~ ⇒ I_5

goto(I_7, F)
I_{10}
T → T*F·

goto(I_7, ( )
~~F → (· E)~~
~~E → · E+T~~
~~E → · T~~ ⇒ I_4
~~T → · T*F~~
~~T → · F~~
~~F → ·(E)~~
~~F → · id~~

goto(I_7, id)
~~F → id·~~
⇒ I_5

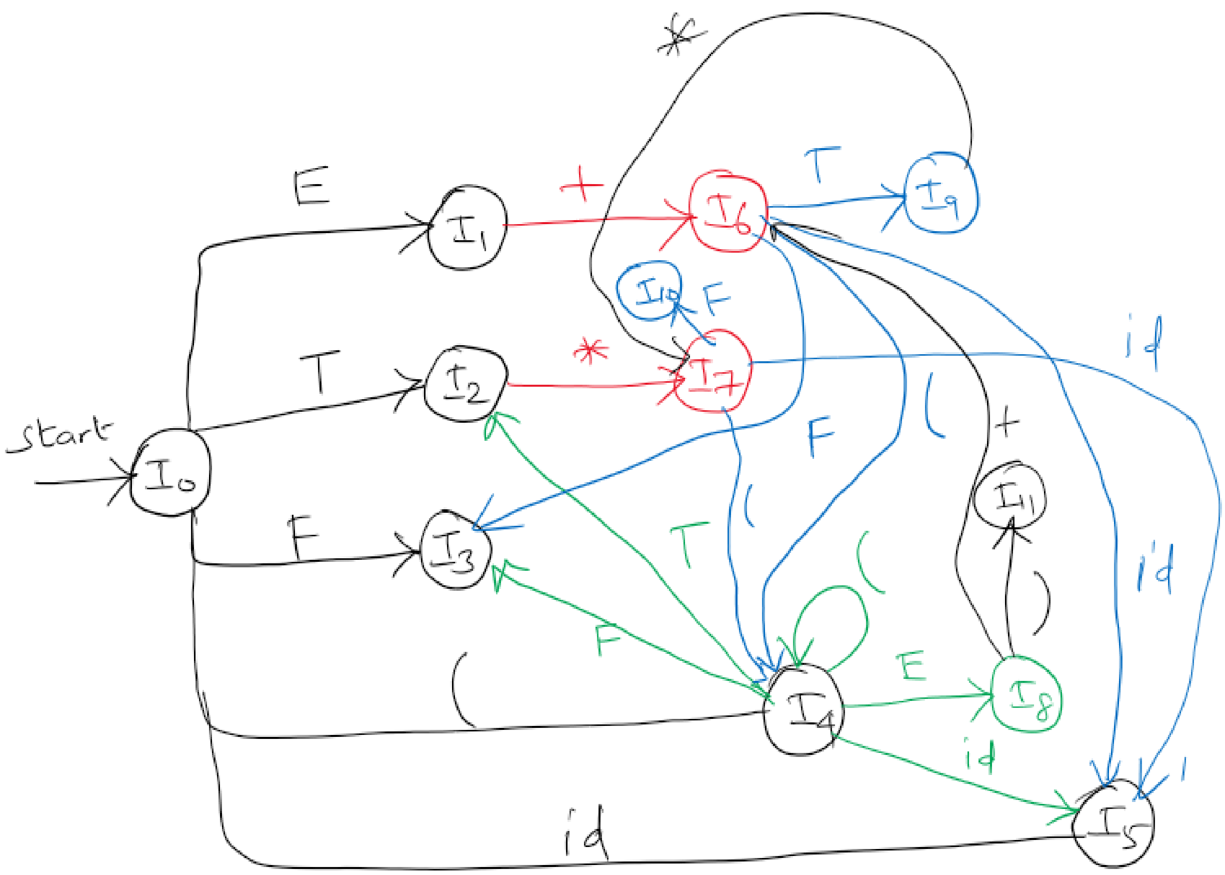
I_{11} goto(I_8,))
F → (E) ·

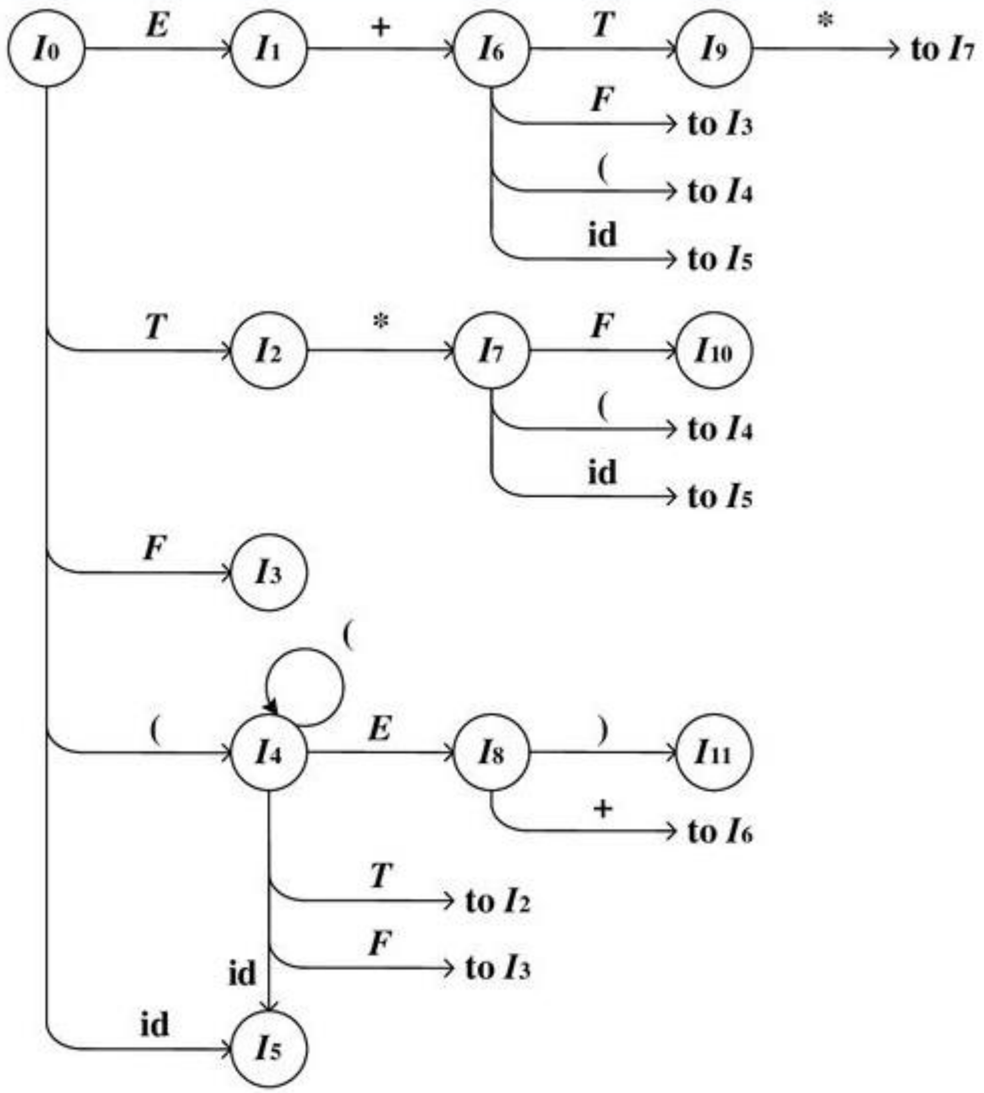
goto(I_8, +)
~~E → E+ · T~~
~~T → · T*F~~
~~T → · F~~ ⇒ I_6
~~F → ·(E)~~
~~F → · id~~

goto(I_9, *)
~~T → T* · F~~
~~F → ·(E)~~ ⇒ I_7
~~F → · id~~

Constructing SLR(1) Parsing Table

Step 3: Draw Transition Diagram

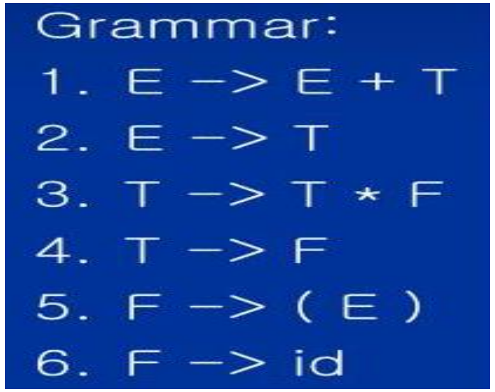




Constructing SLR(1) Parsing Table

Step 4: Find the FOLLOW set for all the Non-Terminals to perform Reduction

FOLLOW(E) = { +,) }
FOLLOW(T) = { +,), * }
FOLLOW(F) = { +,), * }



Grammar:
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

SLR Parsing Table:

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Construction of SLR Parsing Table

2. Construct the SLR Parsing Table for the following Grammar

S → (L)
S → a
L → L,S
L → S

3. Construct the SLR Parsing Table for the following Grammar

S → L=R
S → R
L → *R
L → id
R → L

4. Construct the SLR Parsing Table for the following Grammar

S → aAd
S → bBd
S → aBe
S → bAe
A → c
B → c