Unit – 2

I. CFG:

Overview of Context Free Grammar. Derivations and Parse trees, Ambiguity, Left Recursion, Left factoring.

II. Top down Parsing:

Recursive descent parsing and Predictive parsers.

III. Bottom up Parsing:

Shift-Reduce Parsers. Operator Precedence Parsers, LR Parsers.

IV. YACC:

YACC parser generator

1. SYNTAX ANALYSIS / PARSING

1.1 The Role of the Syntax Analyzer (Parser)

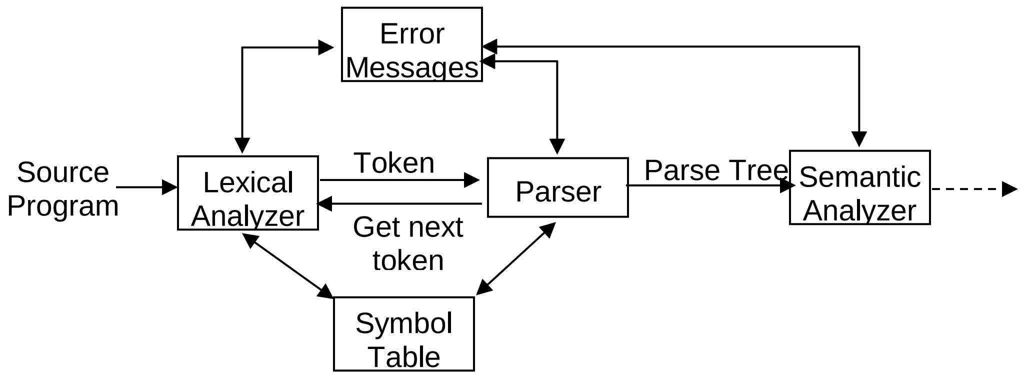


Fig 3.1: Location of the Parser

- The Parser obtains a string of tokens from the lexical analyzer and verifies that the string of tokens can be generated by the grammar for the source language.
- The parser constructs the parse tree and passes it to the rest of the compiler phases for further processing.
- It reports the presence of syntax error and also recovers from commonly occurring errors to continue processing the reminder of the program.
- There are two general types of parsers for grammars
  - Top down parsing – which build the parse tree from root to leaves.
  - Bottom up parsing – which build the parse tree from leaf and work up the root.

Parsing

- Parsing is the process of determining a string of tokens that can be generated by a grammar" or not. For any CFG, there is a parser that parses a string of n tokens.
- Programming language parsers almost always make a single left-to-right scan over the input, looking ahead one token at a time.

1.2 Context Free Grammars

- Grammars are used to systematically describe the syntax of programming language constructs like expressions and statements.

- **Significance of Grammars in any Programming Language:**
  Grammars offer significant benefits for both language designers and Compiler writers.
  - A Grammar gives a precise, easy to understand, syntactic specification of a programming language.
  - The Parser (syntax analyzers) built from a grammar that determines the syntactic structure of the source program.
  - The structure imparted to a language by a properly augmented grammar is useful for
      - translating source program into object program
      - detecting errors
  - A Grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.

- A **Context-Free Grammar** (CFG) consists of Terminals, Non-terminals, a Start symbol, and Productions.
  - **Terminals** are the basic symbols from which strings are formed. The term the terminals are the first components of the tokens output by the lexical analyzer.
  - ***Non-Terminals*** are syntactic variables that denote sets of strings. The sets of strings denoted by non-terminals help define the language generated by the grammar. Non-terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
  - In a grammar, one non-terminal is distinguished as the ***Start symbol,*** and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
  - The **productions** of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.
    - Each *production* consists of:
      - A non-terminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.
      - A *body* or *right side* consisting of zero or more terminals and non-terminals.
      - The components of the body describe one way in which strings of the non-terminal at the head can be constructed.

Notational Conventions:

- **Terminals:** Lowercase letters early in the alphabet, such as *a, b,* Operators, Punctuation symbols such as parentheses and the digits.
- **Non-terminals**: Uppercase letters early in the alphabet, such as *A, B, C.* Lowercase, italic names such as *expr* or *stmt.*
- **Grammar Symbol** that is, either non-terminals or terminals, Uppercase letters late in the alphabet, such as *X,* Y, *Z,* represent single *grammar symbol*. Lowercase Greek letters ***α, β, γ*** represent (possibly empty) strings of grammar symbols.
- A generic **production** can be written as *A→α,* where *A* is the head and '*α' the* body.
  A set of productions *A→ αi, A → α2,... , A→αk* with a common head *A* (call them *A-productions)*, may be written *A* -> *αi* | *α*2 | • • • | *αk.*

1.3 Verifying the language generated by the Grammar

- A Grammar G generates a language L has two parts: Every string generated by G is in L and conversely every string in L can indeed be generated by G.
- The *language generated by* a grammar is its set of sentences. Thus, a string of terminals *w* is in *L(G),* the language generated by *G,* if and only if *w* is a sentence of *G.*
- **Sentential form** may contain both terminals and non-terminals, and may be empty.
- A ***sentence or string* of *G*** is a sentential form with no non-terminals.
- If two grammars generate the same language, the grammars are said to be *equivalent.*
- The techniques that are used to verify the given string is generated by G are,
  - Derivation
  - Reduction

(i) Derivation

- The derivation corresponds to the top down construction of a parse tree.
- Derivation beginning with the Start symbol of the grammar G.
- Each rewriting step replaces a non-terminal by the body of (rhs) one of its productions.
- Derivation can be Left-most and Right-most derivation in which the non-terminal to be replaced at each step is chosen.
  - In **Left-most Derivation (LMD)**, the leftmost non-terminal in each sentential form is always chosen.
  - In ***Rightmost* derivations (RMD),** the rightmost non-terminal is always chosen.

Example:

Consider the Grammar,

S → (L) | a

L → L , S | S

and the input string (a,(a,a))

Left-most Derivation		**Right-most Derivation**	
S	Expanded by S → (L)	**S**	Expanded by S → (L)
(**L**)	Expanded by L→L,S	(**L**)	Expanded by L→L,S
(**L**,S)	Expanded by L → S	(L,**S**)	Expanded by S → (L)
(**S**,S)	Expanded by S →a	(L,(**L**))	Expanded by L → L,S
(a,**S**)	Expanded by S → (L)	(L,(L,**S**))	Expanded by S →a
(a,(**L**))	Expanded by L → L,S	(L,(**L**,a))	Expanded by L → S
(a,(**L**,S))	Expanded by L → S	(L,(**S**,a))	Expanded by S → a
(a,(**S**,S))	Expanded by S →a	(**L**,(a,a))	Expanded by L→ S
(a,(a,**S**))	Expanded by S →a	(**S**,(a,a))	Expanded by S → a
(a,(a,a))	**Verified**	**(a,(a,a))**	**Verified.**

(ii) Reduction

- Reduction is the process of "reducing" a string *w* to the start symbol of the grammar.
- At each *reduction* step, a specific substring matching the body (RHS) of a production is replaced by the non-terminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and- about what production to apply, as the parse proceeds. Formally a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
  - Example Consider the Grammar,

    S → (L) | a

    L → L , S | S

    and the input string (a,(a,a))

Sentential Form	String	Handle	Reduction Steps (handle pruning)
(	(a,(a,a))		
(**a**	,(a,a))	a	Reduce by S → a
(**S**	,(a,a))	S	Reduce by L → S
(L ,	(a,a))		

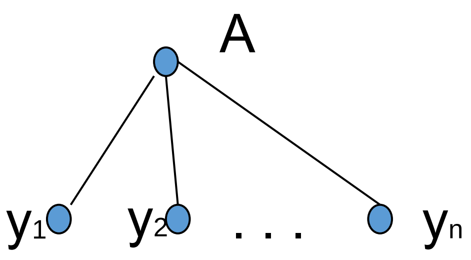
(L, (	a,a))		
(L,(**a**	,a))	a	Reduce by S → a
(L,(**S**	,a))	S	Reduce by L → S
(L,(L,	a))		
(L,(L,**a**	))	a	Reduce by S → a
(L,(**L,S**	))	L,S	Reduce by L → L,S
(L,**(L)**	)	(L)	Reduce by S → (L)
(L,S	)	L,S	Reduce by L → L,S
(L)	End	(L)	Reduce by S → (L)
S			Start Symbol of the Grammar.

1.4 Parse Tree

- A parse tree is a graphical representation for a derivation that filters out the choice regarding replacement order.
- It pictorially shows how the start symbol of a grammar derives a string in a language.
-

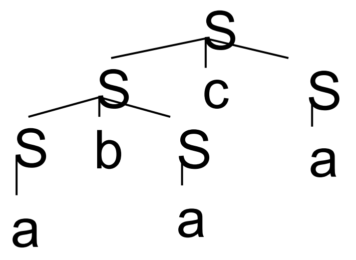
Properties of Parse Tree

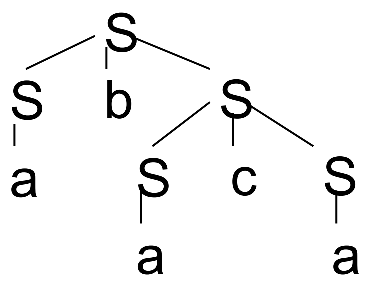
1. The Root is labeled by the start symbol of G.
2. Each leaf is labeled by a Token or by ϵ.
3. Each interior node is labeled by a non-terminal
4. If there is a production A → $Y_1Y_2 \ldots Y_n$ then A is a interior node with the children of that node are X_1, X_2 and $X_3$from left-to-right.



Example: Consider the CFG G: S → SbS | ScS | a and the input string "abaca"

The left-most derivation and right-most derivation of the string can be represented using a parse tree





1.5 ELIMINATION OF LEFT RECURSION

- A Grammar is left recursive if it has a non-terminal A, such that there is derivation A → Aα for some string α. Top Down parsers cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

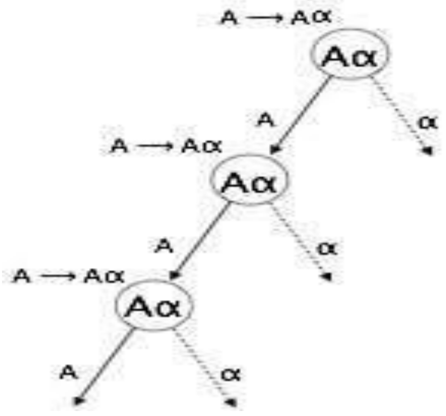
  There is a production of the form

  A → A α | β

  can be replaced by the non-left recursive pairs,

  A → βA'

  A' → αA' | ϵ, where A' is a new non-terminal.



Example:

E → E + T | T

E → $\underline{E + T}$ | $\underline{T}$

A → A α | β

After eliminating left-recursion,

E → T E'

E' → +TE' | ϵ

1.6 ELIMINATION OF LEFT FACTOR

- When the choice between two alternatives A – productions is not clear, a parser may be able to rewrite the productions to defer the decision enough of the input has been seen that can make the right choice.
- For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If there is a nontrivial common prefix replaces all of the A –productions
  A → αβ1| αβ2 | …| αβn | γ, where γ represents all alternatives that do not begin with α, then

  A → αβ1| αβ2 | …| αβn | γ

  can be rewritten as

  A → αA' | γ

  A' → β1| β2 | …| βn

  Where A' is a new non-terminal.

Example:

S → <u>iEtS</u> | <u>iEtS</u>eS |a

A → α β1| α β2 | γ

S → <u>iEtS</u> | <u>iEtS</u>eS |a

A → α β1| α β2 | γ

After eliminating left-factor,

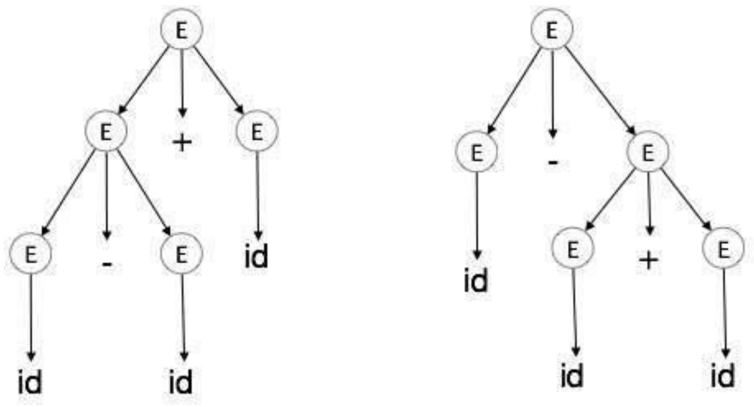
S → iEtS S' | a

S' → eS | ϵ (since β1 is ϵ)

AMBIGUITY:

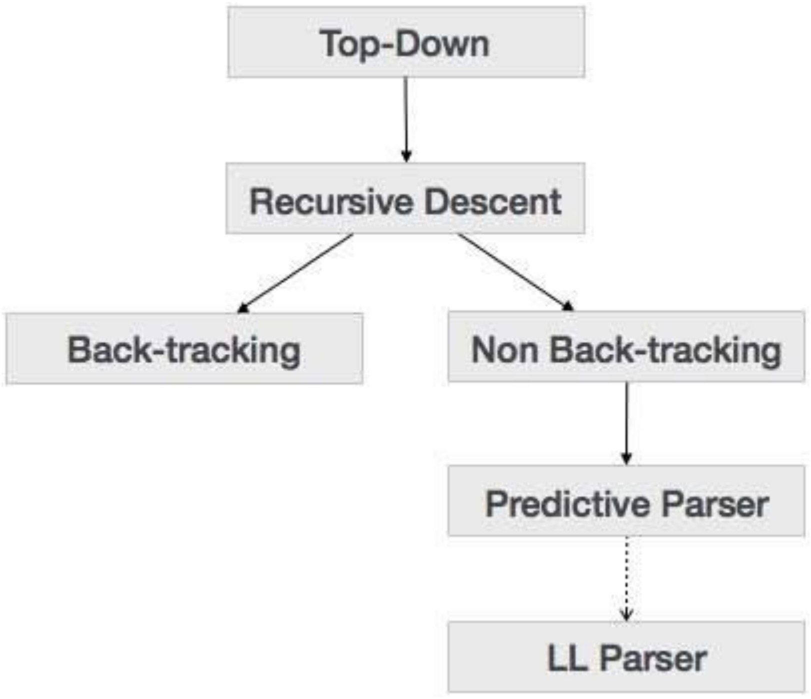
A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example: E → E + E;E → E − E; E → id



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

II. TOP DOWN PARSING



2.1 RECURSIVE DESCENT PARSER

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

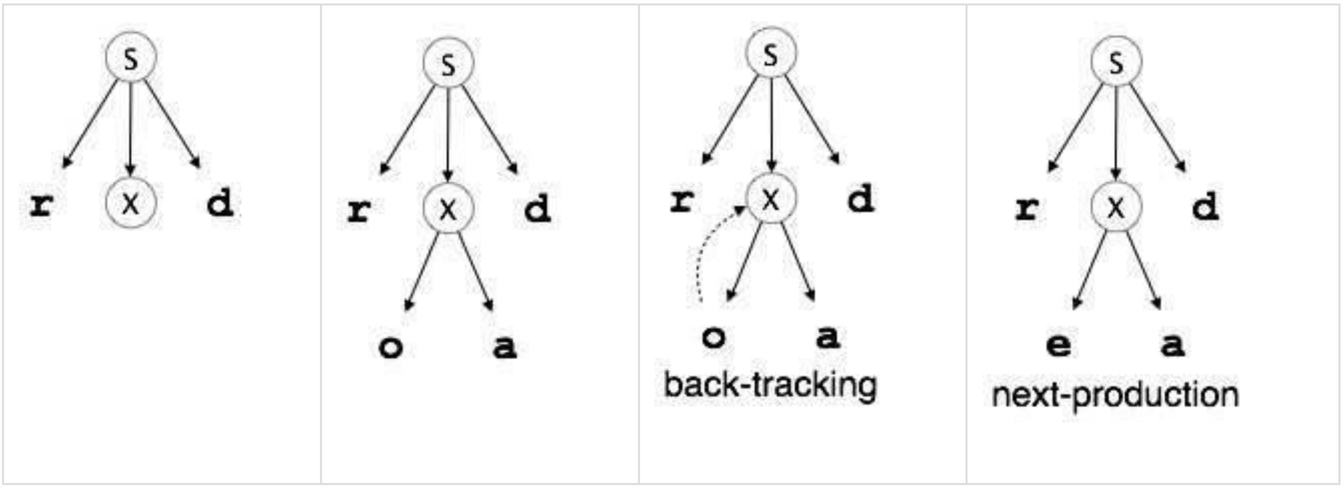
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

```
S → rXd | rZd
X → oa | ea
Z → ai
```

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

Now the parser matches all the input letters in an ordered manner. The string is accepted.



2.2 Non Recursive Predictive Parser / LL (1) Parser

- **Top Down parsing can be viewed as the process of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.**
- **Top-down parsing can also be viewed as finding a leftmost derivation for an input string.**

Construction of Predictive LL (1) Parser

1. Before constructing the Predictive LL (1) parsers we have to
   a. Eliminate ambiguity,
   b. Eliminate left-recursion and
   c. Perform left factor where required.
2. For construction of Predictive LL (1) parser, follow the following steps:
   a. Computation of FIRST
   b. Computation of FOLLOW
   c. Construct the Predictive Parsing Table using FIRST and FOLLOW sets
3. Parse the input string

Computation of FIRST: The Computation of FIRST allows the parser to choose which production to apply, based on the first input symbol.

Definition – FIRST: FIRST (α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

__Algorithm__

(R1) If x is terminal then **FIRST(x) = {x}**

(R2) If there is a production A → ε, then **FIRST (A) = {ε}**

(R3) If A → $X_1X_2X_3 \ldots X_k$ is a production, where X is a grammar symbol, then

If FIRST(X1) ≠ {ε} then **FIRST (A) = FIRST(X1).**

If FIRST(X1) = {ε} then FIRST (A) = FIRST(X2) − {ε}.

If FIRST(X1) = {ε} and FIRST (X2) = {ε} then FIRST(A) = FIRST(X3) − {ε}.

…

If ε is in the FIRST set for every X_k then FIRST (A) = ε

__Computation of FOLLOW__: **Follow (A) for a non-terminal A, to be the set of terminals a such that can appear immediately to the right of A in some sentential form.**

__Algorithm__

(R1) Include **$ in the Follow (S)**, where S is the start symbol of the grammar

(R2) If A → αBβ, then **FOLLOW (B) = FIRST (β) except ϵ**

(R3) If (A → αB) or (A → αBβ and FIRST (β) has ϵ), then **FOLLOW (B) = FOLLOW (A)**

__Construction of Parsing Table – Algorithm__

(R1) For each terminal 'a' in FIRST (α), **add A → α to M [A, a]**

(R2) If ϵ is in FIRST (α), then **add A→ ϵ to M [A, b] for every symbol 'b' in FOLLOW (A).**

(R3) If ϵ is in FIRST (α) and $ is in FOLLOW (A) then **Add A→ α to M [A, $].**

(R4) Make all undefined entries of M be **ERROR.**

__Parsing the Input string - Algorithm__

Let X is the symbol on top of the stack and 'a' is the current input symbol,

(R1) If X is a terminal and X=a=S, then the parser halts and announce the successful completion of parsing.

(R2) If X is a terminal and X=a≠S, then Pop-off X from the stack and advance the input pointer.

(R3) If X is a Non-terminal then consult an entry M [X, a] of the parsing table,

- If M [X, a] = {X → UVW}, then the parser replaces X on top of the stack by UVW, with U on top of the stack.
- If M [X, a] = Undefined, then the parser calls an error recovery routine.

__Example__:

Construct the LL (1) parsing table for the following grammar and parse the string id + id * id.

E → E + T | T

T → T * F | F

F → (E) | id

Given:

G: E → E + T | T

T → T * F | F

F → (E) | id

Step -1: Elimination of Left Recursion

After Eliminating the Left Recursion from E and T productions,

E→TE'

E'→+TE' | ϵ

T→ FT'

T'→*FT' | ϵ

F→(E) | id

Step -2: Computation of FIRST

FIRST (E)

E → TE'

First (E) = First (T)………..(1)

FIRST (E')

E'→+TE'   E'→ ϵ

First (E') = {+, ϵ}

FIRST (T)

T → FT'

First (T) = First (F)………..(2)

FIRST (T')

T'→*FT'   T'→ ϵ

First (T') = {*, ϵ}

FIRST (F)

F→(E)   F→ id

First (F) = {(, id}

Substituting First (F) in (2)

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

Substituting First (T) in (1)

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

Step -3: Computation of FOLLOW

FOLLOW (E)

FOLLOW (E) = {$} by [R1]

F→(E)

FOLLOW (E) =First ()) = {$,)} by [R2]

FOLLOW (E')

E→TE'

Follow(E') = Follow(E) by [R3] ={$,)}

E'→+TE'

Follow(E') = Follow(E') by [R3] ={$,)}

FOLLOW (T)

E→TE'

Follow(T) = First (E') Except ϵ by (R2) = {+}

Follow(T) = Follow (E) Since First(E') has ϵ

Follow(T) = {+,$,) }

E'→+TE'

Follow (T) = Follow (E') by [R3] = {+,$,) }

FOLLOW (T')

T→ FT'

Follow(T') = Follow (T) by [R3] = {+,$,) }

T'→*FT'

Follow(T’) = Follow(T’) by [R3] = {+,$,) }

<u>FOLLOW (F)</u>

T→ FT'

Follow(F) = First (T’) Except ϵ by (R2) = {*}

Follow(F) = Follow (T) Since First(E’) has ϵ

Follow (F) = Follow(T) = {+,$,) }

T'→*FT'

Follow (F) = Follow (T’) by [R3] = {*,+,$,) }

Symbol	FIRST	FOLLOW
E	(, id	$,)
E’	+,ϵ	$,)
T	(, id	+, $,)
T’	*, ϵ	+, $,)
F	(, id	*, +, $,)

Step – 4: Construction of Parsing Table

M	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)	

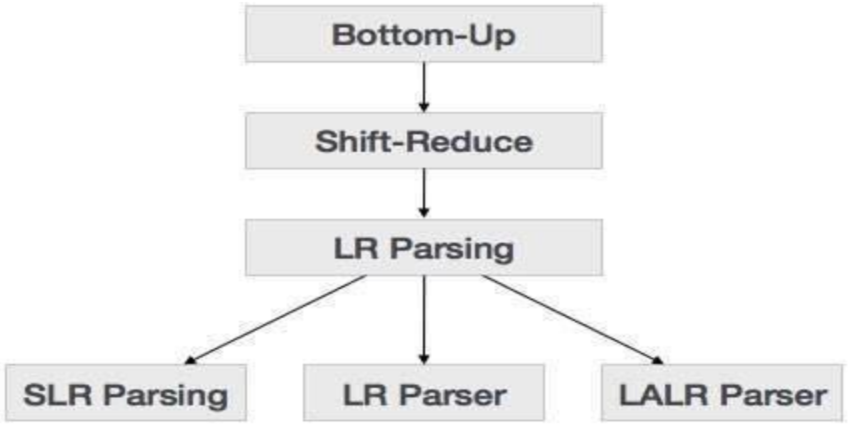
Step 5: Parsing the String id+id*id

Stack	Input buffer	Action taken
$E	id+id*id$	M[E,id] = E → TE’
$E'T	id+id*id$	M[T,id] = T → FT’
$E'T'F	id+id*id$	M[F,id] = F → id
$E'T'~~id~~	~~id~~+id*id$	Pop off & Advance the Input pointer
$E'T'	+id*id$	M[T’,+] = T’ → ϵ
$E'	+id*id$	M[E’,+] = E’→+TE’
$E'T~~+~~	~~+~~id*id$	Pop off & Advance the Input pointer
$E'T	id*id$	M[T,id] = T → FT’
$E'T'F	id*id$	M[F,id] = F → id
$E'T'~~id~~	~~id~~*id$	Pop off & Advance the Input pointer
$E'T'	*id$	M[T’,*] = T’→*FT’
$E'T'F~~*~~	~~*~~id$	Pop off & Advance the Input pointer
$E'T'F	id$	M[F,id] = F → id

$E'T'~~id~~	~~id~~$	Pop off & Advance the Input pointer
$E'T'	$	M[T',$] = T'→ ϵ
$E'	$	M[E',$] = E'→ ϵ
$	$	Accepted

More problems on LL (1) Parser: please refer the class work note-book.

III. BOTTOM-UP PARSING TECHNIQUES



- A bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- Bottom-up parsers generally operate by choosing, on the basis of the next input symbol (look-ahead symbol) and the contents of the stack, whether to shift the next input onto the stack, or to reduce some symbols at the top of the stack.
- A reduce step takes a production body at the top of the stack and replaces it by the head of the production

3.1 Shift Reduce Parsing (SR parser)

- Shift – Reduce Parsing is a form of bottom-up parsing in which a stack holds the grammar symbols and an input buffer holds the rest of the string to be parsed.
- It uses $ to mark the bottom of the stack and also the right end of the input.
- There are four possible actions a Shift-Reduce parser can make,
  1. **SHIFT** – Shift the next input symbol onto the top of the stack.
  2. **REDUCE** – The right end of the handle must be at the top of the stack, locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
  3. **ACCEPT** - Announce the successful completion of parsing
  4. **ERROR** - Discover a syntax error and call an error recovery routine.

Example:

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

Input string: id * id

Stack implementation of SR Parser.

Stack	Buffer	Handle	Action
$	id * id $		Shift
$ id	* id $	id	Reduce by F → id
$ F	* id $	F	Reduce by T → F
$ T	* id $		Shift
$ T *	id $		Shift
$ T * id	$	id	Reduce by F→ id
$ T * F	$	F	Reduce by T → T * F
$ T	$	T	Reduce by E → T
$ E	$		Accept

3.2 OR PARSER – OPERATOR PRECEDENCE PARSER

Operator Grammar: An Operator Grammar is a grammar with the property that

(i) No production has ϵ on its right side

(ii) No production has two adjacent Non-terminals

Operator Precedence Grammar:

- An Operator precedence grammar is an operator grammar with the disjoint relation (includes <, > and =) between any pair of terminals
- There are three precedence relations exists between any pair of terminals a and b as

  a <· b, a gives precedence to b

  a ·> b, a takes precedence over b

  a ≐ b, a has equal precedence as b

OR Parser – Steps

1. Check for Operator Grammar
2. Computation of LEADING

3. Computation of TRAILING
4. Computation of PRECEDENCE RELATION
5. Construction of Precedence Table
6. OR Parsing

Computation of LEADING – Algorithm

(Rule 1) ‘a’ is in LEADING (A) if, **A → aY or A → XaY** where X, Y are grammar symbols.

(Rule 2) If **A → BX** is a production, then
Include every terminal in LEADING (B) to LAEADING (A).

Computation of TRAILING – Algorithm

(Rule 1) ‘a’ is in TRAILING (A) if, **A → Xa or A → XaY** where X, Y are grammar symbols.

(Rule 2) If **A → XB** is a production, then
Include every terminal in TRAILING (B) to TRAILING (A).

Computation of PRECEDENCE RELATION – Algorithm

(Rule 1) Let S is a start symbol of the grammar,
Set $ <· a, for all symbols ‘a’ in LEADING(S)
Set b ·> $, for all symbols ‘b’ in TRAILING(S).

(Rule 2) If A → XY or A → XBY, where X and Y are terminals, then
Set X = Y.

(Rule 3) If A → XB where X is a terminal, then
Set X <· a, for all symbols ‘a’ in LEADING(B)

(Rule 4) If A → BX where X is a terminal, then
Set b ·> X, for all symbols ‘b’ in TRAILING(B).

PRECEDENCE Parsing – Algorithm

(Rule 1) If **Top of the Stack and current Input symbol is $**, then
ACCEPT and announce successful completion of parsing.

(Rule 2) If **Top = a, current Input symbol is b and a <· b or a = b,** then
SHIFT b onto the stack.

(Rule 3) else If **a ·> b,** then **REDUCE.**
Pop off all the symbols until the backward scan reaches the first <· from top of the stack.

(Rule 4) Announce **ERROR** if undefined entry encountered.

Example: Please refer class-work note book.

3.3 LR PARSERS

- The LR parser is a non-recursive, shift-reduce, bottom-up parser.
- It uses a wide class of context free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as **LR(k)** parsers,

  Where
  - L stands for left-to-right scanning of the input stream;
  - R stands for the construction of right-most derivation in reverse, and
  - K denotes the number of look-ahead symbols to make decisions.
- There are three widely used algorithms available for constructing an LR parser:
  1. **SLR(1) – Simple LR Parser**:
     - Works on smallest class of grammar
     - Few number of states, hence very small table
     - Simple and fast construction
  2. **CLR(1) – Canonical LR Parser**:
     - Works on complete set of LR(1) Grammar
     - Generates large table and large number of states
     - Slow construction
  3. **LALR(1) – Look-Ahead LR Parser**:
     - Works on intermediate size of grammar
     - Number of states are same as in SLR(1)

Organization of LR Parsers: The LR Parser consists of
Input Buffer – to hold the input string to be parsed
Stack – holds a sequence of states of LR Items set and the grammar symbols.
Driver Program – is the same for all LR Parsers. Only the parsing table changes from one parser to another.
Parsing table: The parsing table consists of two parts:

(1) a parsing function ACTION and
(2) a goto function GOTO.

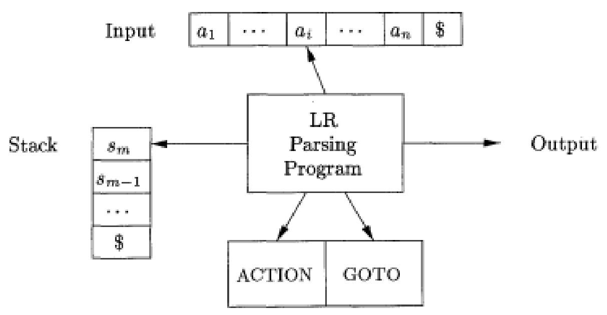


Figure 4.35: Model of an LR parser

1. The ACTION function takes as arguments a state *i* and a terminal *a* (or $, the input end marker). The value of ACTION[*i*, *a*] can have one of four forms:
   a. Shift *j*, where *j* is a state. The action taken by the parser effectively shifts input *a* to the stack, but uses state *j* to represent *a*.
   b. Reduce A → β. The action of the parser effectively reduces β on the top of the stack to head A.
   c. Accept. The parser accepts the input and finishes parsing.
   d. Error. The parser discovers an error in its input and takes some corrective action.
2. Extend the GOTO function, defined on sets of items, to states: if GOTO[I_i, A] = I_j, then GOTO also maps a state *i* and a non terminal A to state *j*.

SLR – Simple LR Parser

- SLR Grammar: A grammar for which an SLR parser can be constructed is said to be SLR Grammar.
- **Steps to construct the SLR Parser**
  1. Computation of LR (0) Items for the given grammar G
  2. Construction of SLR Parsing table using LR (0) items
  3. LR Parsing the Input String

LR (0) Item

- An LR Item of a grammar G is a production of G with a DOT at some position of the RHS.

  (1) A production A→ XYZ yields the four items

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

  (2) A Production A→ ϵ generates only one item as A→ .
- LR (0) Items provides basis for constructing SLR Parser.

LR Items – Classification

The LR Items can be classified into two different classes as

Kernel Items – It include the Initial Item S' → S and all other items whose dots are not at the leftmost end.

Non-Kernel Items - It include items which have their dots at their leftmost end.

Computation of LR (0) Items – Steps

LR (0) Items can be computed using three steps

1. Augmented Grammar
2. Closure Function
3. GOTO Function

(i) Augmented Grammar: If G is a grammar with the start symbol S, then the augmented grammar G' for G is formed with new start symbol S' and production S'→ S.

Example:

G: E→ E + E | id

The augmented grammar

G': **E' → E**

E → E + E | id

Significance of Augmented Grammar:

- An Augmented Grammar indicates to the parser, when it should stop parsing and announce the acceptance of the inputs.
- The parsing process is completed when a parser is about to reduce S' → S.

(ii) CLOSURE Function:

Let I is a set of Items for a grammar G, then Closure (I) is constructed from I by using 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 item B→. γ into I, if it is not there.

(iii) GOTO Function:

The function GOTO (I, X), where X is a Grammar Symbol,

If A → α.Xβ is in I then GOTO (I, X) defined as the CLOSURE (A→ αX.β)

2. Construction of SLR Parsing table – Algorithm

Let C = {I_0, I_1, I_2, …I_n} be the collection of LR (0) Items for G' and Let I_j is a set in C,

(R1) If GOTO (I_j , a) = I_k then set ACTION [j,a] = SHIFT K (or) S_k

(R2) If GOTO (I_j , A) = I_k then set GOTO [j,A] = K

(R3) If A → α. is in set I_j then set ACTION [j,a] = REDUCE BY A → α for every symbol 'a' in FOLLOW (A)

(R4) If S' → S. is in set I_j then set ACTION [j,$] = ACCEPT

(R5) All the undefined entries are ERROR.

Example: Construct the Simple LR Parsing table for the following grammar S → CC, C → eC, C → d. And parse the string eded.

Given:

G: S → CC
C → eC
C → d

<u>Computation of LR (0) Items</u>

<u>1. Augmented Grammar</u>

G': S' → S
S → CC
C → eC
C → d

<u>2. CLOSURE (S'→ S)</u>

S' → .S
S → .CC
C → .eC
C → .d ………..(I_0)

<u>3. GOTO (I_0,S)</u>

S' → S. ………(I_1)

<u>GOTO (I_0,C)</u>

S → C.C
C → .eC
C → .d ………(I_2)

<u>GOTO (I_0,e)</u>

C → e.C
C → .eC
C → .d ………(I_3)

<u>GOTO (I_0,d)</u>

C → d. ………(I_4)

<u>GOTO (I_2,C)</u>

S → CC. ………(I_5)

<u>GOTO (I_2,e)</u>

C → e.C
C → .eC
C → .d ………(I_3)

<u>GOTO (I_2,d)</u>

C → d. ………(I_4)

<u>GOTO (I_3,C)</u>

C → eC. ………(I_6)

<u>GOTO (I_3,e)</u>

C → e.C
C → .eC
C → .d ………(I_3)

<u>GOTO (I_3,d)</u>

C → d. ………(I_4)

<u>Construction of SLR Parsing Table</u>

State	ACTION			GOTO	
	e	d	$	S	C
0	S_3	S_4		1	2

1			ACC		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

SLR Parsing

No	STACK	BUFFER	ACTION
1	0	eded $	ACTION [0,e] = S_3 Shift
2	0e3	ded $	ACTION [3,d] = S_4 Shift
3	0e3d4	ed $	ACTION [4,e] = R_3 Reduce by C $\rightarrow$ d
	0e3C6	ed $	Pop-off d4, Push C. GOTO(3,C)=6. Push 6
4	0e3C6	ed $	ACTION [6,e] = R_2 Reduce by C $\rightarrow$ eC
	0C2	ed $	Pop-off e3C6, Push C. GOTO(0,C)=2. Push 2
5	0C2	ed $	ACTION [2,e] = S_3 Shift
6	0C2e3	d $	ACTION [3,d] = S_4 Shift
7	0C2e3d4	$	ACTION [4,$] = R_3 Reduce by C $\rightarrow$ d
8	0C2e3C6	$	ACTION [6,$] = R_2 Reduce by C $\rightarrow$ eC
9	0C2C5	$	ACTION [5,$] = R_1 Reduce by S $\rightarrow$ CC
10	0S1	$	ACTION [1,$] = Accept

Problems on SLR Parsers – Please refer the Class Work Note-Book.

CLR – Canonical LR Parser

- The CLR parser incorporates extra information in the state by redefining items to include a terminal symbol as a look-ahead as a second component.
- The general form of an Item is
  A → α.Bβ for SLR | LR (0) Item
  becomes
  A → α.Bβ, a
  Where a is a terminal or a right-end marker $, and the item is said to be LR (1) Item.
- **Steps to construct the SLR Parser**
  1. Computation of LR (1) Items for the given grammar G
  2. Construction of CLR Parsing table using LR (1) items
  3. Parsing the Input String

(ii) CLOSURE to compute LR(1) Item:

An Item of the form

[A → α.Bβ, a] in the set I and B→ γ is a production it can be added into I as

[A → α.Bβ, a]
[B→ .γ, b]
Where b = FIRST(βa).

Example: Construct the canonical LR Parsing table for the following grammar
S → CC, C → eC, C → d.

Computation of LR (1) Items

1. Augmented Grammar

G': **S' → S**
S → CC
C → eC
C → d

2. CLOSURE (S'→ S)

S' → .S, $
S → .CC, $
C → .eC, e|d
C → .d, e|d ....(I_0)

3. GOTO (I_0,S)

S' → S., $ ....(I_1)

GOTO (I_0,C)

S → C.C, $
C → .eC, $
C → .d, $ ....(I_2)

GOTO (I_0,e)

C → e.C, e|d
C → .eC, e|d
C → .d, e|d ....(I_3)

GOTO (I_0,d)

C → d., , e|d ....(I_4)

GOTO (I_2,C)

S → CC., $ ....(I_5)

GOTO (I_2,e)

C → e.C, $
C → .eC, $
C → .d, $ ....(I_6)

GOTO (I_2,d)

C $\rightarrow$ d., \$ ....(I_7)

GOTO (I_3,C)

C $\rightarrow$ eC. , e|d ....(I_8)

GOTO (I_3,e)

C $\rightarrow$ e.C, e|d

C $\rightarrow$ .eC, e|d

C $\rightarrow$ .d, e|d ....(I_3)

GOTO (I_3,d)

C $\rightarrow$ d. , e|d ....(I_4)

GOTO (I_6,C)

C $\rightarrow$ eC., \$ ....(I_9)

GOTO (I_6,e)

C $\rightarrow$ e.C, \$

C $\rightarrow$ .eC, \$

C $\rightarrow$ .d, \$ ....(I_6)

GOTO (I_6,d)

C $\rightarrow$ d., \$ ....(I_7)

Construction of CLR Parsing Table

State	ACTION			GOTO	
	e	d	\$	S	C
0	S_3	S_4		1	2
1			ACC		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

More Problems on CLR Parsers – Please refer the Class Work Note-Book.

LALR – Look-ahead LR Parser

- From the LR (1) Items of G' of G,
  - Take a pair of similar looking states; say I_j and I_k each of these states are differentiated only by the look-ahead symbols.
  - Replace I_j and I_k by I_{jk}, the union of I_j and I_k
  - **The Goto's on any symbol X to I_j or I_k from any other states now replaced with I_{jk}.**

Example: Construct the canonical LR Parsing table for the following grammar
S → CC, C → eC, C → d. And parse the string eed.

From Collection of LR (1) Items computed in CLR parser

State I_3 and I_6 are differentiated only by their look-ahead symbols,

(i) C → e.C, e|d
C → .eC, e|d
C → .d, e|d ....(I_3)
and
C → e.C, $
C → .eC, $
C → .d, $ ....(I_6)
becomes
C → e.C, $|e|d
C → .eC, $|e|d
C → .d, $|e|d ....($I_{36}$)

(ii) C → d., e|d.........(I_4)
and
C → d., $ ....(I_7)
becomes
C → d., $|e|d ....($I_{47}$)

(iii) C → eC., e|d.........(I_8)
and
C → eC., $ ....(I_9)
becomes
C → eC., $|e|d ....($I_{89}$)

Construction of LALR Parsing Table

State	ACTION			GOTO	
	e	d	$	S	C
0	S_{36}	S_{47}		1	2
1			ACC		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

LALR Parsing

No	STACK	BUFFER	ACTION
1	0	eed \$	ACTION [0,e] = S_{36} Shift
2	0e36	ed \$	ACTION [36,e] = S_{36} Shift
3	0e36e36	d \$	ACTION [36,d] = S_{47} Shift
4	0e36e36$\underline{\text{d47}}$	\$	ACTION [47,\$] = R_3 Reduce by C → d
5	0e36$\underline{\text{e36C89}}$	\$	ACTION [89,\$] = R_2 Reduce by C → eC
6	0$\underline{\text{e36C89}}$	\$	ACTION [89,\$] = R_2 Reduce by C → eC
7	0C2	\$	ACTION [2,\$] = Undefined, Error

IV. YACC – PARSER GENERATOR

A parser generator is a program that takes as input a specification of a syntax and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided in three parts.

```
/* definitions */
 ....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

- YACC automatically assigns numbers for tokens. YACC also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:
  %start nonterminal

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:
  #include "lex.yy.c"
- YACC input file generally finishes with: * .y

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.