**SYNTAX ANALYSIS (Parser)**

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.
Parsing: determining the syntax or the structure of a program. Parse tree specifies the statements execution sequence.

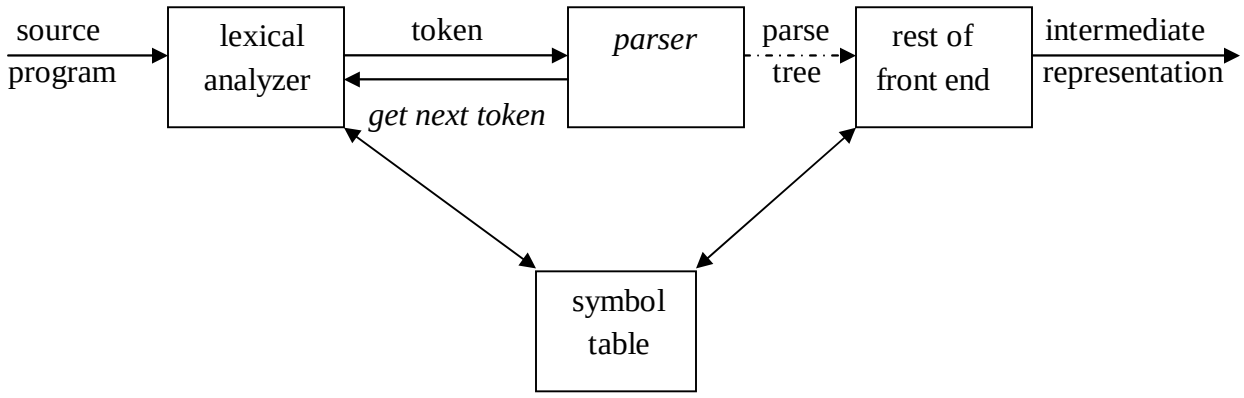
**Advantages of grammar for syntactic specification :**

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

**THE ROLE OF PARSER**

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

***Position of parser in compiler model***




**Functions of the parser :**

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

**Issues :**

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

There are three general types of parsers for grammars:

I. **universal**
II. **top-down**
III. **bottom-up**

Universal parsing methods can parse any grammar. The following two algorithms are the best examples of universal parsers.

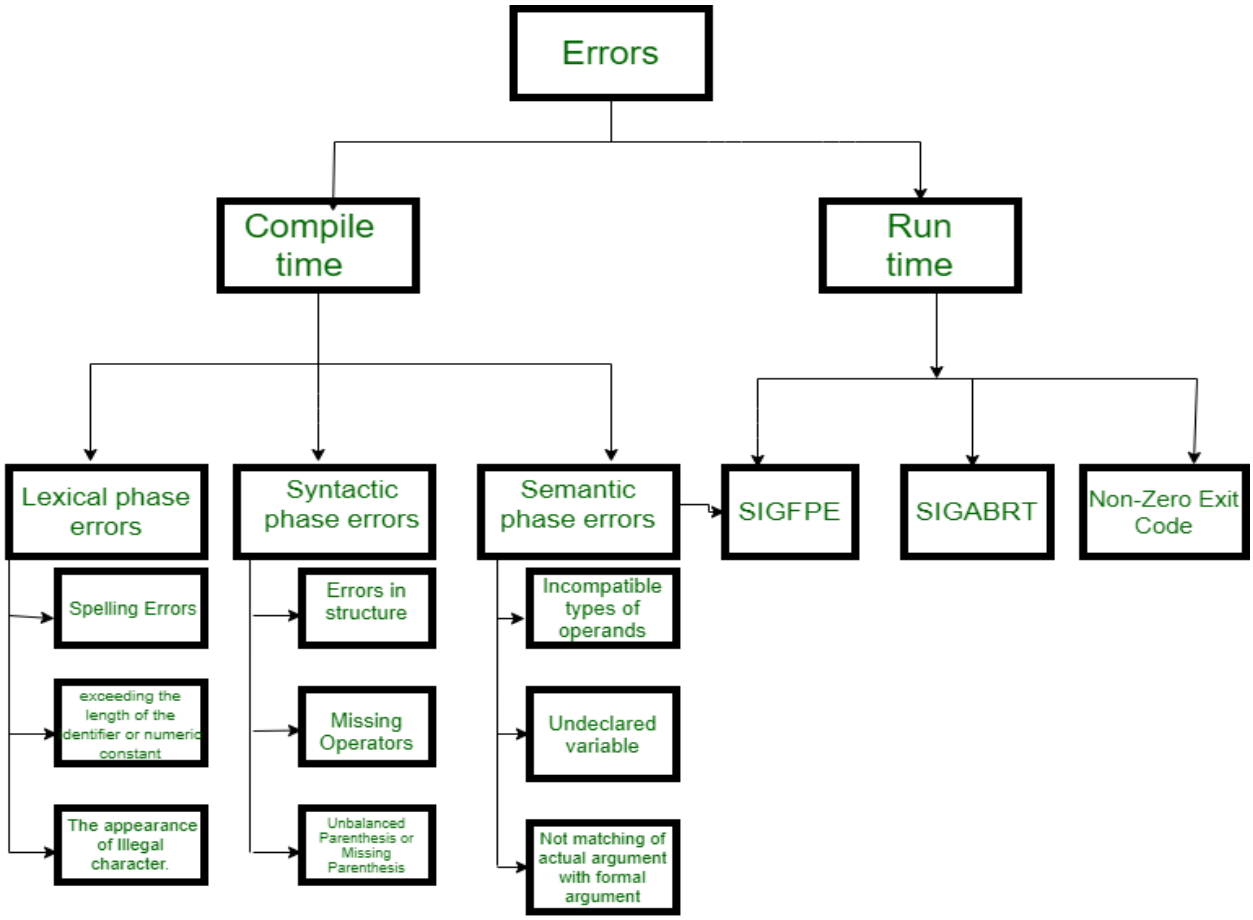
i. the Cocke-Younger-Kasami algorithm
ii. Earley's algorithm

These general methods are, however, too inefficient to use in producing compilers. Commonly used methods in compilers are either top-down or bottom-up.

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

**Syntax Error Handling and Recovery:**
Most of the programming language specifications never describe about error handling, so it is the responsibility of compiler to handle errors. Compiler must track down all the errors done by the programmer while coding the program and handle them.
Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.




Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

**Error recovery strategies :**

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions

4. Global correction

**Panic mode recovery:**

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

**Phrase level recovery:**

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

**Error productions:**

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

**Global correction:**

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

| Error / Recovery Method | Lexical Phase Error | Syntactic Phase Error | Semantic Phase Error |
|---|---|---|---|
| Panic Mode | ✓ | ✓ | x |
| Phrase-Level | x | ✓ | x |
| Error Production | x | ✓ | x |
| Global Production | x | ✓ | x |
| Using Symbol Table | x | x | ✓ |

## 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.

**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.

**Types of parsing:**

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

## TOP-DOWN PARSING

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

### 1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

**Example for backtracking :**

Consider the grammar G : $S \rightarrow cAd$

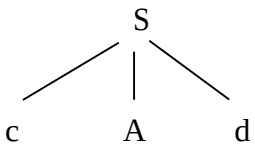
$A \rightarrow ab \mid a$

and the input string w=cad.

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

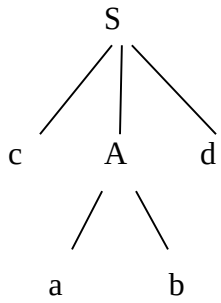
**Step1:**

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.



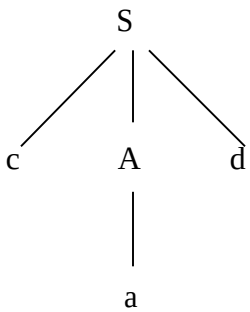

**Step3:**

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 position. This is called **backtracking.**

**Step4:**

Now try the second alternative for A.




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

**Example for recursive decent parsing:**

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

After eliminating the left-recursion the grammar becomes,

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

**Recursive procedure:**

```
Procedure EPRIME( )
begin

If input_symbol='+' then ADVANCE( );
T( );
EPRIME( );
end

Procedure E()
begin
        T( );
        EPRIME( );
end


Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME( )
begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
end

Procedure  F( )
```

**begin**

If input-symbol=’id’ then
ADVANCE( );
else if input-symbol=’(‘ then
ADVANCE( );
E( );
else if input-symbol=’)’ then
ADVANCE( );

**end**

else ERROR( );

**Stack implementation:**

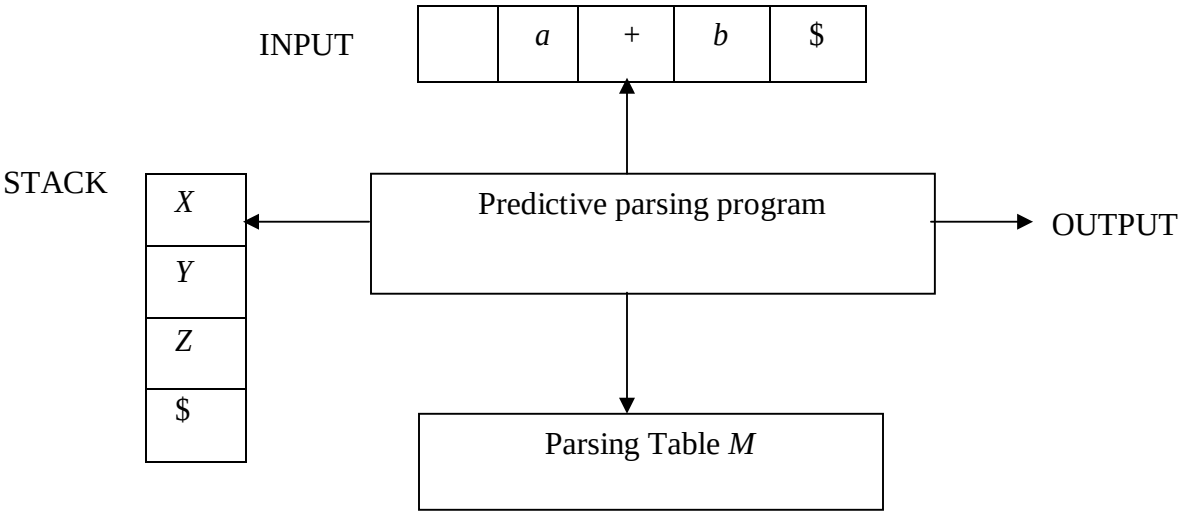
To recognize input **id+id*id :**

| PROCEDURE | INPUT STRING |
|---|---|
| E( ) | **id**+id*id |
| T( ) | **id**+id*id |
| F( ) | **id**+id*id |
| ADVANCE( ) | id**+**id*id |
| TPRIME( ) | id**+**id*id |
| EPRIME( ) | id**+**id*id |
| ADVANCE( ) | id+**id***id |
| T( ) | id+**id***id |
| F( ) | id+**id*** id |
| ADVANCE( ) | id+id***** id |
| TPRIME( ) | id+id***** id |
| ADVANCE( ) | id+id***id** |
| F( ) | id+id***id** |
| ADVANCE( ) | id+id***id** |
| TPRIME( ) | id+id***id** |

**1. PREDICTIVE PARSING**

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

**Non-recursive predictive parser**




The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

**Input buffer:**

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

**Stack:**

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

**Parsing table:**

It is a two-dimensional array $M[A, a]$, where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

**Predictive parsing program:**

The parser is controlled by a program that considers $X$, the symbol on top of stack, and $a$, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops $X$ off the stack and advances the input pointer to the next input symbol.

3. If $X$ is a non-terminal , the program consults entry $M[X, a]$ of the parsing table $M$. This entry will either be an $X$-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces $X$ on top of the stack by $WVU$.
   If $M[X, a]$ = **error**, the parser calls an error recovery routine.

**Predictive parsing table construction:**

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

**Rules for first( ):**

1. If $X$ is terminal, then FIRST($X$) is {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST($X$).

3. If $X$ is non-terminal and $X \rightarrow a\alpha$ is a production then add $a$ to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place $a$ in FIRST($X$) if for some $i$, $a$ is in FIRST($Y_i$), and ε is in all of FIRST($Y_1$),…,FIRST($Y_{i-1}$); that is, $Y_1,\ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,k, then add ε to FIRST($X$).

**Example:**

Construct the FIRST and FOLLOW for the grammar:A

-> BC | EFGH | H
B -> b
C -> c | ε
E -> e | ε
F -> CE
G -> g
H -> h | ε

Solution:

1. Finding first () set:

first (H) = first (h) U first (ε) = {h, ε }

first (G) = first (g) = {g}

first (C) = first (c) U first (ε) = {c, ε }

first (E) = first (e) U first (ε) = {e, ε }

first (F) = first (CE) = (first (c) - { ε }) U first (E)

= ({c, ε } { ε }) U {e, ε } = {c, e, ε }

first (B) = first (b)={b}

first (A) = first (BC) U first (EFGH) U first (H)

= first (B) U (first (E) – { ε }) U first (FGH) U {h, ε}

= { b } U { e } U (first (F) – { ε }) U first (GH) U {h, ε}

= { b } U { e } U {c, e} U first (G) U {h, ε}

= { b } U { e } U {c, e} U { g } U {h, ε}

= {b, c, e, g, h, ε }

## Predictive Parser or LL(1) Parser

**Algorithm for nonrecursive predictive parsing:**

**Input** : A string *w* and a parsing table *M* for grammar *G*.

**Output** : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

**Method** : Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else            /* X is a non-terminal */
        if M[X, a] = X → Y1Y2 … Yk  then begin
            pop X from the stack;
            push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
            output the production X → Y1 Y2 . . . Yk
        end
        else error()
until X = $         /* stack is empty */
```

**Input** : Grammar *G*

**Output** : Parsing table *M*

**Method** :

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add $A \rightarrow \alpha$ to *M*[*A*, *a*].
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add $A \rightarrow \alpha$ to *M*[*A*, $].
4. Make each undefined entry of *M* be **error**.

**Example:**

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

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

**First( ) :**

FIRST(E) = { ( , id}

FIRST(E') ={+ , ε }

FIRST(T) = { ( , id}

FIRST(T') = {*, ε }

FIRST(F) = { ( , id }

**Follow( ):**

FOLLOW(E) = { $, ) }

FOLLOW(E') = { $, ) }

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

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

FOLLOW(F) = {+, * , $ , ) }

**Predictive parsing table :**

| NON-TERMINAL | 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 implementation:**

| 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 |

**LL(1) grammar:**

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called 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}

**Parsing table:**

| NON-TERMINAL | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | S → a | | | S → iEtSS’ | | |
| S’ | | | S’ → eS<br>S’ → ε | | | S’ → ε |
| E | | E → b | | | | |

Since there are more than one production, the grammar is not LL(1) grammar.

**Implementation of predictive parser:**

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

**BOTTOM-UP PARSING**

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**.

**SHIFT-REDUCE PARSING**

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

**Example:**

Consider the grammar:

S → aABe

A → Abc | b

B → d

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

| **REDUCTION (LEFTMOST)** | | **RIGHTMOST DERIVATION** |
|---|---|---|
| a**b**bcde | (A → b) | **S** → aA**B**e |
| a**Abc**de | (A → Abc) | → a**A**de |
| aA**d**e | (B → d) | → a**A**bcde |
| **aABe** | (S → aABe) | → abbcde |
| 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.

**Example:**

Consider the grammar:

E → E+E

E → E*E

E → (E)

E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

E → **E**+**E**

→ E+**E*E**

→ E+E***id**$_3$

→ E+**id**$_2$***id**$_3$

→ **id**$_1$+id$_2$***id**$_3$

In the above derivation the underlined substrings are called **handles.**

**Handle pruning:**

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if $w$ is a sentence or string of the grammar at hand, then $w = \gamma_n$, where $\gamma_n$ is the $n^{th}$ right-sentinel form of some rightmost derivation.

**Stack implementation of shift-reduce parsing :**

| 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* | id3 $ | shift |
| $ E+E*id3 | $ | reduce by E→id |
| $ E+E*E | $ | reduce by E→ E *E |
| $ E+E | $ | reduce by E→ E+E |
| $ E | $ | accept |

**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 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.

**1. Shift-reduce conflict:**

**Example:**

Consider the grammar:

E→E+E | E*E | id and input id+id*id

| Stack | Input | Action | Stack | Input | Action |
|---|---|---|---|---|---|
| $ E+E | *id $ | Reduce by E→E+E | $E+E | *id $ | Shift |
| $ E | *id $ | Shift | $E+E* | id $ | Shift |
| $ E* | id $ | Shift | $E+E*id | $ | Reduce by E→id |
| $ E*id | $ | Reduce by E→id | $E+E*E | $ | Reduce by E→E*E |
| $ E*E | $ | Reduce by E→E*E | $E+E | $ | Reduce by E→E*E |
| $ E | | | $E | | |

1. Reduce-reduce conflict:

Consider the grammar:

M → R+R | R+c | R

R → c

And input c+c

| Stack | Input | Action | Stack | Input | Action |
|---|---|---|---|---|---|
| $ | c+c $ | Shift | $ | c+c $ | Shift |
| $ c | +c $ | Reduce by R→c | $ c | +c $ | Reduce by R→c |
| $ R | +c $ | Shift | $ R | +c $ | Shift |
| $ R+ | c $ | Shift | $ R+ | c $ | Shift |
| $ R+c | $ | Reduce by R→c | $ R+c | $ | Reduce by M→R+c |
| $ R+R | $ | Reduce by M→R+R | $ M | $ | |
| $ M | $ | | | | |

**Viable prefixes:**

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

**OPERATOR-PRECEDENCE PARSING**

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These

grammars have the property that no production on right side is ε or has two adjacent non-terminals.

**Example:**

Consider the grammar:

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

E → E+E | E-E | E*E | E/E | E↑E | -E | id

**Operator precedence relations:**

There are three disjoint precedence relations namely

- $<\cdot$ - less than
- $=$ - equal to
- $\cdot>$ - greater than

The relations give the following meaning:

- $a <\cdot b$ – a yields precedence to b
- $a = b$ – a has the same precedence as b
- $a \cdot> b$ – a takes precedence over b

**Rules for binary operations:**

1. If operator $\theta_1$ has higher precedence than operator $\theta_2$, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators $\theta_1$ and $\theta_2$, are of equal precedence, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if operators are left associative
   $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:
   $\theta <\cdot id$ , $id \cdot> \theta$
   $\theta <\cdot ($ , $( <\cdot \theta$
   $) \cdot> \theta$ , $\theta \cdot> )$
   $\theta \cdot> \$$ , $\$ <\cdot \theta$

   Also make
   $( = )$ , $( <\cdot ( , ) \cdot> )$ , $( <\cdot id , id \cdot> )$ , $\$ <\cdot id , id \cdot> \$$ , $\$ <\cdot ( , ) \cdot> \$$

**Example:**

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

| | + | - | * | / | ↑ | id | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|---|
| + | ·> | ·> | <· | <· | <· | <· | <· | ·> | ·> |
| - | ·> | ·> | <· | <· | <· | <· | <· | ·> | ·> |
| * | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| / | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| ↑ | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| id | ·> | ·> | ·> | ·> | ·> | | | ·> | ·> |
| ( | <· | <· | <· | <· | <· | <· | <· | = | |
| ) | ·> | ·> | ·> | ·> | ·> | | | ·> | ·> |
| $ | <· | <· | <· | <· | <· | <· | <· | | |

**Operator precedence parsing algorithm:**

**Input :** An input string *w* and a table of precedence relations.

**Output :** If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.

**Method :** Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute the following program :

(1) Set *ip* to point to the first symbol of *w*$;
**(2) repeat forever**
**(3) if** $ is on top of the stack and *ip* points to $ **then**
**(4) return**
**else begin**
*(5)* let *a* be the topmost terminal symbol on the stack
and let *b* be the symbol pointed to by *ip;*
**(6) if** *a* <· *b* or *a* = *b* **then begin**
(7) push *b* onto the stack;
(8) advance *ip* to the next input symbol;
**end;**
**(9) else if** *a* · > *b* **then** **/*** reduce ***/**
**(10) repeat**
(11) pop the stack
(12) **until** the top stack terminal is related by <·
to the terminal most recently popped
(13) **else** error( )
**end**

**Stack implementation of operator precedence parsing:**

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

| STACK | INPUT |
|---|---|
| $ | w $ |

where w is the input string to be parsed.

**Example:**

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is **id+id*id** .The implementation is as follows:

| STACK | INPUT | COMMENT |
|---|---|---|
| $ | <· id+id*id $ | shift id |
| $ id | ·> +id*id $ | pop the top of the stack id |
| $ | <· +id*id $ | shift + |
| $ + | <· id*id $ | shift id |
| $ +id | ·> *id $ | pop id |
| $ + | <· *id $ | shift * |
| $ + * | <· id $ | shift id |
| $ + * id | ·> $ | pop id |
| $ + * | ·> $ | pop * |
| $ + | ·> $ | pop + |
| $ | $ | accept |

**Advantages of operator precedence parsing:**

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

**Disadvantages of operator precedence parsing:**

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

**LR PARSERS**

An efficient bottom-up syntax analysis technique that can be used to parse a large class of 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. When '*k*' is omitted, it is assumed to be 1.

**Advantages of LR parsing:**

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

**Drawbacks of LR method:**

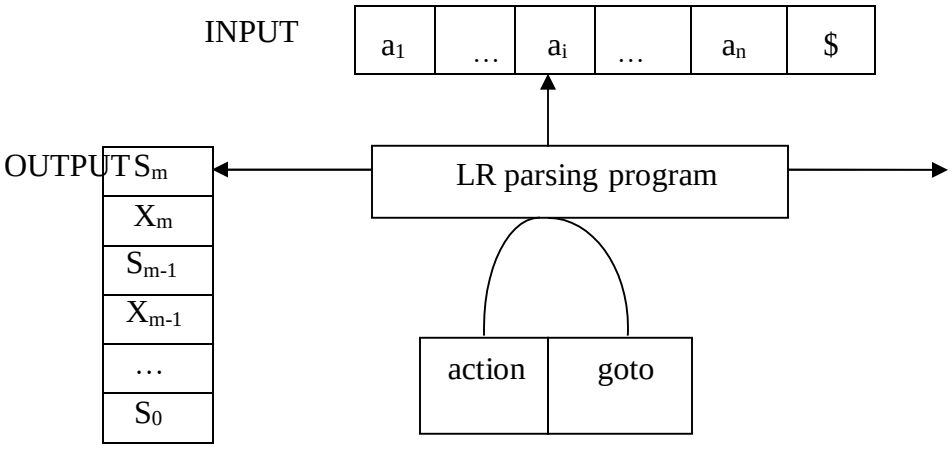
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

**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.

**The LR parsing algorithm:**

The schematic form of an LR parser is as follows:




STACK

It consists of : an input, an output, a stack, a driver program, and a parsing table that has two 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 $s_0X_1s_1X_2s_2\ldots X_ms_m$, where $s_m$ is on top. Each $X_i$ is a grammar symbol and each $s_i$ is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

**Action** : The parsing program determines $s_m$, the state currently on top of stack, and $a_i$, the current input symbol. It then consults *action*$[s_m,a_i]$ in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

**Goto** : The function goto takes a state and grammar symbol as arguments and produces a state.

**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 $s_0$ on its stack, where $s_0$ 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
```

**CONSTRUCTING SLR(1) 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 **.**

**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:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more 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:

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

**<u>Algorithm for construction of SLR parsing table:</u>**

**Input** : An augmented grammar G'
**Output** : The SLR parsing table functions *action* and *goto* for G'
**Method** :

1. Construct C = {$I_0, I_1, \ldots I_n$}, 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·*a*β] 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-terminals A using the rule: If *goto*($I_i$,A) = $I_j$, 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 set of items containing [S'→.S].

**Example for SLR parsing:**

SLR stands for "Simple LR" parser. It is a type of bottom-up parser used in compiler design to parse a given programming language's syntax based on a formal grammar. The SLR parser is simple compared to other LR parsers like LR(1), LALR(1), etc., which makes it easier to implement and understand.

Working of SLR
Parsing Table Construction:
The core of the SLR parser is the parsing table, which is constructed using the given grammar. This parsing table is a finite automaton that guides the parser in making decisions about which actions to take based on the current state and the next input token.

States:
The parser moves through a set of states, with each state representing a set of items. An item is a production rule of the grammar with a dot (·) indicating how much of the rule has been parsed. For example, if we have a production A → XYZ, then the item could be A → X·YZ, indicating that X has been parsed.

Closure Operation:
To construct the parsing table, the closure operation is applied to sets of items. Closure operation expands the set of items by including items for all non-terminal symbols that can be derived from the current set of items.

Goto Operation:
After applying closure, the parser computes the goto function. Goto function determines the state transition given a state and a grammar symbol.

Action and Goto Tables:
The parsing table is divided into two parts: the action table and the goto table.

Action Table: Contains actions to be taken based on the current state and the next input token. Actions can be shift, reduce, or accept.
Goto Table: Contains state transitions for non-terminal symbols.
Parsing:
The parsing process starts with an initial state and iterates through the input tokens. At each step, the parser consults the parsing table to determine whether to shift the current token onto the stack, reduce a portion of the stack to a non-terminal symbol, or accept the input.

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:
Find LR(0) items.
Completing the closure.
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 .

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 ? a . B is in closure(I) and B ? ? is a production, then add the item B ? . ? 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? aX . ] such that [A? a . X ] 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.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

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

Method :

Construct C = {I0, I1, .... In}, the collection of sets of LR(0) items for G'.

State i is constructed from Ii.. The parsing functions for state i are determined as follows:

If [A-->aa] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.

If [A-->a] is in Ii , then set action[i,a] to "reduce A-->a" for all a in FOLLOW(A).

If [S'-->S.] is in Ii, then set action[i,$] to "accept".

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

The goto transitions for state i are constructed for all non-terminals A using the rule: If goto(Ii,A) = Ij, then goto[i,A] = j.

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

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

Construct SLR parsing for the following grammar :

G : E → E + T | T

T → T * F | F

F → (E) | id

The given grammar is :

G : E → E + T ------ (1)

E →T ------ (2)

T → T * F ------ (3)

T → F ------ (4)

F → (E) ------ (5)

F → id ------ (6)

**Step 1 :** Convert given grammar into augmented grammar.

**Augmented grammar :**

E' → E

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

**Step 2 :** Find LR (0) items.

$I_0$ : 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 ( $I_4$ , id )

$I_5$ : F → id **.**

GOTO ( $I_6$ , T )

GOTO ( $I_0$ , T)

$I_2$ : E → T .

T → T . * F

GOTO ( $I_0$ , F)

$I_3$ : T → F .

GOTO ( $I_0$ , ( )

$I_4$ : F → ( . E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO ( $I_0$ , id )

$I_5$ : F → id .

GOTO ( $I_1$ , + )

$I_6$ : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO ( $I_2$ , * )

$I_7$ : T → T * . F

F → . (E)

F → . id

GOTO ( $I_4$ , E )

$I_8$ : F → ( E . )

E → E . + T

GOTO ( $I_4$ , T)

$I_2$ : E → T .

T → T . * F

GOTO ( $I_4$ , F)

$I_3$ : T → F .

$I_9$ : E → E + T .

T → T . * F

GOTO ( $I_6$ , F )

$I_3$ : T → F .

GOTO ( $I_6$ , ( )

$I_4$ : F → ( . E )

GOTO ( $I_6$ , id)

$I_5$ : F → id .

GOTO ( $I_7$ , F )

$I_{10}$ : T → T * F .

GOTO ( $I_7$ , ( )

$I_4$ : F → ( . E )

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO ( $I_7$ , id )

$I_5$ : F → id .

GOTO ( $I_8$ , ) )

$I_{11}$ : F → ( E ) .

GOTO ( $I_8$ , + )

$I_6$ : E → E + . T

T → . T * F

T → . F

F → . ( E )

F → . id

GOTO ( $I_9$ , *)

$I_7$ : T → T * . F

F → . ( E )

F → . id

GOTO ( $I_4$ , ( )

$I_4$ : F → ( . E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → id

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

**SLR parsing table:**

| | **ACTION** | | | | | | **GOTO** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** | **E** | **T** | **F** |
| **$I_0$** | s5 | | | s4 | | | 1 | 2 | 3 |
| **$I_1$** | | s6 | | | | ACC | | | |
| **$I_2$** | | r2 | s7 | | r2 | r2 | | | |
| **$I_3$** | | r4 | r4 | | r4 | r4 | | | |
| **$I_4$** | s5 | | | s4 | | | 8 | 2 | 3 |
| **$I_5$** | | r6 | r6 | | r6 | r6 | | | |
| **$I_6$** | s5 | | | s4 | | | | 9 | 3 |
| **$I_7$** | s5 | | | s4 | | | | | 10 |
| **$I_8$** | | s6 | | | s11 | | | | |
| **$I_9$** | | r1 | s7 | | r1 | r1 | | | |
| **$I_{10}$** | | r3 | r3 | | r3 | r3 | | | |
| **$I_{11}$** | | r5 | r5 | | r5 | r5 | | | |

Blank entries are error entries.

**Stack implementation:**

Check whether the input **id + id * id** is valid or not.

| STACK | INPUT | ACTION |
|---|---|---|
| 0 | id + id * id $ | GOTO ( $I_0$ , id ) = s5 ; **shift** |
| 0 id 5 | + id * id $ | GOTO ( $I_5$ , + ) = r6 ; **reduce** by F→id |
| 0 F 3 | + id * id $ | GOTO ( $I_0$ , F ) = 3<br>GOTO ( $I_3$ , + ) = r4 ; **reduce** by T → F |
| 0 T 2 | + id * id $ | GOTO ( $I_0$ , T ) = 2<br>GOTO ( $I_2$ , + ) = r2 ; **reduce** by E → T |
| 0 E 1 | + id * id $ | GOTO ( $I_0$ , E ) = 1<br>GOTO ( $I_1$ , + ) = s6 ; **shift** |
| 0 E 1 + 6 | id * id $ | GOTO ( $I_6$ , id ) = s5 ; **shift** |
| 0 E 1 + 6 id 5 | * id $ | GOTO ( $I_5$ , * ) = r6 ; **reduce** by F → id |
| 0 E 1 + 6 F 3 | * id $ | GOTO ( $I_6$ , F ) = 3<br>GOTO ( $I_3$ , * ) = r4 ; **reduce** by T → F |
| 0 E 1 + 6 T 9 | * id $ | GOTO ( $I_6$ , T ) = 9<br>GOTO ( $I_9$ , * ) = s7 ; **shift** |
| 0 E 1 + 6 T 9 * 7 | id $ | GOTO ( $I_7$ , id ) = s5 ; **shift** |
| 0 E 1 + 6 T 9 * 7 id 5 | $ | GOTO ( $I_5$ , $ ) = r6 ; **reduce** by F → id |
| 0 E 1 + 6 T 9 * 7 F 10 | $ | GOTO ( $I_7$ , F ) = 10<br>GOTO ( $I_{10}$ , $ ) = r3 ; **reduce** by T → T * F |
| 0 E 1 + 6 T 9 | $ | GOTO ( $I_6$ , T ) = 9<br>GOTO ( $I_9$ , $ ) = r1 ; **reduce** by E → E + T |
| 0 E 1 | $ | GOTO ( $I_0$ , E ) = 1<br>GOTO ( $I_1$ , $ ) = **accept** |

**Conflicts in LR parsres**

There are two types of conflicts in any LR parser

1. **SR conflict** : combination of Complete and incomplete productions

   A→ B.aC

   A→ B.

2. **RR conflict:** contains more than one complete productions

   A→ B.

   C→ B.

If a Grammar G contains any conflict in LR parsing table then the Grammar is not LR grammar.

## CLR Parser

CLR parsers, also known as "Canonical LR" parsers, are a more powerful variation of LR parsers compared to SLR (Simple LR) parsers. They can handle a broader class of grammars while retaining the bottom-up parsing strategy. Let's explore CLR parsers in detail:

1. LR(1) Items:

Like SLR parsers, CLR parsers use LR items to represent possible configurations of the parsing process.

However, CLR parsers use LR(1) items, which include both the LR(0) item (production rule with a dot indicating the current position) and a lookahead symbol representing the next input token.

2. Canonical Collection of LR(1) Items:

The core of a CLR parser is the construction of the canonical collection of LR(1) items.

The canonical collection is a collection of sets of LR(1) items, where each set represents a state of the parser.

The construction process involves applying closure and goto operations similarly to SLR parsers but using LR(1) items.

3. Parsing Table Construction:

Once the canonical collection of LR(1) items is constructed, the parser builds the parsing table.

The parsing table includes both action and goto entries.

Action entries specify whether to shift, reduce, or accept based on the current state and the next input token.

Goto entries determine the next state to transition to when encountering a non-terminal symbol.

Example:
S ->CC
C ->cC/d.
1.Number the grammar productions:

S ->CC
C ->cC
C ->d

2. The Augmented grammar is:

$S^I$ ->S
S ->CC
C ->cC
C ->d.

Constructing the sets of LR(1) items:
We begin with:
SI ->.S,$ begin with look-a-head (LAH) as $.

We match the item [SI ->.S,$] with the term [A ->α.Bβ, a] In the procedure closure,i.e.,

**A = S^I**

α = ∈

**B =S**

β = ∈

**a = $**

Function closure tells us to add [B->.r,b] for each production B->r and terminal b in FIRST (βa). Now β->r must be S->CC, and since β is ∈ and a is $, b may only be $. Thus,

**S->.CC,$**

We continue to compute the closure by adding all items [C->.r,b] for b in FIRST [C$] i.e., matching [S->.CC,$] against [A ->α.Bβ, a] we have, A=S, α = ∈, B=C and a=$.

FIRST (C$) = FIRST (C)
FIRST(C) = {c,d}

We add items:
C->.cC, c
C->cC, d
C->.d, c
C->.d,d

None of the new items have a non-terminal immediately to the right of the dot,so we have completed our first set of LR(1) items. The initial $I_0$ items are:

$I_0$:
$S^I$->.S, $
S->.CC, $
C->.cC, c/d
C->.d, c/d

Now we start computing goto ($I_0$,X) for various non-terminals i.e.,Goto ($I_0$,S):

$I_1$ : $S^I$->S.,$ -> reduced item.

Goto ($I_0$,C): $I_2$ :

S->C.C, $

C->.cC,$

C->.d,$

Goto ($I_0$,c) : $I_3$ :
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto ($I_0$,d) : $I_4$ :
C->d., c/d -> reduced item.

Goto ($I_2$,C) : $I_5$ :
S->CC., $ -> reduced item.

Goto (I2,C) : $I_6$
C->c.C, $
C->.cC, $
C->.d, $

Goto (I2,d) : $I_7$
C->d., $ -> reduced item.

Goto (I3,C) : $I_8$
C->cC., c/d -> reduced item.

Goto (I3,C) : $I_3$
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto (I3,d) : $I_4$
C->d., c/d. -> reduced item.

Goto (I6,C) : $I_9$
C->cC., $ -> reduced item.

Goto (I6,C) : $I_6$
C->c.C, $
C->,cC, $
C->.d, $

Goto (I6,d) : $I_7$
C->d., $ -> reduced item.

All are completely reduced. So now we construct the canonical LR(1) parsingtable – Here there is no need to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

## Constructing LR(1) Parsing table:

| | Action | | | goto | |
|---|---|---|---|---|---|
| State | C | D | $ | S | C |
| $I_0$ | S3 | S4 | | 1 | 2 |
| 1 | | | Accept | | |
| 2 | S6 | S7 | | | 5 |
| 3 | S3 | S4 | | | 8 |
| 4 | R3 | R3 | | | |
| 5 | | | R1 | | |
| 6 | S6 | S7 | | | 9 |
| 7 | | | R3 | | |
| 8 | R2 | R2 | | | |
| 9 | | | R2 | | |

# LALR Parser

**Merging Similar States:**

**CLR Parser:**

Does not merge similar states during construction, resulting in a larger number of distinct states in the canonical collection.

This approach maintains full parsing power but can lead to larger parsing tables and increased memory usage.

**LALR Parser:**

Merges similar states to achieve a more compact representation of LR(1) item sets.

By merging states with identical LR(0) items, LALR parsers reduce the size of the parsing tables while preserving parsing power to a large extent.

Example

S ->CC

C ->cC/d.

1.Number the grammar productions:

S ->CC

C ->cC

C ->d

2. The Augmented grammar is:

$S^I$ ->S

S ->CC

C ->cC

C ->d.

3.Constructing the sets of LR(1) items:

We begin with:

SI ->.S,$ begin with look-a-head (LAH) as $.

We match the item [SI ->.S,$] with the term [A ->α.Bβ, a] In the procedure closure,i.e.,

**$A = S^I$**

α = ∈

**B =S**

β = ∈

**a = $**

Function closure tells us to add [B->.r,b] for each production B->r and terminal b in FIRST (βa).

Now β->r must be S->CC, and since β is ∈ and a is $, b may only be $. Thus,

**S->.CC,$**

We continue to compute the closure by adding all items [C->.r,b] for b in FIRST [C$] i.e.,

matching [S->.CC,$] against [A ->α.Bβ, a] we have, A=S, α = ∈, B=C and a=$.

FIRST (C$) = FIRST (C)

FIRST(C) = {c,d}

We add items:

C->.cC, c

C->cC, d
C->.d, c
C->.d,d

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial $I_0$ items are:

$I_0$:
$S^I$->.S, $
S->.CC, $
C->.cC, c/d
C->.d, c/d

Now we start computing goto ($I_0$,X) for various non-terminals i.e.,

Goto(I0,S):

$I_1$ : $S^I$->S.,$ -> reduced item.

Goto ($I_0$,C): $I_2$ :

S->C.C, $

C->.cC,$

C->.d,$

Goto ($I_0$,c) : $I_3$ :
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto ($I_0$,d) : $I_4$ :
C->d., c/d -> reduced item.

Goto ($I_2$,C) : $I_5$ :
S->CC., $ -> reduced item.

Goto (I2,C) : $I_6$
C->c.C, $
C->.cC, $
C->.d, $

Goto (I2,d) : $I_7$
C->d., $ -> reduced item.

Goto (I3,C) : $I_8$
C->cC., c/d -> reduced item.

Goto (I3,C) : $I_3$
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto (I3,d) : $I_4$
C->d., c/d. -> reduced item.

Goto (I6,C) : $I_9$
C->cC., $ -> reduced item.

Goto (I6,C) : $I_6$
C->c.C, $
C->,cC, $
C->.d, $

Goto (I6,d) : $I_7$
C->d., $ -> reduced item.

4. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (plus them into a single term)

$I_0$ ->same as previous $I_1$
-> same as previous $I_2$
-> same as previous
$I_{36}$ – Clubbing item I3 and I6 into one I36 item.
C ->cC, c/d/$
C->cC, c/d/$
C->d, c/d/$
$I_5$ ->some as previous
$I_{47}$ - Clubbing item I4 and I7 into one I47 item C->d, c/d/$
$I_{89}$ - Clubbing item I8 and I9 into one I89 item C->cC, c/d/$

## LALR Parsing table construction:

| State | Action | | | Goto | |
|---|---|---|---|---|---|
| | c | d | $ | S | C |
| 0 | $S_{36}$ | $S_{47}$ | | 1 | 2 |
| 1 | | | Accept | | |
| 2 | S36 | S47 | | | 5 |
| 36 | S36 | S47 | | | 89 |
| 47 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 89 | r2 | r2 | r2 | | |

comparing SLR (Simple LR), CLR (Canonical LR), and LALR (Look-Ahead LR) parsers

**1. Parsing Power:**

**SLR Parser:** SLR parsers are based on LR(0) items and have the least parsing power among the three types. They can handle a limited subset of grammars and may fail to parse certain ambiguous grammars efficiently.

**CLR Parser:** CLR parsers use LR(1) items and have higher parsing power compared to SLR parsers. They can handle a broader class of grammars and resolve more ambiguities by considering lookahead symbols.

**LALR Parser:** LALR parsers also use LR(1) items but achieve parsing power comparable to CLR parsers through a more compact representation of the parsing tables. LALR parsers can handle a wide range of grammars efficiently, making them popular choices in practice.

**2. Construction of Parsing Tables:**

**SLR Parser:** SLR parsers construct parsing tables based on LR(0) items, resulting in simpler and smaller parsing tables compared to CLR and LALR parsers. However, this simplicity comes with limitations in parsing power.

**CLR Parser:** CLR parsers construct parsing tables based on LR(1) items, which can be more complex and larger than SLR parsing tables due to the consideration of lookahead symbols. This increased complexity enables CLR parsers to handle a wider range of grammars.

**LALR Parser:** LALR parsers construct parsing tables based on a more compact representation of LR(1) items, known as Look-Ahead LR(1) items. This allows LALR parsers to achieve parsing power similar to CLR parsers while maintaining smaller parsing tables compared to CLR parsers.

**3. Implementation Complexity:**

**SLR Parser:** SLR parsers are simpler to implement compared to CLR and LALR parsers due to their use of LR(0) items. They are suitable for educational purposes and straightforward compiler implementations.

**CLR Parser**: CLR parsers are more complex to implement than SLR parsers due to the consideration of lookahead symbols and the construction of canonical LR(1) item sets. Implementing a CLR parser requires a deeper understanding of parsing theory.

**LALR Parser:** LALR parsers strike a balance between parsing power and implementation complexity. They are more complex than SLR parsers but simpler than CLR parsers due to their more compact representation of LR(1) items.

**4. Parsing Performance:**
**SLR Parser**: SLR parsers are efficient in terms of parsing performance for grammars within their parsing power. However, they may exhibit inefficiencies or fail to parse certain grammars efficiently due to their limited parsing power.

**CLR Parser:** CLR parsers may have higher parsing overhead compared to SLR parsers due to the larger parsing tables and the consideration of lookahead symbols. However, they offer greater parsing power and can handle a wider range of grammars.

**LALR Parser:** LALR parsers offer parsing performance comparable to CLR parsers while maintaining smaller parsing tables. They are efficient in terms of both parsing performance and memory usage, making them popular choices for real-world compiler implementations.

**Conclusion:**
SLR Parser: Simple and easy to implement but limited in parsing power.
CLR Parser: More powerful but complex to implement with potentially larger parsing tables.
LALR Parser: Balances parsing power and implementation complexity with efficient parsing performance and memory usage, making it a popular choice for practical compiler implementations.

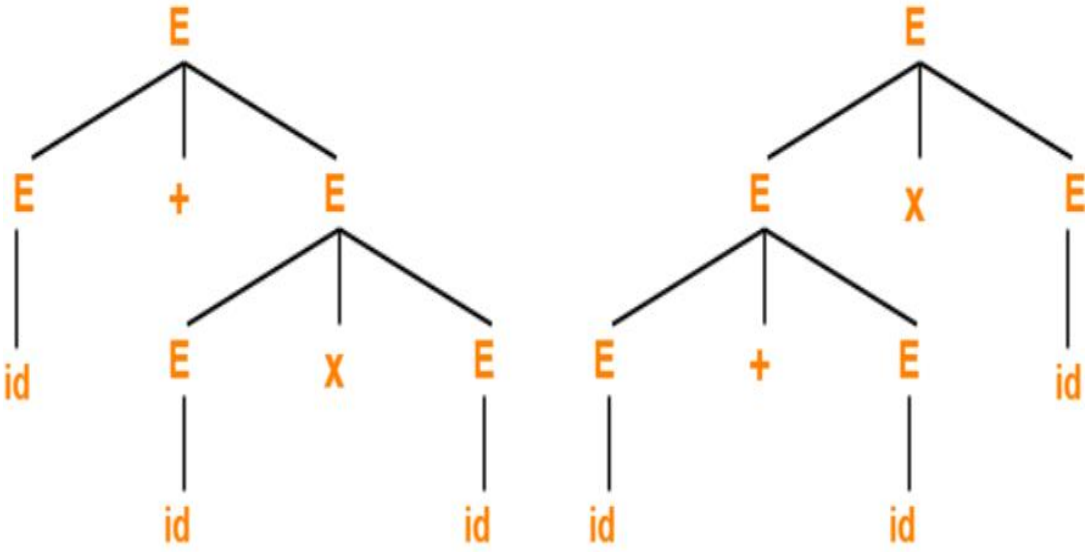
**Ambiguity:**

A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**. **(or)** more than one LMD or more than one RMD for some string is also called ambiguous grammar.

Example : Given grammar G : E → E+E | E*E | ( E ) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

| | |
|---|---|
| E → E+ E | E → E* E |
| E → id + E | E → E + E * E |
| E → id + E * E | E → id + E * E |
| E → id + id * E | E → id + id * E |
| E → id + id * id | E → id + id * id |




Parse Tree-01 Parse Tree-02

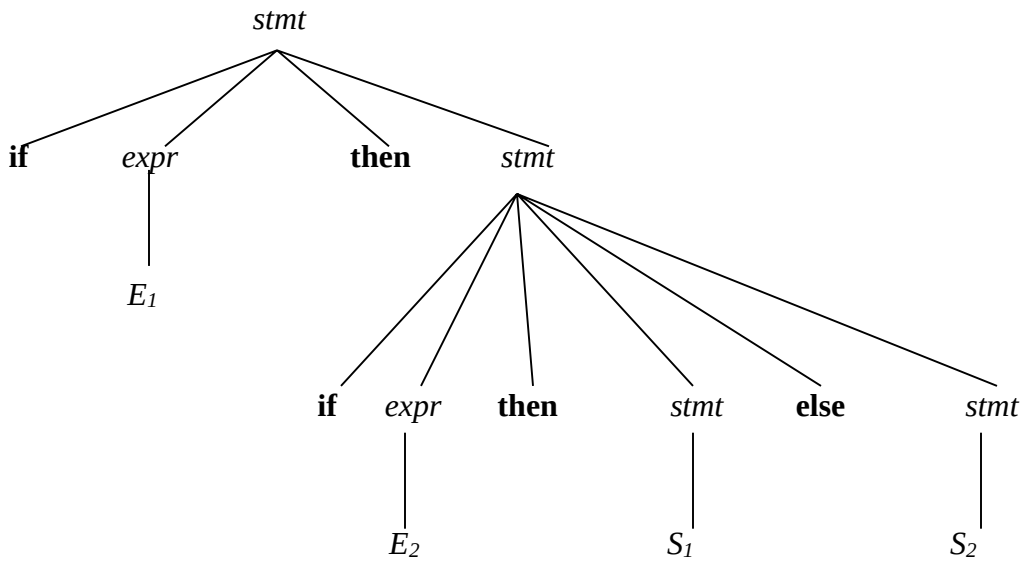
**Eliminating ambiguity:**

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

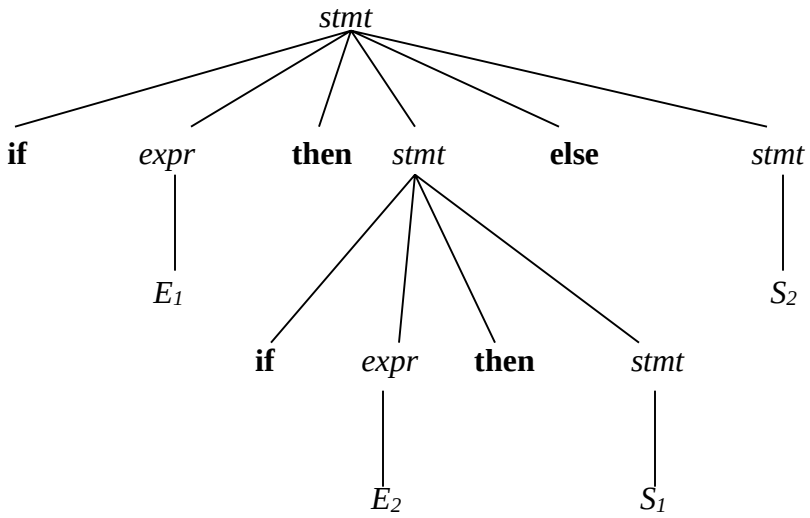
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if $E_1$ then if $E_2$ then $S_1$ else $S_2$** has the following two parse trees for leftmost derivation :

*1.*




*2.*




To eliminate ambiguity, the following grammar may be used:

*stmt* → *matched_stmt* | *unmatched_stmt*

*matched_stmt* → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

*unmatched_stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

**Eliminating Left Recursion:**

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

**If there is a production A → Aα | β it can be replaced with a sequence of two productions**

**A → βA'**

**A' → αA' | ε**

without changing the set of strings derivable from A.

**Example** : Consider the following grammar for arithmetic expressions:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

First eliminate the left recursion for E as

$E \rightarrow TE'$

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

Then eliminate for T as

$T \rightarrow FT'$

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

Thus the obtained grammar after eliminating left recursion is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

**Algorithm to eliminate left recursion:**

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. **for** *i* := 1 **to** *n* **do begin**
    **for** *j* := 1 **to** *i*-1 **do begin**
        replace each production of the form $A_i \rightarrow A_j \gamma$
            by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current $A_j$-productions;
    **end**
    eliminate the immediate left recursion among the $A_i$-productions
**end**

**Left factoring:**

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

**If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ , it can be rewritten as**

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar , G : S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS' | a
S' → eS | ε
E → b

Using Ambiguous Grammars:

If the grammar is ambiguous, it creates conflicts, we cannot parse the input string. Ambiguous grammar generates the below to kinds of conflicts while parsing:

Shift-Reduce Conflict

Reduce-Reduce Conflict

But for arithmetic expressions, ambiguous grammars are more compact, provides more natural specification.

Ambiguous grammar can add any new production for special constructs. So, ambiguous grammars must be handled carefully to obtain only one parse tree for the specific input (i.e., precedence and associativity for arithmetic expressions)

Ambiguous grammars can pose challenges when used with LR parsers, which are a type of bottom-up parsing technique. However, ambiguous grammars can still be used in LR parsers with some considerations and techniques.

Here's how ambiguous grammars are handled in LR parsers:

1. Shift-Reduce Conflicts:

Ambiguous grammars can lead to shift-reduce conflicts in LR parsers. These conflicts occur when the parser faces a choice between shifting the next input token onto the stack or reducing the current stack symbols to a non-terminal.

LR parsers typically resolve shift-reduce conflicts by favoring the shift operation, meaning that the parser will choose to shift the input token onto the stack rather than reducing the symbols.

However, this resolution strategy may not always produce the desired parse tree for ambiguous inputs, leading to potentially incorrect parsing results.

2. Reduce-Reduce Conflicts:

Ambiguous grammars can also lead to reduce-reduce conflicts in LR parsers. These conflicts occur when the parser faces a choice between two or more reduction operations for the current stack symbols.

LR parsers typically resolve reduce-reduce conflicts by applying specific rules or heuristics defined in the parsing algorithm or by prioritizing reductions based on the order of rules in the grammar.

Similar to shift-reduce conflicts, the resolution of reduce-reduce conflicts may not always produce the desired parse tree for ambiguous inputs, potentially leading to incorrect parsing

results.

3. Use of Ambiguity-Resolving Techniques:

In some cases, ambiguous grammars can be used with LR parsers by applying ambiguity-resolving techniques during parsing.

Techniques such as operator precedence parsing, associativity rules, and explicit disambiguation rules can help resolve ambiguity and guide the parser in making informed decisions during parsing.

By incorporating these techniques into the parsing process, LR parsers can handle certain types of ambiguity more effectively and produce more accurate parsing results.

4. Preprocessing or Grammar Modification:

Ambiguous grammars can sometimes be preprocessed or modified to make them suitable for parsing with LR parsers.

Techniques such as left factoring, left recursion elimination, and grammar refactoring can help transform an ambiguous grammar into an unambiguous or less ambiguous form that is compatible with LR parsing techniques.

By preprocessing or modifying the grammar, the parser can avoid or minimize conflicts and ambiguity during parsing.

While LR parsers are typically designed to handle unambiguous grammars, they can still be used with ambiguous grammars with careful consideration and the application of appropriate techniques. By understanding the nature of ambiguity and employing ambiguity-resolving techniques or preprocessing methods, LR parsers can parse certain types of ambiguous grammars effectively and produce accurate parsing results. However, it's important to note that not all ambiguous grammars can be effectively handled by LR parsers, and in some cases, other parsing techniques may be more suitable.

## Parser Generator YAAC:

To automate the process of parsing an input string by parser, certain automation tools for parser generation are available.

YACC (Yet Another Compiler Compiler) is one such automatic tool for parser generation. Itis an UNIX based utility tool for LALR parser generator. LEX and YACC work together to analyze the program syntactically, can report conflicts or ambiguities in the form of error messages.

Originally developed by Stephen C. Johnson at Bell Labs in the 1970s, Yacc has become one of the most widely used tools for building parsers.

**How Yacc Works:**

**Grammar Specification:**

Yacc takes as input a grammar specification file written in a formalism similar to BNF (Backus-Naur Form). This grammar defines the syntax rules of the language to be parsed.

**Parsing Table Generation:**

Yacc analyzes the grammar specification and generates the parsing tables needed for a parser to recognize and parse input according to the specified grammar.

These parsing tables typically consist of action and goto entries, which guide the parsing process by specifying whether to shift, reduce, or accept input tokens and how to transition between parser states.

**Code Generation:**

Yacc generates source code for a parser in a target programming language, such as C, C++, or Java.

The generated parser code includes functions for initializing the parsing tables, executing parsing actions, and managing the parsing stack.

**Integration with Lexical Analyzer:**

Yacc parsers are typically used in conjunction with lexical analyzers generated by tools like Lex (or Flex). Lexical analyzers tokenize input streams into individual tokens, which are then fed into the Yacc parser for syntactic analysis.

**Error Handling:**

Yacc-generated parsers often include mechanisms for error handling, such as syntax error detection and recovery routines.

Error messages and recovery strategies can be customized by the user to provide meaningful feedback to users of the compiler or interpreter.

**YACC Specification**

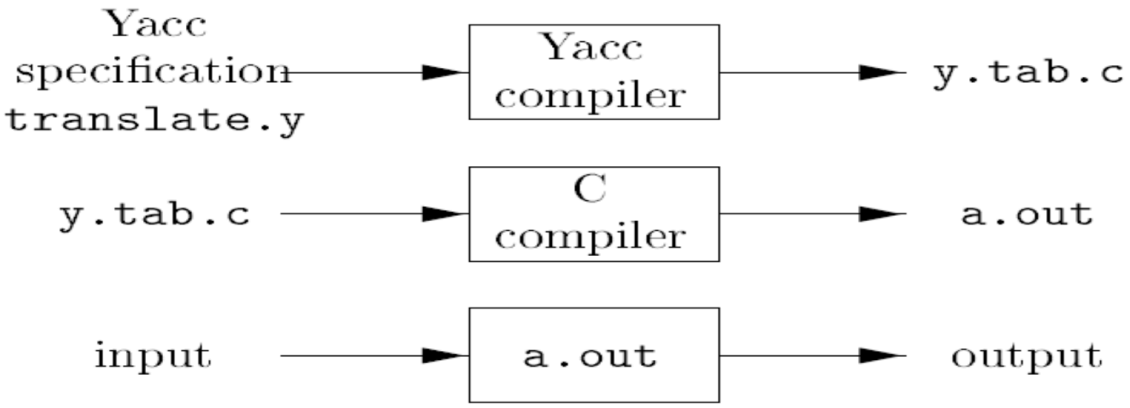
```
%{ declarations %}        required header files are declared
 % token                    required tokens are declared

%%                         translation rules consist of a grammar production
translation rules          and the associated semantic action.
%%

supporting C routines      //main() method, yylex() function is called from main.
```

A translator is constructed using YACC specification. After translator is prepared, it is tobe saved with **.y** extension.

**Compilation and execution steps:**

- The UNIX system command to compile this specification is **yacc filename.y**
- it transforms the specification into **y.tab.c** and **y.tab.h** using LALR algorithm.
- Compile y.tab.c with ly library using CC compiler

  **cc y.tab.c -ly**
- Compilation produces **a.out** file. execute it the input program to get the desired output.
- To execute use **./a.out** command




A set of productions of the below form

<head> -> <body>1 | <body>2 | . . . | <body>nwould

be written in YACC as

```
<head> : <body>1 { <semantic action>1 }
       | <body>2 { <semantic action>2 }
        . . .
       | <body>n { <semantic action>n }
       ;
```

In a YACC production, unquoted strings of letters and digits not declared as tokens are considered as non-terminals.

A quoted single character, e.g., 'c', is considered as terminal symbol c, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for 'c' to the parser, as an integer).

Second part of YACC specification contains semantic action which are a sequence of C statements.

In this, the symbol $$ refers to the attribute value associated with the nonterminal of the head, while $i refers to the value associated with the $i^{th}$ grammar symbol (terminal or nonterminal) of the body.

In the YACC specification, for the written E -productionsE -
> E + T | T

and their associated semantic actions as:

```
expr : expr '+' term { $$ = $1 + $3; } // Note:  + is $2
     | term          { $$ = $1; }      // Note: we can omit this
     ;
```

The third part of a YACC specification consists of supporting C-routines. A lexical analyzer by the name yylex() must be provided.

**Example:**

YACC source program for a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value.

Grammar for arithmetic expressions

E -> E + T | T

T -> T * F | F

F -> ( E ) | digit

The token digit is a single digit between 0 and 9.

```
%{
#include<stdio.h>
#include<ctype.h>
%}
%token DIGIT
%%
line : expr '\n' { printf("%d\n", $1); }
     ;
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
term : term '*' factor { $$ = $1 * $3; }
     | factor
     ;
factor : '(' expr ')' { $$ = $2; }
       | DIGIT
       ;
%%
void main()
{
yyparse();
}
void yyerror(char *s)
{
fprintf(stderr, "\nerror\n");
}
int yylex() {
int c;
c = getchar();if
(isdigit(c)) {
yylval = c-'0';
return DIGIT;
}
return c;
}
```

**Advantages of Yacc:**

Simplicity: Yacc simplifies the process of building parsers by automating much of the parser generation process based on a formal grammar specification.

Efficiency: Yacc-generated parsers are often highly efficient, with parsing performance comparable to hand-written parsers.

Modularity: Yacc promotes modularity by separating the lexical analysis and parsing phases, allowing developers to focus on each aspect independently.

**Limitations of Yacc:**

Limited Expressiveness: Yacc is limited to parsing context-free grammars and may not be suitable for languages with more complex syntactic features.

Learning Curve: While Yacc automates much of the parser generation process, learning how to use Yacc effectively and understanding the generated parser code may require some initial effort.

**Use Cases:**

Yacc is commonly used in the development of compilers, interpreters, and other language processing tools.

It has been used to build parsers for a wide range of programming languages and domain-specific languages, including C, Pascal, SQL, and more.

**Rules for follow( ):**

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is placed in follow(*B*).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

## Example:

Construct FOLLOW for the grammar:

A -> BC | EFGH | H
B -> b
C -> c | ε
E -> e | ε
F -> CE
G -> g
H -> h | ε

**Solution:**

Finding follow() sets:

follow(A) = {$}

follow(B) = first(C) – { ε } U follow(A) = {C, $}

follow(G) = first(H) – { ε } U follow(A)

={h, ε } – { ε } U {$} = {h, $}

follow(H) = follow(A) = {$}

follow(F) = first(GH) = {g} follow(E)

= first(FGH) U follow(F)

= ((first(F) – { ε }) U first(GH)) U follow(F)

= {c, e} U {g} U {g}

= {c, e, g}

follow(C) = follow(A) U (first (E) – { ε }) U follow (F)

= { $ } U { e } U {g}

= {e, g, $}

Calculate the first and follow functions for the given grammar-

$$S \rightarrow (L) / a$$

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' / \in$$

**Solution-**

The first and follow functions are as follows-

***First Functions-***

- First(S) = { ( , a }
- First(L) = First(S) = { ( , a }
- First(L') = { , , ∈ }

***Follow Functions-***

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