UNIT-2

SYNTAX ANALYSIS & SYNTAX DIRECTED TRANSLATION

1. Role of a Parser

   SOURCEPROGRAM → LEXICAL ANALYZER → PARSER → REST OF FRONT END → intermediate representation
   ↓     ↓             ↓                      ↓
   token                  8th next token tree             SYMBOL TABLE

2. CFGs (CONTEXT-FREE GRAMMERS)

   A CFG consists of terminals, non-terminals, a start symbol, and productions.
   (token) Terminal → Basic symbols from which things are formed.
   Non-terminal → Syntactic variables that denote set of strings.
   Start symbol → A non-terminal starting symbol.
   Productions → Specify how the non-terminals can be combined to form strings.

   Parse Tree - Graphical representation for a derivation
   Leftmost derivation - Only the leftmost non-terminal is replaced at each step.
   Rightmost derivation - Rightmost non-terminal is replaced at each step.
   Ambiguity - A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

3. Top Down Parsing

   A top-down parsing algorithm parses an input string of tokens by tracing out the steps in a leftmost derivation. Such an algorithm is called top-down because the implied traversal of the parse tree is a pre-order traversal and, thus, occurs from the root to the leaves.

   Top-down parsers come in two forms:
   (1) Backtracking Parser
   (2) Predictive Parser
4) **Brute-Force Approach**

Top-down parsing with full back-up is a 'brute-force' method of parsing. In using full back-up we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached.

\[
S \rightarrow aA | bB | ab | cB \\
A \rightarrow aA | b | c |
\]

Tree of a brute force top down parse for string 'aced'.

Due to left-recursive grammar, it causes an infinite loop for the Top-down parser. Left recursion can be removed using left factoring which is given as:

\[
A \rightarrow A_1 | A_2 | \ldots | A_n | B_1 | B_2 | \ldots | B_m \\
\]

which is converted to:

\[
A \rightarrow B_1 | B_2 | \ldots | B_m | B_1 A' | \ldots | B_m A' \\
A' \rightarrow A_1 | A_2 | \ldots | A_n | A_1 A' | \ldots | A_n A' \\
\]

Error recovery is very poor and very inefficient (time consuming).

5) **Recursive-descent Parsing**

It is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each non-terminal of a grammar.

\[
S \rightarrow aA | AB | A \rightarrow b | c | B \rightarrow cD | dD |
\]

procedures CS:

begin
match (a); procedure if stmt;
if tok begin
match (if);
match (C);
exp;

if (exp) statement else statement
end

if stmt
end
match (1);
  statement;
  if token = else then
    match (else);
    statement;
  endif;
end ifstmt;

procedure match (expected token);
begin
  if token = expected token then
    getToken;
    else
      error;
  endif;
end match;

Each function returns a value of true or false depending on whether or not it recognizes a substring which is an expansion of that non-terminal.

#### Predictive Parsing

It is a special form of recursive-descent parsing, in which current input token unambiguously determines the production to be applied at each step.

- No backtracking, efficient and uses LL (1) grammar
- We have to eliminate the left recursion which is not enough for predictive parsing

**Two types**

1. **Recursive Predictive Parsing**
   - Each non-terminal corresponds to a procedure

   \[ A \rightarrow aBB \mid bAB \]

   procedure A ?
   case of the current token ?
   'a': match the current token with a, and move to next token
   - call 'B'
   - match the current token with b, and move to the next token
   'b': match the current token with b, and move to the next token
   - call 'A'
   - call 'B' ?
(2) Non-Recursive Predictive Parsing (also known as LL(1) parsers) -

It is a table driven form. It looks up the production to be applied in a parsing table.

```
INPUT BUFFER
STACK ───> NON-RECURSIVE PREDICTIVE PARSER
      ↓
PARSING TABLE
```

```
Eq: S → aBa, B → bB, ε

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → aBa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B → ε</td>
<td>B → bB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

← LL(1) parsing table.
```

```
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S</td>
<td>abba$</td>
<td>S → aBa</td>
</tr>
<tr>
<td>$aBa</td>
<td>abba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>bba$</td>
<td>B → bB</td>
</tr>
<tr>
<td>$aBb</td>
<td>bba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>ba$</td>
<td>B → bB</td>
</tr>
<tr>
<td>$aBb</td>
<td>ba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>a$</td>
<td>B → a</td>
</tr>
<tr>
<td>$a</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
```

**Constructing LL(1) Parsing Table** -

Two functions are used that is FIRST and FOLLOW.

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from last non-terminal α.
- **FOLLOW(α)** is a set of terminals which occur immediately after the non-terminal α in the strings derived from the starting symbol.

Input read from left to right, one input symbol used to as a look-ahead to determine parse action.
LL(1) Grammar -
- A grammar whose parsing table has no multiple entries
  → A left-recursive grammar can not be a LL(1) grammar
  → A grammar is not left factored, it cannot be a LL(1) grammar.
  → An ambiguous grammar cannot be a LL(1) grammar.

Computing FIRST for any string X -
1. If X is a terminal, then FIRST(X) = \{ X \}
2. If X -> E is a production, then add E to FIRST(X)
3. If X is nonterminal and X -> Y_1 Y_2 ... Y_k is a production then,
   i. Place a in FIRST(X), if for some i, a is in FIRST(Y_i)
   and Y_1 Y_2 ... Y_{i-1} \Rightarrow E \implies E \in \text{all of FIRST}(Y_i)...
   ii. Place E in FIRST(X), if Y_1 Y_2 ... Y_k \Rightarrow E and \n   \text{means E is in all of FIRST}(Y_i)...

Computing FOLLOW for non-terminals -
1. Place $ in FOLLOW(S), where S is the start symbol and $ is 
   the input right endmarker.
2. If there is a production A -> \alpha B \beta, then everything in FIRST(B) 
   except for E is placed in FOLLOW(B).
3. If there is a production A -> \alpha \beta or a production A -> \alpha B \beta 
   where FIRST(B) = \{ E \}, then everything in FOLLOW(A) is in 
   FOLLOW(B).

Algorithm for constructing LL(1) parsing table -
For each production rule A -> \alpha of a grammar G -
→ For each terminal a in FIRST(\alpha), add A -> \alpha to M[A,a]
→ If E \in FIRST(\alpha) then for each terminal a in FOLLOW(A), add 
   A -> \alpha to M[A,a]
→ If E \in FIRST(\alpha) and $ \in FOLLOW(A), then add A -> \alpha to M[A,$]
All other undefined entries of the parsing table are error entries.

My companion
\[ E \rightarrow TE', \quad E' \rightarrow +TE', \quad E, T \rightarrow FT', \quad T' \rightarrow \ast FT' / \epsilon, \quad F \rightarrow (E) \]

\[ \text{Sol.} \quad \text{FIRST}(E) = \{ \epsilon, \text{id} \} \]
\[ \text{FIRST}(E') = \{ +, \epsilon \} \]
\[ \text{FIRST}(T) = \{ \epsilon, \text{id} \} \]
\[ \text{FIRST}(T') = \{ \ast, \epsilon \} \]
\[ \text{FIRST}(E) = \{ \epsilon, \text{id} \} \]

\[ \text{FOLLOW}(E) = \{ \$, \} \]
\[ \text{FOLLOW}(E') = \{ \$, \} \]
\[ \text{FOLLOW}(T) = \{ +, \$, \} \]
\[ \text{FOLLOW}(T') = \{ +, \$, \} \]
\[ \text{FOLLOW}(F) = \{ \$, +, \}, \$ \}

<table>
<thead>
<tr>
<th>\text{id}</th>
<th>+</th>
<th>\ast</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow TE' )</td>
<td>( E \rightarrow TE' )</td>
<td>( E' \rightarrow +TE' )</td>
<td>( E' \rightarrow \epsilon )</td>
<td>( E' \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow FT' )</td>
<td>( T \rightarrow FT' )</td>
<td>( T \rightarrow \ast FT' )</td>
<td>( T \rightarrow \ast FT' )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E \rightarrow \text{id} )</td>
<td>( E \rightarrow (E) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bottom-up Parsing** — (Shift-Reduce parsing)

It attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

A "handle" of a string is a substring that matches the right side of a production.

If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

The approach to reduce the string by a step in the reverse of rightmost derivation is known as **handle pruning**.
Configuration of shift-reduce parser on input id* id$_2$:

Grammar is given as: $E \rightarrow E + T / T$, $T \rightarrow T * F / F$, $F \rightarrow id / (E)$

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>id$_1$ * id$_2$ $$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ id$_1$</td>
<td>* id$_2$ $$</td>
<td>reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$$ F</td>
<td>* id$_2$ $$</td>
<td>reduce by $T \rightarrow F$</td>
</tr>
<tr>
<td>$$ T</td>
<td>* id$_2$ $$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ T*</td>
<td>id$_2$ $$</td>
<td>shift</td>
</tr>
<tr>
<td>$$ T* id$_2$</td>
<td>$$</td>
<td>reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$$ T * F</td>
<td>$$</td>
<td>reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$$ T</td>
<td>$$</td>
<td>reduce by $E \rightarrow T$</td>
</tr>
<tr>
<td>$$ E</td>
<td>$$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Four possible actions a shift-reduce parser can make:
1. Shift  
2. Reduce  
3. Accept  
4. Error

**Conflict during shift-reduce parsing:**
1. **Shift/Reduce conflict** → whether make a shift operation or reduce operation.
2. **Reduce/Reduce conflict** → Parser cannot decide which of several reductions to make.

If a shift-reduce parser cannot be used for a grammar, that grammar is called as Non-LR(k) grammar.

An ambiguous grammar cannot be a LR grammar.

**Types of shift reducing parsers:**
1. Operator-Precidence Parsers
2. LR parsers - covers wide range of grammars -
   1. SLR (Simple LR)
   2. CLR (Canonical or most general LR)
   3. LALR (Look Ahead or Intermediate LR)
8. **Operator Precedence Parsing**

There are two rules in this grammar of operator grammar:

1. No production right side is $\epsilon$.
2. No production has two adjacent non-terminals.

In operator precedence parsing, we define three disjoint precedence relations, $<$, $=$ and $>$ between certain pairs of terminals.

Precedence relation table is given as,

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td></td>
<td>&lt;</td>
</tr>
</tbody>
</table>

Using precedence relations to find handle:
1. Scan the input from left to right until first $>$ is encountered.
2. Scan backwards over $=$ until $<$ is encountered.
3. The handle is a string between $<$ and $>$.

Example: $id + id * id \Rightarrow$ $<$id$>$ + $<$id$>$ * $<$id$>$ $>$

**Advantages:**
1. Simple to implement

**Disadvantages:**
1. The operator like minus has two different precedence (Unary & Binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of program.

Application - **SNOBOL**

9. **LR (1R(k) parsing)**

- $L$ for left-to-right scanning of the input
- $R$ for constructing rightmost derivation in reverse
- $k$ - Number of input symbols of lookahead

It is non-recursive backtracking shift reduce parsing.
Properties of LR parsers:

1. It can be constructed to recognize most of the programming languages for which CFn can be written.
2. The class of grammars that can be parsed by LR parsers is a superset of the class of grammars that can be parsed using predictive parsers.
3. LR parsers work using non-backtracking shift-reduce technique yet it is efficient.

- It detects syntactical errors very efficiently.
- It leads to table-driven parsers.

Structure of LR parsers:

```
[ a | + | b | $ ] INPUT TOKEN
```

```
LR Parser

Top

STACK

State
S0
S1
Sn

Action

Goto

PARSING TABLE

```

\[ \Rightarrow \]

\[ SLR(1) \leq LALR(1) \leq LR(1) \]

\[ \Rightarrow \]

```
SLR Parsing (Simple LR Parsing) - SLR(1)
```

Working of SLR(1):

```
CFn → Construction of canonical set of items → Construction of SLR parsing table → Parsing of input string
```

Definition of LR(0) items and related items:

1. The LR(0) item for grammar G is production rule in which symbol \( \cdot \) is included at some point in RHS of the rule.
   
   The production \( S \rightarrow \cdot \) generates only one item \( S \rightarrow \cdot \).

2. Augmented Grammar - Grammar G have S as start symbol then augmented grammar is \( S' \rightarrow S + \text{grammar G} \).
Augmented grammar indicates the acceptance of input. That is when parser is about to reduce \( S' \rightarrow S \) it reaches to acceptance state.

3. Kernel item - collection of items \( S' \rightarrow \epsilon S \) and all the items whose dots are not at the leftmost end of RHS of the rule.

Non-kernel item - collection of all the items in which one of the leftmost end of RHS of the rule.

4. Function closure and goto - Required to create collection of canonical set of items.

5. Visible prefix - Set of prefixes in the right sentential form of production \( A \rightarrow \alpha \). This set can appear on the stack during shift/reduce action.

→ Closure Operation →

For CFG, if \( I \rightarrow \) set of items then function closure(\( I \)) can be constructed using following rules:

1. Consider \( I \) is a set of canonical items & initially every item \( I \) is added to closure(\( I \)).

2. If rule \( A \rightarrow \alpha \cdot B \beta \) is a rule in closure(\( I \)) and there is another rule for \( B \) such as \( B \rightarrow Y \) then
   closure(\( I \)): \( A \rightarrow \alpha \cdot B \beta \)
   \( B \rightarrow Y \)
   
   This rule is applied until no more new items can be added to closure(\( I \)).

→ Goto Operation →

If there is a production \( A \rightarrow \alpha \cdot B \beta \) then

goto \((A \rightarrow \alpha \cdot B \beta, B) = A \rightarrow \epsilon \cdot B \beta \)

That means simply shifting of one position ahead over the grammar symbol.
Construction of canonical collection of set of item -
(1) For the grammar G initially add $S \rightarrow \cdot S$ in the set of item $C$.
(2) For each set of items $I_i$ in $C$ and for each grammar symbol $X$,
add clause $(I_i, X)$. This process should be repeated by
applying goto $(I_i, X)$ for each $X$ in $I_i$ such that goto $(I_i, X)$
is not empty and not in $C$.

Construction of SLR parsing table -
Input - An Augmented Grammar $G'$
Output - SLR parsing table
Algorithm:
(1) Initially construct set of items $C = \{I_0, I_1, \ldots, I_n\}$ where $C$ is a
collection of set of LR(0) items for the input grammar $G'$.
(2) The parsing actions are based on each item $I_i$. The action are
as given below:
(a) If $A \rightarrow \alpha \cdot A \beta$ is in $I_i$ and goto $(I_i, A) = I_j$ then
set action $[i, A]$ as "shift $j$". Note $\alpha$ is a terminal.
(b) If the $A \rightarrow \alpha \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \··· \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \··· \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·...
Algorithm:
(1) Initially push 0 as initial state onto the stack and place the input string with $\$ as end marker on the input tape.
(2) If $S$ is the state on the top of the stack and $a$ is the symbol from input buffer pointed by a lookaheads pointer then
   a) If action $[S, a]$ is equal to -
      (a) shift $j$ → then push $a$, then push $j$ onto the stack. Advance
          the input lookahead pointer.
      (b) reduce $A \rightarrow \beta$ → then pop $\beta$ symbols. If $i$ is on the top of the stack, then push $i$, then push goto $[i, \beta]$ onto the stack.
   b) accept → then halt the process. It indicates successful parsing.

→ CLR Pass
(Canonical LR Pass or LR(1) Pass)

→ Working of LR(1)

CFG → Construction of canonical set of items along with lookahead → Construction of CLR parsing table → Parsing of input string

→ Construction of canonical set of items along with the lookahead
(1) For the grammar $G$ initially add $S' \rightarrow S$ in the set of item $C$
(2) For each set of items $I_i$ in $\mathcal{C}$ and for each grammar symbol $X$, add closure $(I_i, X)$. This process should be repeated by applying goto $(I_i, X)$ for each $X$ in $I_i$ such that goto $(I_i, X)$ is not empty and not in $C$.
(3) The closure function can be computed as follows
   For each item, $A \rightarrow \alpha \cdot X \beta$ and rule $X \rightarrow Y$ and $b \in \text{FIRST}(X)$ such that $X \rightarrow \cdot Y$ and $b$ is not in $I_i$ then add $X \rightarrow \cdot Y, b$ to $I_i$
(4) Similarly the goto function can be computed as
   For each item $[A \rightarrow \alpha \cdot X \beta, a]$ is in $I_i$ and rule $[A \rightarrow \alpha \cdot X \beta, a]$ is not in goto items then add $[A \rightarrow \alpha \cdot X \beta, a]$ to goto items.
Construction of canonical LR parsing table -

Input - An augmented grammar G'

Output - The canonical LR parsing table

Algorithm -

1. Initially construct set of items C = {I₀, I₁, ..., Iₙ} where C is a collection of sets of LR(1) items for the input grammar G'.

2. The parsing actions are based on each item Iᵢ. The action are as given below-
   
   (a) If [A → α · aβ, b] is in Iᵢ and goto (Iᵢ, a) = Iⱼ then set action [Iᵢ, a] as "shift Iⱼ". Note - A is terminal.
   
   (b) If [A → α · β] is in Iᵢ then set action [Iᵢ, a] to "reduce A → α". Here A should not be $S'$.
   
   (c) If S' → S · $ b in Iᵢ then set action [Iᵢ, $] = "accept".

3. The goto part of the LR table can be filled as -

   The goto transitions for state i is considered for non-terminal only.

4. All the entries not defined by rule 2 & 3 are considered to be "error".

Parsing the input -

Algorithm is same as SLR(1) parsing input algorithm.

LALR Parsing - (lookahead LR Parsing or LALR(1) Parsing)

Working of LALR(1) -

1. Construction of canonical set of items along with lookahead
2. Construction of LR parsing table
3. Parsing of input string

Construction set of LR(1) items with the lookahead -

Same as CLR (LR(1)) form. But only difference is that -

In construction of LR(1) items for LR parse, we have different two states if the second component is different but in this case, we will merge the two states by merging of first and second components from both the states.
Construction of LALR parsing table -

Same as CLR parsing table algorithm. But there is another step is included between Step 1 and Step 2 of CLR parsing table algorithm that is -

Merge the two state Ii and Ij if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as Ij = I; U Ii.

Comparison of LR parsers -

<table>
<thead>
<tr>
<th>SLR PARSER</th>
<th>LALR PARSER</th>
<th>CLR PARSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) SLR parser is smallest in size</td>
<td>LALR and SLR have same in size</td>
<td>CLR parser is largest in size</td>
</tr>
<tr>
<td>(2) Earliest method based on FOLLOW function</td>
<td>Applicable to wider class than SLR</td>
<td>Most powerful than SLR and LALR</td>
</tr>
<tr>
<td>(3) imposes less syntactic features than that of LR parser</td>
<td>Features of a language are incorporated in LALR</td>
<td>Features than that of LR parser</td>
</tr>
<tr>
<td>(4) Error detection is not immediate</td>
<td>Error detection is not immediate</td>
<td>Immediate error detection is done</td>
</tr>
<tr>
<td>(5) Requires both time and space complexity</td>
<td>Time and space complexity is more, but efficient methods exist for constructing LALR</td>
<td>Time and space complexity is more</td>
</tr>
</tbody>
</table>

Graphical representation for the class of LR family is given below -

[Diagram of LR(0), LR(1), LALR(1), and SLR sets]
**YACC Specification** - (YACC - Automatic Parser Generator)

YACC stands for Yet Another Compiler Compiler which is basically a utility available for UNIX.

**YACC** is a parser generator. The YACC can accept conflicts or ambiguities (if at all) in the form of error messages.

Lex and YACC work together to analyze the program syntactically.

- **Specification file**
  - `x.y`
- **YACC Compiler**
  - `y.tab.c` and `y.tab.h`
  - `y.tab.h` file contains all the tokens
- **Parser program**
  - `y.tab.c`
- **CC - The C Compiler**
  - `a.out`
- **Input string**
  - `a.out`
- **Executable program**
  - `Output`

**UNIX Commands** -

- `yacc x.y` → Generate a parse.g program (y.tab.c) using YACC specification file.
- `yacc -d x.y` → Two files generator i.e. `y.tab.c` and `y.tab.h`

YACC specification file contains the CFEs and using the production rules of CFEs the parsing of the input string can be done by `y.tab.c`

Extension to YACC program i.e. `y.tab.c`

**YACC specification** -

It has three sections which can be written as -

```c
/* % */
/* declaration section */
/* % */
/* % */
/* * Translation rule section */
/* % */
/* % */
/* * Required C functions */
/* % */
/* % */
```
(1) Declaration part - Ordinary C and grammar token declaration.

(2) Translation Rule section - All production rule of CFG:

\[
S \rightarrow A b / B a, \quad A \rightarrow a, \quad B \rightarrow b \quad \text{can be written as}
\]

\[
S : A b \\
\quad / / \text{rule 1 action 1}
\]

\[
B a \\
\quad / / \text{rule 1 action 2}
\]

\[
A : a \\
\quad / / \text{rule 2 action 3}
\]

\[
B : b \\
\quad / / \text{rule 3 action 4}
\]

(3) C function section - Consists of one main function in which the routine `yyparse()` will be called. Also it consists of required C function.

Compiling and running of LEX and YACC program:

```
# lex calc1.1  //create lex.yy.c
# yacc -d calc1.1 //create y.tab.c and y.tab.h
# cc y.tab.c lex.yy.c -ll -y -lm
    //compile with lex.yy.c and y.tab.c by linking common library file
# ./a.out    //will run executable file of program
```

- Shift/reduce conflict can be resolved by YACC with the help of precedence rules. If there is same precedence then YACC checks for associativity. For left associativity shift action will be performed and for right associativity reduce action will be perform.

- Rule section - `$1$` is used as attributed value at C/H5 of the grammar.

\[
gr \rightarrow E + E \Rightarrow $1 + $2, \quad $2 = +
\]

- Subroutine section - `yyparse()` is called which in turn calls `yylex()` when it requires tokens. The routine `yyerror()` is used to print the error message when an error is occurred in parsing of input.
**Syntax Directed Definitions Translation -**

1. **Syntax Directed Definition (SDD)** -

<table>
<thead>
<tr>
<th>INPUT STRING</th>
<th>SYNTAX TREE</th>
<th>DEPENDENCY GRAPH</th>
<th>EVALUATION ORDER FOR SEMANTIC RULES</th>
</tr>
</thead>
</table>

SDD is a kind of abstract specification which is used while doing the static analysis of the language. It means an augmented CFG is generated.

A parse tree containing the values of the attributes at each node is called an annotated or decorated parse tree.

**Definition -** Syntax directed definition is a generalization of CFG in which each grammar production \( X \rightarrow \alpha \) is associated with a set of semantic rules of the form \( a := f(b_1, b_2, \ldots, b_k) \), where \( a \) is an attribute obtained from the function \( f \).

**Attribute -** It can be a string, a number, a type, a memory location or anything else.

Consider \( X \rightarrow \alpha \) be a CFG and \( a := f(b_1, b_2, \ldots, b_k) \) where \( a \) is the attribute. There are two types of attributes:

1. **Synthesized attribute** -
   - The attribute \( a \) is called synthesized attribute of \( X \) and \( b_1, b_2, \ldots, b_k \) are attributes belonging to the production symbols.
   - The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.
2. **Inherited attribute** -
   - The attribute \( a \) is called inherited attribute of one of the grammar symbols on the right side of production (i.e., \( \alpha \)) and \( b_1, b_2, \ldots, b_k \) are belonging to either \( X \) or \( \alpha \).
   - The inherited attribute can be computed from the values of the attributes at the sibling and parent of that node.

---

*my companion*
### Syntax-directed definition

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E1 + T</td>
<td>( E.\text{val} := E_1.\text{val} + T.\text{val} )</td>
</tr>
<tr>
<td>E → E1 - T</td>
<td>( E.\text{val} := E_1.\text{val} - T.\text{val} )</td>
</tr>
<tr>
<td>E → T</td>
<td>( E.\text{val} := T.\text{val} )</td>
</tr>
<tr>
<td>T → T1 * E</td>
<td>( T.\text{val} := T_1.\text{val} \times E.\text{val} )</td>
</tr>
<tr>
<td>T → T1 / E</td>
<td>( T.\text{val} := T_1.\text{val} / E.\text{val} )</td>
</tr>
<tr>
<td>T → F</td>
<td>( T.\text{val} := E.\text{val} )</td>
</tr>
<tr>
<td>E → (E)</td>
<td>( E.\text{val} := E.\text{val} )</td>
</tr>
<tr>
<td>E → digit</td>
<td>( E.\text{val} := \text{digit}.\text{eval} )</td>
</tr>
<tr>
<td>N → ;</td>
<td>Can be ignored by lexical analyzer (terminal)</td>
</tr>
</tbody>
</table>

The token digit has synthesized attribute \( \text{eval} \) whose value can be obtained from lexical analysis.

In SDD, terminals have synthesized attributes only. No definition of \( \text{eval} \) is needed. The SDD that uses only synthesized attribute is called **s-attribute definition**.

To compute s-attributes definition -

1. Write SDD
2. Annotated parse tree is generated & attributes' value are computed in bottom up manner.
3. The value of obtained at root node is the final output.

To compute inherited attributes -

1. Write SDD
2. Annotate the parse tree with inherited attributes by processing in top down fashion.
Example of computation of $S$-attributed definer:

Annotated Parse Tree →  $S$

\[
E_{val} = 11 \quad N
\]
\[
E_{val} = 5 \quad + \quad T_{val} = 6
\]
\[
T_{val} = 5
\]
\[
E_{val} = 6
\]
\[
digit_{:} lenval = 6
\]
\[
digit_{:} lenval = 5
\]

Example of computation of inherited attributes:

Annotated Parse Tree →

```
S

<table>
<thead>
<tr>
<th>Value obtained from child to parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>T: Type = int</td>
</tr>
<tr>
<td>L: Lin = int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value obtained from sibling to child</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>Lin = int</td>
</tr>
</tbody>
</table>
```

```
S → TL
T → int
T → float
T → char
T → double
L = L₁, id
L₁ : Lin = Lin
id
```

Production Rule | Semantic Action
-----------------|-----------------|
$S \rightarrow TL$ | Lin := T.Type
$T \rightarrow int$ | T.Type := int
$T \rightarrow float$ | T.Type := float
$T \rightarrow char$ | T.Type := char
$T \rightarrow double$ | T.Type := double
$L := L_1, id$ | L₁ : Lin := Lin
$id$ | EnterType (identifier, Lin)

Dependency Graph:

The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called a dependency graph.

```
E \rightarrow E₁ + E₂
```

- Solid arrows → depending
- Dotted lines → parse tree

\[
E_{val} \quad + \quad E_{₁ val} + \quad E₂ \quad val
\]

mycompanion
-> Evaluation Order -

The topological sort of the dependency graph decided the evaluation order in a parse tree. In deciding evaluation order, the semantic rules of the syntax directed definitions are used. Thus the translation is decided by SDD. Therefore, precise definition of SDD is required.

2) Construction of Syntax Trees -

Syntax tree is an abstract representation of the language constructs.

-> For Enpumen -

Following functions are used in syntax tree for enpumen:
1. mknode (op, left, right)
2. mkleaf (id, entry)
3. mkleaf (num, val)

-> Directed Acyclic Graph for enpumen - (DAG)

DAG is directed graph drawn by identifying common subexpressions.

e.g. Enpumen \[ k := k + 5 \]

```
\[
\text{SYNTAX TREE:} \quad k \quad + \quad 5
\]
```

```
\[
\text{DAG:} \quad k \quad + \quad 5
\]
```

Sequence of operation -
1. \( p_1 = \text{mkleaf}(\text{id}, k) \)
2. \( p_2 = \text{mkleaf}(\text{num}, 5) \)
3. \( p_3 = \text{mknode}(\+, p_1, p_2) \)
4. \( p_4 = \text{mkleaf}(\text{num}, \text{id}, k) \)
5. \( p_5 = \text{mknode}(\text{=}, p_4, p_3) \)

3) Bottom-Up Evaluation of S-Attribute Definition -

-> Synthesized Attributes on the pass stack -

(1) A translator for S-attribute definition is implemented using LR parse generator.
(2) A bottom-up method is used to parse the input string.
(3) A parser stack is used to hold the values of synthesized attributes.
(4) After reduction, top is decremented as top = top + 1 for r symbols.
(5) If the symbol has no attribute, then the corresponding entry in the value array will be kept undefined.

Example - \( X \rightarrow ABC \)

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \rightarrow ABC )</td>
<td>( X.x = f(A.a, B.b, C.c) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Reduction</th>
<th>After Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>VALUE</td>
</tr>
<tr>
<td>A</td>
<td>A.a</td>
</tr>
<tr>
<td>B</td>
<td>B.b</td>
</tr>
<tr>
<td>C</td>
<td>C.c</td>
</tr>
</tbody>
</table>

PARSER STACK

\[ \text{PARSER STACK} \]

Passthrough input:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input String</td>
<td>State Value</td>
</tr>
</tbody>
</table>

\[ \text{Input String} \]

L-attribute Definition (evaluated in depth-first order).

The SDD can be defined as the L-attribute for the production rule \( A \rightarrow X_1X_2...X_n \) where the inherited attribute \( X_i \) is such that \( 1 \leq i \leq n \). The production \( A \rightarrow X_1X_2...X_i \) is such that:

1. It depends upon the attributes of the symbols \( X_{i+1}, X_{i+2}, ..., X_n \) to left of \( X_i \).
2. It also depends upon the inherited attribute.

L-attribute Definition (evaluated in depth-first order).

A SDD is L-attribute if each inherited attribute of \( X_j \), \( 1 \leq j \leq n \) on the right side of the production \( A \rightarrow X_1X_2...X_n \) defines only on:

1. The attributes of the symbols \( X_{i+1}, X_{i+2}, ..., X_n \) to left of \( X_j \) in the production
2. The inherited attributes of \( A \).

Every S-attribute definition is L-attribute.
Translation Scheme -

The process of execution of code fragment semantic actions from the SDD is called syntax directed translation. Thus, the execution of SDD can be done by syntax directed translation scheme.

A translation scheme generates the output by executing the semantic actions in an ordered manner.

While designing the translation scheme -

We have to follow one restriction that is for every semantic action if it refers to some attribute then that attribute value must be computed before that attribute gets defined.

5 Top-down Translation -

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁ + T</td>
<td>{E.val := E₁.val + T.val}</td>
</tr>
<tr>
<td>E → E₁ - T</td>
<td>{E.val := E₁.val - T.val}</td>
</tr>
<tr>
<td>E → T</td>
<td>{E.val := T.val}</td>
</tr>
<tr>
<td>T → (E)</td>
<td>{T.val := E.val}</td>
</tr>
<tr>
<td>T → digit</td>
<td>{T.val := digit.val}</td>
</tr>
</tbody>
</table>

Removing left recursion & rewrite the translation scheme for non left recursion grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → T</td>
<td>{P.in := T.val}</td>
</tr>
<tr>
<td>P</td>
<td>{E.val := P.s}</td>
</tr>
<tr>
<td>P → + T</td>
<td>{P₁.in := P₁.in + T.val}</td>
</tr>
<tr>
<td>P₁</td>
<td>{P₁.s := P₁.s}</td>
</tr>
<tr>
<td>P → - T</td>
<td>{P₁.in := P₁.in - T.val}</td>
</tr>
<tr>
<td>P₁</td>
<td>{P₁.s := P₁.s}</td>
</tr>
<tr>
<td>P → E</td>
<td>{Pₛ := P.in}</td>
</tr>
<tr>
<td>T → (E)</td>
<td>{T.val := E.val}</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>{T.val := digit.len.val}</td>
</tr>
</tbody>
</table>

my companion digit
Annotated parse tree can be drawn as:

```
E           E
T           T
P           P
+           +
digit       digit
T           T
-           -
digit       digit
E
```

**TOP DOWN TRANSLATION**

In the parse tree, the top-down translation takes place. The blue lines show the parse tree whereas the black arrow lines show the way of computing values of expression.

We can construct syntactic tree for the translation scheme using mknod, mknclt (op, left, right), mknclt (id, entry) & mknclt (num, val)

**Bottom up Evaluation of Inherited Attributes** -

The bottom-up parse reduces the right side of the production X -> ABC by removing C, B and A from the parse stack.

<table>
<thead>
<tr>
<th>PRODUCTION RULE</th>
<th>CODE FRAGMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow T\text{-}list;$</td>
<td>\text{value}[\text{top}] = \text{int}</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>\text{value}[\text{top}] = \text{float}</td>
</tr>
<tr>
<td>$T \rightarrow \text{float}$</td>
<td>Enter-type(\text{value}[\text{top}], \text{value}[\text{top}-3])</td>
</tr>
<tr>
<td>$\text{List} \rightarrow \text{List}, id$</td>
<td>Enter-type(\text{value}[\text{top}], \text{value}[\text{top}-1])</td>
</tr>
</tbody>
</table>

**Recursive Evaluation** -

Recursive function that evaluate attributes as they traverse a parse tree can be constructed from a SDD using a generalization of the techniques for predictive translation.

Such functions allows us to implement SDD that cannot be
implemented simultaneously with parsing.

In a translation specified by this definition, the children of
a node for one production need to be visited from left to right,
while the children of a node for the other productions need to be
visited from right to left.

The functions need not depend on the order in which the
same tree nodes are visited.

3. Analysis of Syntax Directed Definition

→ Strongly non-circular SDD

A SDD is said to be strongly noncircular if for each
nonterminal A we can find a partial order RA on the attributes of
A such that for each production p in the left side A and nonterminal
A₁, A₂, ..., Aₙ occurring on the right side:

1) Dₚ[RA₁, RA₂, ..., RAₙ] is acyclic, and
2) if there is an edge from attribute Aₖ to Aₗ in Dₚ[RA₁,RA₂,...,RAₙ],
   then RA orders Aₖ before Aₗ.

→ Circularity Test

A SDD is said to be circular if the dependency graph
for some parse tree has a cycle.
To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

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

2. If X → ε is a production, then add ε to FIRST(X).

3. If X is nonterminal and X → Y₁Y₂⋯Yₖ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yᵢ), and ε is in all of FIRST(Y₁), …, FIRST(Yᵢ₋₁); that is, Y₁⋯Yᵢ₋₁ \[\Rightarrow\] ε. If ε is in FIRST(Yⱼ) for all j = 1, 2, ⋯, k, then add ε to FIRST(X). For example, everything in FIRST(Y₁) is surely in FIRST(X). If Y₁ does not derive ε, then we add nothing more to FIRST(X), but if Y₁ \[\Rightarrow\] ε, then we add FIRST(Y₂) and so on.

Now, we can compute FIRST for any string X₁X₂⋯Xₙ as follows. Add to FIRST(X₁X₂⋯Xₙ) all the non-ε symbols of FIRST(X₁). Also add the non-ε symbols of FIRST(X₂) if ε is in FIRST(X₁), the non-ε symbols of FIRST(Xₙ) if ε is in both FIRST(X₁) and FIRST(X₂), and so on. Finally, add ε to FIRST(X₁X₂⋯Xₙ) if, for all i, FIRST(Xᵢ) contains ε.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.

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

3. If there is a production A → αβ, or a production A → αββ where FIRST(β) contains ε (i.e., β \[\Rightarrow\] ε), then everything in FOLLOW(A) is in FOLLOW(β).
Example 4.11: Construct the LR(1) parsing table for the following grammar:

1. \( S \rightarrow CC \)
2. \( C \rightarrow aC \)
3. \( C \rightarrow d \)

Solution: First we will construct the set of LR(1) items:

\[
\begin{align*}
I_0: & \\
S \rightarrow S, \epsilon & \rightarrow S, $ \\
S \rightarrow CC, \epsilon & \rightarrow S, $ \\
C \rightarrow aC, \epsilon & \rightarrow C, a \\
C \rightarrow d, \epsilon & \rightarrow C, d \\
S \rightarrow S, \epsilon & \rightarrow S, $ \\
S \rightarrow C, \epsilon & \rightarrow S, $ \\
C \rightarrow aC, \epsilon & \rightarrow C, a \\
C \rightarrow d, \epsilon & \rightarrow C, d \\
I_0 \text{ goto } (I_0, S) & \rightarrow I_0, S, $ \\
I_0 \text{ goto } (I_0, C) & \rightarrow I_0, C, $ \\
I_0 \text{ goto } (I_0, a) & \rightarrow I_0, aC, $ \\
I_0 \text{ goto } (I_0, d) & \rightarrow I_0, d, $ \\
I_0 \text{ goto } (I_0, \epsilon) & \rightarrow I_0, \epsilon, $ \\
I_0 \text{ goto } (I_0, \epsilon) & \rightarrow I_0, \epsilon, $ \\
I_0 \text{ goto } (I_0, $) & \rightarrow I_0, $, $ \\
\end{align*}
\]

We will initially add \( S \rightarrow S, $ \) as the first rule in \( I_0 \). Now match

\( S \rightarrow S, $ \) with

\( \{A \rightarrow a \times (a, a)\} \)

Hence, \( S \rightarrow S, $ \) and

\( A \rightarrow a \times (a) \)

\( A = S, a = \epsilon, S = a = $ \)

If there is a production \( X \rightarrow \gamma, b \) then add \( X \rightarrow \gamma, b \)

\( :S \rightarrow CC, $ \)

\( \beta \in \text{FIRST}(b) \)

\( \beta \in \text{FIRST}(S) \) as \( eS = $ \)

\( \beta \in \text{FIRST}(S) \)

\( b = $ \)

\( :S \rightarrow CC, $ \) will be added in \( I_0 \)

The LR(1) parsing table as follows:

<table>
<thead>
<tr>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
</tbody>
</table>

The remaining blank entries in the table are considered as syntactical error.

Parsing the input using LR(1) parsing table:

Using above parsing table we can parse the input string "an$ax" as:

\[ \begin{array}{l}
\text{Stack} \quad \text{Input buffer} \quad \text{action table} \quad \text{goto table} \quad \text{Parsing action} \\
50 \quad \text{add} \, 8 \quad \text{action}[3, 8] = 63 \quad \text{Shift} \\
50a3 \quad \text{a} \quad \text{action}[3, a] = 63 \quad \text{Shift} \\
50a3a6 \quad \text{d} \quad \text{action}[6, d] = 64 \quad \text{Shift} \\
50a3a6d4 \quad \text{d} \quad \text{action}[6, d] = 64 \quad \text{Shift} \\
50a3a6C8 \quad \text{d} \quad \text{action}[6, d] = 64 \quad \text{Shift} \\
50C2 \quad \text{d} \quad \text{action}[2, d] = 67 \quad \text{Shift} \\
50C2a5 \quad \text{d} \quad \text{action}[5, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{a} \quad \text{action}[7, a] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\
50C2a5d7 \quad \text{d} \quad \text{action}[7, d] = 67 \quad \text{Shift} \\n\end{array}\]

Thus the given input string is successfully parsed using LR parser or canonical LR parser.
Example 4.43:

$ S \rightarrow CC $

$ C \rightarrow aC $

$ C \rightarrow d $

Construct the parsing table for LALR(1) parser.

Solution: First the set LR(1) items can be constructed as follows with merged states.

```
<table>
<thead>
<tr>
<th>$ S \rightarrow S' C $</th>
<th>$ S' \rightarrow S \ V $</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ S \rightarrow aC $</td>
<td>$ S \rightarrow aC, a \ $</td>
</tr>
<tr>
<td>$ C \rightarrow d $</td>
<td>$ C \rightarrow d, a \ $</td>
</tr>
</tbody>
</table>

$ S \rightarrow S' \ V $ $ \rightarrow d \ V T $ $ \rightarrow d \ V S $ $ \rightarrow d \ V d $
```

Now consider state $ I_0 $ there is a match with the rule $ A \rightarrow a \ $ and goto $ (I_0, a) = I_1 $.

$ C \rightarrow aC, a/d/S $ and if the goto is applied on $ a $ then we got the state $ I_3 $. Hence we will create entry action(0, a) = goto $ S $.

Similarly,

In $ I_1 $

$ C \rightarrow d, a/d/S $

$ A \rightarrow a \ $ and $ A \rightarrow a \ $ if the goto is applied on $ a $ then we got the state $ I_3 $. Hence we will create entry action(0, a) = goto $ S $.

For state $ I_2 $

$ C \rightarrow d, a/d/S $ $ A \rightarrow a \ $ $ A \rightarrow a \ $ $ A \rightarrow a \ $ $ A \rightarrow a \ $ $ A \rightarrow a \ $ $ A \rightarrow a \ $ $ action(47, a) = reduce by C \rightarrow d $ i.e. rule 3

$ action(47, d) = reduce by C \rightarrow d $ i.e. rule 3

$ action(47, S) = reduce by C \rightarrow d $ i.e. rule 3

LALR(1) parsing table as follows:

<table>
<thead>
<tr>
<th>States</th>
<th>Action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>538</td>
<td>847</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>538</td>
<td>847</td>
</tr>
<tr>
<td>3</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>5</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>6</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>7</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>8</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>10</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>11</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>12</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>13</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>14</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>15</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>16</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>17</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>18</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>19</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>20</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>21</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>22</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>23</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>24</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>25</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>26</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>27</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>28</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>29</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>30</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>31</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>32</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>33</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>34</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>35</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>36</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>37</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>38</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>39</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>40</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>41</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>42</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>43</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>44</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>45</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>46</td>
<td>88</td>
<td>r2</td>
</tr>
<tr>
<td>47</td>
<td>88</td>
<td>r2</td>
</tr>
</tbody>
</table>

Parsing the input string using LALR parser

The string having regular expression $ a^* d a^* d a^* d $ grammar $ G $. We will consider input string as $ "a add d " $ for parsing by using LALR parsing table.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input buffer</th>
<th>Action table</th>
<th>goto table</th>
<th>Parsing action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 0 $</td>
<td>$ a d d $</td>
<td>$ action(0, a) = 538 $</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d $</td>
<td>$ d $</td>
<td>$ action(36, d) = 847 $</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(47, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow d</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(88, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow aC</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(47, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow aC</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(88, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow aC</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(47, d) = 847 $</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(36, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow d</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(47, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow aC</td>
<td></td>
</tr>
<tr>
<td>$ 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d 0 a d d $</td>
<td>$ d $</td>
<td>$ action(88, d) = 847 $</td>
<td>(88, C) = 89 Reduce by C \rightarrow aC</td>
<td></td>
</tr>
</tbody>
</table>

Thus the LALR and LR parser will mimic one another on the same input.
Example 4.7: Construct the SLR(1) parsing table for

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

Solution: We will first construct a collection of canonical set of items for the above grammar. The set of items generated by this method are also called SLR(0) items. As there is no lookahead symbol in this set of items.

| $i_0$ | $E \rightarrow E + T$
| $i_1$ | $E \rightarrow T$
| $i_2$ | $T \rightarrow T * F$
| $i_3$ | $T \rightarrow F$
| $i_4$ | $F \rightarrow (E)$
| $i_5$ | $F \rightarrow id$

| goto ($i_0$) | $i_2$
| $i_1$ | $E \rightarrow E + T$
| $i_2$ | $T \rightarrow T * F$
| $i_3$ | $T \rightarrow F$
| $i_4$ | $F \rightarrow (E)$
| $i_5$ | $F \rightarrow id$

| goto ($i_1$) | $i_3$
| $i_2$ | $T \rightarrow T * F$
| $i_3$ | $T \rightarrow F$
| $i_4$ | $F \rightarrow (E)$
| $i_5$ | $F \rightarrow id$

| goto ($i_2$) | $i_3$
| $i_4$ | $T \rightarrow F$
| $i_5$ | $F \rightarrow id$

| goto ($i_4$) | $i_1$
| $i_2$ | $T \rightarrow T * F$
| $i_3$ | $T \rightarrow F$
| $i_4$ | $F \rightarrow (E)$
| $i_5$ | $F \rightarrow id$

Finally the SLR(1) parsing table will look as:

<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S_0$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$S_0$</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$E \rightarrow E + T$</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>$T \rightarrow T * F$</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$S_5$</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>$F \rightarrow (E)$</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>$F \rightarrow id$</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>$S_5$</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>$S_5$</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>$S_5$</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>$T \rightarrow T * F$</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>$F \rightarrow (E)$</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>$F \rightarrow id$</td>
<td>13</td>
</tr>
</tbody>
</table>

Remaining blank entries in the table are considered as syntactical errors.

Input string: id + id + id

We will consider two data structures while taking the parsing actions and those are - stack and input buffer.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input buffer</th>
<th>action table</th>
<th>goto table</th>
<th>Parsing action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>id</td>
<td>id + id</td>
<td>0</td>
<td>Reduce by $E \rightarrow E + T$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id + id</td>
<td>$T \rightarrow T * F$</td>
<td>1</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id + id</td>
<td>$F \rightarrow (E)$</td>
<td>2</td>
<td>Reduce by $F \rightarrow (E)$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>id</td>
<td>id + id</td>
<td>3</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>id + id</td>
<td>$F \rightarrow (E)$</td>
<td>4</td>
<td>Reduce by $F \rightarrow (E)$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>id + id</td>
<td>$F \rightarrow id$</td>
<td>5</td>
<td>Reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>S12</td>
<td>id + id</td>
<td>$T \rightarrow T * F$</td>
<td>6</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>S12</td>
<td>id + id</td>
<td>$F \rightarrow (E)$</td>
<td>7</td>
<td>Reduce by $F \rightarrow (E)$</td>
</tr>
<tr>
<td>S12</td>
<td>id + id</td>
<td>$F \rightarrow id$</td>
<td>8</td>
<td>Reduce by $F \rightarrow id$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id</td>
<td>id + id</td>
<td>9</td>
<td>Reduce by $E \rightarrow E + T$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id + id</td>
<td>$T \rightarrow T * F$</td>
<td>10</td>
<td>Reduce by $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id + id</td>
<td>$F \rightarrow (E)$</td>
<td>11</td>
<td>Reduce by $F \rightarrow (E)$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>id + id</td>
<td>$F \rightarrow id$</td>
<td>12</td>
<td>Reduce by $F \rightarrow id$</td>
</tr>
</tbody>
</table>