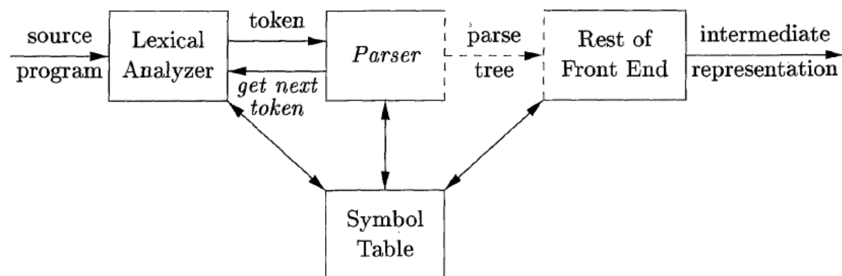
UNIT- II

Syntax Analysis

Role of Parser:

- Parser gets a string of tokens from lexical analyzer then construct parse tree and passes it to rest of the compiler for further processing.
- Checking and translation actions can be a part of parsing. So parse tree need not constructed explicitly.
- Parser can report any syntax error. It also recovers from commonly occurring errors to continue parsing.



- There are three types of parsers for grammar
    1. Universal
    2. Top down
    3. Bottom up
- Universal type of parser is inefficient to use in production compilers.
- Parsers commonly used in compiler are either top down or bottom up
    1. Top down parser built parse tree from top (root) to bottom (leaves).
    2. Bottom up method start from leaves work up to root
- In either case, input to parser is scanned from left to right, one symbol at a time.
- LL and LR grammar are enough to describe most syntactic structures in programming language.
- Parsers for class of LL grammar are constructed by hand and parsers for larger class of LR grammar are constructed by automated tools.

Syntax error handling:

- If a compiler process only correct programs, its design and implementation would be simplified greatly.
- Compiler is expected to assist programmer in locating and tracking down errors and error handling is left to compiler designer.
- Parsing methods like LL and LR methods detect syntactic error efficiently.

- Common Programming errors occur at different levels of compiler are
    1. Lexical errors include misspellings as identifier, keyword and operators.
    2. Syntactic errors include misplaced semicolons and extra or missing braces.
    3. Semantic errors include type mismatches between operators and operands.
    4. Logical errors can be anything from incorrect reasoning
- Accurate detection of semantic and logical errors at compile time is a difficult task.
- Error handling in parser has some goals those are

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly to detect subsequent errors.
3. Add minimum overhead to processing of correct programs.

- Error handler report the line in which error is detected, because of this there is a good chance to detect actual error occurred within previous few tokens.

Error Recovery Strategies:

- Simplest approach for parser is to quit from processing with informative error message when it detects first error.
- If number of errors is larger, it is better for compiler to quit after exceeding error limit.
- Some error recovery strategies are :

1. Panic mode
2. Phrase level
3. Error production
4. Global corrections

1. Panic mode:

- In this method, on discussing error, parser discards input symbols one at a time until designated set of tokens is found.
- Panic mode correction skips considerable amount of input without checking it for additional errors.
- It is very simple but it is guaranteed not to go into infinite loop.

2. Phrase level recovery :

- On discovering error, Parser perform local corrections on remaining input like replace prefix of remain input with a string.
- Local correction is to replace common by semicolon, delete semicolon or inserting missing semicolon.
- We must be careful to choose replacements that do not lead to infinite loops.
- Major drawback is very difficult to identify situation in which actual error has occurred before point of detection.

3. **Error Production :**
   - By expecting common errors that might encounter, we construct grammar for language at hand with production that generates error part.
   - These error productions detect errors when parser using these production. It also provides appropriate error diagnostics for errors those recognized in input.
4. **Global Correction:**
   - Global Correction contains algorithms; those are used for choosing minimal subsequent changes to obtain globally least cost correction.
   - These provides small number of changes to convert incorrect string x to correct string y.
   - These methods are too costly to implement in terms of time and space. So these techniques are currently only theoretical.

Context Free Grammar:

- Many programming language constructs have inherently recursive structure that can be defined by context free grammar.
- CFG consists of terminals, non terminals, stat symbol and productions.
  1. Terminals are basic symbols from which strings are formed. When we are talking about grammars of Programming language if , then and else keywords are terminals.
  2. Non terminals are syntactic variables that denote set of strings. These non terminals are helpful in define language generated by grammar. Statement and expression are non terminals.
  3. In grammar one non terminal is indicated as start symbol and set of strings it denotes is language generated by grammar.
  4. Productions of grammar specify the manner in which terminals and non terminals can be combined to form string. Each production of
     - Non terminal called head or left side of production.
     - Symbol → or ::=
     - Body or right side of production consists of zero or more terminals and non terminals.

expr → expr op expr
expr → (expr)
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑

Start symbol: expr
Terminals: id, +, -, *, /,↑
Non terminals: expr, op

National conventions:

1. Normally lower case letters, operators, digits, punctuation symbols (parenthesis, comma etc), boldface strings, if and id are terminals.
2. Normally uppercase letters, lowercase italic names such as expr or stmt are non terminals. letter s is starting symbol.
3. Uppercase letters x, y, z represent grammar symbol i.e either terminal or non terminals.
4. Lowercase letters u, v, w, ----z represent (empty) strings of terminals.
5. Lowercase Greek letters α, β, γ represent (empty) strings of grammar symbols.
6. If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ---- $A \rightarrow \alpha_k$ are productions with A on left then we write $A \rightarrow \alpha_1 | \alpha_2 |$-----$\alpha_k$.
7. Unless stated otherwise, left side of the first production is start symbol.

Example:

expression → expression + term

expression → expression – term

expression → term

term → term * factor

term → term / factor

term → factor

factor → (expression)

factor → id

Using the above conventions given grammar is rewritten as

E → E + T | E - T | T

T → T * F | T / F | F

F → (E) | id

Derivations:

- Construction of parse tree can be made exactly by taking a derivational view, in which productions are treated as rewriting rules.
- In derivation, we start with starting symbol; each rewriting step replaces a non-terminal by body of one of its productions.
- This derivational view corresponds to top down construction of parse tree, but the correctness afforded by derivations will helpful when bottom up parsing is discussed.
- At each step in derivation, there are two choices to be made. We need to choose which non terminal to replace .Based on this derivations are two types
  1. leftmost derivation
  2. rightmost derivation

- In leftmost derivation, the left most non terminal in each sentential is always chosen .If $\alpha \Rightarrow \beta$ is step in which leftmost non terminal in α is replaced, we write as $\alpha \underset{lm}{\Rightarrow} \beta$.
- In rightmost derivation the right most non terminal is always chosen, we write as $\alpha \underset{lm}{\Rightarrow} \beta$.

Example: construct leftmost and rightmost derivations for given grammar for string id + id.

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

Leftmost derivation is

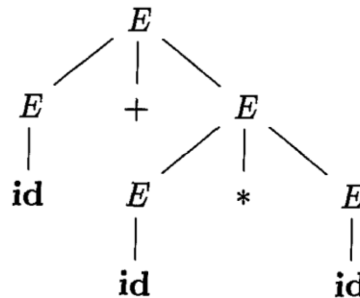
$$E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} id + E \underset{lm}{\Rightarrow} id + id$$

Rightmost derivation is

$$E \underset{rm}{\Rightarrow} E + E \underset{rm}{\Rightarrow} E + id \underset{rm}{\Rightarrow} id + id$$

Parse Tree:

- Parse tree is graphical representation of derivation that filters out the order which productions are applied to replace non terminals.
- Interior node is labelled with non terminal in the head of production.
- Leaves of parse tree are labelled by non terminal or terminals.
    - Parse tree of the string id + id * id for given grammar E → E + E | E * E | (E) | id is



- There is a many to one relationship between parse trees and derivations.

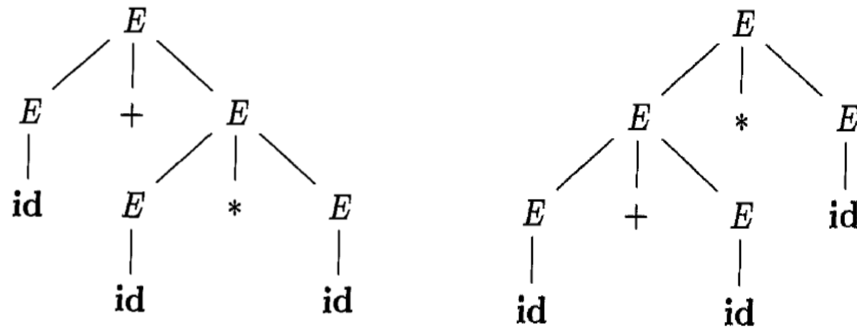
Ambiguity:

- A grammar that produces more than one parse tree for some input string Is said to be ambiguous.
- Ambiguous grammar is one that produces more than one left most derivation or more than one right most derivation for some input string.
- Below grammar permits two distinct left most derivations for input string "id + id *id ".

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

$$\begin{aligned} E &\underset{lm}{\Rightarrow} E + E \\ &\underset{lm}{\Rightarrow} id + E \\ &\underset{lm}{\Rightarrow} id + E * E \\ &\underset{lm}{\Rightarrow} id + id * E \\ &\underset{lm}{\Rightarrow} id + id * id \end{aligned}$$

$$\begin{aligned} E &\underset{rm}{\Rightarrow} E * E \\ &\underset{rm}{\Rightarrow} E + E * E \\ &\underset{rm}{\Rightarrow} id + E * E \\ &\underset{rm}{\Rightarrow} id + id * E \\ &\underset{rm}{\Rightarrow} id + id * id \end{aligned}$$



Context Free Grammar Vs Regular Expression:

- Context free grammars are strictly more powerful than regular expressions.
- Any language that can be generated using regular expressions can be generated by context free grammar but not vice versa.
- Every regular language is context free language but not vice versa.
- Regular expression $(a|b)^*$ abb and grammar.

  $A_0 \rightarrow aA_0|bA_0|aA_1$

  $A_1 \rightarrow bA_2$

  $A_2 \rightarrow bA_3$

  $A_3 \rightarrow \varepsilon$

  Describe the same language, the set of strings as a 's and b 's ending in abb.
- The usage of regular expression is in context of lexical analysis phase, where as context free grammar is in context of syntax analysis phase.
- Regular expressions are very easy to understand when compare to context free grammar.

Lexical Vs Syntax Analysis:

- Separating syntactic structure of language into lexical and non lexical parts provides compiler front end into two manageable sized components.
- Lexical rules of language are frequently quite simple; we do not need notation as grammar to describe them.
- Regular expressions are easier to understand notation for tokens than grammars.
- More efficient lexical analyzers can be constructed automatically from regular expressions than from grammars.
- Regular expressions are useful for describing construct like identifier, constants, keywords and whitespaces. Grammars, on other hand useful for describing if-then-else, balanced parenthesis, matching begin-ends.

Eliminating Ambiguity:

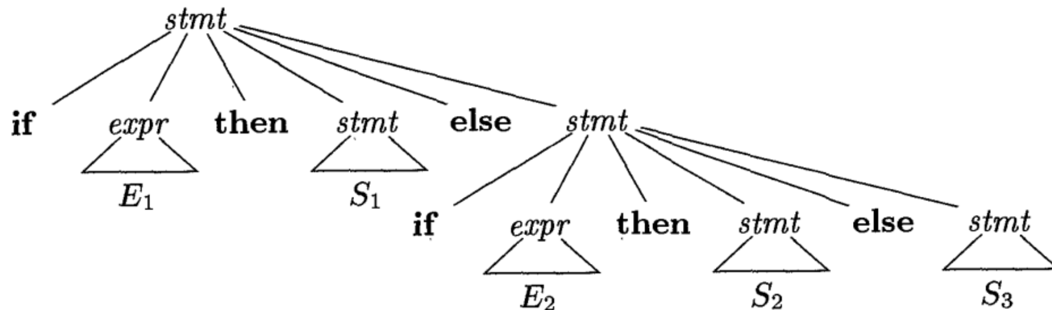
- Sometimes an ambiguous grammar can be rewritten to eliminate ambiguity.
- Eliminate the ambiguity from following dangling else grammar:

  Stmt→ if expression then statement

  | if expr then stmt else stmt

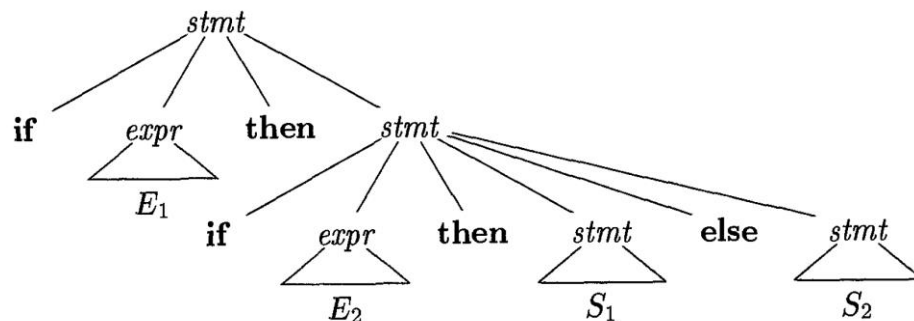
  | other
- According to this grammar, the compound conditional statement ,

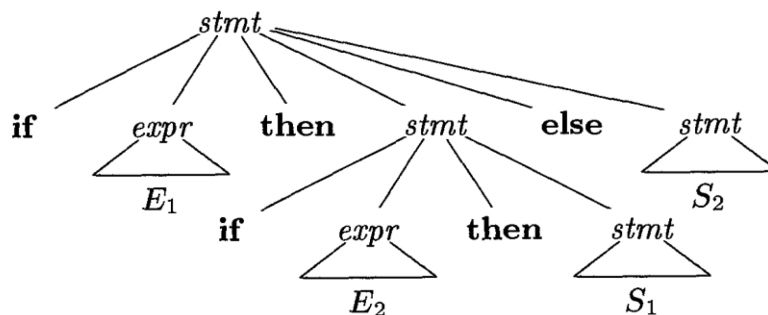
  If E_1 then S_1 else if E_2 then S_2 else S_3.



- Grammar is ambiguous since the string,

  If E1 then if E_2 then S_1 else S_2.





- To eliminate the ambiguity for above grammar, we can reconstruct the above grammar as shown below.

  stmt → matched_stmt

  |open_stmt

  matched_stmt → if expr then matched_stmt else matched_stmt

  | other

  open_stmt → if expr then stmt

  | If expr then matched_stmt else open_stmt

Elimination of Left Recursion:

- Grammar is left recursive if it has non terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
- Top down parsing methods can't handle left recursive grammars .So, eliminate left recursion.
- To eliminate left recursion ,each left recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by non left recursive productions:

$A \rightarrow \beta A^I$

$A \rightarrow \alpha A^I \mid \varepsilon$

Example: Eliminate the left recursion for the below grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Left recursion elimination process

$E \rightarrow E + T \mid T$	$T \rightarrow T * F \mid F$	$F \rightarrow (E) \mid id$
$E \rightarrow T E^I$	$T \rightarrow F T^I$	No left recursion
$E^I \rightarrow T E^I \mid \varepsilon$	$T^I \rightarrow * F T^I \mid \varepsilon$	

After elimination grammar is

$E \rightarrow T E^I$
$E^I \rightarrow + T E^I \mid \varepsilon$
$T \rightarrow F T^I$
$T^I \rightarrow * F T^I \mid \varepsilon$
$F \rightarrow (E) \mid id$

Algorithm for Left Recursion:

Input: Grammar G with no cycles or ε-production.

Output: Equivalent grammar with no left recursion.

Method: Resulting non left recursive grammar may have €-productions.

1. Arrange the non terminals in some order $A_1, A_2, \ldots\ldots, A_n$.
2. For(each i from 1 to n){
3. For(each j from 1 to i-1){
4. Replace each production of form $A_i \rightarrow A_j\, \gamma$ by productions
   $A_i \rightarrow \delta_1 \gamma \mid \delta_2\, \gamma \mid \ldots\ldots \mid \delta_k\, \gamma$
   Where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots\ldots \mid \delta_k$ are all current A_j productions
5. }
6. Eliminate left recursion among A_i productions.
7. }

Left Factoring:

- When the choice between two alternative A-productions is not clear by reading initial elements in input .This situation is called non-deterministic.
- Top down parsing methods can't be non-deterministic situation .So, eliminate non-deterministic by left factoring.
- To implement left factoring, each non-deterministic productions $A \rightarrow \alpha\beta_1 / \alpha\beta_2$ can be replaced by

$$A \rightarrow \alpha A^I$$
$$A^I \rightarrow \beta_1 \mid \beta_2$$

Example: Eliminate non-deterministic (left factoring) on below grammar

$$S \rightarrow i\ E + S \mid i\ E + S\ e\ S \mid a$$
$$E \rightarrow b$$

Left recursion elimination process

$S \rightarrow i\ E + S \mid i\ E + S\ e\ S \mid a$	$E \rightarrow b$
$S \rightarrow i\ E + S\ S^I \mid a$	No non deterministic
$S \rightarrow e\ S \mid \varepsilon$	

After elimination grammar is

$$S \rightarrow i\ E + S\ S^I \mid a$$
$$S \rightarrow e\ S \mid \varepsilon$$
$$E \rightarrow b$$

Left Factoring Grammar:

Input: Grammar G

Output: Equivalent left factored grammar

Method: For each non terminal A, find longest prefix α common to two or more its alternatives. Replace all A-productions $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \ldots\ldots \mid \alpha\ \beta_n \mid \gamma$ Where r represents all alternative not begin with α by

$$A \rightarrow \alpha\ A^I \mid \gamma$$
$$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

Non Context Free Language Constructs:

- Few syntactic constructs found in typical programming languages cant by specified using grammars alone. Two of these constructs are shown below
  1. Language consists of form WCW, where first W represents declaration of identifier W, C represents program fragment, and second W represents the user of identifier.
  2. Problem by checking number of formal parameters i_n declaration of function agrees with number of actual parameters in the use of function. Language consists of string of

form $a^n b^m c^n d^m$. $a^n b^m$ represents formal parameter list and $c^n d^m$ represents actual parameter list

Top down parsing:

- Top down parsing can be viewed as problem of constructing parse tree for input, starting from root and creating nodes for parse tree in pre order.
- Top down parsing can be viewed as finding left most derivation for input string.
- At each step, determining production to be applied for non terminal say A is key problem. Once A production is chosen, rest of process consists matching terminals in production body with input.

Example: sequence of parse trees of Top down approach for input id + id * id with

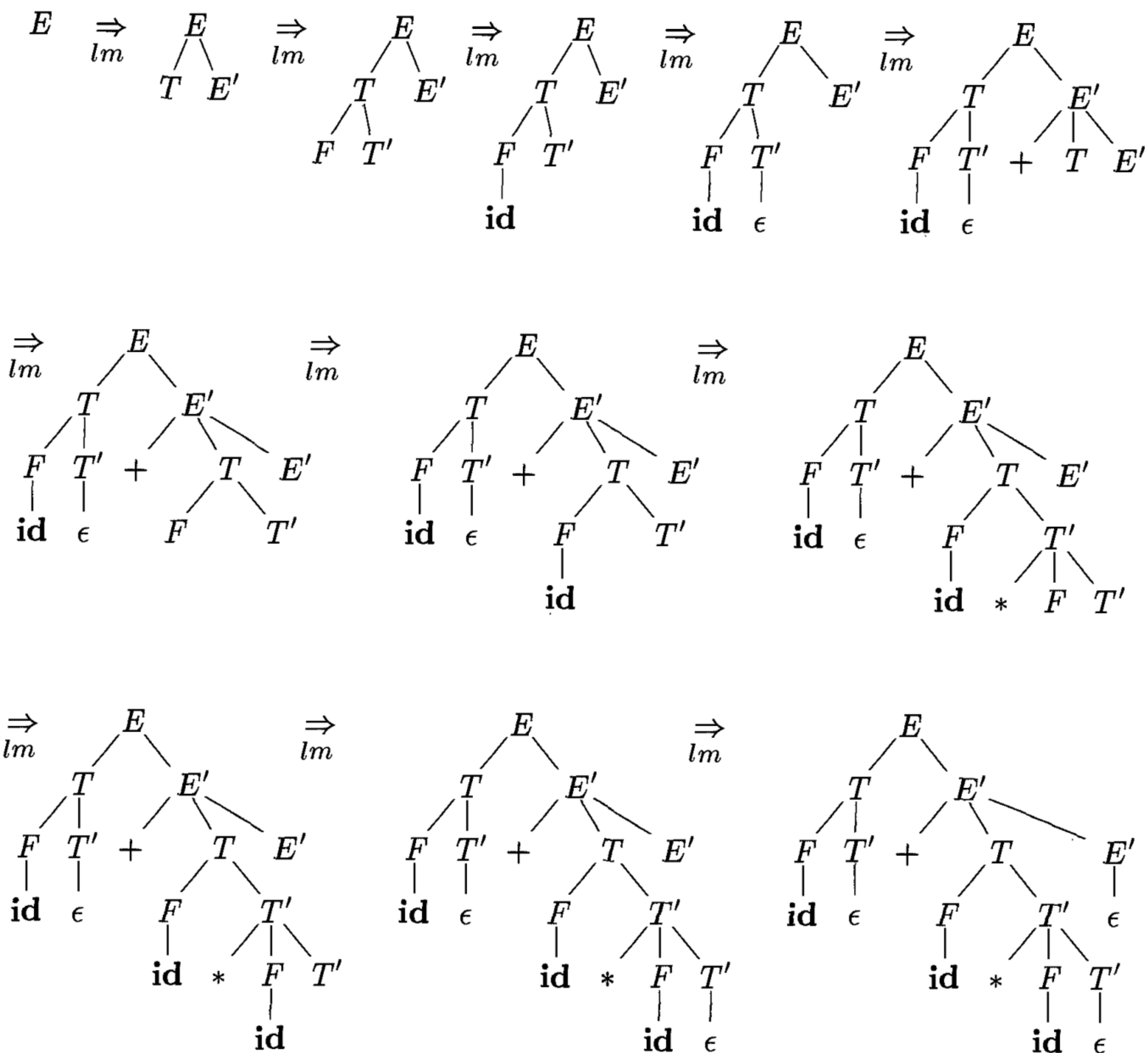
$E \rightarrow T E^I$

$E^I \rightarrow + T E^I \mid \varepsilon$

$T \rightarrow F T^I$

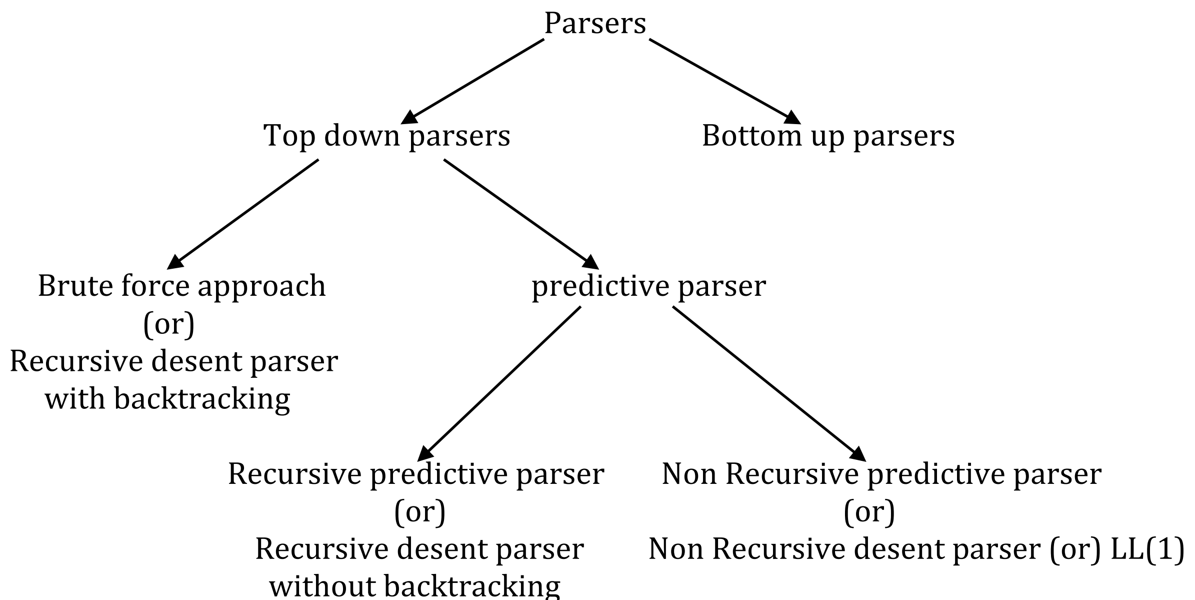
$T^I \rightarrow * F T^I \mid \varepsilon$

$F \rightarrow (E) \mid id$



Procedure for non terminal in top down parser:

```
   Void A () {
1) Choose A productions A → X1X2X3......Xk;
2) For(i=1 to K) {
3)       if(Xi is a nonterminal)
4)           Call procedure Xi();
5)       else if(Xi equal current input symbol a)
6)           Advance the input to next symbol;
7)       else    /*an error has occurred*/;
      }
    }
```



- Problem with Top Down parsing are
  1. back tracking
  2. left recursion
  3. left factoring
  4. ambiguity

Brute force approach:

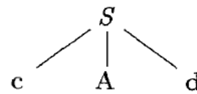
- It requires back tracking: that is it may require repeated scans over input.
- Back tracking parsers are not seen frequently because reading input number of times may be complex and time consuming task.
- Brute force approach do not keep restriction on grammar that is grammar can have left recursion and left factoring

Example consider grammar

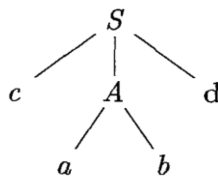
S → cAd

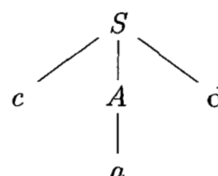
A → ab | a Construct parse tree for input string w=cad.

- Being with root element labelled S, and input pointer pointing to C first symbol of w. S has only one production expands it.



- Left most leaf labelled c, matches first symbol of input w. so, we advance pointer to a, second symbol of w and next leaf element labelled A.
- We expand A using first alternative, a match for second symbol a, so we advances pointer to d, third input symbol and compare d against next has labelled b. b doesn't match d. We report failure and go back to A to check another alternative.
- In going back to A, we must reset input pointer to position 2, then proceed with alternative production. To store input pointer position we use a local variable.
- In alternative production leaf a match 2nd symbol as W and leaf d match 3rd symbol of w then halt and announces completion of parsing.





- Biggest drawback of brute force in recursive descent with backtracking parser is, if one of a phase enters into infinite loop due to backtracking in left recursion lead compiler or machine to crash.

Predictive Parsing:

- Predictive parse doesn't allow grammar, that has left recursion, non deterministic and backtracking.
- Predictive parser is special type of recursive descent parser.
- It can predicted with production is suitable for completion of parsing based on input symbol.

Recursive Predictive Parsing:

- Recursive descent parsing program consists of set of procedures. One for each non terminal.
- It consists input buffer contains string to be parsed, followed by end marker $.
- It can be built by maintaining stack implicitly via recursive calls by using one input symbol of look ahead at each step to make parsing decision.

Example: Let us consider grammar

E → id T

T → + id T | ε   to parse input string id + id with recursive descent approach.

Procedure:

```
E()
{
    if (lookahead=='id')
  {
     match ('id');
       T();
}
else
       return;
}

T()
{
if (lookahead=='+')
  {
     match('+');
    if (lookahead=='id')
     {
          match('id
');
              T();
      }
     else
            return ;
   }
   else
         return;
}

match(chart t)
 {
    if (lookahead=='t')
         lookahead==next-token;
        else
          printf("error")
 }
main()
 {
    E();
    if(lookahead=='$')
    printf("Success") ;
 }
```

First and Follow:

- Construction of both top down and bottom up parser is by two functions, First and Follow, associated with grammar G.
- During top down parsing, First and Follow allow us to choose which production to apply based on the next input symbol.
- During panic mode error recovery, sets of tokens produced by follow can be used as synchronizing token.
- Define first(A), consider two A-productions $A \rightarrow \alpha \mid \beta$ where first(α) and first(β) are disjoint sets .we can choose between these A-productions by looking at next input symbol a. since a can be in at most one of first(α) and first(β) not both.
- To compose 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 a terminal then first(x) = {x}
  2. If X is non terminal and $X \rightarrow Y_1Y_2.........Y_k$ is a production for k >= 1 then add all non ε symbols of first(Y_1) to first(X). If first(Y_1) contains ε then add non ε symbols of first(Y_2) to first(X) and so on.
  3. If $X \rightarrow \varepsilon$ is production ,then add ε to first(X)
- Define follow(A), for non terminal A, to be set of terminals a that can appear immediately to right of A in some sentential form $S \Rightarrow \alpha A a \beta$, for some α and β. If A can be rightmost symbol in some sentential form then $ is in follow(A).
- To compute follow(A) for all non terminals A, apply the following rules until nothing can be added to any follow set.
  1. Place $ in follow(S), where S is the starting symbol and $ is input right end marker.
  2. If there is a production $A \rightarrow \alpha B \beta$, where first(β) doesn't contains ε, then everything in first(β) is in follow(B)
  3. If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$, where first(β) contains ε, then everything in follow(A) is in follow(B)

Example: Consider grammar

$E \rightarrow T E^I$

$E^I \rightarrow + T E^I \mid \varepsilon$

$T \rightarrow F T^I$

$T^I \rightarrow * F T^I \mid \varepsilon$

$F \rightarrow (E) \mid id$ construct first and follow for the given grammar

Procedure:

Terminals = {+, *, id, (,)}

Non-Terminals= {E, E^I, T, T^I, F}

first of terminals:

first(+)={ + }

first(*)={ * }

first(id)={ id }

first(()={ (}

first())={) }

first(**ε**)={ **ε** }

first of non terminals:

<u>first(E):</u>

first(E)=first(TE^I) (∴ rule 2)

=first(T) =first(FT^I) (∴ rule 2)

=first(F)

F → (E)	F → id
(rule 2) first(F)=first((E)) ={ (}	(rule 1) first(F)=first(id) ={ id }

∴ first(F)={ (, id }

∴ first(E) = first(T) = first(F) = {(,id}

<u>first(E^I):</u>

E^I → + T E^I	E^I → **ε**
(rule 2) first(E^I)=first(+TE^I) =first(+) ={ + }	(rule 1) first(E^I)=first(**ε**) ={ **ε** }

∴ first(E^I)={ +, **ε** }

<u>first(T^I):</u>

T^I → * F T^I	T^I → **ε**
(rule2) first(T^I)=first(*FT^I) =first(*) ={ * }	(rule 1) first(T^I)=first(**ε**) ={ **ε** }

∴ first (T^I)={ *, **ε** }

follow of non terminals:

: Non terminals={E, E^I, T, T^I, F}

Follow(E):

(rule 1) If E is starting symbol follow(E)={ $ }	F→ (E) (rule 2) follow(E)= first( )) ={) }

∴ follow(E)={ $ } ∪ {) }

= { $,) }

Follow(E^I):

E→ TE^I (rule 3) follow (E^I)=follow(E) ={ $,) }	E^I→+TE^I (rule 3) follow(E^I)=follow(E^I) ={ $,) }

∴ follow(E^I)={ $,) } ∪ { $,) }

={ $,) }

∴ follow(E)=follow(E^I)={ $,) }

Follow(T):

E →TE^I (rule 2) follow(T)=first(E^I) if ε is there ={ first(E^I) - { ε }} ∪ follow(E) ={{+, ε} - { ε }} ∪ { $,) } ={ +, $,) }	E^I→+TE^I (rule 2) follow(T)=first(E^I) if ε is there ={ first(E^I) - { ε }} ∪ follow(E^I) ={{+, ε} - { ε }} ∪ { $,) } ={ +, $,) }

∴ follow(T)={ +, $,) } ∪ { +, $,) }

= { +, $,) }

∴ follow(T)={ +, $,) }

Follow(T^I):

T→FT^I (rule 3) follow(T^I)=follow(T) ={ +, $,) }	T^I→*FT^I (rule 3) follow(T^I)=follow(T^I) ={ +, $,) }

∴ follow (T^I)={ +, $,) }

Follow(F):

$T \rightarrow FT^I$	$T^I \rightarrow *FT^I$
(rule 2) follow(F)= first(T^I) if ε is there ={ first(T^I) - { ε }} ∪ follow(T) ={{*, ε} - { ε }} ∪ { +, $,) } ={ *, +, $,) }	(rule 2) follow(F)= first(T^I) if ε is there ={ first(T^I) - { ε }} ∪ follow(T^I) ={{*, ε} - { ε }} ∪ { +, $,) } ={ *, +, $,) }

∴ follow (F)= { *, +, $,) } ∪ { *, +, $,) }

= { *, +, $,) }

∴ follow(T)={ *, +, $,) }

	First	Follow
E	(, id	$,)
E^I	+, **ε**	$,)
T	(, id	+, $,)
T^I	*, **ε**	+, $,)
F	(, id	*, +, $,)

LL(1) grammar:

- Non recursive predictive parsing can be constructed for class as grammar called LL(1).
- First 'L' in LL(1) stands for scanning input from left to right, second 'L' for producing left most derivation and '1' for using one input symbol of lookahead at each step to make parsing decision.
- Grammar G is LL(1) if and only if whenever A→ α | β are two distinct productions as G ,by holding following conditions.
  1. For terminal a, both α and β cannot drive strings beginning with a.
  2. At most one as α and β can drive empty string.
- Next algorithm collects information from FIRST and FOLLOW sets into predictive passing table M [A, a], it is two-dimensional array, where A is no terminal, a is terminal and $ is end marker.

Algorithm for construction of parsing table

INPUT: Grammar G.

OUTPUT: Parsing table M .

Method: Each production A → α of grammar, do the following:

1. For each terminal a in first (α), add A → α to M [A, a].
2. If ε in first(α) ,then for each terminal b in follow(A) add A → α to M[A ,b].
3. There is no production at all in M [A, a] then set M [A, a] to error.

- In LL(1) grammar ,each table entry uniquely identifies a production or signal an error. Some grammar are not LL(1),because table entry multiply defined.

Example: Construct predictive passing table for the below grammar

$E \rightarrow T E^I$

$E^I \rightarrow + T E^I \mid \varepsilon$

$T \rightarrow F T^I$

$T^I \rightarrow * F T^I \mid \varepsilon$

$F \rightarrow (E) \mid id$

Procedure:

"Construct FIRST and FOLLOW"

	First	Follow
E	(, id	$,)
E^I	+, **ε**	$,)
T	(, id	+, $,)
T^I	*, **ε**	+, $,)
F	(, id	*, +, $,)

Production $E \rightarrow TE^I$ is in the form $A \rightarrow \alpha$

first(TE^I)=first (T)

={ (, id }

Therefore $E \rightarrow TE^I$ is place in M[E, ( ] and M[E, id].

Production $E^I \rightarrow +TE^I$

first($+TE^I$)={ + }

Therefore $E^I \rightarrow +TE^I$ is place in M[E^I, +].

Production $E^I \rightarrow \varepsilon$

first(ε)={ ε }

Therefore $E^I \rightarrow \varepsilon$ is placed in M[E^I, follow(E^I)]

$E^I \rightarrow \varepsilon$ is placed in M[E^I, ( ] and M[E^I, id].

Production $T \rightarrow FT^I$

first(FT^I)=first(F)

={ (, id }

Therefore $T \rightarrow FT^I$ is placed in M[T, ( ] and M[T, id].

Production $T^I \rightarrow *FT^I$

first($*FT^I$)=first(*)

={ * }

Therefore $T^I \rightarrow *FT^I$ is placed in M[T^I, *].

Production $T^I \rightarrow \varepsilon$

first(ε)={ ε }

Therefore $T^I \rightarrow \varepsilon$ is placed in M[T^I, FOLLOW(T^I)]

$T^I \rightarrow \varepsilon$ is placed in M[T^I, +], M[T^I, $] and M[T^I,)].

Production F → (E)

first((E))= first (()={ (}

Therefore F → (E) is placed in M[F, ( ].

Production F → id

first(id)={id}

Therefore F → id is placed in M[F, id].

Non Terminal	Input Symbols					
	Id	+	*	(	)	$
E	$E \rightarrow TE^I$			$E \rightarrow TE^I$		
E^I		$E^I \rightarrow +TE^I$			$E^I \rightarrow \varepsilon$	$E^I \rightarrow \varepsilon$
T	$T \rightarrow FT^I$			$T \rightarrow FT^I$		
T^I		$T^I \rightarrow \varepsilon$	$T^I \rightarrow *FT^I$		$T^I \rightarrow \varepsilon$	$T^I \rightarrow \varepsilon$
F	F → id			F → (E)		

Therefore The given grammar is LL(1) because every entry in table is unique production.

Example: consider a grammar

$S \rightarrow i\ E\ t\ S\ S^I \mid a$

$S^I \rightarrow e\ S \mid \varepsilon$

E → b

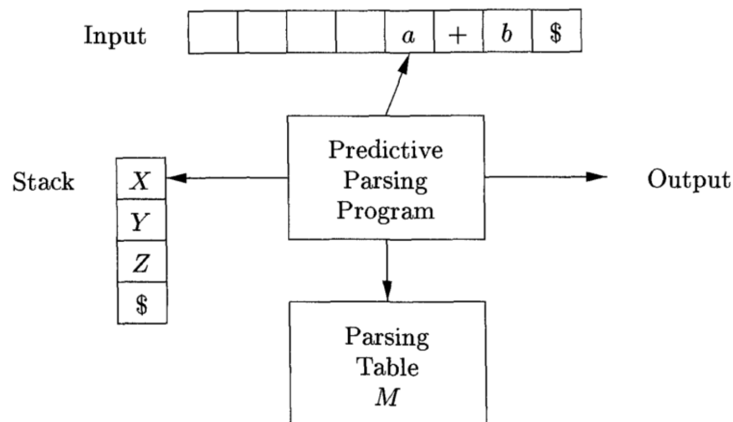
Check whether the given grammar is LL(1) or not

Non Terminal	Input Symbols					
	A	b	e	i	t	$
S	S → a			$S \rightarrow i\ E\ t\ S\ S^I$		
S^I			$S^I \rightarrow \varepsilon$ $S^I \rightarrow e\ S$			$S^I \rightarrow \varepsilon$
E		E → b				

Therefore Given grammar is not LL(1) because multiple production entries in M[S^I,e].

Non recursive predictive parsing:

- Non recursive predictive parser can be built by maintaining stack explicitly rather than implicitly via recursive calls.
- w is input that has been matched so far, stack holds sequence of grammar symbols.
- Double driven parser has input buffer, stack a parsing table constructed based on LL(1)grammar.
- Input buffer contains string to parse followed by end marker $. we use symbol $ to mark the bottom of the stack and initially top of stack is starting symbol.



- Program considers X, top of stack, a is current input symbol, then parser chooses x productions by consulting M[X, a] in parsing table m. otherwise check for matching to input if X is terminal.

Algorithm for table driven predictive parsing:

Input: string W and parsing table M for grammar G.

Output: if w is in L [G] leftmost derivation of W or error.

Method: program takes input and parsing table for parsing the input.

```
let a be first symbol of W
let X be top of stack
while(X!= $) {/*stack is not empty*/
    if(X=a) pop the stack and a be the next symbol of W.
        else if(X is terminal) error();
        else if(M [X, a] is error entry) error();
        else if(M[X, a] = X→Y1Y2……..Yk) {
            pop the stack;
            push Yk,Yk-1…….Y1, on to stack with Y1 on top.
        }
  let X be top stack symbol;
 }
```

Example: consider grammar

$E \rightarrow T E^{I}$

$E^{I} \rightarrow + T E^{I} \mid \varepsilon$

$T \rightarrow F T^{I}$

$T^{I} \rightarrow * F T^{I} \mid \varepsilon$

$F \rightarrow (E) \mid id$

Construct parsing table and check id+id*id string is accepted by grammar or not.

Stack	Input	Action
$ E	id + id * id $	output $E \rightarrow T E^{I}$
$ E^{I} T	id + id * id $	output $T \rightarrow F T^{I}$
$ E^{I} T^{I} F	id + id * id $	output $F \rightarrow id$
$ E^{I} T^{I} id	id + id * id $	match id
$ E^{I} T^{I}	+ id * id $	output $T^{I} \rightarrow \varepsilon$
$ E^{I}	+ id * id $	Output $E^{I} \rightarrow + T E^{I}$
$ E^{I} T +	+ id * id $	match +
$ E^{I} T	id * id $	output $T \rightarrow F T^{I}$
$ E^{I} T^{I} F	id * id $	output $F \rightarrow id$
$ E^{I} T^{I} id	id * id $	match id
$ E^{I} T^{I}	* id $	output $T^{I} \rightarrow * F T^{I}$
$ E^{I} T^{I} F *	* id $	match *
$ E^{I} T^{I} F	id $	output $F \rightarrow id$
$ E^{I} T^{I} id	id $	match id
$ E^{I} T^{I}	$	output $T^{I} \rightarrow \varepsilon$
$ E^{I}	$	output $E^{I} \rightarrow \varepsilon$
$	$	accepted

Error recovery in predictive parsing:

- Error is detected during predictive parsing when top of stack does not match next input symbol or M [A, a] is error.
- Error recovery in predictive parsing is done in 2 ways.
    1. Panic mode
    2. phase level recovery

Panic mode:

- It is based on idea of skipping over symbols on input until a set of synchronizing tokens appear.
- Some ways are as follows.

1. place all symbols in follow(A) in to the synchronizing set for non terminal. If we skip tokens until an element of follow(A) is seen and pop A from stack.

2. if we add symbols in first(A) to synchronizing set for non terminal A, then it may possible to resume parsing to A if symbol in first(A) appears in input.
3. if non terminal can generate empty string, then deriving e can be used as default. It postpones some error but not missed.
4. if terminal on top of stack cannot be matched, simple idea is to pop terminal, issue message saying terminal was inserted.

Example: consider grammar

$E \rightarrow T E^I$

$E^I \rightarrow + T E^I \mid \varepsilon$

$T \rightarrow F T^I$

$T^I \rightarrow * F T^I \mid \varepsilon$

$F \rightarrow (E) \mid id$

Construct parsing table and apply follow and first symbols into synchronizing sets and check the acceptance of)id*+id.

Non Termi	Input Symbols					
	Id	+	*	(	)	$
E	$E \rightarrow TE^I$			$E \rightarrow TE^I$	synch	synch
E^I		$E^I \rightarrow +TE^I$			$E^I \rightarrow \varepsilon$	$E^I \rightarrow \varepsilon$
T	$T \rightarrow FT^I$	synch		$T \rightarrow FT^I$	synch	synch
T^I		$T^I \rightarrow \varepsilon$	$T^I \rightarrow *FT^I$		$T^I \rightarrow \varepsilon$	$T^I \rightarrow \varepsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

Stack	Input	Action
$ E	) id * + id $	error , skip)
$ E	id * + id $	id is in first(E)
$ E^I T	id * + id $	
$ E^I T^I F	id * + id $	
$ E^I T^I id	id * + id $	
$ E^I T^I	* + id $	
$ E^I T^I F *	* + id $	
$ E^I T^I F	+ id $	error , M[F, +] = synch,
$ E^I T^I	+ id $	F has been popped
$ E^I	+ id $	
$ E^I T+	+ id $	
$ E^I T	id $	
$ E^I T^I F	id $	
$ E^I T^I id	id $	
$ E^I T^I	$	
$ E^I	$	
$	$	

Phrase Level recovery:

- It is implemented by filling in the blank entries by error routines. These routines may change, insert or delete symbols on input and issue error messages.
- They pop from stack alteration of stack symbols or inserting new symbols on stack is not supported for several reasons.
    1. Steps carried out by parser might then not correspond to derivation of any word in language.
    2. We must ensure that there is no possibility of an infinite loop.

Types of Grammar:

Type 0(Unrestricted Grammar):

- If there is no restriction on any grammar then that grammar is categorized as type 0 or Un-restricted grammar.
- In this grammar, non terminal and terminals in production has no limit.

  α → β
  α ∈ (N+T)+
  β ∈ (N+T)*
  Example: aAb → bB
  aA → ε

Type 1 (Context Sensitive):

- Apply some restrictions to type 0 grammar is called as type 1 or Context sensitive grammar.
- Context sensitive means before and after of non terminal should be a terminal or non terminal.
- Right hand side of grammar is always greater than in length of length hand side.

  α → β (∴|α|<=|β|)
  α, β ∈ (N+T)+
  Example: aAb → bbb
  aA → bB

Type 2(Context Free):

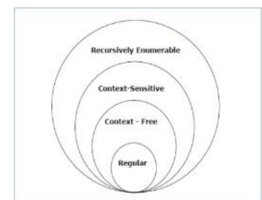
- Apply context free restriction on type 1 grammar is called Context free grammar.
- Context free means before or after of any non terminal on left hand should be empty.
- In this grammar, left hand side of a production should be only one non terminal.

  α → β (∴|α|=1)
  β ∈ (N+T)*
  Example: A → BCD
  B → a

Type 3(Regular):

- If any grammar is left linear, right linear and middle linear then it is called linear grammar.
- If any grammar is left linear and right linear, but not middle linear then it is called regular grammar.

  Examples: A → xB/y (∴ RL)
  A → Bx/y (∴ LL)
  A, B ∈ N
  x, y ∈ T*



Bottom up parsing:

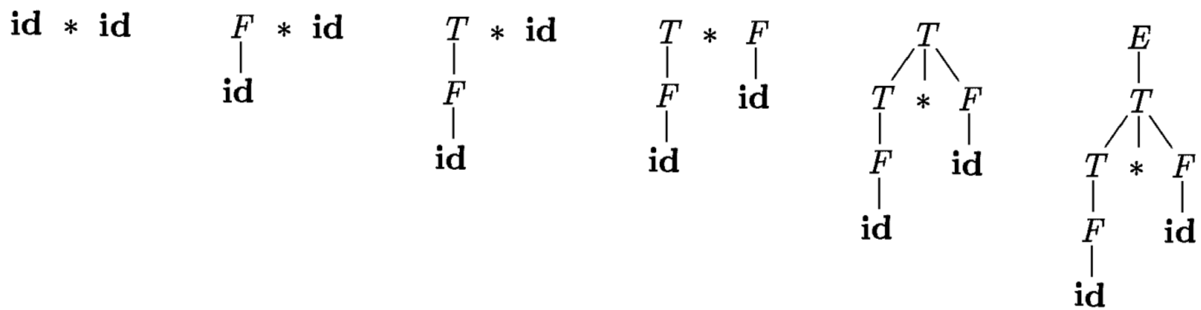
- Bottom up parse corresponds to the construction of parse tree for input string beginning at leaves and working up towards root.
- Largest class of grammars for which shift reduce parser can be built is LR grammars
- It is too much work to built LR parser by hand, tools like automated parser generators make it is to construct LR parser from suitable grammars.

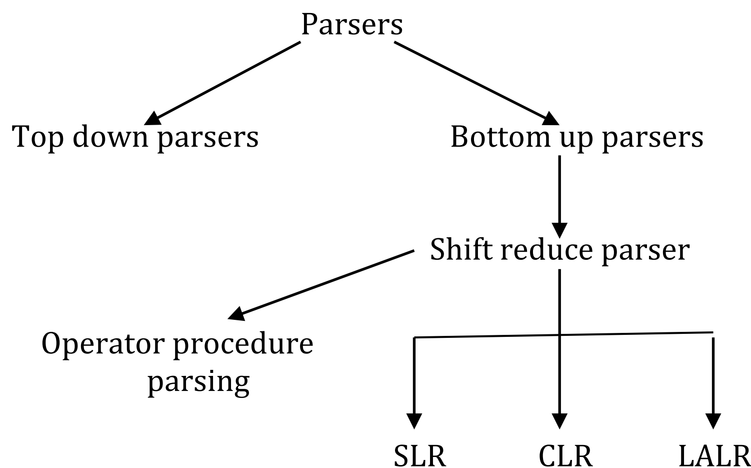
Example: Sequence of parse tree of bottom up approach for input id * id with

E → E + T | T

T → T * F | F

F → (E) | id





Reduction:

- bottom up parsing is process of reducing string w to start symbol of grammer.at each reduction step ,substring matching body is replace by head of production.
- Key decisions during bottom up parsing are about when to reduce and about what production to apply.

Example: consider grammar

E → E + T | T

T → T * F | F

F → (E) | id

and sequence of reduction of above grammar for string id*id is

Id * id F * id T * id T * F T E

Handle pruning:

- Handle is substring that matches the body of production and whose reduction represents one step along the reverse of right most derivation.
- Process of replacing handle with head of production is called handle pruning.
- Leftmost substring that matches the body of some production need not be handle.

Right Sentential Form	Handle	Reducing Production
$id_1 * id_2$	id_1	F → id
F * id_2	F	T → F
T * id_2	Id_2	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

Shift Reduce parsing:

- It is the form of bottom up parsing in which stack holds grammar symbols and input buffer holds rest of string to be parsed.
- Handle always appear at top of stack. We use $ to make bottom of stack and right end of input.
- Initially, stack is empty and string w on input.

Stack	input
$	w$

- During parsing, parser shifts zero or more input symbols onto stack, until it is ready to reduce, then reduce it with head of production.
- The parser repeats cycle until it has detected an error or until stack contain start symbol and input is empty.

Stack	input
$S	$

- there are actually four possible actions shift reduce parser can make is as follows
  1. Shift: shift next input symbol on to top of stack
  2. Reduce: right end of string to be reduced with head of production, which matches the body of that production.
  3. Accept: Announce successful completion of string.
  4. Error: Discover syntax error and call error recovery method.

Example: consider grammar

E → E + T | T

T → T * F | F

F → (E) | id

then check the acceptance of id*id with shift-reduce parser.

Stack	Input	Action
$	$id_1 * id_2$ $	shift
$ id_1	$* id_2$ $	reduce F → id
$ F	$* id_2$ $	reduce T → F
$ T	$* id_2$ $	shift
$ T *	id_2 $	shift
$ T * id_2	$	reduce F → id
$ T * F	$	reduce T → T * F
$ T	$	reduce E → T
$ E	$	accept

Conflicts during shift reduce parsing:

- In shift reduce parsing; parser cannot decide which action to be taken. this situation is called conflict.
- Two types of conflicts should be possible in shift reduce parser.
  1. shift/reduce conflict
  2. reduce/reduce conflict

Shift or reduce conflict:

- Parser cannot decide whether to use shift or reduce conflict will occur.

stack	input
$ E+T	*id$

- To solve the above problem, we take actions based on operator precedence and associativity.

Reduce/Reduce conflict:

- Parser cannot decide which one of several reductions will use, then reduce/reduce conflict will occur.

stack	input
$ E+T*F	$

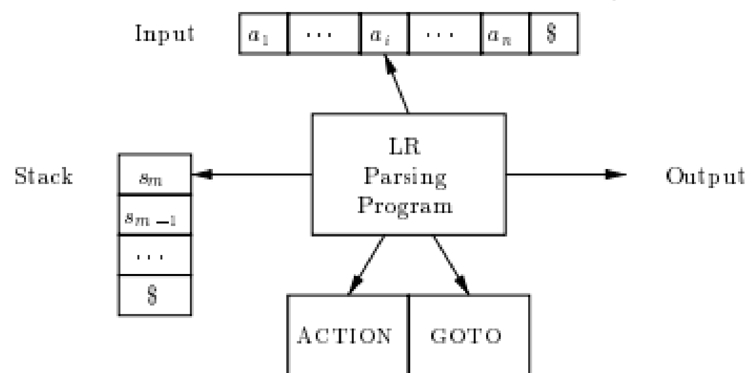
- To solve the above problem, we will take action based on rightmost elements of stack should reduce first.
- These conflicts will encountered for those grammars which are not LR or those grammars are ambiguous.

LR Parsers:

- Bottom-up syntax analysis technique that can be used to parse a large class of context free grammar is called LR(K) Parsing or LR parser.
- The 'L' is left to right scanning of the input, 'R' is constructing rightmost derivation in reverse and 'K' is the number of input symbols of lookahead that are used in making paring decisions.
- The principal drawback of the method is too much work to construct LR Parser by hand for a typical programming language grammar. We present 3 techniques for constructing an LR parsing table for a grammar.
- The first method called Simple LR (SLR), is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other method succeeds.
- The second method, called Canonical LR is the most powerful and the most expensive.
- The third method called Lookahead LR (LALR), is the intermediate in power and cost between the other

SLR ≤ LALR≤CLR

- The LR Parser consist of a stack, an input buffer, an output stream, a driven program and parsing table consists of two columns that are 'action' and 'goto'.



- The program driving the LR Parser behaves as follows, it determine s_m, state currently on top of the stack, and a_i, the current input symbol. It then consults action [s_m,a_i], the parsing action table entry in state sm and input ai, which can have one of four values.
  1. shift s, where s is state.
  2. reduce by a grammar production A→β.
  3. accept,
  4. error.
- The function goto takes a state and grammar symbol as arguments and produces a state.

SLR Parser (Simple LR):

- The grammar for which an SLR Parser can be constructed is said to be SLR grammar. The other 2 methods argument the SLR method with lookahead information.
- SLR method is a good starting point for studying LR Parser. SLR grammar is called LR(0) grammar. Here '0' indicates no lookahead.

Augmented Grammar:

- If G is a grammar with start symbol S then G', the augmented grammar for G, is G with a new start symbol S' and production S'→S.

- The purpose of new starting production is to indicate to the parser where is should stop parsing and announce acceptance of the input. It happen where the parser is about to reduce by S'→S.
- In SLR grammar G is production of G with a dot at some portion of right side. These items are called LR(0) items

A→.XYZ
A→X.YZ
A→XY.Z
A→XYZ.

Closure Operation:

- If 'I' is 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→.α is production then add item B→.α to I, if it is not already there and apply rule for no more new item can be added to closure (I).

Example: Construct SLR parsing table for the following grammar

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

Procedure: The given grammar G:

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

Augmented grammar G':

1. E' → E
2. E → E + T
3. E → T
4. T → T * F
5. T → F
6. F → (E)
7. F → id

States:

I0 : E' →. E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I0 , E)
I1 : E' → E .
E → E . + T

GOTO (I0 , T)
I2 : E → T .
T →T . * F

GOTO (I0 , F)
I3 : T →F .

GOTO (I0 , ( )
I4 : F →(. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I0 , id)
I5 : F → id .

GOTO (I1 , +)
I6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I2 , *)
I7 : T →T * . F
F → . (E)
F → . id

GOTO (I4 , E)
I8 : F →(E .)
E →E . + T

GOTO (I6 , T)
I9 : E →E + T .
T →T . * F

GOTO (I7 , F)
I10 : T → T * F .

GOTO (I8 ,))
I11 : F →(E) .

SLR Parsing Table:

	ACTION						GOTO		
	Id	+	*	(	)	$	E	T	F
I0	s5			s4			1	2	3
I1		s6				accept			
I2		r2	s7		r2	r2			
I3		r4	r4		r4	r4			
I4	s5			s4			8	2	3
I5		r6	r6		r6	r6			
I6	s5			s4				9	3
I7	s5			s4					10
I8		s6			s11				
I9		r1	s7		r1	r1			
I10		r3	r3		r3	r3			
I11		r5	r5		r5	r5			

Check the String acceptance of Id * id + id

	STACK	SYMBOLS	INPUT	ACTION
1	0		Id * id + id $	shift
2	5	Id	* id + id $	reduce by F → id
3	3	F	* id + id $	reduce by T → F
4	2	T	* id + id $	shift
5	27	T *	Id + id $	shift
6	275	T * id	+ id $	reduce by F → id
7	2710	T * F	+ id $	reduce by T → T *F
8	2	T	+ id $	reduce by E → T
9	1	E	+ id $	shift
10	16	E +	Id $	shift
11	165	E + id	$	reduce by F → id
12	163	E + F	$	reduce by T → F
13	169	E + T	$	reduce by E → E+T
14	1	E	$	accept

CLR Parser (Canonical LR):

- CLR Parser is an LR parser that uses a lookahead symbol during parsing. Items in CLR parsing are of the form

  A → .αBβ,a
  A → α.Bβ,a
  A → αB.β,a
  A → . αBβ.,a

- Here A → αBβ is a production and 'a' is a lookahead symbol. CLR items are also known as LR(1) items. Where 1 indicates the number of lookaheads. CLR grammar is also Called as LR(1) grammar

Example: construct a CLR parsing table for the grammar S→CC C→cC C→c/d

Procedure:

Given Grammar G:

1. S → CC
2. C → cC
3. C → d

Augmented grammar G' of G:

1. S' → S
2. S → CC
3. C → cC
4. C → d

To find lookahead:

Take S' → .S, $ because S' is the starting symbol of the augmented grammar

Rule finding Lookahead of B:

A → α.Bβ,a
Lookahead of B = first(βa)
Then production is
B→ .ϒ, first(βa)

Lookahead of S:

S’ → .S, $

Lookahead of S = first(ε$)=$

Then production is

S → .CC, $

Lookahead of C:

S → .CC, $

Lookahead of S = first(C$)={c,d}

Then production is

C → .cC, c/d

C → .d, c/d

States:

I0 : S’ → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c /d

GOTO (I0 , S)
I1: S’ → S., $

GOTO (I0 , C)
I2: S → C.C, $
C → .Cc, $
C → .d, $

GOTO (I0 , c)
I3: C → c.C, c/d
C → .Cc, c/d
C → .d, c/d

GOTO (I0 , d)
I4: C → d., c/d

GOTO (I2 , C)
I5: S → CC., $

GOTO (I2 , c)
I6: C → c.C, $
C → .cC, $
C → .d, $

GOTO (I2 , d)
I7: C → d., $

GOTO (I3 , C)
I8: C → cC., c/d

GOTO (I6 , C)
I9: C → cC., $

Canonical Parsing Table:

STATE	Actions			Goto	
	C	d	$	S	C
0	s3	s4		1	2
1			acc		
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 (lookahead LR):

- LALR method is based on the LR(0)sets of items, and has many fewer states than typical parsers based on the LR(1) items. By carefully introducing lookaheads into the LR(0) items.
- we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables. LALR is the method of choice in most situations.

Example: construct a LALR parsing table for the grammar S→CC C→cC C→c/d
Procedure:

Given Grammar G:

1. S → CC
2. C → cC
3. C → d

Augmented grammar G' of G:

1. S' → S
2. S → CC
3. C → cC
4. C → d

To find lookahead:

Take S' → .S, $ because S' is the starting symbol of the augmented grammar

Rule finding Lookahead of B:

A → α.Bβ,a

Lookahead of B = first(βa)

Then production is

B→ .ϒ, first(βa)

Lookahead of S:

S' → .S, $

Lookahead of S = first(ε$)=$

Then production is

S → .CC, $

Lookahead of C:

S → .CC, $

Lookahead of S = first(C$)={c,d}

Then production is

C → .cC, c/d
C → .d, c/d

States:

I0 : S' → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c /d

GOTO (I0 , S)
I1: S' → S., $

GOTO (I0 , C)
I2: S → C.C, $
C → .Cc, $
C → .d, $

GOTO (I0 , c)
I3: C → c.C, c/d
C → .Cc, c/d
C → .d, c/d

GOTO (I0 , d)
I4: C → d., c/d

GOTO (I2 , C)
I5: S → CC., $

GOTO (I2 , c)
I6: C → c.C, $
C → .cC, $
C → .d, $

GOTO (I2 , d)
I7: C → d., $

GOTO (I3 , C)
I8: C → cC., c/d

GOTO (I6 , C)
I9: C → cC., $

CLR Parsing Table:

STATE	Actions			Goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- By observing above states I3 and I6 are similar except the lookahead. so merge these 2 states and form a new state I36 . in the same manner
  I4 and I7 merge and form a new state I47
  I8 and I9 merge and form a new state I89
- After merging the states the computed CLR or LALR parsing table is

LALR Parsing Table:

START	Actions			Goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Using Ambiguous Grammars:

- Construct SLR paring table for given Ambiguous Grammar and solve the conflict occur in the parsing table by precedence and associativity then construct final parsing table.

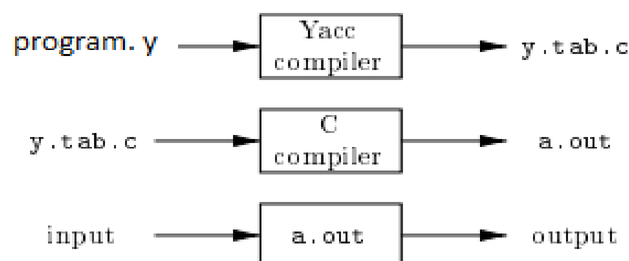
  $E \rightarrow E + E \ / \ E*E \ / \ (E) \ / \ id$

Parser Generator:

- Various tools are used for Parser Generators to describe syntax of given expression.
- A tool called Yacc used for construction of Parser Generators. Yacc stands for yet another compiler-compiler; basically it is a unix utility.

Use of Yacc:

- In yacc tool we write input file with program.y then it forwarded to yacc compiler.
- Yacc compiler transforms program.y to C program file y.tab.c. later this file is compiled by C compiler into a file called a.out



Structure of Yacc program:

Declarations
%%
Translation rules
%%
Supporting C routines

Example: a simple desk calculator program for the given grammar is

E→E+T/ T   T→T *F/F   F→ (E)/digit

Program:

```
%{
#include<ctype.h>
%}
%tokenDIGIT

%%
line     : expr'\n'          {printf("%d\n",$1);}
         ;
expr     : expr'+'term       {$$=$1+$3;}
         | term
         ;
term     : term'*'factor     {$$=$1*$3;}
         | factor
         ;
f actor  : '('expr')'        {$$=$2;}
         | DIGIT
         ;

%%
yylex(){
    intc;
    c = getchar();
    if(isdigit(c)){
        yylval=c-'0';
        return DIGIT;
    }
    return c;
}
```

Declarations Part:

- There are two sections in the declarations part of a Yacc program; both are optional. In first section, we put ordinary C declarations, delimited by %{ and %}.

#include<ctype.h>

- it causes the C preprocessor to include the standard header file <ctype.h> that contains the predicate isdigit.

%token DIGIT

- Declares DIGIT to be a token. Tokens declared in this section can then be used in the second and third parts of the Yacc specification.

Translation Rules Part:

- In the part of the Yacc specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing:

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

- A Yacc semantic action is a sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the non terminal of the head, while $i refers to the value associated with the ith grammar symbol (terminal or non terminal) of the body.
- The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for $$ in terms of the $i's. In the Yacc specification, we have written the two E-productions

$$E \rightarrow E+T \ / \ T$$

and their associated semantic actions as:

```
expr: expr'+'term             {$$=$1+$3;}
     |term
     ;
```

- Note that the non terminal term in the first production is the third grammar symbol of the body, while + is the second. We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol in the body. In general,{$$=$1;}is the default semantic action.
- Notice that we have added a new starting production to the Yacc specification

```
line: expr'\n'          {printf("%d\n",$1);}
```

- This production says that an input to the desk calculator is to be an expression followed by a newline character. The semantic action associated with this production prints the decimal value of the expression followed by a new line character.

Supporting C-Routines Part:

- The third part of a Yacc specification consists of supporting C-routines. A lexical analyzer by the name yylex() must be provided.
- Using Lex to produce yylex() is a common choice. Other procedures such as error recovery routines may be added as necessary.
- The lexical analyzer yylex() produces tokens consisting of a token name and its associated attribute value. The lexical analyzer reads input characters one at a time using the C-function getchar().
- If the character is a digit, the value of the digit is stored in the variable yylval, and the token name DIGIT is returned.

Using Yacc with Ambiguous Grammars:

- Let us now modify the Yacc specification so that the resulting desk calculator becomes more useful. First, we shall allow the desk calculator to evaluate a sequence of expressions, one to a line. We shall also allow blank lines between expressions

```
lines: lines expr'\n'          {printf("%g\n",$2);}
     |lines'\n'
     |/*empty*/
     ;
```

- In Yacc, an empty alternative, as the third line is, denotes. Second, we shall enlarge the class of expressions to include numbers with a decimal point instead of single digits and to include the arithmetic operators +, -,* and /.
- The easiest way to specify this class of expressions is to use the ambiguous grammar

$$E \rightarrow E + E \ / \ E-E \ / \ E*E \ / \ E/E \ / \ (E) \ / \ number$$

The resultant Yacc Specification is

```
%{
#include<ctype.h>#include<stdio.h>
#defineYYSTYPEdouble          /*doubletypeforYaccstack*/
%}
%token NUMBER
%left '+''-'
%left '*''/'

%%
lines: lines expr'\n'     {printf("%g\n",$2);}
     | lines'\n'
     |/*empty*/
     ;
expr : expr'+'expr        {$$=$1+$3;}
     | expr'-'expr        {$$=$1-$3;}
     | expr'*'expr        {$$=$1*$3;}
     | expr'+'expr        {$$=$1/$3;}
     | '('expr')'         {$$=$2;}
     | NUMBER
     ;

%%
yylex(){
int c;
while((c=getchar())=='');
 if ((c =='.') || (isdigit(c)) )
  {
    ungetc(c,stdin);
    scanf("%lf",&yylval);
    return NUMBER;
   }
  return c;
}
```

- Unless otherwise instructed Yacc will resolve all parsing action conflicts using the following two rules:
    1. A reduce/reduce conflict is resolved by choosing the conflicting production listed first in the Yacc specification.
    2. A shift/reduce conflict is resolved in favour of shift. This rule resolves the shift/reduce conflict arising from the dangling-else ambiguity correctly.
- Since these default rules may not always be what the compiler writer wants, Yacc provides a general mechanism for resolving shift/reduce conflicts.
- In the declarations portion, we can assign precedence and associativity to terminals. The declaration %left '+''-' makes + and - be of the same precedence and be left associative.