4.1 Introduction:

The Role of the Parser:

- The syntax analyzer obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.
- i.e., *Syntax Analyzer* creates the syntactic structure of the given source program. This syntactic structure is mostly a *parse tree*.
- Thus Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*.
- The syntax analyzer checks whether a given source program satisfies the rules implied by a context-free grammar or not.
    - ❖ If it satisfies, the parser creates the parse tree of that program.
    - ❖ Otherwise the parser gives the error messages.
- It then passes parse tree to the rest of the compiler for further processing
- A context-free grammar
    - ❖ Gives a precise, easy-to-understand, syntactic specification of a programming language.
    - ❖ Can be used effectively to construct an efficient parser that determines the syntactic structure of a source program. The parser-construction process can reveal syntactic ambiguities and trouble spots that might have noticed in the initial design phase of a language.
    - ❖ Useful for translating source programs into correct object code and for detecting errors.
    - ❖ Allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.

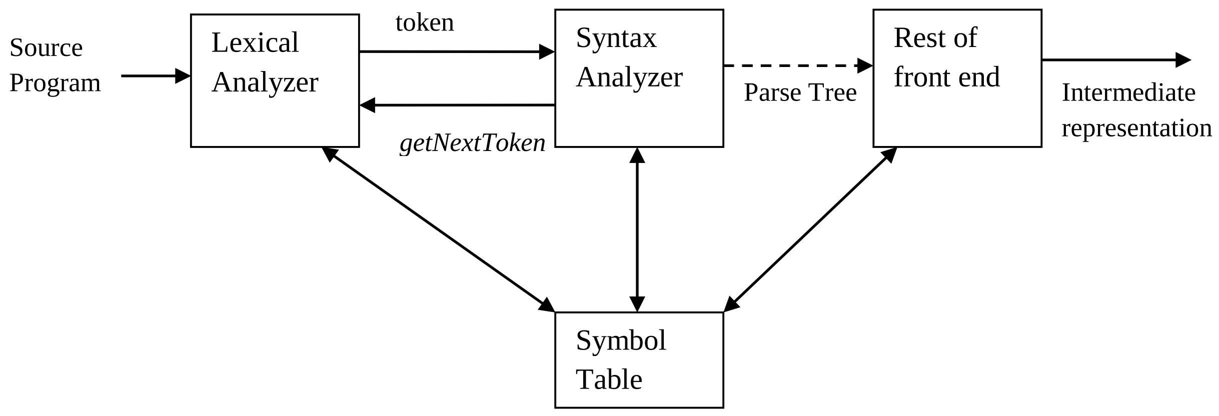


Fig :Position of Parser in Compiler model

- There are three general types of parsers for grammars:
    1) ***Universal:*** Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar . however, too inefficient to use in production compilers.
    2) ***top-down***: build parse trees from the top (root) to the bottom (leaves)

3) ***bottom-up:*** Build parse tree from leaves and work their way up to the root. We categorize the parsers into two groups:

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for subclasses of CFG's:
  - ❖ LL grammars for top-down parsing
  - ❖ LR grammars for bottom-up parsing

Representative grammars:

- The Expression Grammar used for top-down parsing:
  $E \rightarrow TE'$
  $E' \rightarrow +TE' \mid e$
  $T \rightarrow FT'$
  $T' \rightarrow *FT' \mid e$
  $F \rightarrow (E) \mid id$
- The expression grammar used for bottom-up parsing:
  $E \rightarrow E+T \mid T$
  $T \rightarrow T*F \mid F$
  $F \rightarrow (E) \mid id$

Syntax Error Handling:

- Common Programming errors can occur at many different levels:
  1. ***Lexical errors***: include misspelling of identifiers, keywords, or operators.
  2. ***Syntactic errors***: include misplaced semicolons or extra or missing braces.
  3. ***Semantic errors***: include type mismatches between operators and operands.
  4. ***Logical errors***:can be anything from incorrect reasoning on the part of the programmer.

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

Error-Recovery Strategies:

1. Panic-Mode Recovery:

- o In this method, on discovering an error, the parser ***discards input symbols*** one at a time until one of a designated set of ***Synchronizing tokens*** is found.
- o Synchronizing tokens are usually delimiters.
  Ex: ***}*** or ; whose role in the source program is clear and unambiguous.
- o *Advantage:*
  - ➢ Simple method
  - ➢ Is guaranteed not to go into an infinite loop

- *Disadvantage:*
  - It often skips a considerable amount of input without checking it for additional errors.
  - Careful selection of synchronizing tokens

2. **Phrase-Level Recovery:**
   - In this method, A parser may ***perform local correction*** on the remaining input. i.e it may replace a prefix of the remaining input by some string that allows the parser to continue.
   - Ex: replace a comma by a semicolon, insert a missing semicolon
   - *Advantage:*
     - It is used in several error-repairing compliers, as it can correct any input string.
   - *Disadvantage:*
     - Difficulty in coping with the situations in which the actual error has occurred before the point of detection.
     - This method is not guaranteed to not to go into an infinite loop.

3. **Error Productions:**
   - Augment the grammar for the language with productions that would generate the erroneous constructs.
   - Then use this grammar augmented by the error productions to construct a parser.
   - If an error production is used by the parser, we can generate appropriate **error diagnostics** to indicate the erroneous construct that has been recognized in the input.

4. **Global Correction:**
   - We use algorithms that perform minimal sequence of changes to obtain a globally least cost correction.
   - Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.
   - It is too costly to implement in terms of time space, so these techniques only of theoretical interest.

4.2 Context-Free Grammars:

- Grammars describe the syntax of programming language constructs like expression, statements.
- A CFG is defined as G= (V,T,P,S) where
  V is the finite set of non-terminals (variables)
  T is the finite set of terminals (tokens)
  P is the finite set of productions rules in the following form
  $A \rightarrow \alpha$ where
  A is a non-terminal and
  α is a string of terminals and non-terminals (including the empty string)
  S is the start symbol (one of the non-terminal symbol)

Notational conventions:

1. Symbols used for terminals are :
   a. Lower case letters early in the alphabet (such as a, b, c, . . .)

b. Operator symbols (such as +, *, . . .)
c. Punctuation symbols (such as parenthesis, comma and so on)
d. The digits(0…9)
e. Boldface strings and keywords (such as **id** or **if**) each of which represents a single terminal symbol

2. Symbols used for non terminals are:
   a. Uppercase letters early in the alphabet (such as A, B, C, …)
   b. The letter S, which when it appears is usually the start symbol.
   c. Lowercase, italic names (such as *expr* or *stmt).*

3. Lower case greek letters such as α, β, γ represent (possibly empty) strings of grammar symbols.
4. X, Y, Z represent grammar symbols(Terminal or Nonterminal)
5. u,v,…,z represent strings of terminals.
6. A $\rightarrow \alpha_1$, A $\rightarrow \alpha_2$, A $\rightarrow \alpha_3$ can be written as A $\rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$

Ex: The following grammar defines the arithmetic expression

expression **->** *expression + term*
expression -> expression - term
expression **->** *term*
term **->** *term * factor*
term **->** *term / factor*
term **->** *factor*
factor-> (expression)
factor **->** ***id***

Using the conventions listed above, the above grammar can be written as,

E -> E+T | E-T | T
*T -> T*F | T/F | F*
F -> (E) | id

Derivations:

- Consider the following grammar,
  E -> E+E | E*E | -E | (E) | id

- A sequence of replacements of non-terminal symbols by its production body is called as **derivation**
  Ex: E **=>** E+E => id+E => id+id

- In general, a derivation step is **αAβ => αγβ** where A $\rightarrow$ γ is a production

- Since in the above example, multiple derivation steps do exist, it can also be written as E=> id+id

- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**(LMD)
  Ex: E **=>** E+E
  => id+E
  => id+id
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.(RMD)
  Ex: E **=>** E+E
  => E+id
  => id+id

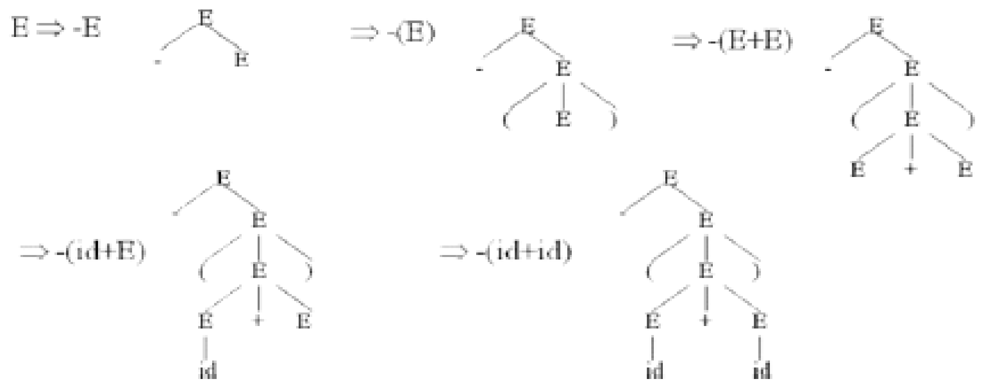
- If *S* =>**α**, where *S* is the start symbol of a grammar *G*, we say that **α** is a **sentential form** of G. A sentential form may contain both terminals and nonterminals, and may be empty.
  Eg: In the above example, the sentential forms are E+E and E+id.

- The sentential forms obtained in a LMD are said to be as **left sentential form** whereas the sentential forms obtained in a RMD are called as **right sentential form**.

- A **sentence** of *G* is a sentential form with no nonterminals.
  Eg: In the above example, sentence is id+id
- The ***language generated*** by a grammar, *L(G)* is its set of sentences.
  Thus, a string of terminals *w* is in *L(G),* if and only if *w* is a sentence of *G* (or *S* => *w).*
- If G is a context-free grammar, L(G) is a **context-free language.**
- Two grammars are **equivalent** if they produce the same language.

<u>**Parse Trees and derivations:**</u>

- Root node has the Start variable
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- The leaves of a parse tree when read from left to right constitute a sentential form, called **yield** or **frontier** of the tree.
- A parse tree can be seen as a graphical representation of a derivation.
- Ex: Parse tree construction for the string **–(id+id)** is shown below along with the derivation



Problems:

1. Consider the grammar S → (L) | a
   L→ L , S | S
   - i. What are the terminals, non terminal and the start symbol?
   - ii. Construct parse tree for the following sentence
     - a. (a , a)
     - b. (a, (a, a))
     - c. (a, ((a, a) , (a , a)))
     - d. ((a, a) , a , (a))
   - iii. Obtain LMD and RMD for each.

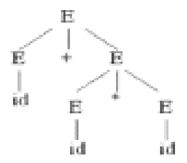
2. Do the above steps for the following grammars:

a)	S → aS | aSbS | ε	for the string aaabaab
b)	S → S S + | S S * | a	for the string aa+a*
c)	S → 0 S 1 | 0 1	with string 000111.
d)	S → + SS | * S S | a	with string + * aaa.
e)	S → S (S) S | ε	with string (()()).
f)	S →S + S | S S | (S) | S *| a	with string (a + a) * a.
g)	S → a S b S | b S a S | ε	with string aabbab.

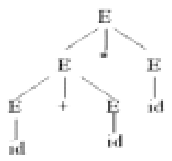
Ambiguity:

- A grammar that produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.
- Ex:

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



- For the most parsers, the grammar must be unambiguous.
- i.e.,We should eliminate the ambiguity in the grammar during the design phase of the compiler.

Verifying the Language Generated by a Grammar

Although compiler designers rarely do so for a complete programming-language grammar, it is useful to be able to reason that a given set of productions generates a particular language. Troublesome constructs can be studied by writing a concise, abstract grammar and studying the language that it generates.
We shall construct such a grammar for conditional statements below.
A proof that a grammar *G* generates a language *L* has two parts: show that every string generated by *G* is in *L*, and conversely that every string in *L* can indeed be generated by *G*.

Context-Free Grammars versus Regular Expressions
Grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa.
Alternatively, every regular language is a context-free language, but not vice-versa.

4.3 Writing a Grammar

Grammars are capable of describing most of the syntax of programming languages. The sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Lexical Versus Syntactic Analysis
"Why do we use regular expressions to define the lexical syntax of a language?" There are several reasons.

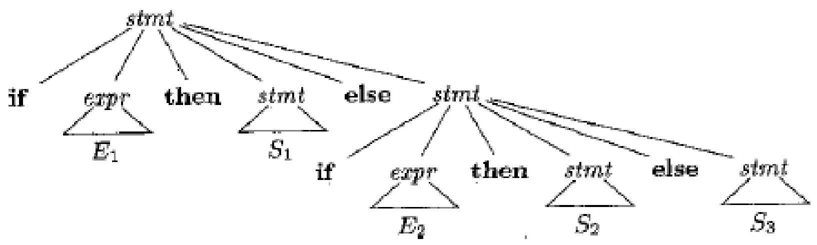
1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

- Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space.
- Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

4.3.2 Eliminating Ambiguity
An ambiguous grammar can be rewritten to eliminate the ambiguity.
As an example, we shall eliminate the ambiguity from the following "dangling else" grammar:

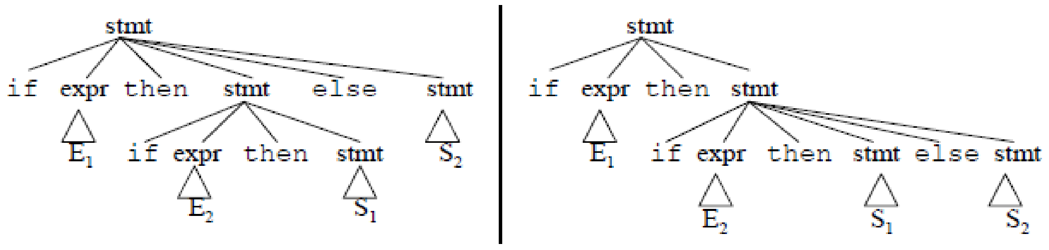
stmt —> **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

Here **"other"** stands for any other statement. According to this grammar, the compound conditional statement " **if** E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3" has the following parse tree:



However, the Grammar is ambiguous since the string

if E_1 **then if** E_2 **then** S_1 **else** S_2 has the following two parse trees.



In all programming languages with conditional statements of this form, the second parse tree is preferred. The general rule is, "Match each **else** with the closest unmatched **then."**

We can rewrite the above dangling-else grammar as the following unambiguous grammar.

- The idea is that a statement appearing between a **then** and an **else** must be "matched"; that is, the interior statement must not end with an unmatched or open **then.**
- A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement.
- Thus, we may use the following grammar , that allows only one parsing for string; namely, the one that associates each **else** with the closest previous unmatched **then.**

stmt -> *matchedstmt* | *openstmt*
matchedstmt -> **if** *expr* **then** *matchedstmt* **else** *matchedstmt* **| other**
openstmt -> **if** *expr* **then** *stmt* **| if** *expr* **then** *matchedstmt* **else** *openstmt*

Ambiguity – Operator Precedence

Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

E -> E+E | E*E | E^E | id | (E)

disambiguate the grammar

precedence: ^ (right to left)
* (left to right)
+ (left to right)

E -> E+T | T
T -> T*F | F
F -> G^F | G
G -> id | (E)

Elimination of Left Recursion

- A grammar is *left recursive* if it has a nonterminal *A* such that there is a derivation
  $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- ***immediate left recursion****:* If there is a production of the form $\boldsymbol{A \text{ -> } A\alpha \mid \beta}$ then it could be replaced by the non-left-recursive productions:
  $\boldsymbol{A \text{ -> } \beta A'}$
  $\boldsymbol{A' \text{->} \alpha A' \mid \varepsilon}$

  Example: Consider the expression grammar,
  E → **E+T** | T
  T → **T*F** | F
  F → (E) | id

  The non-left-recursive expression grammar is
  E -> *T E'*
  E' -> + *T E'* | **ε**
  T -> *FT'*
  T' -> * *F T'* | **ε**
  F -> *(E)* | id

  Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions.

  First, group the productions as
  A -> $A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$ where no β_i begins with an *A*.

  Then, replace the A-productions by
  A -> $\beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
  A' -> $\alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

  This procedure eliminates all left recursion from the *A* and *A'* productions (provided no α_i is **ε**).

- But the above procedure does not eliminate left recursion involving derivations of two or more steps.
  For example, consider the grammar
  S -> Aa | b
  A -> Ac | Sd | ε

The nonterminal *S* is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

The following Algorithm below, systematically eliminates left recursion from a grammar. It is guaranteed to work if

- The grammar has no cycles (derivations of the form $A \overset{+}{\Rightarrow} A$)
- The grammar has no ε -productions (productions of the form *A* -> ε).

Algorithm: Eliminating left recursion.

INPUT: Grammar *G* with no cycles or ε-productions.
OUTPUT: An equivalent grammar with no left recursion.
METHOD**:** Apply the below algorithm to *G*. Note that the resulting non-left-recursive grammar may have ε -productions.

```
1)  Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2)  for ( each $i$ from 1 to $n$ ) {
3)      for ( each $j$ from 1 to $i$-1 ) {
4)          replace each production of the form $A_i \to A_j\gamma$ by the
            productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
            $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current $A_j$-productions
5)      }
6)      eliminate the immediate left recursion among the $A_i$ -productions
7)  }
```

Example:

Consider the grammar

S -> Aa | b
A -> Ac | Sd | ε

- We order the nonterminals *S, A.*
- For *i=1*, There is no immediate left recursion among the 5-productions, so nothing happens.
- For *i* = 2, we substitute for *S* in *A* -> *Sd* to obtain the following A-productions.

  A-> Ac | *Aad* | *bd* | ε

  Eliminating the immediate left recursion among these A-productions yields the following grammar.

  S -> *Aa* | *b*
  A -> *bdA'* | *A'*
  A' -> *cA'* | *adA'* | ε

<u>**Left Factoring**</u>

- Left factoring is a grammar transformation that is ***useful for producing a grammar suitable for predictive, or top-down, parsing***.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

  For example, if we have the two productions
  stmt **->** **if** *expr* **then** *stmt* **else** *stmt* **| if** *expr* **then** *stmt*

  on seeing the input **if,** we cannot immediately tell which production to choose to expand *stmt.*

- In general, if $\boldsymbol{A}$ **->** $\boldsymbol{\alpha\beta_1 \mid \alpha\beta_2}$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\beta_1\ or\ \alpha\beta_2$.
- However, we may defer the decision by left factoring, so that the original productions become
  A -> αA'
  A' -> $\beta_1 \mid \beta_2$

Algorithm: Left factoring a grammar.
INPUT: Grammar *G.*
OUTPUT: An equivalent left-factored grammar.
METHOD:

- For each nonterminal *A*, find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \varepsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions A -> $\alpha\beta_1 \mid \alpha\beta_2 \mid \bullet\bullet\bullet \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by
  A -> $\alpha A' \mid \gamma$
  A' -> $\beta_1 \mid \beta_2 \mid \bullet\bullet\bullet \mid \beta_n$
  Here A' is a new nonterminal.
- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example: Left factor the following grammar which abstracts the "dangling-else" problem:
S -> *iEtS* | *iEtSeS* | *a*
E -> *b*
Here, i, *t*, and e stand for if, then, and else; *E* and S stand for "conditional expression" and "statement."

Left-factored, this grammar becomes:
S -> i*EtSS'* | *a*
S' -> *eS* | ε
E -> b

<u>Non-Context-Free Language Constructs</u>

- A few syntactic constructs found in typical programming languages cannot be specified using grammars alone.

- Example 1:
    - The language in this example abstracts the problem of checking that *identifiers are declared before they are used in a program.*
    - The language consists of strings of the form *wcw*, where the first *w* represents the declaration of an identifier *w*, *c* represents an intervening program fragment, and the second *w* represents the use of the identifier.
    - The abstract language is *L= {wcw \ w* is in (a|b)*}.
    - *L* consists of all words composed of a repeated string of a's and b's separated by c, such as *aabcaab.*
    - *T*he noncontext-freedom of *L* directly implies the non-context-freedom of programming languages like C and Java, which require declaration of identifiers before their use and which allow identifiers of arbitrary length. For this reason, a grammar for C or Java does not distinguish among identifiers that are different character strings. Instead, all identifiers are represented by a token such as **id** in the grammar. In a compiler for such a language, the semantic-analysis phase checks that identifiers are declared before they are used.

- Example 2 :
    - The problem of checking that the *number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function*.
    - The language consists of strings of the form $a^nb^mc^nd^m$. Here a^n and b^m could represent the formal-parameter lists of two functions declared to have *n* and m arguments, respectively, while c^n and d^m represent the actual-parameter lists in calls to these two functions.
    - The abstract language is $L2 = \{ a^nb^mc^nd^m \mid n > 1$ and $m > 1\}$. That is, *L2* consists of strings in the language generated by the regular expression **a*b*c*d*** such that the number of a's and c's are equal and the number of b's and d's are equal.
    - This language is not context free.
    - The typical syntax of function declarations and uses does not concern itself with counting the number of parameters. For example, a function call in C-like language might be specified by

        Stmt **-> id (** *expr_list)*
        expr_list -> expr_list , expr | expr

      with suitable productions for *expr.* Checking that the number of parameters in a call is correct is usually done during the semantic-analysis phase.

<u>Excercises:</u>

For each of the following grammars,
a) Left factor the grammar.
b) In addition to left factoring, eliminate left recursion from the original grammar.

1) *rexpr* → *rexpr + rterm | rterm*
   rterm → *rterm rfactor \ rfactor*
   rfactor → *rfactor * | rprimary*
   rprimary → **a | b**

2) S → S S + | S S * | a

3) S → 0 S 1 | 0 1

4) S → (L) | a
   L→ L , S | S

5) *bexpr* → *bexpr* **or** *bterm* | *bterm*
   bterm → *bterm* **and** *bfactor* \ *bfactor*
   bfactor → **not** *bfactor* | (*bexpr*) | **true** | **false**

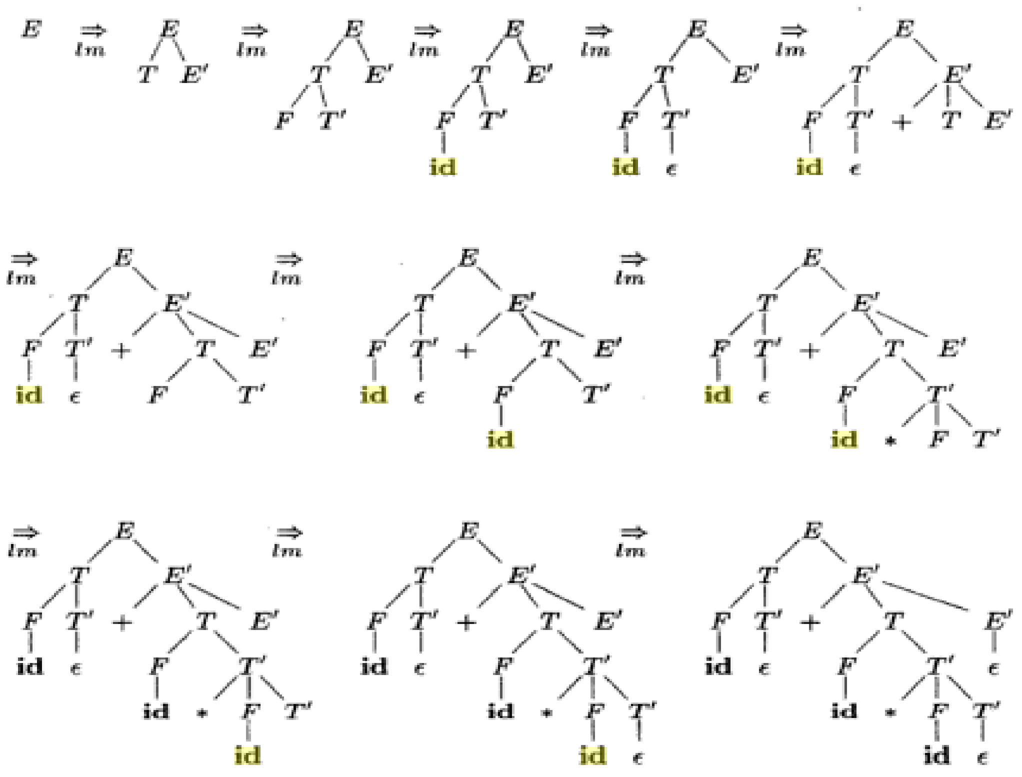
4.4 Top-Down Parsing:

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

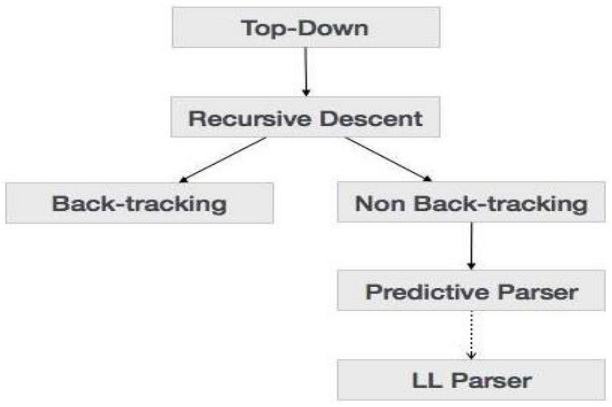
Example: Consider the grammar,

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

The sequence of parse trees for the input **id+id*id** actually corresponds leftmost derivation of the input(see below)



High level classification of Top-Down Parser:



Recursive-Descent Parsing:

```
   void A( ) {
1)     Choose an A-production, $A \to X_1 X_2 \bullet\bullet X_k$;
2)     for ( $i$ = 1 to $k$ ) {
3)         if ( $X_i$ is a nonterminal )
4)             call procedure $X_i( )$;
5)         else if ( $X_i$ equals the current input symbol $a$ )
6)             advance the input to the next symbol;
7)         else /* an error has occurred */;
       }
   }
```

Figure: A typical procedure for a nonterminal in a top-down parser

- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
- Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Pseudocode for a typical nonterminal is shown in the above figure.
- General recursive-descent may require **backtracking**; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.
- To allow backtracking, the above code needs to be modified.
    - First, we cannot choose a unique A-production at line (1), so we must try each of several Productions in some order.
    - Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production.
    - Only if there are no more A-productions to try do we declare that an input error has been found.
    - In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

    - **Example**: Consider the grammar

S -> cAd
A -> ab | a

- ✓ To construct a parse tree top-down for the input string *w = cad*, begin with a tree consisting of a single node labeled *S*, and the input pointer pointing to c, the first symbol *of w*.
- ✓ *S* has only one production, so we use it to expand *S* and obtain the tree of Fig. 4.14(a).
- ✓ The leftmost leaf, labeled c, matches the first symbol of input *w*, so we advance the input pointer to a, the second symbol of *w*, and consider the next leaf, labeled *A*.
- ✓ Now, we expand *A* using the first alternative *A -> ab* to obtain the tree of Fig. 4.14(b). We have a match for the second input symbol, *a*, so we advance the input pointer to d, the third input symbol, and compare *d* against the next leaf, labeled *b*.
- ✓ Since *b* does not match *d*, we report failure and go back to *A* to see whether there is another alternative for *A* that has not been tried, but that might produce a match.
- ✓ In going back to *A*, we must reset the input pointer to position 2, the position it had when we first came to *A*, which means that the procedure for *A* must store the input pointer in a local variable. The second alternative for *A* produces the tree of Fig. 4.14(c).
- ✓ The leaf *a* matches the second symbol of *w* and the leaf *d* matches the third symbol.
- ✓ Since we have produced a parse tree for *w*, we halt and announce successful completion of parsing.

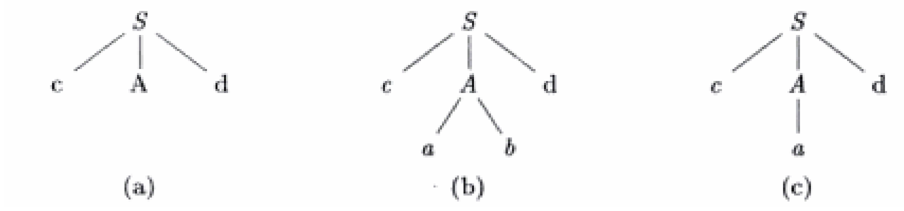


Figure 4.14 : Steps in a top-down parser

- ***A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop***. That is, when we try to expand a nonterminal *A*, we may eventually find ourselves again trying to expand *A* without having consumed any input.

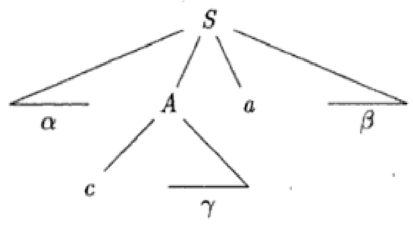
FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW,** associated with a 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 tokens.

- ***FIRST(α) is defined as*** **the set of terminals that begin strings derived from *α*. , where *α* is any string of grammar symbols. If *α* $\overset{*}{\Rightarrow}$ε, then ε is also in FIRST(α).**

- ***For nonterminal A, we define* FOLLOW(A) , to be the set of the terminals *a* which occur immediately after (follow) the *non-terminal A* in some sentential form;**

  That is, the set of terminals *a* such that there exists a derivation of the form $S\overset{*}{\Rightarrow}\alpha Aa\beta$, for some α and β.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

- 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 a terminal, then FIRST(X) = { X }.
  2. If *X* is a nonterminal and *X* -> $Y_1Y_2 \bullet\bullet\bullet Y_k$ is a production for some $k \geq 1$, then
     - ✓ place *a* in FIRST(X) if for some *i,*
       a is in FIRST(Y_i), and ε is in all of FlRST(Y_1), ... , FIRST(Y_{i-1});
       that is, $Y_1\bullet\bullet\bullet Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$.
     - ✓ If ε is in FIRST(Y_j) for all j = 1,2,... , *k,* then add ε to FIRST(X).
  3. If *X* -> ε is a production, then add ε to FIRST(X).

- Now, we can compute FIRST for any string $X_1X_2 \bullet\bullet\bullet X_n$ as follows.
  - ✓ Add to FlRST($X_1X_2 \bullet\bullet\bullet X_n$) all non-ε symbols of FIRST (X_1).
  - ✓ Also add the non- ε symbols of FIRST(X_2), if ε is in FlRST(X_1);
  - ✓ Add the non- ε symbols of FIRST(X_3), if ε is in FIRST(X_1) and FIRST(X_2); and so on.
  - ✓ Finally, add ε to FIRST($X_1X_2 \bullet\bullet\bullet X_n$) if, for all *i,* ε is in FIRST (X_i).

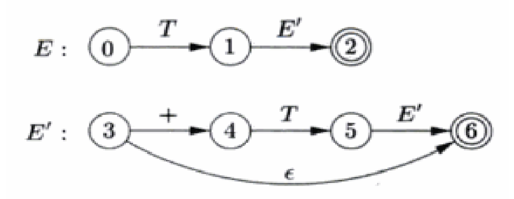
- To **compute FOLLOW(A)** for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.
  1. Place $ in FOLLOW(5), where *S* is the start symbol, and $ is the input right endmarker.
  2. If there is a production *A->αBβ,* then everything in FIRST(β) except ε is in FOLLOW(B).
  3. If there is a production *A->αB,* or a production *A* -> *αBβ,* where FIRST (β) contains ε, then everything in **FOLLOW**(A) is in **FOLLOW**(B) .

Transition Diagrams for Predictive Parsers

- Transition diagrams are useful for visualizing predictive parsers.
- To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar.
- Then, for each nonterminal *A,*
  1. Create an initial and final (return) state.

2. For each production $A \to X_1X_2 \ldots X_k$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots X_k$. , If $A \to \epsilon$, the path is an edge labeled ϵ.

- Transition diagrams for predictive parsers have one diagram for each nonterminal. The labels of edges can be tokens or nonterminals.
- A transition on a token (terminal) means that we take that transition if that token is the next input symbol.
- A transition on a nonterminal A is a call of the procedure for *A*.
- Example: Transition diagrams for non terminals E and E'



<u>LL(1) Grammars</u>

- Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).
- The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the " 1 " for using one input symbol of lookahead at each step to make parsing action decisions.
- No left-recursive or ambiguous grammar can be LL(1).
- A grammar *G* is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of *G*, the following conditions hold:
  1. For no terminal *a* do both α and $\boldsymbol{\beta}$ derive strings beginning with *a*.
  2. At most one of α and β can derive the empty string.
  3. If $\boldsymbol{\beta} \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW (A). Likewise, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then $\boldsymbol{\beta}$ does not derive any string beginning with a terminal in FOLLOW(A).

  The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets.

  The third condition is equivalent to stating that if ε is in FIRST(β), then FIRST (α) and FOLLOW(A) are disjoint sets, and likewise if ε is in FIRST*(α)*.Type equation here.

Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.

<u>Algorithm</u>: **<u>Construction of a predictive parsing table</u>**,
INPUT: Grammar *G*.
OUTPUT: Parsing table *M.A* two-dimensional array, *M[A,a]*, where A is a nonterminal, and *a* is a terminal or the symbol $, the input endmarker
METHOD: For each production A -> α of the grammar, do the following:

1. For each terminal *a* in FIRST(A), add A -> α to M[A, *a]*.
2. If ϵ is in FIRST(α), then for each terminal *b* in FOLLOW(A), add A -> α to *M[A,b]*.
   If ϵ is in FIRST(α) and $ is in FOLLOW(A), add A -> α to *M[A*, $] as well.

If, after performing the above, there is no production at all in *M[A,* a], then set *M[A, a]* to error (which we normally represent by an empty entry in the table).

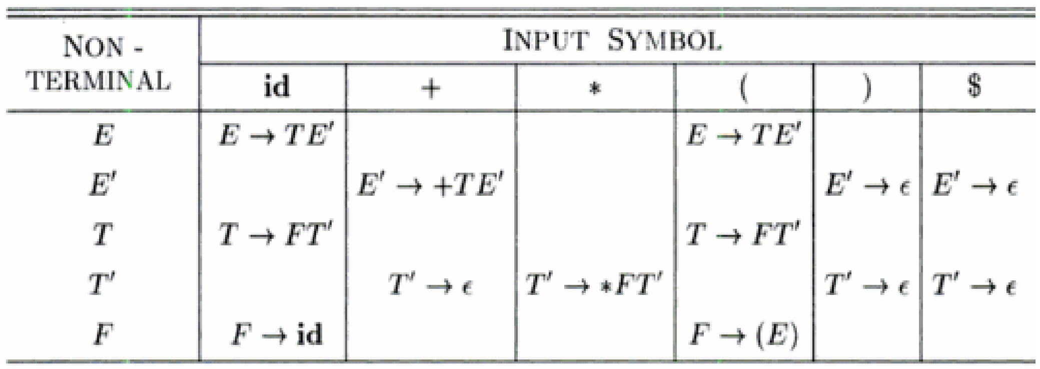
Example 1 : For the following expression grammar,

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

a) Compute FIRST & FOLLOW
b) Construct predictive parsing table

a) FIRST sets:
FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = { +, ε }
FIRST(T') = { *, ε }
FOLLOW sets:
FOLLOW (E) = {$,) }
FOLLOW (E') = {$,) }
FOLLOW (T) = {+,), $}
FOLLOW (T') = {+,), $}
FOLLOW (F) = {+, *,), $}

b) Parsing Table :



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

- The construction of predictive parsing table Algorithm can be applied to any grammar *G* to produce a parsing table M.
- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.
- For some grammars, however, *M* may have some entries that are multiply defined. For example, if *G* is left-recursive or ambiguous, then *M* will have at least one multiply defined entry.
- Although left recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an LL(1) grammar.

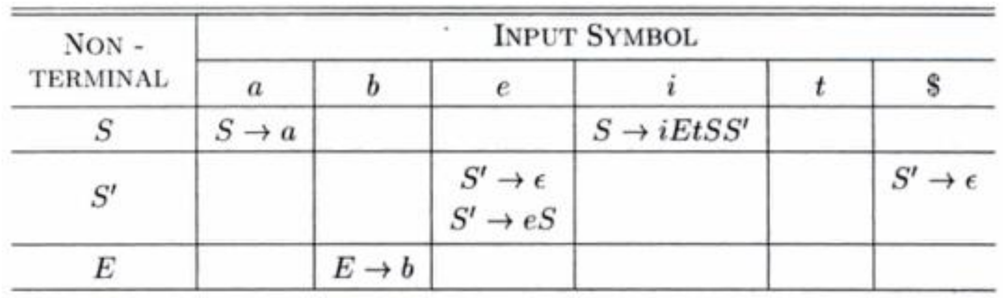
- **Example 2**: Show that the following grammar is not LL(1).

S -> iEtSS' | a
S' -> eS | ϵ
E -> b

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

FOLLOW(S) = {
FOLLOW(S') = {
FOLLOW(E) = {

The parsing table:



NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

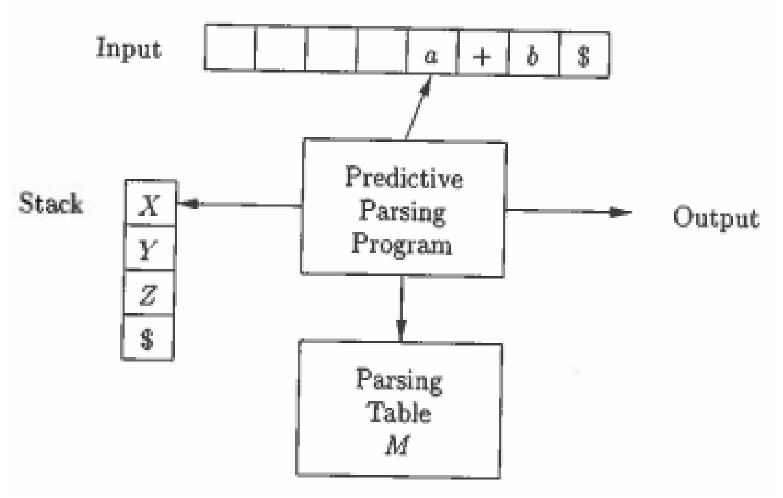
- The entry for *M[S', e]* contains both *S' —> eS* and *S' —>ϵ*.
- Therefore, <u>the grammar is not LL(1).</u>
- The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an *e* (else) is seen.
- We can resolve this ambiguity by choosing *S'->eS*. This choice corresponds to associating an **else** with the closest previous **then.**

<u>Nonrecursive Predictive Parsing:</u>

- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The parser mimics a leftmost derivation. If *w* is the input that has been matched so far, then the stack holds a sequence of grammar symbols *α* such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

- The following is a model of a table-driven predictive parser



- The parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm (*Construction of a predictive parsing table)* and an output stream.
- The input buffer contains the string to be parsed, followed by the endmarker $. We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.
- The parser is controlled by a program that considers *X,* the symbol on top of the stack, and *a,* the current input symbol.
  - ❖ If *X* is a nonterminal, the parser chooses an X-production by consulting entry *M[X, a]* of the parsing table *M.*
  - ❖ Otherwise, it checks for a match between the terminal *X* and current input symbol *a.*
- The behavior of the parser can be described in terms of its ***configurations,*** which give the stack contents and the remaining input.

Algorithm: Table-driven predictive parsing (Describes how *configurations* are manipulated)

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

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

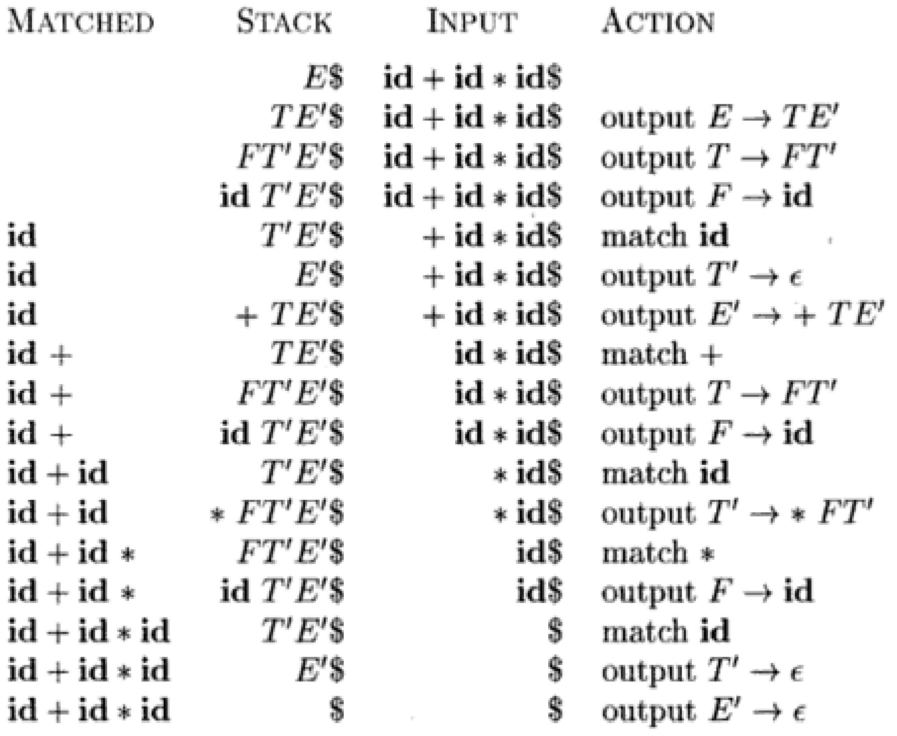
METHOD: Initially, the parser is in a configuration with *w$* in the input buffer and the start symbol *S* of *G* on top of the stack, above $. The program in above figure uses the predictive parsing table *M* to produce a predictive parse for the input.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a )
                pop the stack and advance ip;
        else if ( X is a terminal )
                error( );
        else if ( $M[X,a]$ is an error entry )
                $error( );$
        else if ( $M[X,a] = X \rightarrow Y_1Y_2 \ldots Y_k$ ) {
                output the production $X \rightarrow Y_1Y_2 \ldots Y_k$;
                pop the stack;
                push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with $Y_1$ on top;
        }
        set $X$ to the top stack symbol;
}
```

Example: Consider the Parsing Table constructed for the grammar

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

On input **id + id * id,** the nonrecursive predictive parser makes the sequence of moves as follows. These moves correspond to a leftmost derivation



MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

- The sentential forms in this derivation correspond to the input that has already been matched (in column **MATCHED)** followed by the stack contents.
- The matched input is shown only to highlight the correspondence. The input pointer points to the leftmost symbol of the string in the **INPUT** column.

Error Recovery in Predictive Parsing:

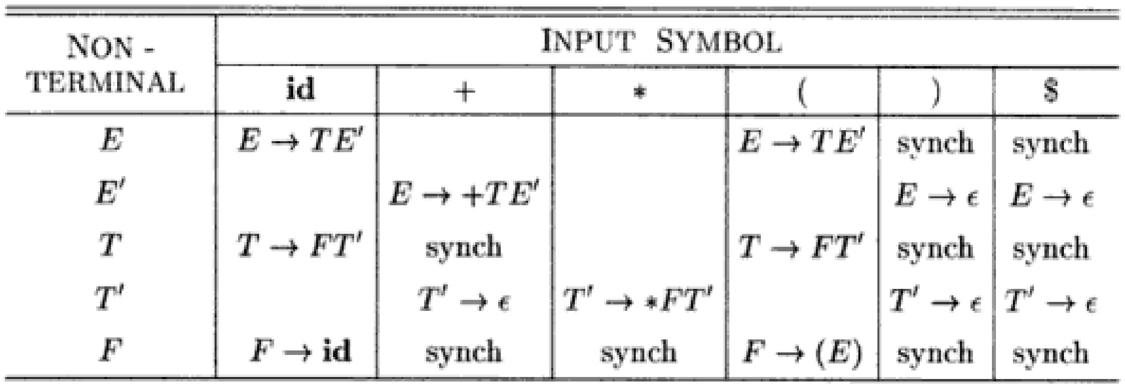
- An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal *A* is on top of the stack, *a* is the next input symbol, and *M[A, a]* is **error** (i.e., the parsing-table entry is empty).

- **Panic Mode**
  - Panic-mode error recovery is based on the idea of skipping symbols on the the input until a token in a selected set of synchronizing tokens appears.
  - Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.
  - Some heuristics are as follows:
    1. As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal *A.* If we skip tokens until an element of FOLLOW(A) is seen and pop *A* from the stack, it is likely that parsing can continue.
    2. It is not enough to use FOLLOW(A) as the synchronizing set for *A.* For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; for example, expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower-level construct the symbols that begin higher-level constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.
    3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal *A,* then it may be possible to resume parsing according to A if a symbol in FIRST (A) appears in the input.
    4. If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
    5. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

<u>Example:</u> Using FIRST and FOLLOW symbols as synchronizing tokens works reasonably well when expressions are parsed according to grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

The parsing table for this grammar in is repeated with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the nonterminal

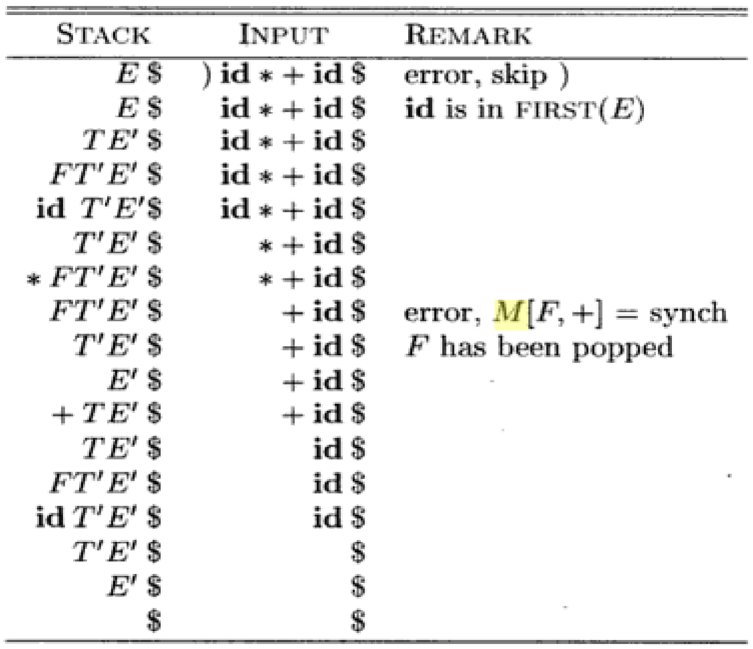


NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$	synch	synch	$F \to (E)$	synch	synch

The table is to be used as follows.

- If the parser looks up entry *M[A,a]* and finds that it is blank, then the input symbol *a* is skipped.
- If the entry is "synch," then the nonterminal on top of the stack is popped in an attempt to resume parsing.
- If a token on top of the stack does not match the input symbol, then we pop the token from the stack, as mentioned above.

On the erroneous input) **id * + id,** the parser and error recovery mechanism of the above table, behaves as follows:



STACK	INPUT	REMARK
$E\,\$$	$)\,\textbf{id} * + \textbf{id}\,\$$	error, skip)
$E\,\$$	$\textbf{id} * + \textbf{id}\,\$$	**id** is in FIRST(E)
$TE'\,\$$	$\textbf{id} * + \textbf{id}\,\$$	
$FT'E'\,\$$	$\textbf{id} * + \textbf{id}\,\$$	
$\textbf{id}\ T'E'\$$	$\textbf{id} * + \textbf{id}\,\$$	
$T'E'\,\$$	$* + \textbf{id}\,\$$	
$* FT'E'\,\$$	$* + \textbf{id}\,\$$	
$FT'E'\,\$$	$+ \textbf{id}\,\$$	error, $M[F,+]$ = synch
$T'E'\,\$$	$+ \textbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \textbf{id}\,\$$	
$+TE'\,\$$	$+ \textbf{id}\,\$$	
$TE'\,\$$	$\textbf{id}\,\$$	
$FT'E'\,\$$	$\textbf{id}\,\$$	
$\textbf{id}\,T'E'\,\$$	$\textbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

- **Phrase-level Recovery**
  - Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.
  - These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

- Alteration of stack symbols or the pushing of new symbols onto the stack is questionable for several reasons.
  - ❖ First, the steps carried out by the parser might then not correspond to the derivation of any word in the language at all.
  - ❖ Second, we must ensure that there is no possibility of an infinite loop. Checking that any recovery action eventually results in an input symbol being consumed (or the stack being shortened if the end of the input has been reached) is a good way to protect against such loops.

Construct the predictive parser LL (1) for the following grammar and parse the given string

1. S -> S(S)S | ϵ String= (( ) ( ))

2. S -> +SS | * SS | a String= +*aaa

3. S -> aSbS | bSaS | ϵ String=aabbbab

4. bexpr →bexpr **or** bterm | bterm
   bterm →bterm **and** bfactor \ bfactor
   bfactor →**not** bfactor | (bexpr) | **true** | **false** String= not (true or false)

5. S → 0S1 | 01 String=00011

6. S -> aB | aC | Sd | Se
   B -> bBc | f
   C -> g

7. P -> Ra | Qba
   R -> aba | caba | Rbc
   Q -> bbc | bc String= cababca

8. S -> PQR
   P -> a | Rb | ϵ
   Q -> c | dP | ϵ
   R -> e | f String= adeb

9. E -> E+T | T
   T -> id | id[ ] | id[X]
   X -> E,E | E String= id[id]

10. S -> (A) | 0
    A -> SB
    B -> ,SB | ϵ String= (0,(0,0))

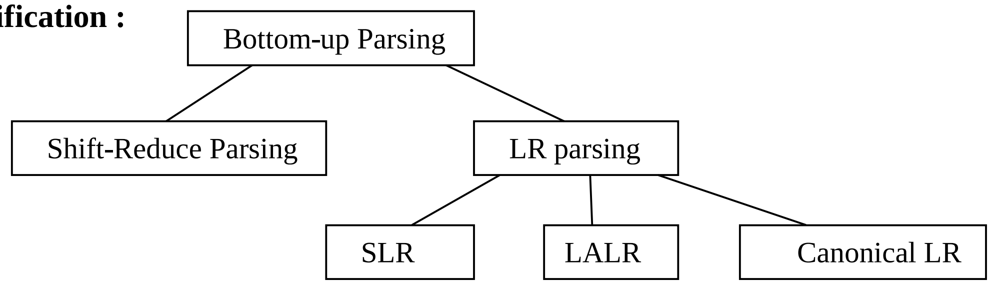
11. S -> a | ↑ | (T) String= (a,(a,a))
    T -> T,S | S = ((a,a),↑,(a),a)

<u>4.5 Bottom-Up Parsing:</u>

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- Ex:
  Consider the grammar,
  E → E+T | T
  T → T*F | F
  F → (E) | id
  Let the string be **id*id**
  The following illustrates construction of parse-tree using bottom-up parsing.

```
id * id      F * id      T * id      T * F        T          E
             |           |           |   |        |          |
             id          F           F   id     T * F        T
                         |           |          |   |        |
                         id          id         F   id     T * F
                                                |          |   |
                                                id         F   id
                                                           |
                                                           id
```

- **Classification :**



- **Reductions:**
  - ❖ We can think of bottom-up parsing as the process of "reducing" a string *w* to the start symbol of the grammar.
  - ❖ At each *reduction* step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.
  - ❖ The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.
  - ❖ Ex: sequence of reductions in the above example: **id * id,** F*** id ,** T***id ,** T*F, T, E
  - ❖ By definition, a ***reduction is the reverse of a step in a derivation***. The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in the above example.

    E => T
    => T * F
    => T *** id**
    => F *** id**
    => **id* id**

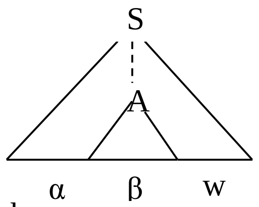
This derivation is in fact a rightmost derivation.

- ❖ Thus, ***Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.***

- **Handle Pruning:**
  - ❖ A "**handle**" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
  - ❖ For example, the handles during the parse of **id_1*id_2** according to the above grammar are as shown in the following:

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

- ❖ Formally, if $S \Rightarrow \alpha Aw \Rightarrow \alpha\beta w$, then the production $A \rightarrow \beta$ in the position following α is a *handle* of $\alpha\beta w$. i.e., a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.



- ❖ If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- ❖ A rightmost derivation in reverse can be obtained by "handle pruning."
  - o That is, we start with a string of terminals w to be parsed. If w is a sentence of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation,
    $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \ldots\ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$
  - o To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \rightarrow \beta_n$ to obtain the previous right-sentential form γ_{n-1}.
  - o We then repeat this process.
  - o If by continuing this process we produce a right-sentential form consisting only of the start symbol *S*, then we halt and announce successful completion of parsing.
  - o The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

- **Shift-Reduce Parsing:**
  - ❖ Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
  - ❖ We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right.

- ❖ Initially, the stack is empty, and the string *w* is on the input, as follows:

Stack	Input
$	w$

- ❖ During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

Stack	Input
$S	$

  Upon entering this configuration, the parser halts and announces successful completion of parsing.
- ❖ There are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.
  1. *Shift.* Shift the next input symbol onto the top of the stack.
  2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
  3. Accept. Announce successful completion of parsing.
  4. *Error.* Discover a syntax error and call an error recovery routine.
- ❖ The actions of a shift-reduce parser in parsing the input string id_1*id_2 according to the expression grammar is shown here:

Stack	**Input**	**Action**
$	$id_1*id_2$$	shift
$$id_1$	$*id_2$$	reduce by *F* -> id
$F	$*id_2$$	reduce by T -> F
$T	$*id_2$$	shift
$T*	$id_2$$	shift
$$T*id_2$	$	reduce by F -> id
$T*F	$	reduce by T -> T * F
$T	$	reduce by E -> T
$E	$	accept

Note: The handle will always eventually appear on top of the stack, never inside.

Question: Consider the following grammar and parse the respective strings using shift-reduce parser.

(1) S -> TL;
T -> int | float
L -> L, id | id
String : **int id, id;**

(2) S -> (L) | a
L -> L,S | S
String : **(a,(a,a))**

- **Conflicts During Shift-Reduce Parsing:**
  - ❖ There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot make certain decisions. Accordingly there are two types of conflicts:

1) *shift/reduce conflict:* This conflict arises when a parser can not decide whether to perform shift action or reduce action.
   Ex: Consider the grammar,
   stmt -> if *expr* **then** *stmt*
   | if *expr* **then** *stmt* **else** *stmt*
   | **other**

   If we have a shift-reduce parser in configuration

Stack	Input
• • • if *expr* then *stmt*	else • • $

   We cannot tell whether if *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the **else** on the input, it might be correct to reduce *if expr **then** stmt* to *stmt*, or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative *if expr **then** stmt **else** stmt.*

2) *reduce/reduce conflict*: This conflict arises when a parser cannot decide which of several reductions to make.
   Ex 1: Consider the following grammar,
   S -> AB
   A -> aA | ab
   B -> bB | ab
   Suppose the string is ***abab***
   Then the actions of a shift-reduce parser will be

Stack	**Input**	**Action**
$	abab$	shift
$a	bab$	shift
$ab	ab$	***reduce by A -> ab or B ->ab [conflict]***

   Here, parser will have a confusion as to which production to use for reduce action.

   Ex 2: Suppose we have a lexical analyzer that returns the token name *id* for all names, regardless of their type. Suppose also that our language invokes procedures by giving their names, with parameters surrounded by parentheses, and that arrays are referenced by the same syntax. Our grammar might therefore have (among others) productions such as shown below:

(1)	*stmt*	->	**id** (*parameterJist)*
(2)	*stmt*		expr := errpr
(3)	*parameter-list*		*parameterJist* , *parameter*
(4)	*parameter-list*	->	*parameter*
(5)	*parameter*		**id**
(6)	*expr*		**id** (*exprJist)*
(?)	*expr*		**id**
(8)	*exprJist*		*exprJist* , *expr*
(9)	*exprJist*	->	*expr*

A statement beginning with *p(i , j)* would appear as the token stream *id(id, id)* to the parser.
After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

<u>Stack</u>	<u>Input</u>
• • • id (id	**, id) • • •**

It is evident that the **id** on top of the stack must be reduced, but by which production? The correct choice is production (5) if p is a procedure, but production (7) if p is an array. The stack does not tell which; information in the symbol table obtained from the declaration of p must be used.

Exercises :
For the following grammars, indicate the handle in each of the following right-sentential forms:

1)S -> 0 S 1 | 0 1	a) 000111
	b) 00S11
2)S -> SS + | SS * | a	a) SSS+a*+
	b) SS+a*a+
	c) aaa*a++

<u>Introduction to LR Parsing: Simple LR(SLR)</u>

- The most of bottom-up parser today is based on a concept called LR(k) parsing;
  "L" is for left-to-right scanning of the input,
  "R" for constructing a rightmost derivation in reverse, and
  k for the number of input symbols of lookahead that are used in making parsing decisions.
  When *(k)* is omitted, *k* is assumed to be 1.

Why LR Parsers?

- For a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.
- LR parsing is attractive for a variety of reasons:
  1) LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.

2) The LR-parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.
3) An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
4) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. i.e.,LR grammars can describe more languages than LL grammars.

- The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar.
- A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available, Ex: YACC: These generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

Items and the LR(0) Automaton:

- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
- States represent sets of "items."
- An item indicates how much of a production we have seen at a given point in the parsing process.
- An ***LR(0)item*** *(item* for short) of a grammar *G* is a production of *G* with a dot at some position of the body. Thus, production *A*-> *XYZ* yields the four items

  A ->•XYZ
  A -> X•YZ
  A -> XY•Z
  A -> XYZ•

  The production *A* ->ε generates only one item, *A*->• .

  Item A ->•XYZ indicates that we hope to see a string derivable from *XYZ* next on the input.
  Item A -> X•YZ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.
  Item A -> XYZ• indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

LR(0) automaton:

- Canonical LR(0) collection provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an ***LR(0) automaton***.
- In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.
- To construct the canonical LR(0) collection for a grammar, we define the following:
  - ➢ An augmented grammar
  - ➢ CLOSURE and GOTO functions.

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 a production *S'* —> *S.*
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. i.e., acceptance occurs when and only when the parser is about to reduce by *S'* ->*S*

Closure of Item Sets:

- If *I* is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from *I* by the two rules:
  1.Initially, add every item in *I* to CLOSURE(I).
  2.If *A* -> *α•Bβ* is in CLOSURE(I) and *B* -> γ is a production, then add the item *B*->•γ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).
- The closure can be computed as :

```
SetOfltems CLOSURE (J)
{
        J = I;
        repeat
                for ( each item A -> α•Bβ  in J )
                        for ( each production B ->γ of G )
                                if ( B ->•γ is not in J )
                                        add B->•γ to J;
        until no more items are added to J on one round;
        return J;
}
```

- There are two classes of the sets of items :
  1. ***Kernel items***: The initial item, *S'* -> •*S*, and all items whose dots are not at the left end.
  2. ***Nonkernel items***: All items with their dots at the left end, except for *S'* -> •*S*

The Function GOTO:

- The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol.
- GOTO (I , X) is defined to be the closure of the set of all items [A -> αX•β] such that [A->α•Xβ] is in I.
- i.e., GOTO function is used to define the transitions in the LR(0) automaton for a grammar.
- The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input *X*.

With the above information, it is possible to construct *C*, the canonical collection of sets of LR(0) items for an augmented grammar *G'* — the algorithm is:

```
void items(G')
{
        C = CLOSURE({[S' -> •S]});
        repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO (I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Ex1: For the following grammar,

a) Compute GOTO function and construct LR(0) item sets.

b) Construct LR(0) automaton

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

Answer:

Augmented Grammar is:

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

Construct initial item set,

I_0 : S' -> •S
S' -> •CC
C -> •cC
C -> •d

Construct other item sets by computing GOTO.

GOTO(I_0,S) : { S' -> S• } : I_1

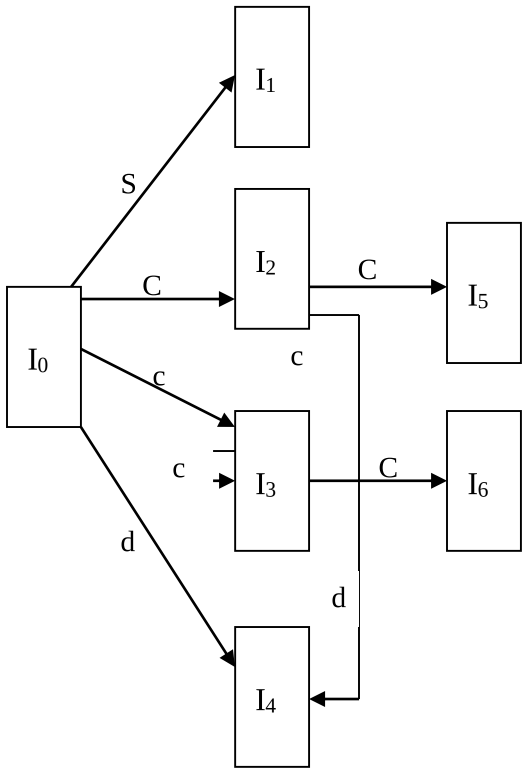
GOTO(I_0,C) : { S' -> C•C
C -> •cC
C -> •d
} : I_2

GOTO(I_0,c) : { C -> c•C
C -> •cC
C -> •d
} : I_3

GOTO(I_0,d) : { C -> d• } : I_4

GOTO(I_2,C) : { S' -> CC• } : I_5

GOTO(I_2,c) : { C -> c•C
C -> •cC
C -> •d
} : I_3



GOTO(I_2,d) : { C -> d• } : I_4

GOTO(I_3,C) : { C -> cC• } : I_6

GOTO(I_3,c) : { C -> c•C
C -> •cC
C -> •d
} : I_3

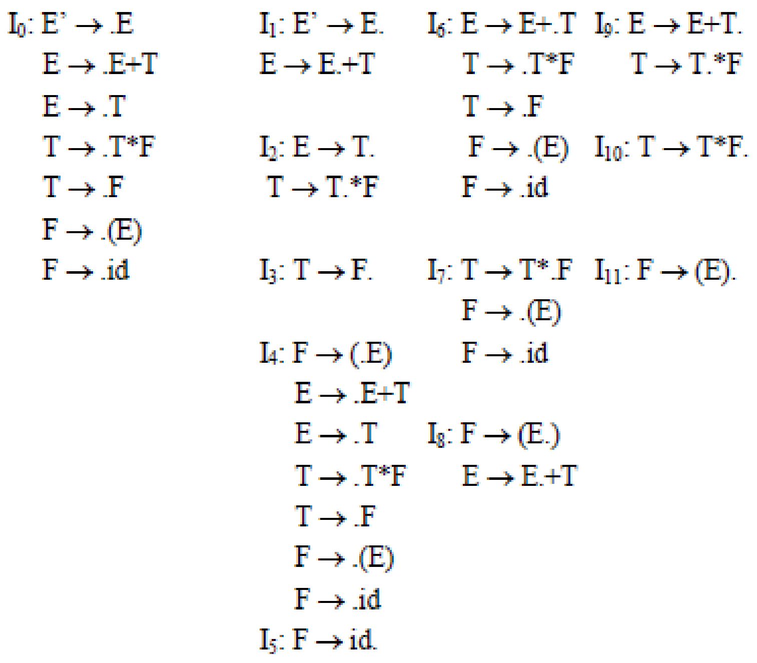
Ex2: Consider the grammar,

E → E+T | T
T → T*F | F
F → (E) | id

a) Compute Canonical Collection of LR(0) items b) Construct LR(0) automaton.

<u>Answer:</u>

Canonical collection of LR(0) items:



I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

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

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

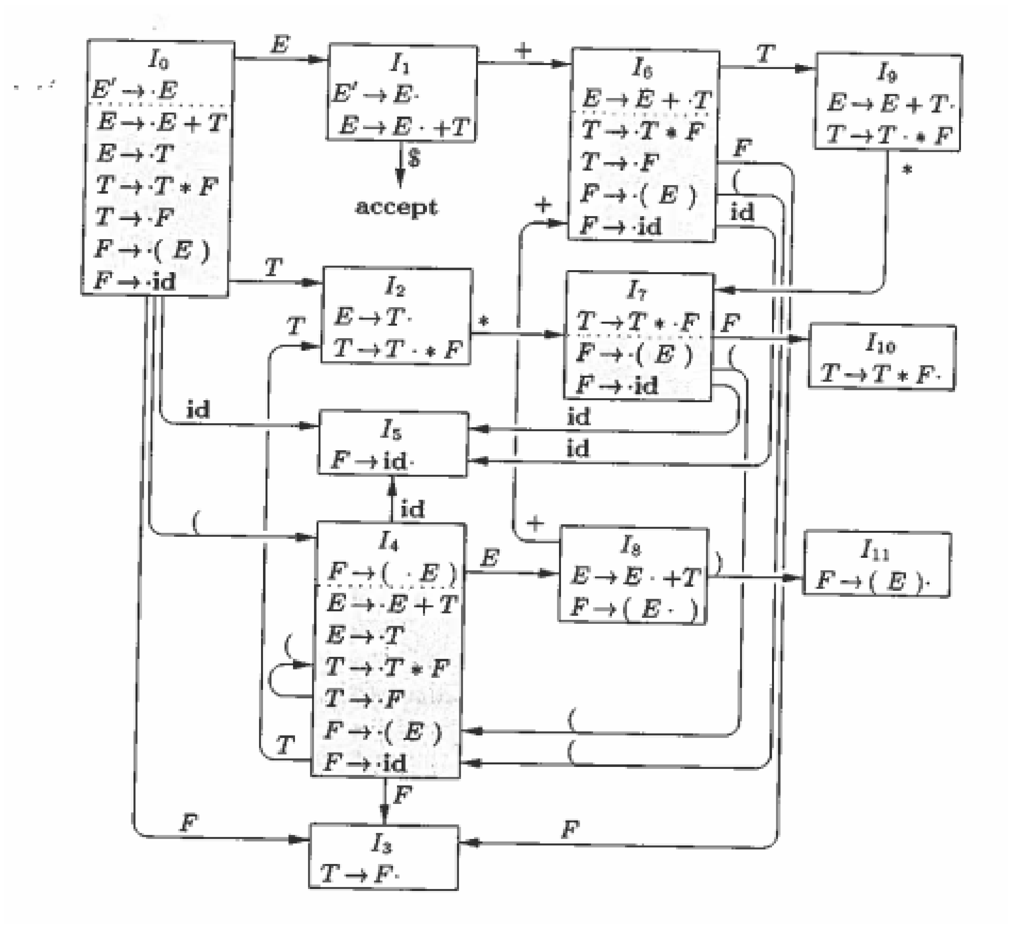
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

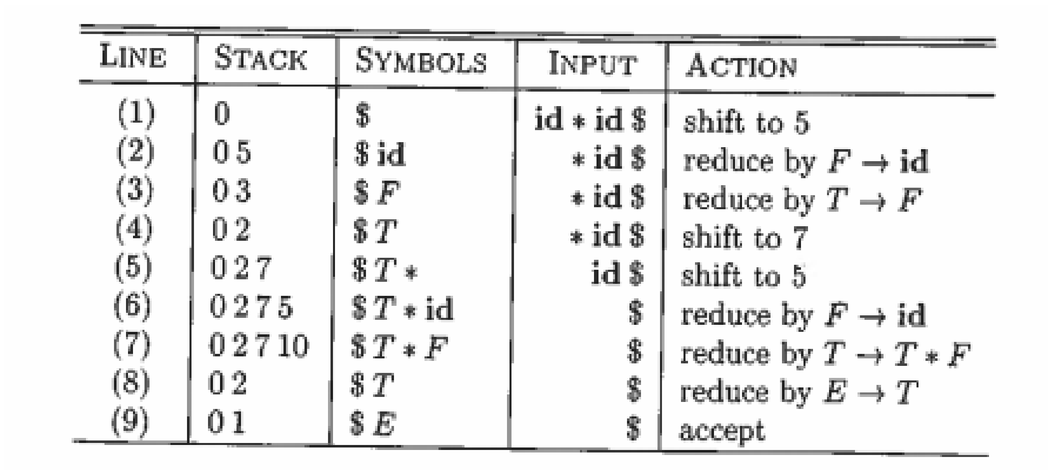
I_{11}: F → (E).

LR(0) automaton:



Use of the LR(0) Automaton:

- LR(0) automaton is used in the construction of SLR parsing table.
- LR(0) automata helps in shift-reduce decisions. The decisions can be made as follows.
  - ❖ Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state *j*.
  - ❖ Then, shift on next input symbol *a* if state *j* has a transition on *a*.
  - ❖ Otherwise, we choose to reduce; the items in state *j* will tell us which production to use.
- **Ex:** The following table illustrates the actions of a shift-reduce parser on input **id * id,** using the LR(0) automaton for an expression grammar.(Refer: above diagram)
  - ❖ We use a stack to hold states; for clarity, the grammar symbols corresponding to the states on the stack appear in column SYMBOLS.
  - ❖ Initially, the stack holds the start state 0 of the automaton; the corresponding symbol is the bottom-of-stack marker $.

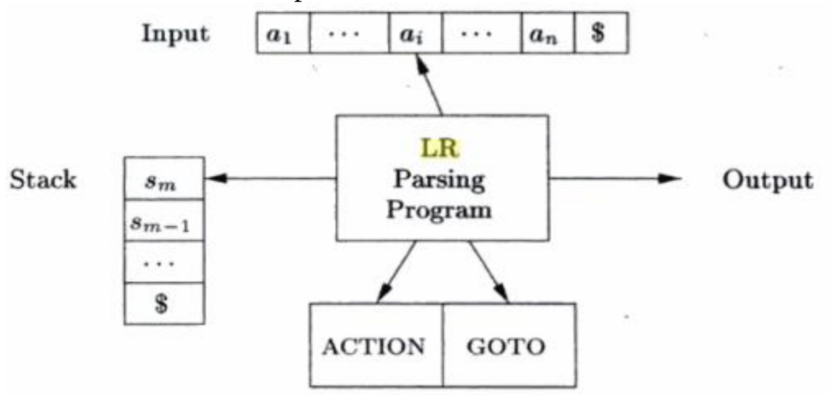


LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	id * id $	shift to 5
(2)	0 5	$ id	* id $	reduce by $F \to$ id
(3)	0 3	$ F	* id $	reduce by $T \to F$
(4)	0 2	$ T	* id $	shift to 7
(5)	0 2 7	$ $T *$	id $	shift to 5
(6)	0 2 7 5	$ $T *$ id	$	reduce by $F \to$ id
(7)	0 2 7 10	$ $T * F$	$	reduce by $T \to T * F$
(8)	0 2	$ T	$	reduce by $E \to T$
(9)	0 1	$ E	$	accept

NOTE: When reduce operation is performed look for the current state and topmost symbol. If there is no suitable production to be applied, just pop the stack.

<u>The LR-Parsing Algorithm</u>

- A schematic of an LR parser is shown below.



- It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO).
- The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state.*
- Each state summarizes the information contained in the stack below it
- The stack holds a sequence of states, $s_0 \, s_1 \ldots s_m$, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton. By construction, each state has a corresponding grammar symbol.

<u>Structure of the LR Parsing Table:</u>

- The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state *i* and a terminal *a* (or $, the input endmarker). The value of ACTION[i,a] can have one of four forms:

(a) **Shift** *j*, where *j* is a state. The action taken by the parser effectively shifts input *a* to the stack, but uses state *j* to represent *a*.
(b) **Reduce** *A->β*. The action of the parser effectively reduces *β* on the top of the stack to head *A*.
(c) **Accept**. The parser accepts the input and finishes parsing.
(d) **Error**. The parser discovers an error in its input and takes some corrective action.

2. We extend the GOTO function, defined on sets of items, to states: if GOTO[I_i, *A] = Ij*, then GOTO also maps a state *i* and a nonterminal *A* to state *j* .

LR-Parser Configurations:

- A *configuration* of ah LR parser is a pair:
  $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n\$)$
  where the first component is the stack contents (top on the right), and the second component is the remaining input.
- This configuration represents the right-sentential form
  $X_1X_2 \bullet\bullet\bullet X_m a_i a_{i+1} \bullet\bullet\bullet a_n$
- Here, the stack holds states from which grammar symbols can be recovered. s_0, the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parse.

Behavior of the LR Parser:

The configurations resulting after each of the four types of move are as follows

1. If ACTION[s_m,a_i] = shift ***s,*** the parser executes a shift move; it shifts the next state ***s*** onto the stack, entering the configuration

   $(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n\$)$

   The symbol a_i need not be held on the stack, since it can be recovered from ***s,*** The current input symbol i s now a_{i+1}.

2. If ACTION $[s_m, a_i]$ = reduce *A -> β*, then the parser executes a reduce move, entering the configuration

   $(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n\$)$

   where *r* is the length of *β*, and ***s*** = GOTO$[s_{m-r}, A]$.
   Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for GOTO$[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move.
   For the LR parsers we shall construct, the sequence of grammar symbols corresponding to the states popped off the stack, will always match *β*, the right side of the reducing production.

3. If ACTlON$[s_m, a_i]$ = accept, parsing is completed.

4. If ACTlON*[s_m, a_i]* = error, the parser has discovered an error and calls an error recovery routine.

Algorithm: LR-parsing algorithm:

INPUT: An input string *w* and an LR-parsing table with functions ACTION and GOTO for a grammar *G.*

OUTPUT: If *w* is in *L(G),* the reduction steps of a bottom-up parse for *w;* otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w$* in the input buffer. The parser then executes the following program

```
Let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION [s, a] = reduce A->β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A ->β;
        } else if ( ACTION[s, a] = accept ) break;     /* parsing is done */
        else call error-recovery routine;
}
```

Constructing SLR-Parsing Tables:

- The SLR method begins with LR(0) items and LR(0) automata. That is, given a grammar, *G,* we augment *G* to produce *G',* with a new start symbol *S'.* From *G',* we construct C, the canonical collection of sets of items for *G'* together with the *GOTO* function.
- The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm.
- **Algorithm: Constructing an SLR-parsing table**

  INPUT: An augmented grammar *G'.*

  OUTPUT: The SLR-parsing table functions ACTION and GOTO for *G'.*

  METHOD:
  1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for *G'.*

2. State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows:
   (a) If *[A -> α.aβ]* is in I_i and GOTO(I_i, a) = I_j, then set ACTlON[*i*, a*]* to "shift j". Here *a* must be a terminal.
   (b) If *[A -> α.]* is in I_i, then set ACTION[*i*, a] to "reduce *A* ->α" for all *a* in FOLLOW(*A*); here *A* may not be *S'*.
   (c) If *[S' -> S.]* is in I_i, then set ACTION[*i*, $] to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(l). The algorithm fails to produce a parser in this case.

3. The goto transitions for state *i* are constructed for all nonterminals *A* using the rule: If *GOTO*(I_i*,A) =* I_j, then GOTO[*i, A] = j.*

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

5. The initial state of the parser is the one constructed from the set of items containing *[S' ->.S]*.

- The parsing table consisting of the ACTION and GOTO functions determined by the above Algorithm is called the *SLR(l) table for G.*
- An LR parser using the SLR(l) table for *G* is called the SLR(l) parser for *G,* and a grammar having an SLR(l) parsing table is said to be *SLR(1) or SLR*

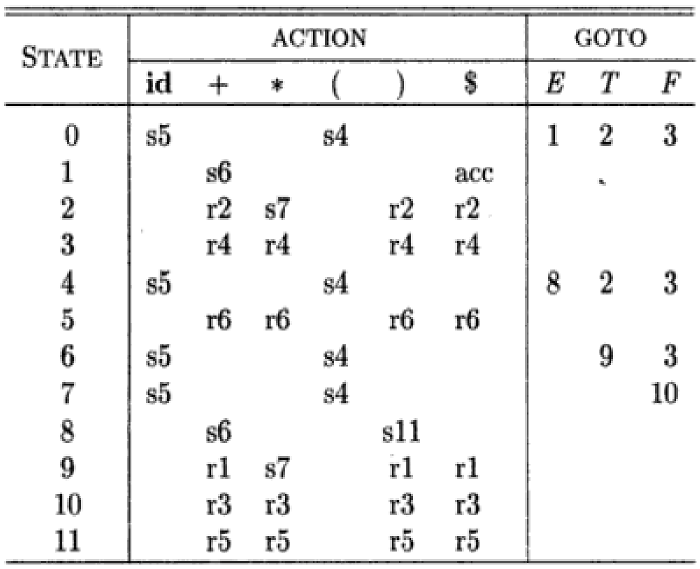
Example 1: For the following grammar

a) Compute canonical collection of sets of LR(0) items
b) Construct SLR Parsing Table
c) Show the parser moves for the input id*id+id

E → E+T | T
T → T*F | F
F → (E) | id

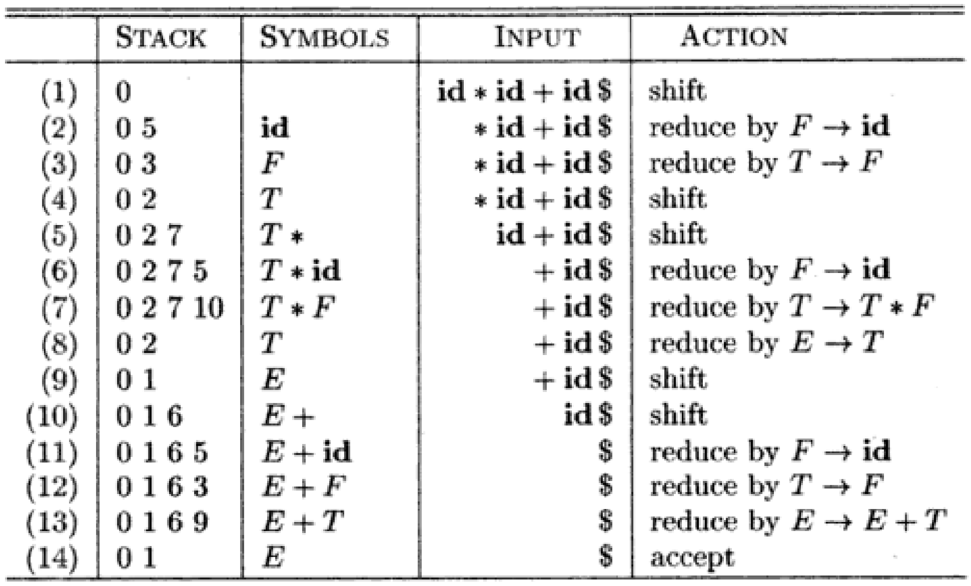
<u>Answer:</u>

a) Already computed (Refer Page No 33 & 34)
b) SLR Parsing Table:



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

c) Parser Configuration for the input **id*id+id**



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

- Every SLR(l) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(l).

<u>Example 2:</u> Is the following grammar SLR(1)?

S* -> *L* = *R* | *R
***L* -> **R* | id**
R* -> *L

Note:Consider *L* and *R* as standing for *l*-value and r-value, respectively, and * as an operator indicating "contents of."

Answer:
The canonical collection of sets of LR(0) items for the grammar:

$$\begin{array}{llll}
I_0: & S' \to \cdot S & I_5: & L \to \mathbf{id}\cdot \\
 & S \to \cdot L = R & & \\
 & S \to \cdot R & I_6: & S \to L = \cdot R \\
 & L \to \cdot * R & & R \to \cdot L \\
 & L \to \cdot \mathbf{id} & & L \to \cdot * R \\
 & R \to \cdot L & & L \to \cdot \mathbf{id} \\
 & & & \\
I_1: & S' \to S\cdot & I_7: & L \to *R\cdot \\
 & & & \\
I_2: & S \to L\cdot = R & I_8: & R \to L\cdot \\
 & R \to L\cdot & & \\
 & & I_9: & S \to L = R\cdot \\
I_3: & S \to R\cdot & & \\
 & & & \\
I_4: & L \to *\cdot R & & \\
 & R \to \cdot L & & \\
 & L \to \cdot * R & & \\
 & L \to \cdot \mathbf{id} & &
\end{array}$$

- Consider the set of items I_2. The first item in this set makes **ACTION[2,=] be "shift 6."** Since FOLLOW**(**R) contains =, the second item sets **ACTlON[2, =] to "reduce *R* -> L."**
- ***Since there is both a shift and a reduce entry in ACTION[2,=], state 2 has a shift/reduce conflict on input symbol =. Therefore the grammar is not SLR(1).***
- This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input =, having seen a string reducible to *L*.

<u>Viable Prefixes</u>

- Why can LR(0) automata be used to make shift-reduce decisions? The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar.
- The stack contents must be a prefix of a right-sentential form. If the stack holds α and the rest of the input is *x*, then a sequence of reductions will take *αx* to *S*. i.e, $S \underset{rm}{\Rightarrow} \alpha x$.
- Not all prefixes of right-sentential forms can appear on the stack, however, since the parser must not shift past the handle.
- For example, suppose $E \underset{rm}{\Rightarrow} F * id \underset{rm}{\Rightarrow} (E) * id$ Then, at various times during the parse, the stack will hold (, *(E,* and *(E),* but it must not hold *(E)**, since *(E)* is a handle, which the parser must reduce to *F* before shifting *.
- **The prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called *<u>viable prefixes</u>*.**
- A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form. By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.
- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
- We say item $A \to \beta_1.\beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \underset{rm}{\Rightarrow} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. In general, an item will be valid for many viable prefixes.

- In particular, if $\beta_2 \neq \epsilon$, then it suggests that we have not yet shifted the handle onto the stack, so shift is our move.
- If $\beta_2 = \epsilon$, then it looks as if $A \rightarrow \beta_1$ is the handle, and we should reduce by this production.
- The set of valid items for a viable prefix γ is exactly the set of items reached from the initial state along the path labeled γ in the LR(0) automaton for the grammar.
- Example: Let us consider the augmented expression grammar again. Clearly, the string E + T* is a viable prefix of the grammar. The automaton will be in state 7 after having read *E + T**. State 7 contains the items

  T ->T .F*
  F -> .(E)
  F-> .id

  which are precisely the items valid for *E+T**. To see why, consider the following three rightmost derivations

$$\begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E+T \\ &\underset{rm}{\Rightarrow} E+T*F \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E+T \\ &\underset{rm}{\Rightarrow} E+T*F \\ &\underset{rm}{\Rightarrow} E+T*(E) \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E+T \\ &\underset{rm}{\Rightarrow} E+T*F \\ &\underset{rm}{\Rightarrow} E+T*\mathbf{id} \end{aligned}$$

  The first derivation shows the validity of *T* -> T * *.F,* the second the validity of *F -> .(E),* and the third the validity of *F ->.id.* and there are no other valid items for *E + T**.

Operator Precedence Parsing

- **Precedence Relations**
  - Bottom-up parsers for a large class of context-free grammars can be easily developed using *operator grammars*.
  - *Operator grammars* have the property that no production right side is empty or has two adjacent nonterminals. This property enables the implementation of efficient *operator-precedence parsers*.
  - These parser rely on the following three precedence relations:

Relation	Meaning
a <· *b*	*a* yields precedence to *b*
a =· *b*	*a* has the same precedence as *b*
a ·> *b*	*a* takes precedence over *b*

  - These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.
  - Let's assume that between the symbols a_i and a_{i+1} there is exactly one precedence relation. Suppose that $ is the end of the string.

Then for all terminals we can write: $ <· *b* and *b* ·> $.

- If we remove all nonterminals and place the correct precedence relation: <·, =·, ·> between the remaining terminals, there remain strings that can be analyzed by easily developed parser.

  For example, the following operator precedence relations can be introduced for simple expressions:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

  Example: The input string: $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$

  After inserting precedence, relations becomes $\$ <\cdot\ \mathbf{id}_1 \cdot> + <\cdot\ \mathbf{id}_2 \cdot> * <\cdot\ \mathbf{id}_3 \cdot> \$$

- Having precedence relations allows to identify handles as follows:
  - scan the string from left until seeing ·>

  - scan backwards the string from right to left until seeing <·

  - Everything between the two relations <· and ·> forms the handle

  Note that not the entire sentential form is scanned to find the handle.

- **<u>Operator Precedence Parsing Algorithm</u>**

```
Initialize: Set ip to point to the first symbol of w$
Repeat: Let X be the top stack symbol, and a the symbol pointed to by ip
        if $ is on the top of the stack and ip points to $
            then return
        else
            Let a be the top terminal on the stack, and b the symbol pointed to by ip
                if a <· b or a =· b then
                    push b onto the stack
                    advance ip to the next input symbol

                else if a ·> b then
                    repeat
                    pop the stack
                until the top stack terminal is related by <·
                      to the terminal most recently popped
            else error()
        end
```

- **Making Operator Precedence Relations**
    - The operator precedence parsers usually do not store the precedence table with the relations, rather they are implemented in a special way.
    - Operator precedence parsers use precedence functions that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison.
    - Not every table of precedence relations has precedence functions but in practice for most grammars such functions can be designed.

- **Algorithm for Constructing Precedence Functions**

  1. Create functions f_a for each grammar terminal *a* and for the end of string symbol;

  2. Partition the symbols in groups so that f_a and g_b are in the same group if *a* =· *b* (there can be symbols in the same group even if they are not connected by this relation);

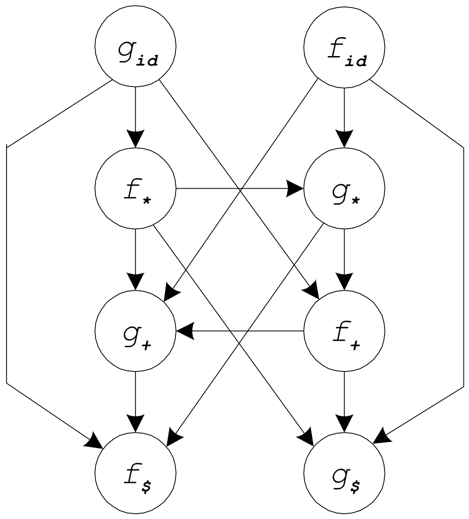
  3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if *a* <· *b*, otherwise if *a* ·> *b* place an edge from the group of f_a to that of g_b;

  4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

  Example: Consider the above table

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Using the algorithm leads to the following graph:



from which we extract the following precedence functions:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0