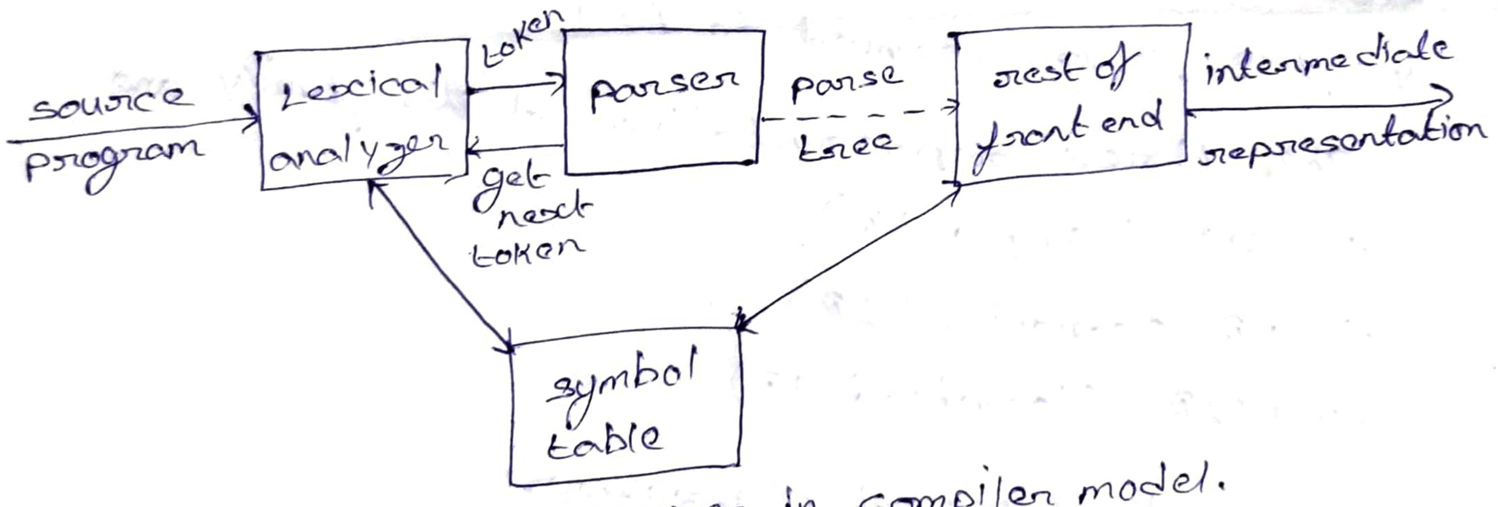
6) What is parser? Explain LR parser and SLR parser with examples. (2004, 2003, 2007, 2002

<u>Parser</u>:- The parser is a program that obtains a string of tokens from the lexical analyzer, as shown in the below figure and verifies that the string can be generated by the grammar for the source language. The parser reports any syntax errors in an intelligible fashion. It should also recover from commonly occuring errors so that it can continue processing the remainder of its input.



position of parser in compiler model.

There are three general types of parsers for grammar

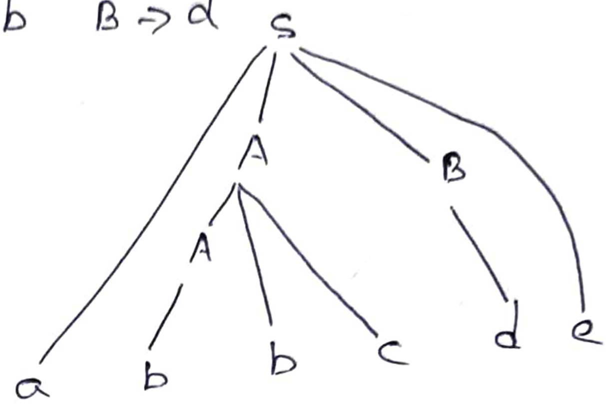
a) Universal parser
b) TOP-down parser
c) Bottom-up parser.

Bottom up parsing

The process of constructing a parse tree for an input string begining at the leaves and proceding towards the root is called Bottom up parsing.

Ex:- $S \rightarrow aABe$ $A \rightarrow Abc/b$ $B \rightarrow d$

string abbcde



$\Rightarrow$ abbcde

$\Rightarrow$ aAbcde

$\Rightarrow$ aAde

$\Rightarrow$ aABe

= S

Handle:- Handle of a string is a substring that matches the right side of a production and which can be reduced to the non terminal on the leftside of the production.

$S \rightarrow aABe$, $A \rightarrow Abc/b$ $B \rightarrow d$

string $\rightarrow$ abbcde

b & d are called Handle

Handle Pruning:-

The Process of reducing the string to the starting symbol is called handle pruning. A rightmost derivation in reverse can be obtained by handle pruning.

$\Rightarrow$ abbcde

$\Rightarrow$ aAbcde

$\Rightarrow$ aAde

$\Rightarrow$ aABe

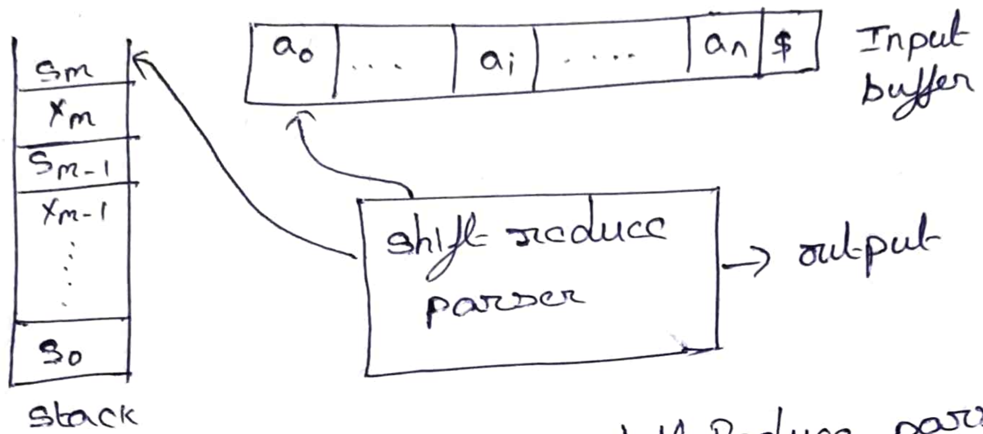
$\Rightarrow$ S

Bottom up parsing

Shift Reduce Parsing :-

Shift reduce parser use the principle of Bottom-up parsing. It attempts to construct a parse tree for an input string beginning at the leaves and working upwards the root.

Stack implementation of shift Reduce parsing :-



structure of shift Reduce parser.

A shift reduce parser consists of four actions.

i) shift ii) Reduce iii) Accept iv) Error.

i) Shift :- In a shift action, the next input symbol is shifted onto the top of the stack.

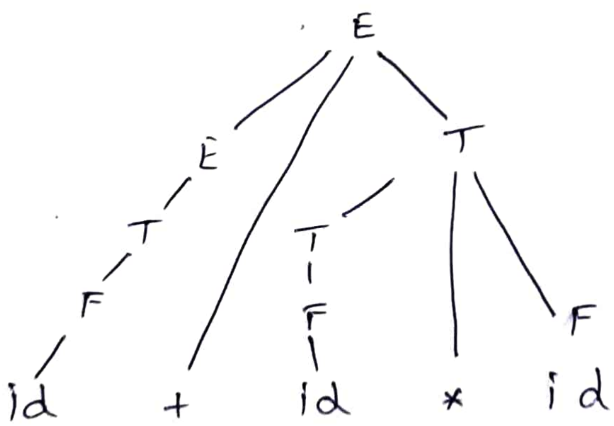
ii) Reduce :- In a reduce action, the element on the top of the stack is reduced to the non terminal of the left side of the production.

iii) Accept :- The parser announces successful completion of parsing.

iv) Error :- The parser discovers that a syntax error has occured and calls an error recovery routine.

ex:- $E \to E+T \mid T$ $\quad T \to T*F \mid F$ $\quad F \to (E) \mid id$.

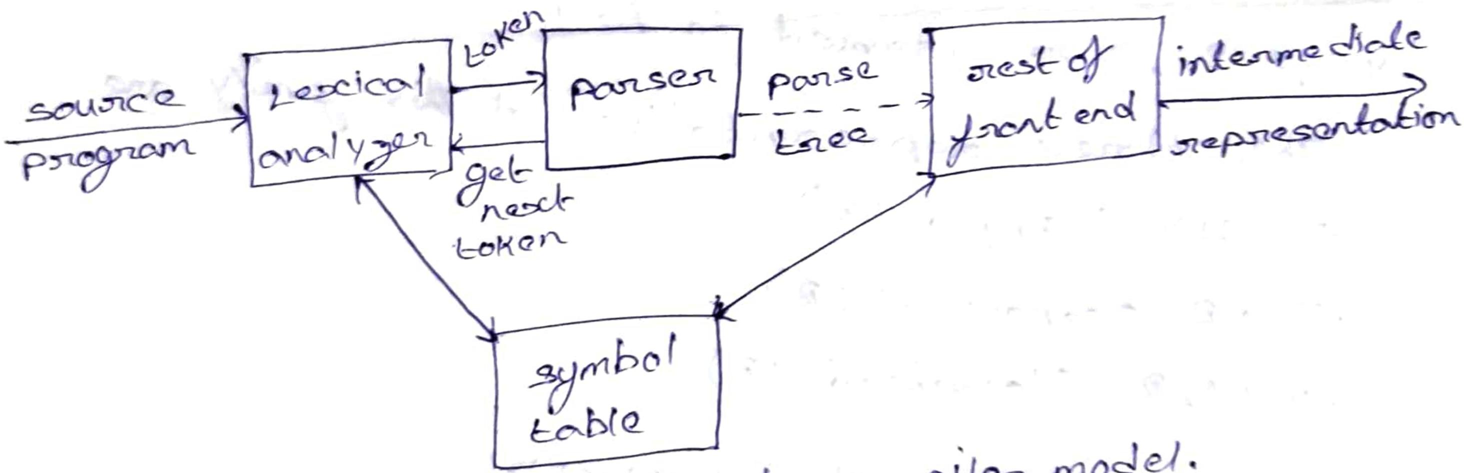
Stack	Input	action
$	id+id*id$	
$id	+id*id$	shift
$F	+id*id$	reduce F→id
$T	+id*id$	reduce T→F
$E	+id*id$	reduce E→T
$E+	id*id$	shift
$E+id	*id$	shift
$E+F	*id$	reduce F→id
$E+T	*id$	reduce T→F
$E+T*	id$	shift
$E+T*id	$	shift
$E+T*F	$	reduce F→id.
$E+T	$	reduce T→T*F
$E	$	reduce E→E+T
		accept



BOTTOM UP PARSING

6) What is parser? Explain LR parser and SLR parser with examples. (2004, 2003, 2007, 2002

Parser:- The parser is a program that obtains a string of tokens from the lexical analyzer, as shown in the below figure and verifies that the string can be generated by the grammar for the source language. The parser reports any syntax errors in an intelligible fashion. It should also recover from commonly occuring errors so that it can continue processing the remainder of its input.



position of parser in compiler model.

There are three general types of parsers for grammar

a) Universal parser
b) TOP-down parser
c) Bottom-up parser.

LR PARSERS :-

LR parsing is a bottom up parsing technique which can be used to parse a large class of context free grammars. The technique of parsing is called as LR(K) parsing.

where 'L' stands for left-to-right scanning of the input.

'R' stands for constructing a right most derivation in reverse.

'K' stands for number of input symbols of lookahead that are used in making parsing decisions. If K is not specified, it is assumed to be 1.

The three representatives of this family are.

-> SLR (simple LR)

-> LR (canonical LR)

-> LALR (lookahead LR)

The Advantages of LR parsers:-

-> LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammar can be written.

-> The LR parsing method is the most general non back tracking shift reduce parsing method.

3) The class of grammar that can be parsed using LR method is a proper super set of the class of grammars that can be passed with predictive parsers.

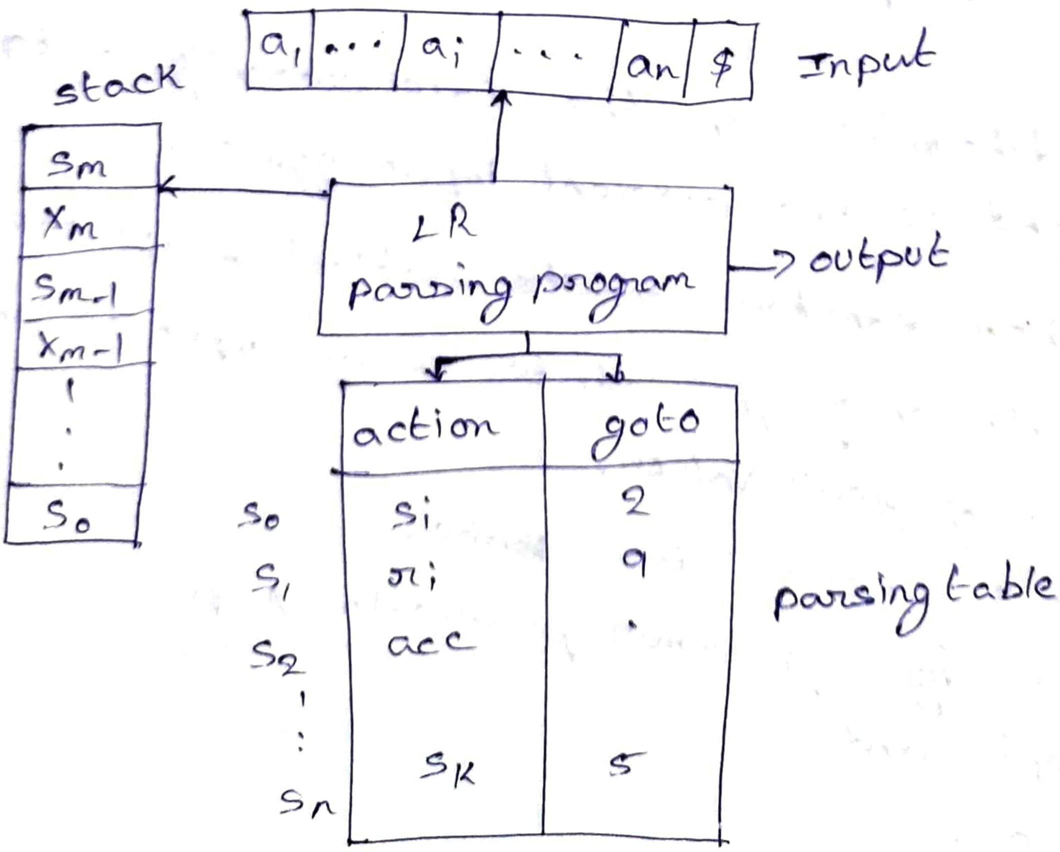
4) An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

Disadvantages of LR parsers

1) It is lengthy to construct an LR parser by hand for a programming language grammar.

2) A special tool called as LR-parser generator is required to construct a LR-parser.

structure of a LR-parser:-



The LR parsing Algorithm :-

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 the characters from an input buffer one at a time. The program uses a stack to store a string of the form $S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m$ where S_m is on top. Each X_i is a grammar symbol and each S_i is a symbol called a state.

The program driving the LR parser behaves as follows.

-> It determines S_m, the state currently on top of the stack, and a_i, the current input symbol.

-> It then ~~construct~~ consults action$[S_m, a_i]$, the parsing action table entry for state S_m and input a_i, which can have one of four values.

a) shift s, where s is a state.

b) reduce by a grammar production $A \rightarrow \beta$.

c) accept, and

d) error.

LR Parsing procedure :-

i) If action[sm, ai] = shift s, the parser executes a shift move.

ii) If action[sm, ai] = reduce A->β, then the parser executes a reduce move.

iii) If action[sm, ai] = accept, parsing is completed.

iv) If action[sm, ai] = error, the parser has discovered an error and calls an error recovery routine.

constructing SLR Parsing Tables

SLR or simple LR is the simplest parser in LR family. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus production S->XYZ gives the four items.

S->•XYZ , S->X•YZ, S->XY•Z, S->XYZ•

The production A->ε generates only one item,

A->•

A LR(0) item is complete if we have seen the complete right hand side of the rule ie if the dot is the last symbol in the right hand side as S->XYZ•. otherwise it is an incomplete item.

→ collection of sets of LR(0) items are called as: canonical LR(0) collection which provides the basis for constructing SLR parsers.

Augmented grammar:-

If G is a grammar with start symbol S then G' is the augmented grammar for G, ~~is G~~ with a new start symbol S' and production $S' \to S$.

The purpose of the new symbol is to indicate the parser when it has to stop parsing and announces acceptance of the input.

The closure operation:-

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules.

i) Initially, every item in I is added to closure(I).

ii) If $A \to \alpha \cdot B\beta$ is in closure(I) and $B \to \gamma$ is a production, then add the item $B \to \gamma$ to I, if it is not already there. This rule is applied until no more new items can be added to closure(I).

The Goto operation:-

-> 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 -> α X.β] such that [A -> α.Xβ] is in I.

-> If I is the set of items that are valid for some viable prefix γ, then goto (I, X) is the set of items that are valid for the viable prefix γX.

Algorithm :- constructing an SLR parsing table:

Input:- An augmented grammar G'.

output:- The SLR parsing table function 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 action[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 [$S' \to S.$] is in I_i, the set action[i, \$] to "accept".

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

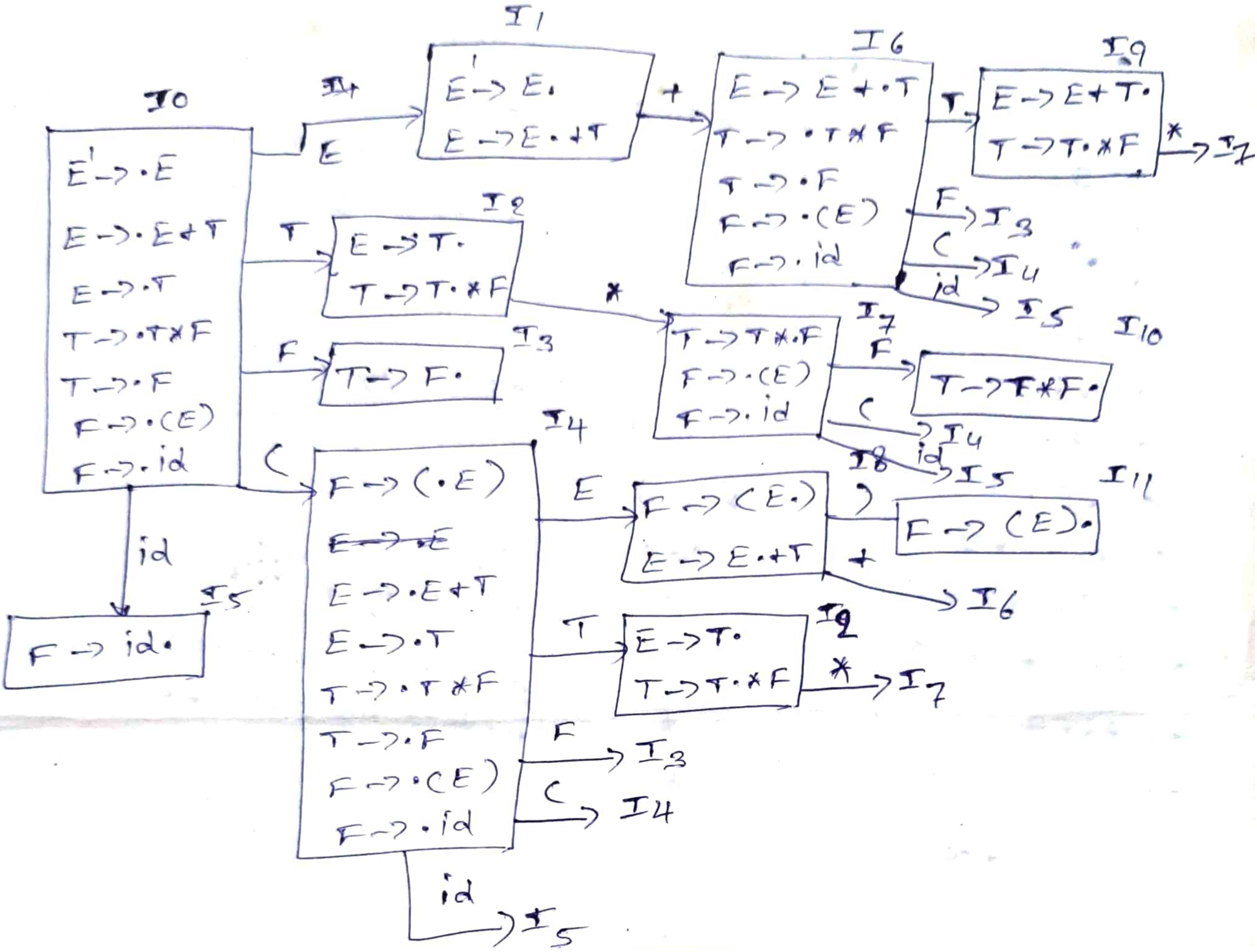
4) All entries not defined by rules (2) and (3) are made "error".

5) The initial state of the parser is the one constructed from the set of items containing [$S' \to .S$].

Ex:- $E' \to E$ $E \to E+T/T$ $T \to T*F/F$ $F \to (E)/id$

first construct the canonical collection of sets of LR(0) items.

	follow
E'	\$
E	+,), \$
T	*, +,), \$
F	*, +,), \$



state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4~~s7~~		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			

id + id * id

stack	Input	action
0	id + id * id $	shift
0 id 5	+ id * id $	reduce by F->id
0 F 3	+ id * id $	reduce by T->F
0 T 2	+ id * id $	reduce by E->T
0 E 1	+ id * id $	shift
0 E 1 + 6	id * id $	shift
0 E 1 + 6 id 5	* id $	reduce by F->id
0 E 1 + 6 F 3	* id $	reduce by T->F
0 E 1 + 6 T 9	* id $	shift
0 E 1 + 6 T 9 * 7	id $	shift
0 E 1 + 6 T 9 * 7 id 5	$	reduce by F->id
0 E 1 + 6 T 9 * 7 F 10	$	reduce by T->T*F
0 E 1 + 6 T 9	$	reduce by E->E+T
0 E 1	$	
acc		

UNIT-II
SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

Advantages of grammar for syntactic specification:

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

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals: These are the basic symbols from which strings are formed.

Non-Terminals: These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol: One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

Productions: It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar: The following grammar defines **simple** arithmetic expressions:

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

In this grammar,

- **id** + - * / ↑ ( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

DERIVATIONS:

Two basic requirements for a grammar are:
1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example: Consider the following grammar for arithmetic expressions:E
→ E+E | E*E | (E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)

3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E |E*E | (E) | - E | id
Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

Given a grammar G with start symbol S, if S → α, where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

AMBIGUITY:

A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

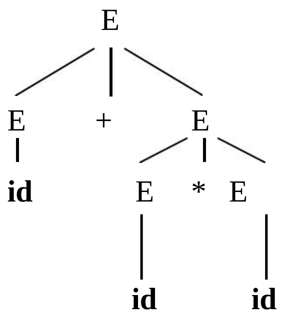
A grammar that produces more than one left most derivation or more than one right most derivation for some sentence is said to be **ambiguousgrammar**.

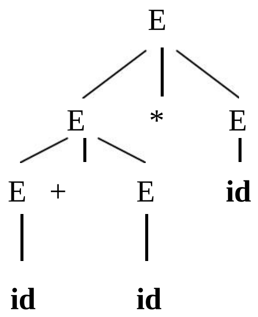
Example: Given grammar G : E → E+E | E*E | (E) | - E | id

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

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are:





WRITING A GRAMMAR:

There are four categories in writing a grammar:

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence; the initial grammar may have to be rewritten to make it parsable.

Regular Expressions vs. Context-Free Grammars:

Regular Expression	Context Free Grammar
• It is used to describe the tokens of programming language	• It consists of a quadruple, where S→ Symbol, P→ Productions, T→ Terminal, V→ variable or Non-terminal.
• It is used to check whether the given input isvalid or not using transition diagram.	• It is used to check whether the given input isvalid or not using derivation.
• The transition diagram has set of states and edges	• The context-free grammar has set of productions
• It has no start symbol.	• It has start symbol.
• It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	• It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

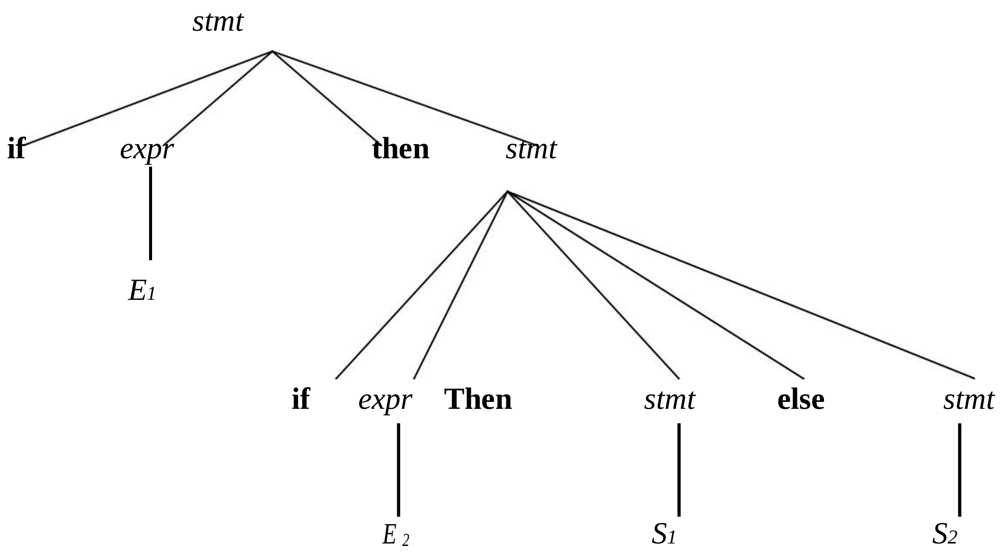
- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

ELIMINATING AMBIGUITY:

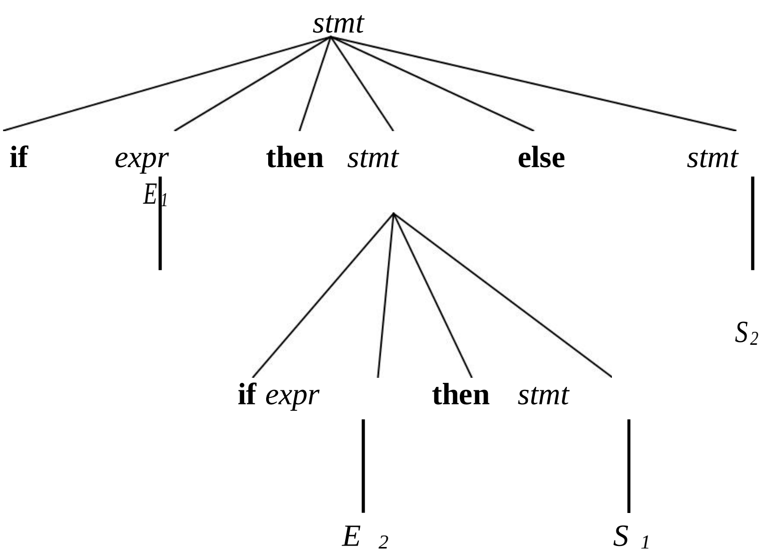
Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar. Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

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

1.



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

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

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

ELIMINATING LEFT RECURSION:

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

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

A → βA'
A' → αA' | ε

without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T T → T*F | F F → (E) | id

First eliminate the left recursion for E as

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

Then eliminate for T as

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

Thus the obtained grammar after eliminating left recursion is

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

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.

2. for *i* := 1 **to** *n* **do begin**

  for *j* := 1 **to** *i*-1 **do begin**

    replace each production of the form $A_i \rightarrow A_j \gamma$ by the

      productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$

      where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions;

  end

  eliminate the immediate left recursion among the A_i-productions

end

LEFT FACTORING:

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

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

$\mathbf{A \rightarrow \alpha A'}$
$\mathbf{A' \rightarrow \beta_1 \mid \beta_2}$

Consider the grammar , G : $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

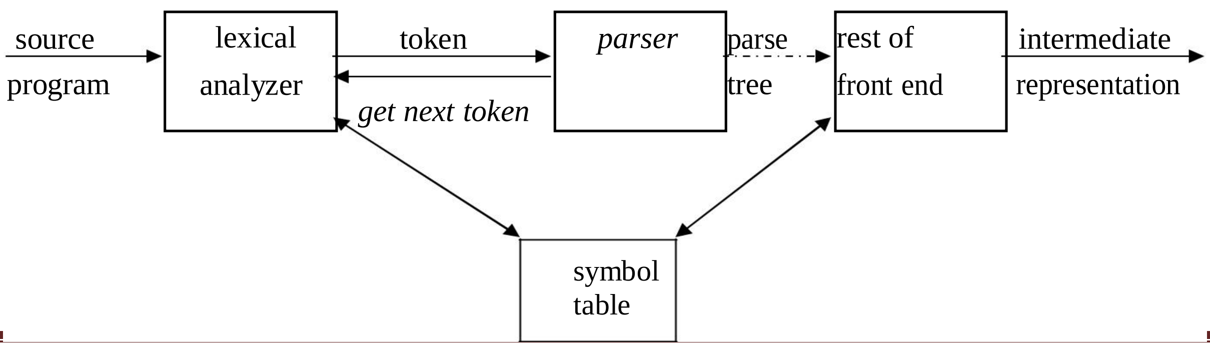
Left factored, this grammar becomes

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

THE ROLE OF PARSER

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

Position of parser in compiler model



Functions of the parser:

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

Issues:

Parser cannot detect errors such as:

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

The above issues are handled by Semantic Analysis phase.

Syntax error handling:

Programs can contain errors at many different levels. For example:

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

Functions of error handler:

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

Error recovery strategies:

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

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

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

Phrase level recovery:

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

Error productions:

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

Global correction:

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

PARSING:

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

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

1. Bottom–up parsing: TOP-DOWN PARSING:

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing:

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

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

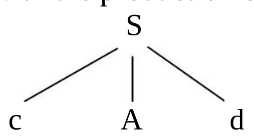
Example for backtracking:

Consider the grammar G: S → cAd
A → ab | a

and the input string w=cad.

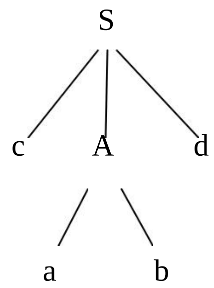
The parse tree can be constructed using the following top-down approach:

Step1: Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.

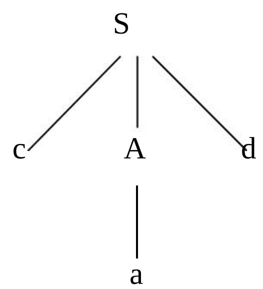


Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the inpu pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol.

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

Step4: Now try the second alternative for A.



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

Example for recursive decent parsing:

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

Consider the grammar for arithmetic expressions

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating the left-recursion the grammar becomes,

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

Now we can write the procedure for grammar as follows:

Recursive procedure:

```
Procedure    E()
begin
        T();
        EPRIME( );
End
Procedure EPRIME( )
 begin
        If input_symbol='+' then ADVANCE( );
        T( );
        EPRIME( );
end
```

```
Procedure T( ) begin
        F( );
        TPRIME( );
end
Procedure TPRIME( ) begin
        If input_symbol='*' then ADVANCE( );
        F( );
        TPRIME( );
end
Procedure F( ) begin
        If input-symbol='id' then ADVANCE( );
        else if input-symbol='(' then ADVANCE( );
        E( );
        else if input-symbol=')' then ADVANCE( );
end
else ERROR( );
```

Stack implementation:

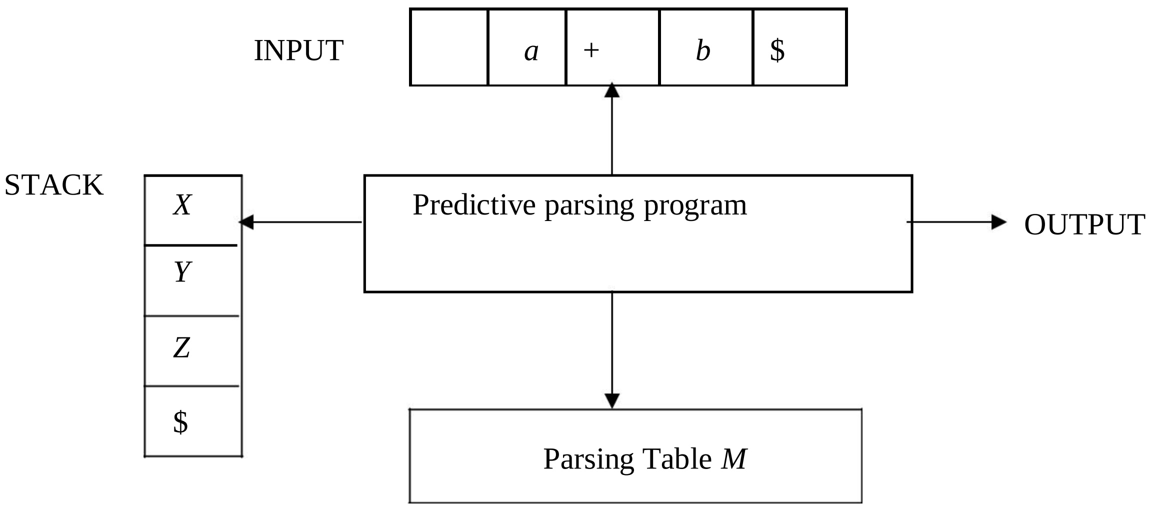
To recognize input **id+id*id :**

Procedure	**Input String**
E()	**<u>id</u>+id*id**
T()	**<u>id</u>+id*id**
F()	**<u>id</u>+id*id**
ADVANCE()	**id<u>+</u>id*id**
TPRIME()	**id<u>+</u>id*id**
EPRIME()	**id<u>+</u>id*id**
ADVANCE()	**id+<u>id</u>*id**
T()	**id+<u>id</u>*id**
F()	**id+<u>id</u>*id**
ADVANCE()	**id+id<u>*</u>id**
TPRIME()	**id+id<u>*</u>id**
ADVANCE()	**id+id*<u>id</u>**
F()	**id+id*<u>id</u>**
ADVANCE()	**id+id*<u>id</u>**
TPRIME()	**id+id*<u>id</u>**

PREDICTIVE PARSING

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

Non-recursive predictive parser



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

Input buffer:

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

Stack:

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

Parsing table:

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

Predictive parsing program:

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

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by WVU. If $M[X, a] =$ **error**, the parser calls an error recovery routine.

Algorithm for non_recursive predictive parsing:

Input : A string w and a parsing table M for grammar G.
Output : If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.
Method : Initially, the parser has $\$S$ on the stack with S, the start symbol of G on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

set *ip* to point to the first symbol of *w*$;
repeat
let X be the top stack symbol and a the symbol pointed to by *ip*; **if**

X is a terminal or \$ **then**
if $X = a$ **then**
pop X from the stack and advance *ip*
else *error*()
else /* X is a non-terminal */
if $M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then begin**
pop X from the stack;
push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
end
else *error*()
until X = \$ /* stack is empty */

Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

Rules for first ( ):

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

Rules for follow ( ):

1. If S is a start symbol, then FOLLOW(S) contains \$.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method :

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

Example: Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

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

First( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
\$E	\$E	
\$E'T	id+id*id \$	E → TE'
\$E'T'F	id+id*id \$	T → FT'
\$E'T'id	id+id*id \$	F → id
\$E'T'	+id*id \$	
\$E'	+id*id \$	T' → ε
\$E'T+	+id*id \$	E' → +TE'
\$E'T	id*id \$	
\$E'T'F	id*id \$	T → FT'
\$E'T'id	id*id \$	F → id
\$E'T'	*id \$	
\$E'T'F*	*id \$	T' → *FT'
\$E'T'F	id \$	
\$E'T'id	id \$	F → id
\$E'T'	\$	
\$E'	\$	T' → ε
\$	\$	E' → ε

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' | a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }
FIRST(S') = {e, ε }
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:

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

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

Actions performed in predictive parsing:

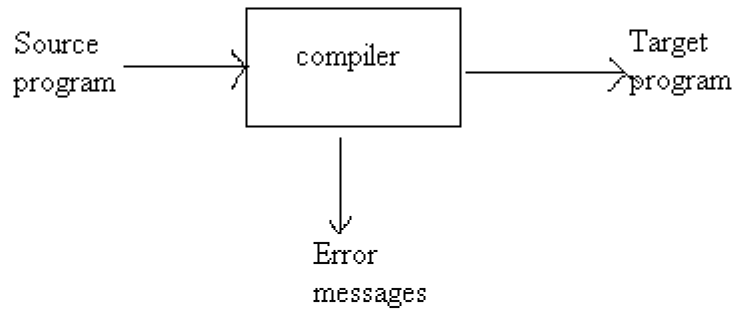
1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

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

Introduction to Compilers

A compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language). The compiler reports errors present in the source program.



A Compiler

The target language may be another programming language or the machine language of a processor. Compilers are some times classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

The Analysis-Synthesis Model of Compilation:

There are two parts to compilation

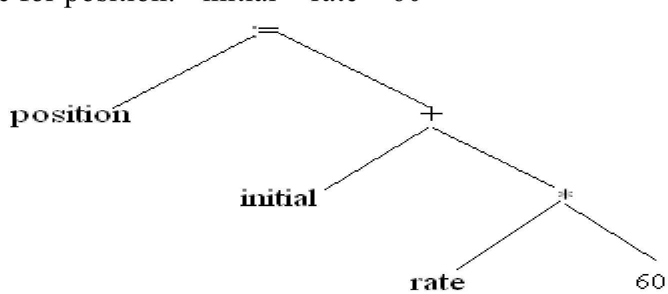
a) Analysis
b) Synthesis

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

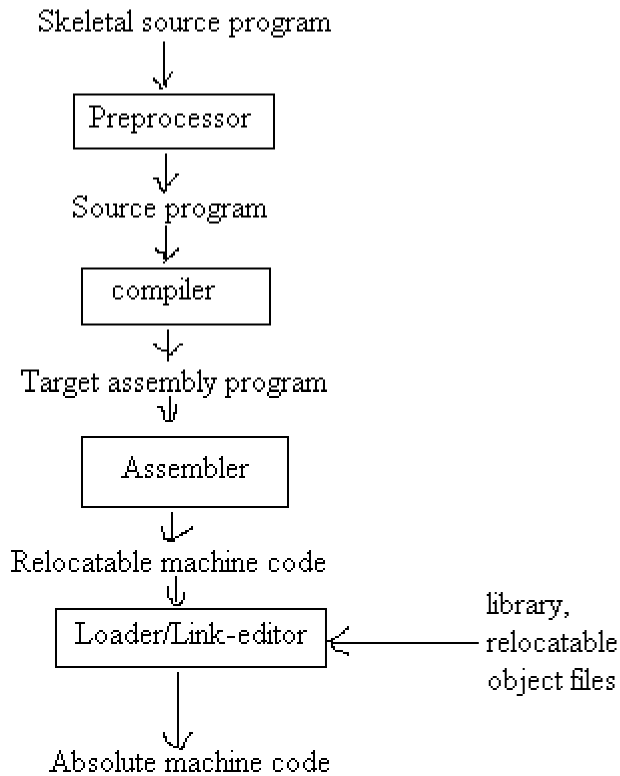
During analysis, the operations implemented by the source program are determined and recorded in a hierarchical structure called a tree. A special kind of tree called a syntax tree is used, in which each node represents operation and children of the node represent the arguments of the operation.

Eg: Syntax tree for position: =initial + rate * 60



The Context of a Compiler:

In addition to a compiler several other programs may be required to create an executable target program. A source program may be divided in to modules stored in separate files. The task of collecting the source program is some times entrusted to a distinct program, called a preprocessors.



A language-processing system

The above figure shows a typical compiler. The target program created by the compiler may require further processing before it can be run. The compiler creates assembly code that is translated by an assembly into machine code and then linked together with some library routines into the code that actually runs on the machine.

Analysis of the Source program:

Analysis consists of three phases.

a) Linear analysis – Divides the source program in to tokens.
b) Hierarchical analysis – Generates parse tree to check syntax.
c) Semantic analysis – The semantic analysis phase checks the source program for semantic errors.

Lexical Analysis:

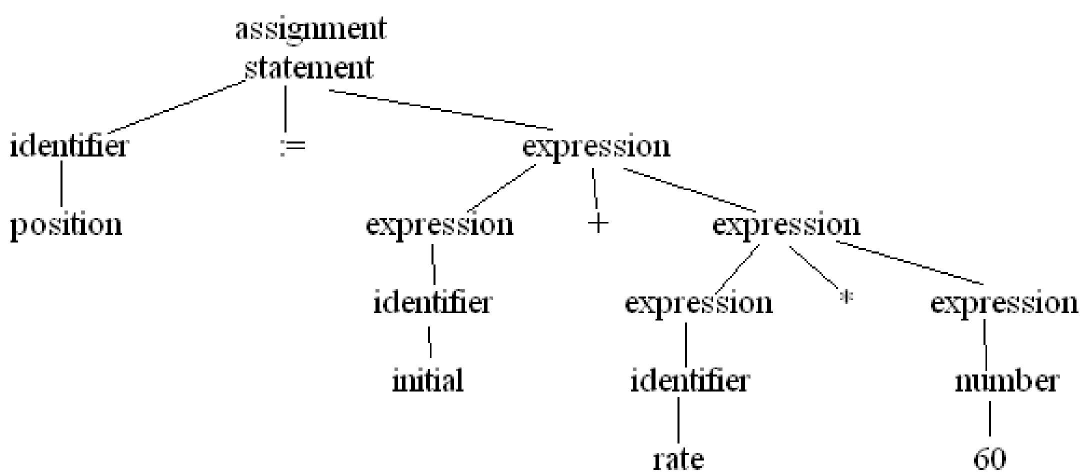
In a compiler, linear analysis is called lexical analysis or scanning.
Eg: position: = initial + rate * 60
Would be grouped in to following tokens.

a) The identifier position
b) The assignment symbol :=
c) The identifier initial
d) The plus sign +
e) The identifier rate
f) The multiplication sign *
g) The number 60

The blanks separating the characters of there tokens would normally be eliminated during lexical analysis.

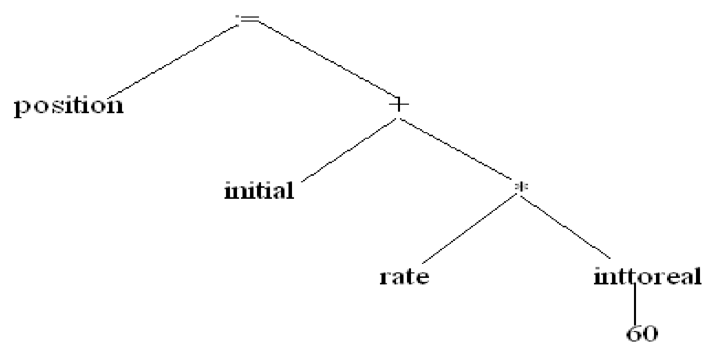
Syntax Analysis:

Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output. The grammatical phrases of the source program are represented by a parse tree.



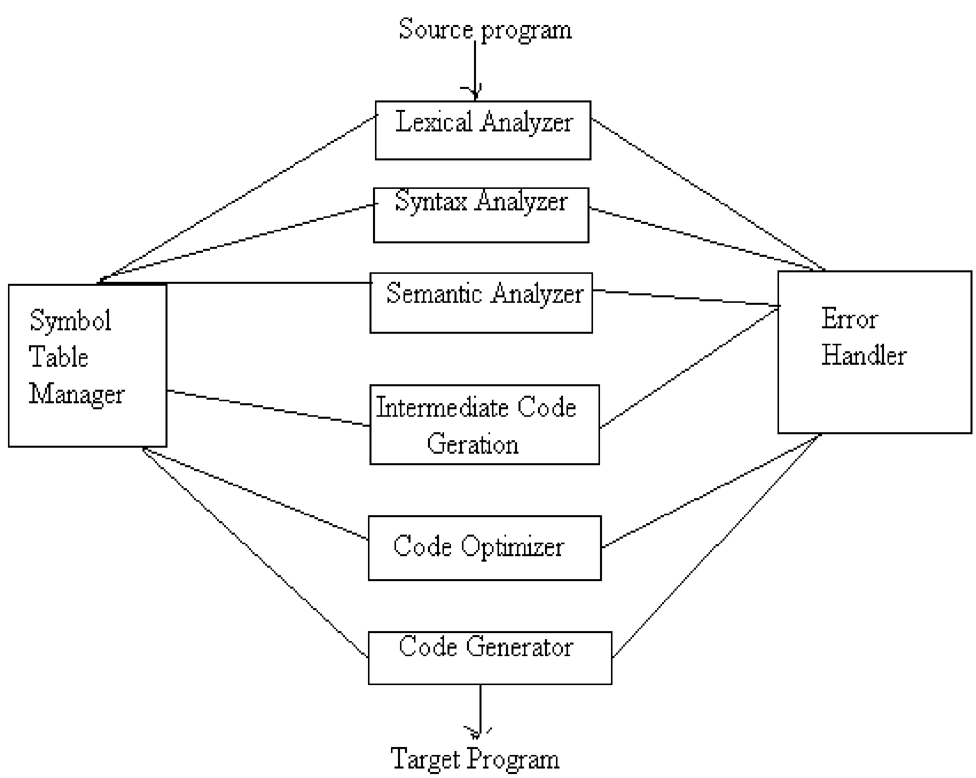
Semantic Analysis:

The semantic analysis phase checks the source program for semantic errors and gathers the information for the subsequent code generation phase.
An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. For example, many programming languages definitions require a compiler to report an error every time a real number is used to index an array.



The Phases of a Compiler:

The compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in the below figure.



The compiler implementation process is divided into two parts.

1) Analysis of source program
   a) Lexical analysis
   b) Syntax analysis
   c) Semantic analysis
2) Synthesis of target program
   a) Intermediate code generation
   b) Code optimization
   d) Code generation

Analysis of the source program involves analyzing the different constructs of the program by breaking them into irreducible pieces and evaluating its syntax and semantics.

Synthesis of target program includes the process of representing source program in intermediate form and optimizing it to get improved machine code. It includes all machine independent phases of compiler.

Symbol Table Management:

An essential function of a compiler to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocation for an identifier, its type, its scope etc.

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered in to the symbol table. However, the attribute of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

Error Detection and reporting:

Each phase can encounter errors. However, after detecting an error, a phase must some how deal with that error, so that compiler can proceed, allowing further errors in the source program to be detected.

The syntax and semantic analysis phase usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors, where the characters remaining in the input do not form any token of the language.

Errors when the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. E.g. if we add two identifiers, one of which is the name of an array and the other the name of a procedure.

The Analysis Phase:

a) **Lexical Analysis:**

The lexical analysis phase reads the character in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters forming a token, such as an identifier, a keyword, a punctuation character, or a multi character operator like :=. The character sequence forming a token is called the lexeme for the token.

The lexical analysis, not only recognizes tokens but also the code value for that tokens. The value place contains a pointer to the symbol table where the actual value is stored.

E.g. position:= initial + rate * 60

Would be grouped in to the following tokens.

Position	identifier
:=	assignment
Initial	identifier

+	plus sign
Rate	identifier
‘*’	multiplication sign
60	constant.

b) **Syntax Analysis:**

Syntax analysis involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually the grammatical phrases of the source program are represented by a parse tree.

c) **Semantic Analysis:**

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase.

d) **Intermediate Code Generation:**

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. We can think of this intermediate representation as a program for an abstract machine. This intermediate representation should have two important properties, it should be easy to produce, and easy to translate into the target program.

e) **Code Optimization:**

The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result.

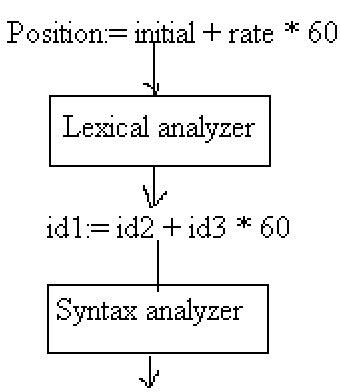
Eg: temp: = id3 * 60.0
id1:= id2 +temp1

The compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated.

f) **Code Generation:**

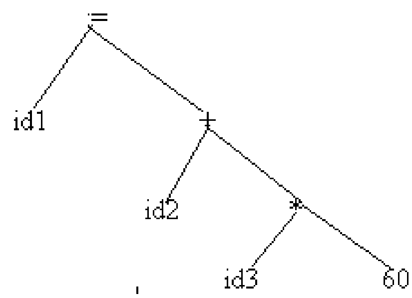
The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

Example:

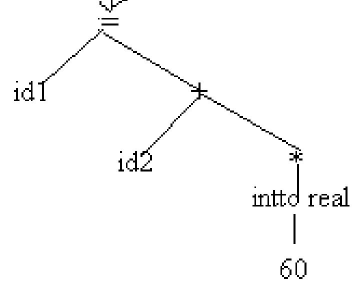


Symbol table

position	
initial	
rate	



Semantic Analyzer



Intermediate code generator

```
temp1:= inttoreal(60)
temp2:=id3 * temp1
temp3:= id2 + temp2
id1:= temp3
```

Code Optimizer

```
temp1:= id3 * 60.0
id1:= id2 + temp1
```

Code generator

```
MOVF   id3,R2
MULF   #60.0,R2
MOVE   id2,R1
ADDF   R2,R1
MOVF   R1,id1
```

Cousins of the compiler:

Preprocessors:

A preprocessor produces input to compiler. They may perform the following functions.

a) Macro processing: - A preprocessor may allow a user to define macros that are short hands fro longer constructs.
b) File inclusion: - A preprocessor may include header files into the program text.
c) Rational preprocessors: - These preprocessors expand older languages with more modern floe of control and data structuring facilities.
d) Language extensions – These processors attempt to add capabilities to the language by what amount top built in macros.

Assemblers:

Some compilers produce assembly code that is passed to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.

Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory address.

E.g.:

```
MOV    a, R1
ADD    #2, R1
MOV    R1, b
```

This code moves the contents of the address a into R1, then adds the constant 2 to it, and finally stores the result in the location named by b. thus it computes b: = a + 2.

Two pass Assembly:

The simplest form of assembler makes two passes over the input. In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table. Identifiers are assigned storage locations as they encountered for the first time, so after reading, for example the symbol table might contain the entries shown in the figure below.

identifier	address
a	0
b	4

In the second pass, the assembler scans the input again. This time, it translates each operation code in to the sequence of bits representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table. The output of the second pass is usually relocatable machine code, i.e. it can be loaded starting at any location L in memory.

Loader and Link Editors:

Usually, a program called a loader performs the two functions of loading and link editing. The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper locations.

The link editor allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations and one or more may be library files of routines provided by the system and available to any program that needs them.

<u>The Grouping of Phases:</u>

Front and Back Ends:

The phases are collected into a front end and a back end. The front end consists of those phases that depends primarily on the source language and are largely independent of the target machine. These normally include lexical and syntax analysis, the creation of symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well as the front end also includes the error handling that goes along with each of these phases.

The back end includes those portions of the compiler that depend on the target machine, and generally these portions don't depend on the source language, just the intermediate language. In the back end we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol table operations.

<u>Compiler construction tools:</u>

a) Parser Generators: - These produce syntax analyzer normally from input that is based on a context free grammar.
b) Scanner Generators: - These automatically generate lexical analyzer, normally from a specification based on regular expressions.
c) Syntax Directed translation engines: - These generates the intermediate code by traversing through parse tree.
d) Automatic code Generators: - This converts intermediate code to assembly code of the target machine based on the architecture and instruction sets.
e) Data flow Engines: - Much of the information needed to perform good code optimization involves "data flow analysis", the gathering of information about how values are transmitted from one part of program to each other part.