1. Explain overview of translation process.

- A translator is a kind of program that takes one form of program as input and converts it into another form.
- The input is called *source program* and output is called *target program.*
- The source language can be assembly language or higher level language like C, C++, FORTRAN, etc...
- There are three types of translators,
    1. Compiler
    2. Interpreter
    3. Assembler

2. What is compiler? List major functions done by compiler.

- A compiler is a program that reads a program written in one language and translates into an equivalent program in another language.

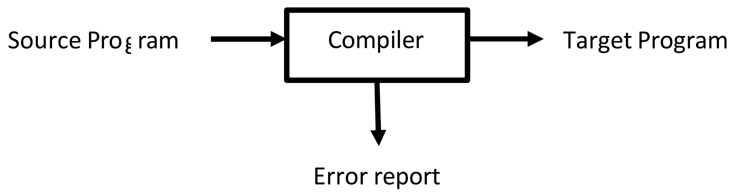


Fig.1.1. A Compiler

Major functions done by compiler:

- Compiler is used to convert one form of program to another.
- A compiler should convert the source program to a target machine code in such a way that the generated target code should be easy to understand.
- Compiler should preserve the meaning of source code.
- Compiler should report errors that occur during compilation process.
- The compilation must be done efficiently.

3. Write the difference between compiler, interpreter and assembler.

1. *Compiler v/s Interpreter*

No.	Compiler	Interpreter
1	Compiler takes entire program as an input.	Interpreter takes single instruction as an input.
2	Intermediate code is generated.	No Intermediate code is generated.
3	Memory requirement is more.	Memory requirement is less.
4	Error is displayed after entire program is checked.	Error is displayed for every instruction interpreted.
5	Example: C compiler	Example: BASIC

Table 1.1 Difference between Compiler & Interpreter

2. *Compiler v/s Assembler*

No.	Compiler	Assembler
1	It translates higher level language to machine code.	It translates mnemonic operation code to machine code.
2	Types of compiler, • Single pass compiler • Multi pass compiler	Types of assembler, • Single pass assembler • Two pass assembler
3	Example: C compiler	Example: 8085, 8086 instruction set

Table 1.2 Difference between Compiler & Assembler

4. Analysis synthesis model of compilation. OR Explain structure of compiler. OR Explain phases of compiler. OR Write output of phases of a complier. for a = a + b * c * 2; type of a, b, c are float

There are mainly two parts of compilation process.

1. **Analysis phase**: The main objective of the analysis phase is to break the source code into parts and then arranges these pieces into a meaningful structure.
2. **Synthesis phase**: Synthesis phase is concerned with generation of target language statement which has the same meaning as the source statement.

Analysis Phase: Analysis part is divided into three sub parts,

I. Lexical analysis
II. Syntax analysis
III. Semantic analysis

Lexical analysis:

- Lexical analysis is also called linear analysis or scanning.

- Lexical analyzer reads the source program and then it is broken into stream of units. Such units are called token.
- Then it classifies the units into different lexical classes. E.g. id's, constants, keyword etc...and enters them into different tables.
- For example, in lexical analysis the assignment statement **a: = a + b * c * 2** would be grouped into the following tokens:

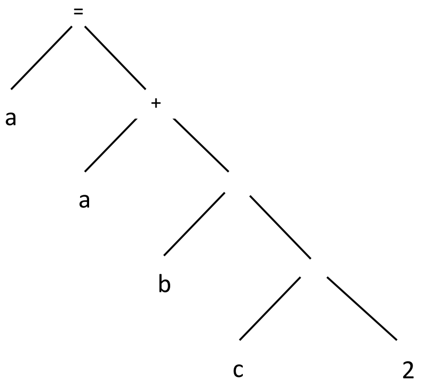
A	Identifier 1
=	Assignment sign
A	Identifier 1
+	The plus sign
b	Identifier 2
*	Multiplication sign
c	Identifier 3

*	Multiplication

	sign
2	Number 2

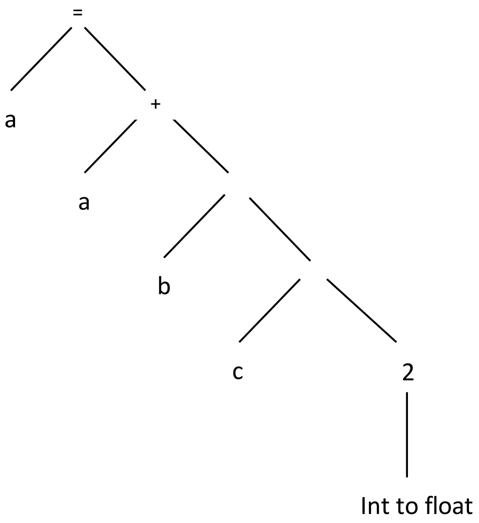
Syntax Analysis:

- Syntax analysis is also called hierarchical analysis or parsing.
- The syntax analyzer checks each line of the code and spots every tiny mistake that the programmer has committed while typing the code.
- If code is error free then syntax analyzer generates the tree.



Semantic analysis:

- Semantic analyzer determines the meaning of a source string.
- For example matching of parenthesis in the expression, or matching of if..else statement or performing arithmetic operation that are type compatible, or checking the scope of operation.



Synthesis phase: synthesis part is divided into three sub parts,

I. Intermediate code generation
II. Code optimization
III. Code generation

Intermediate code generation:

- The intermediate representation should have two important properties, it should be easy to produce and easy to translate into target program.

- We consider intermediate form called “three address code”.
- Three address code consist of a sequence of instruction, each of which has at most three operands.
- The source program might appear in three address code as,

```
t1= int to real(2)
t2= id3 * t1
t3= t2 * id2
t4= t3 + id1
id1= t4
```

Code optimization:

- The code optimization phase attempt to improve the intermediate code.
- This is necessary to have a faster executing code or less consumption of memory.
- Thus by optimizing the code the overall running time of a target program can be improved.

```
t1= id3 * 2.0
t2= id2 * t1
id1 = id1 + t2
```

Code generation:

- In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instruction.

```
MOV id3, R1
MUL #2.0, R1
MOV id2, R2
MUL R2, R1
MOV id1, R2
ADD R2, R1
MOV R1, id1
```

Symbol Table

- A **symbol table** is a data structure used by a language translator such as a compiler or interpreter.
- It is used to store names encountered in the source program, along with the relevant attributes for those names.
- Information about following entities is stored in the symbol table.
  - ✓ Variable/Identifier
  - ✓ Procedure/function
  - ✓ Keyword
  - ✓ Constant
  - ✓ Class name
  - ✓ Label name

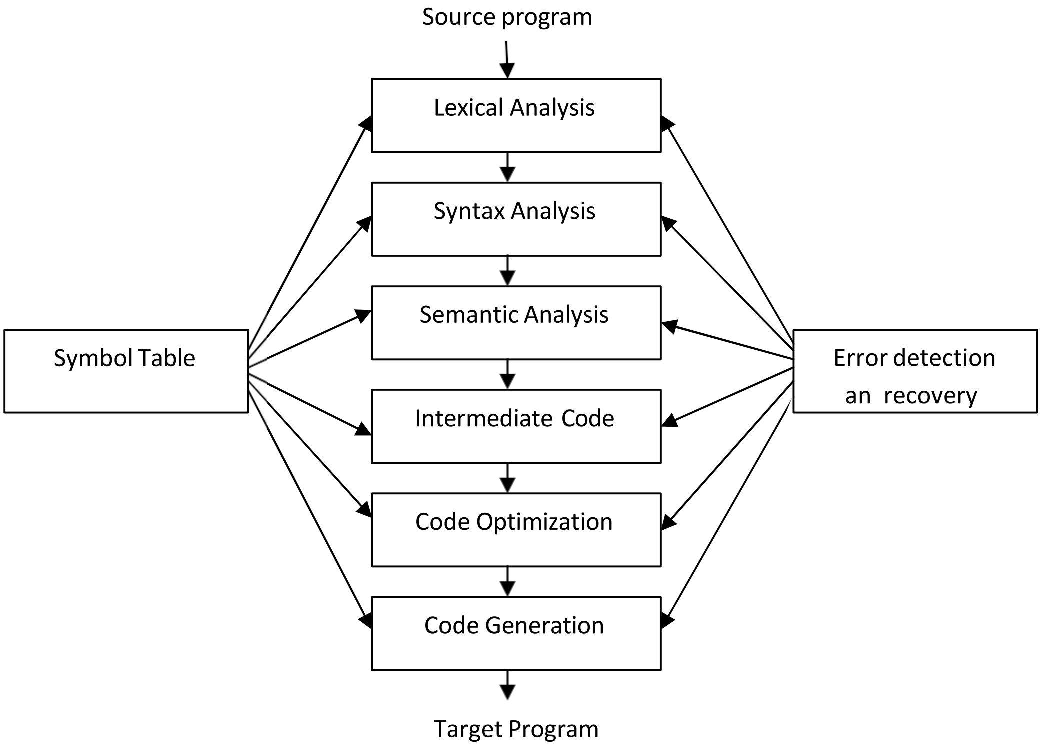


Fig.1.2. Phases of Compiler

5. The context of a compiler. OR Cousins of compiler. OR What does the linker do? What does the loader do? What does the Preprocessor do? Explain their role(s) in compilation process.

- In addition to a compiler, several other programs may be required to create an executable target program.

Preprocessor

Preprocessor produces input to compiler. They may perform the following functions,

1. Macro processing: A preprocessor may allow user to define macros that are shorthand for longer constructs.
2. File inclusion: A preprocessor may include the header file into the program text.
3. Rational preprocessor: Such a preprocessor provides the user with built in macro for construct like while statement or if statement.
4. Language extensions: this processors attempt to add capabilities to the language by what amount to built-in macros. Ex: the language equal is a database query language embedded in C. statement beginning with ## are taken by preprocessor to be database access statement unrelated to C and translated into procedure call on routines that perform the database access.

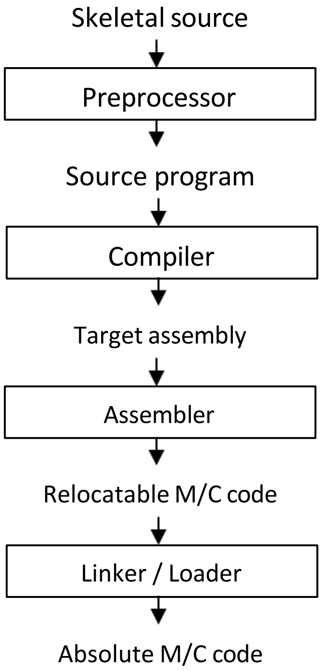


Fig.1.3. Context of Compiler

Assembler

Assembler is a translator which takes the assembly program as an input and generates the machine code as a output. An assembly is a mnemonic version of machine code, in which names are used instead of binary codes for operations.

Linker

Linker allows us to make a single program from a several files of relocatable machine code. These file may have been the result of several different compilation, and one or more may be library files of routine provided by a system.

Loader

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 location.

6. Explain front end and back end in brief. (Grouping of phases)

- The phases are collected into a front end and back end.

Front end

- The front end consist of those phases, that depends primarily on source language and largely independent of the target machine.
- Front end includes lexical analysis, syntax analysis, semantic analysis, intermediate code generation and creation of symbol table.
- Certain amount of code optimization can be done by front end.

Back end

- The back end consists of those phases, that depends on target machine and do not depend on source program.

- Back end includes code optimization and code generation phase with necessary error handling and symbol table operation.

7. What is the pass of compiler? Explain how the single and multi-pass compilers work? What is the effect of reducing the number of passes?

- One complete scan of a source program is called pass.
- Pass include reading an input file and writing to the output file.
- In a single pass compiler analysis of source statement is immediately followed by synthesis of equivalent target statement.
- It is difficult to compile the source program into single pass due to:
- **Forward reference:** a forward reference of a program entity is a reference to the entity which precedes its definition in the program.
- This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- It leads to multi pass model of compilation.
- In Pass I: Perform analysis of the source program and note relevant information.
- In Pass II: Generate target code using information noted in pass I.

Effect of reducing the number of passes

- It is desirable to have a few passes, because it takes time to read and write intermediate file.
- On the other hand if we group several phases into one pass we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

8. Explain types of compiler. OR Write difference between single pass and multi pass compiler.

Single pass compiler v/s Multi pass Compiler

No.	Single pass compiler	Multi pass compiler
1	A one-pass compiler is a compiler that passes through the source code of each compilation unit only once.	A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times.
2	A one-pass compiler is faster than multi-pass compiler.	A multi-pass compiler is slower than single-pass compiler.
3	One-pass compiler are sometimes called narrow compiler.	Multi-pass compilers are sometimes called wide compiler.
4	Language like Pascal can be implemented with a single pass compiler.	Languages like Java require a multi-pass compiler.

Table 1.3 Difference between Single Pass Compiler & Multi Pass Compiler

9. Write the difference between phase and pass.

Phase v/s Pass

No.	Phase	Pass
1	The process of compilation is carried out in various step is called phase.	Various phases are logically grouped together to form a pass.
2	The phases of compilation are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.	The process of compilation can be carried out in a single pass or in multiple passes.

Table 1.3 Difference between Phase & Pass

1. Role of lexical analysis and its issues. OR How do the parser and scanner communicate? Explain with the block diagram communication between them.

- The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- This interaction is given in figure 2.1,

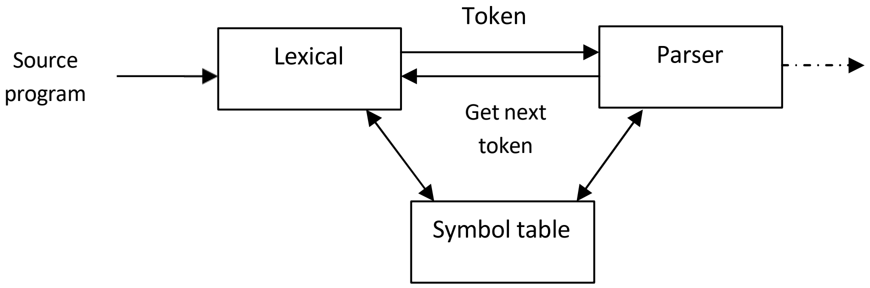


Fig. 2.1 Communication between Scanner & Parser

- It is implemented by making lexical analyzer be a subroutine.
- Upon receiving a "get next token" command from parser, the lexical analyzer reads the input character until it can identify the next token.
- It may also perform secondary task at user interface.
- One such task is stripping out from the source program comments and white space in the form of blanks, tabs, and newline characters.
- Some lexical analyzer are divided into cascade of two phases, the first called scanning and second is "lexical analysis".
- The scanner is responsible for doing simple task while lexical analysis does the more complex task.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

- Simpler design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one or other of these phases.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

2. Explain token, pattern and lexemes.

Token: Sequence of character having a collective meaning is known as *token*.

Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: The set of rules called *pattern* associated with a token.

Lexeme: The sequence of character in a source program matched with a pattern for a token is called lexeme.

Token	Lexeme	Pattern
Const	Const	Const
If	If	If
Relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or >
Id	Pi, count, n, I	letter followed by letters and digits.
Number	3.14159, 0, 6.02e23	Any numeric constant
Literal	"Darshan Institute"	Any character between “ and “ except “

Table 2.1. Examples of Tokens

Example:

total = sum + 12.5

Tokens are: total (id),

= (relation)

Sum (id)

+ (operator)

12.5 (num)

Lexemes are: total, =, sum, +, 12.5

3. What is input buffering? Explain technique of buffer pair. OR Which technique is used for speeding up the lexical analyzer?

There are mainly two techniques for input buffering,

- ✓ Buffer pair
- ✓ Sentinels

1. **Buffer pair:**

- The lexical analysis scans the input string from left to right one character at a time.
- So, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- We use a buffer divided into two N-character halves, as shown in figure 2.2. N is the number of character on one disk block.

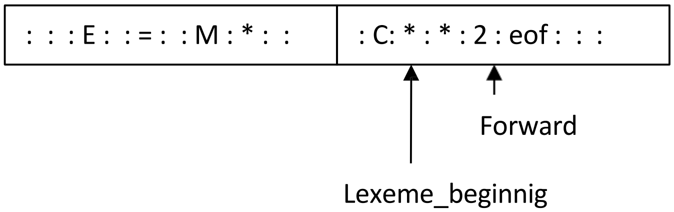


Fig. 2.2 An input buffer in two halves

- We read N input character into each half of the buffer.
- Two pointers to the input are maintained and string between two pointers is the current lexemes.

- Pointer *Lexeme Begin*, marks the beginning of the current lexeme.
- Pointer *Forward,* scans ahead until a pattern match is found.
- If forward pointer is at the end of first buffer half then second is filled with N input character.
- If forward pointer is at the end of second buffer half then first is filled with N input character.

code to advance forward pointer is given below,

```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1;
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half;
end
else forward := forward + 1;
```

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, Lexeme Begin is set to the character immediately after the lexeme just found.

2. **Sentinels:**

- If we use the scheme of Buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers; if we do, then we must reload the other buffer. Thus, for each character read, we make two tests.
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF.**

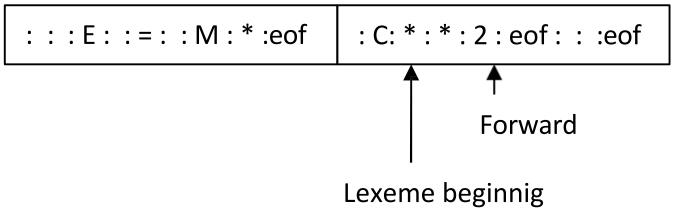


Fig.2.3. Sentinels at end of each buffer half

- Look ahead code with sentinels is given below:

```
forward  := forward + 1;
if forward  = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward  := forward + 1;
    end
    else if forward at the second half then begin
        reload first half;
```

```
                move forward to beginning of first half;
            end
            else terminate lexical analysis;
        end;
```

4. Specification of token.

Strings and languages

Terms for a part of string

Term	Definition
Prefix of S	A string obtained by removing zero or more trailing symbol of string S. e.g., ban is prefix of banana.
Suffix of S	A string obtained by removing zero or more leading symbol of string S. e.g., nana is suffix of banana.
Sub string of S	A string obtained by removing prefix and suffix from S. e.g., nan is substring of banana.
Proper prefix, suffix and substring of S	Any nonempty string x that is respectively prefix, suffix or substring of S, such that s≠x
Subsequence of S	A string obtained by removing zero or more not necessarily contiguous symbol from S. e.g., baaa is subsequence of banana.

Table 2.2. Terms for a part of a string

Operation on languages

Definition of operation on language

Operation	Definition
Union of L and M Written L U M	L U M = {s | s is in L or s is in M }
concatenation of L and M Written LM	LM = {st | s is in L and t is in M }
Kleene closure of L written L*	L* denotes “zero or more concatenation of” L.
Positive closure of L written L+	L+ denotes “one or more concatenation of” L.

Table 2.3. Definitions of operations on languages

5. Regular Expression & Regular Definition.

Regular Expression

1. 0 or 1
   0+1
2. 0 or 11 or 111
   0+11+111

3. Regular expression over ∑={a,b,c} that represent all string of length 3.
   (a+b+c)(a+b+c)(a+b+c)
4. String having zero or more a.
   a*
5. String having one or more a.
   a^+
6. All binary string.
   (0+1)*
7. 0 or more occurrence of either a or b or both
   (a+b)*
8. 1 or more occurrence of either a or b or both
   $(a+b)^+$
9. Binary no. end with 0
   (0+1)*0
10. Binary no. end with 1
    (0+1)*1
11. Binary no. starts and end with 1.
    1(0+1)*1
12. String starts and ends with same character.
    0(0+1)*0 or a(a+b)*a
    1(0+1)*1 b(a+b)*b

13. All string of a and b starting with a
    a(a/b)*
14. String of 0 and 1 end with 00.
    (0+1)*00
15. String end with abb.
    (a+b)*abb
16. String start with 1 and end with 0.
    1(0+1)*0
17. All binary string with at least 3 characters and 3rd character should be zero.
    (0+1)(0+1)0(0+1)*
18. Language which consist of exactly two b's over the set ∑={a,b}
    a*ba*ba*
19. ∑={a,b} such that 3rd character from right end of the string is always a.
    (a+b)*a(a+b)(a+b)
20. Any no. of a followed by any no. of b followed by any no. of c.
    a*b*c*
21. It should contain at least 3 one.
    (0+1)*1(0+1)*1(0+1)*1(0+1)*
22. String should contain exactly Two 1's
    0*10*10*
23. Length should be at least be 1 and at most 3.

(0+1) + (0+1) (0+1) + (0+1) (0+1) (0+1)

24. No.of zero should be multiple of 3
    (1*01*01*01*)*+1*
25. ∑={a,b,c} where a are multiple of 3.
    ((b+c)*a (b+c)*a (b+c)*a (b+c)*)*
26. Even no. of 0.
    (1*01*01*)*
27. Odd no. of 1.
    0*(10*10*)*10*
28. String should have odd length.
    (0+1)((0+1)(0+1))*
29. String should have even length.
    ((0+1)(0+1))*
30. String start with 0 and has odd length.
    0((0+1)(0+1))*
31. String start with 1 and has even length.
    1(0+1)((0+1)(0+1))*
32. Even no of 1
    (0*10*10*)*
33. String of length 6 or less
    $(0+1+^\wedge)^6$
34. String ending with 1 and not contain 00.
    $(1+01)^+$
35. All string begins or ends with 00 or 11.
    (00+11)(0+1)*+(0+1)*(00+11)
36. All string not contains the substring 00.
    (1+01)* (^+0)
37. Language of all string containing both 11 and 00 as substring.
    ((0+1)*00(0+1)*11(0+1)*)+ ((0+1)*11(0+1)*00(0+1)*)
38. Language of C identifier.
    (_+L)(_+L+D)*

Regular Definition

- A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
- Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letters and digits beginning with a letters.

  letter → A | B | . . . | Z | a | b | . . . | z
  digit → 0 | 1 | 2 | . . . | 9
  id → letter (letter | digit)*

- The regular expression id is the pattern for the identifier token and defines letter and digit. Where letter is a regular expression for the set of all upper-case and lower case letters in the alphabet and digit is the regular expression for the set of all decimal digits.

6. Reorganization of Token.

- Here we address how to recognize token.
- We use the language generated by following grammar,

stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ϵ

expr → term **relop** term
| term
term → **id** | **num**

- Where the terminals if, then, else, relop, id and num generates the set of strings given by the following regular definitions,

if → if
then → then
relop → < | <= | = | <> | > | >=
letter → A | B | . . . | Z | a | b | . . . | z
digit → 0 | 1 | 2 | . . . | 9
id → letter (letter | digit)*
num → **digit**$^{+}$ ()?(E(+/-)?**digit**$^{+}$)?

- For this language the lexical analyzer will recognize the keyword if, then, else, as well as the lexeme denoted by **relop, id** and **num**.
- num represents the unsigned integer and real numbers of pascal.
- Lexical analyzer will isolate the next token in the input buffer and produces token and attribute value as an output.

7. Transition Diagram.

- A stylized flowchart is called transition diagram.
- Positions in a transition diagram are drawn as a circle and are called states.
- States are connected by arrows called edges.
- Edge leaving state have label indicating the input character.
- The transition diagram for unsigned number is given in Fig.2.4.

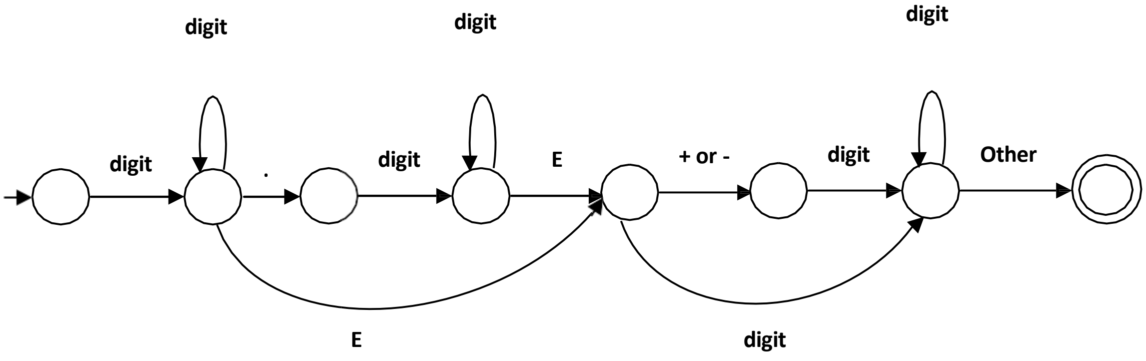


Fig. 2.4. Transition diagram for unsigned number

- Transition diagram for the token **relop** is shown in figure 2.5.

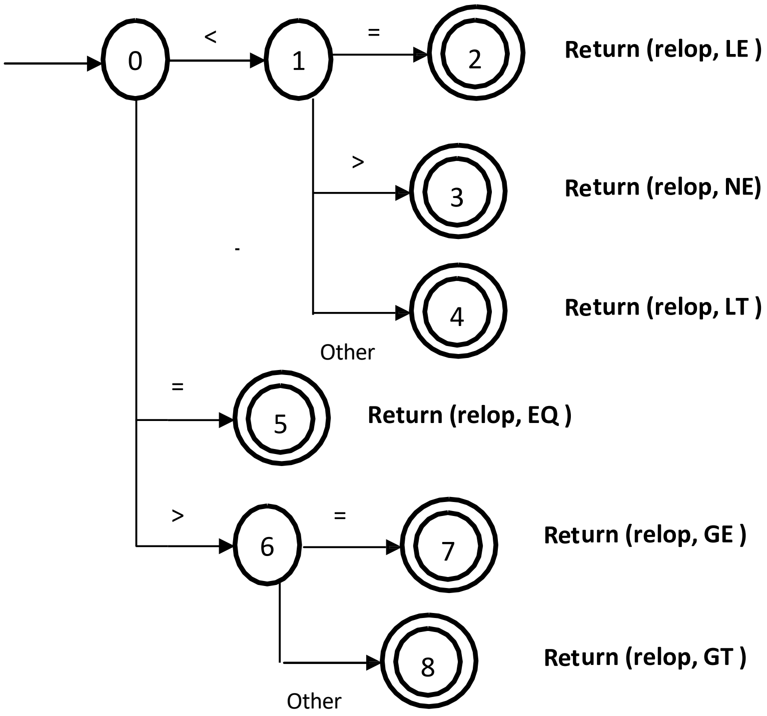


Fig. 2.5. Transition diagram for relational operator

8. Explain Finite automata. (NFA & DFA)

- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.
- A finite Automata or finite state machine is a 5-tuple(S,∑,S₀,F,δ) where

  S is finite set of states
  ∑ is finite alphabet of input symbol
  S_0 ∈ S (Initial state)
  F (set of accepting states)
  δ is a transition function

- There are two types of finite automata,

1. Deterministic finite automata (DFA) have for each state (circle in the diagram) exactly one edge leaving out for each symbol.
2. Nondeterministic finite automata (NFA) are the other kind. There are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with ε.

9. Conversion from NFA to DFA using Thompson's rule.

Ex:1 (a+b)*abb

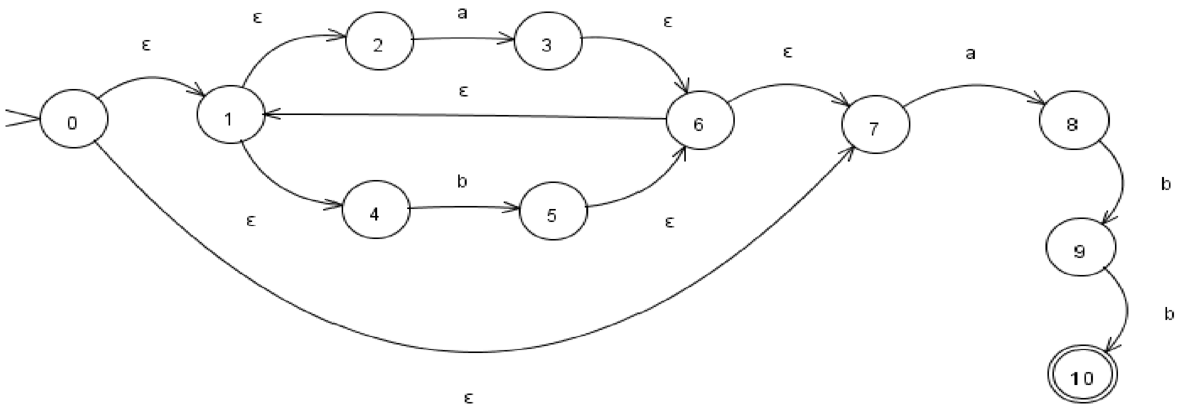


Fig. 2.6. NFA for (a+b)*abb

- ε – closure (0) = {0,1,2,4,7} ---- Let A
- Move(A,a) = {3,8}
  ε – closure (Move(A,a)) = {1,2,3,4,6,7,8}---- Let B
  Move(A,b) = {5}
  ε – closure (Move(A,b)) = {1,2,4,5,6,7}---- Let C

- Move(B,a) = {3,8}
  ε – closure (Move(B,a)) = {1,2,3,4,6,7,8}---- B
  Move(B,b) = {5,9}
  ε – closure (Move(B,b)) = {1,2,4,5,6,7,9}---- Let D

- Move(C,a) = {3,8}
  ε – closure (Move(C,a)) = {1,2,3,4,6,7,8}---- B
  Move(C,b) = {5}
  ε – closure (Move(C,b)) = {1,2,4,5,6,7}---- C

- Move(D,a) = {3,8}
  ε – closure (Move(D,a)) = {1,2,3,4,6,7,8}-----B
  Move(D,b) = {5,10}
  ε – closure (Move(D,b)) = {1,2,4,5,6,7,10}---- Let E

- Move(E,a) = {3,8}
  ε – closure (Move(E,a)) = {1,2,3,4,6,7,8}---- B
  Move(E,b) = {5}
  ε – closure (Move(E,b)) = {1,2,4,5,6,7} ---- C

States	**a**	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Table 2.4. Transition table for (a+b)*abb

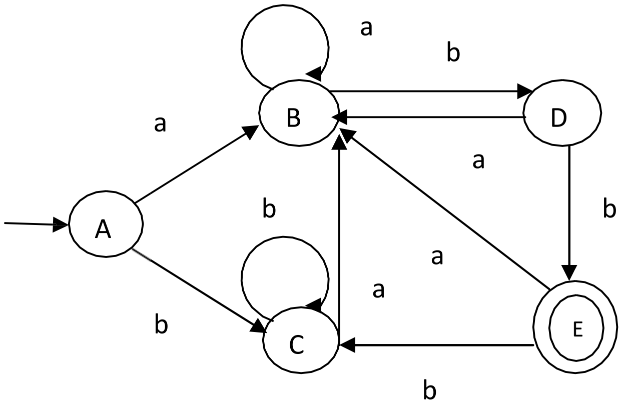


Fig.2.7. DFA for (a+b)*abb

DFA Optimization

- Algorithm to minimizing the number of states of a DFA

1. *Construct an initial partition Π of the set of states with two groups: the accepting states F and the non-accepting states S - F.*
2. *Apply the repartition procedure to Π to construct a new partition Πnew.*
3. *If Π new = Π, let Πfinal= Π and continue with step (4). Otherwise, repeat step (2) with Π = Πnew.*

```
for each group G of Π do begin
        partition G into subgroups such that two states s and t
                of G are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π.
        replace G in Πnew by the set of all subgroups formed
```

4. *Choose one state in each group of the partition Πfinal as the representative for that group. The representatives will be the states of M'. Let s be a representative state, and suppose on input a there is a transition of M from s to t. Let r be the representative of t's group. Then M' has a transition from s to r on a. Let the start state of M' be the representative of the group containing start state s_0 of M, and let the accepting states of M' be the representatives that are in F.*
5. *If M' has a dead state d (non-accepting, all transitions to self), then remove d from M'. Also remove any state not reachable from the start state.*

Example: Consider transition table of above example and apply algorithm.

- Initial partition consists of two groups (E) accepting state and non accepting states

(ABCD).

- E is single state so, cannot be split further.
- For (ABCD), on input a each of these state has transition to B. but on input b, however A, B and C go to member of the group (ABCD), while D goes to E, a member of other group.
- Thus, (ABCD) split into two groups, (ABC) and (D). so, new groups are (ABC)(D) and (E).
- Apply same procedure again no splitting on input a, but (ABC) must be splitting into two group (AC) and (B), since on input b, A and C each have a transition to C, while B has transition to D. so, new groups (AC)(B)(D)(E).
- Now, no more splitting is possible.
- If we chose A as the representative for group (AC), then we obtain reduced transition table shown in table 2.5,

States	**a**	**b**
A	B	C
B	B	D
D	B	E
E	B	C

Table 2.5. Optimized Transition table for (a+b)*abb

10. Conversion from Regular Expression to DFA without constructing NFA.

Ex:1 (a+b)*abb#

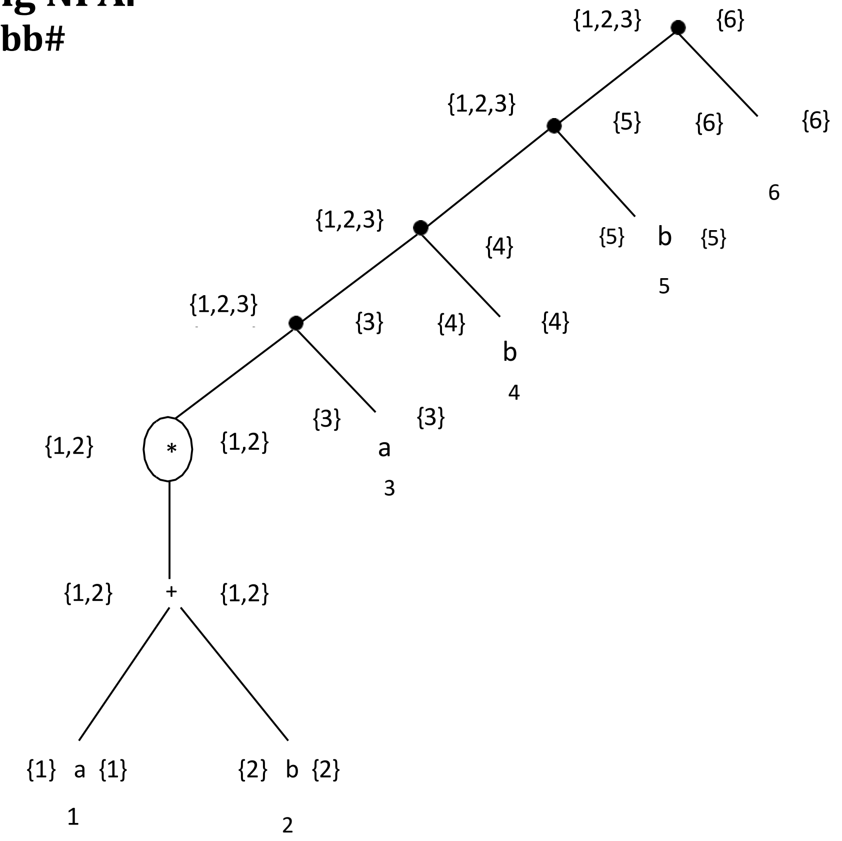


Fig.2.8. Syntax tree for (a+b)*abb#

- To find followpos traverse concatenation and star node in depth first search order.

1.
i=lastpos(c1)={5}
firstpos(c2)={6}
followpos(i)=firstpos(c2)
followpos(5)={6}
	2.
	i=lastpos(c1)={4}
	firstpos(c2)={5}
	followpos(i)=firstpos(c2)
	followpos(4)={5}
		3.
		i=lastpos(c1)={3}
		firstpos(c2)={4}
		followpos(i)=firstpos(c2)
		followpos(3)={4}

4.
i=lastpos(c1)={1,2}
firstpos(c2)={3}
followpos(i)=firstpos(c2)
followpos(1)={3}
followpos(2)={3}
	5.
	i=lastpos(c1)={1,2}
	firstpos(c1)={1,2}
	followpos(i)=firstpos(c1)
	followpos(1)={1,2}
	followpos(2)={1,2}	

Position	Followpos(i)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}

Table 2.6. follow pos table

Construct DFA

Initial node = firstpos (root node)= {1,2,3} -- A

δ(A,a) = followpos(1) U followpos(3)
= {1,2,3} U {4}
= {1,2,3,4} --- B
δ (A,b) = followpos(2)
= {1,2,3} ---- A

δ (B,a) = followpos(1) U followpos(3)
= {1,2,3} U {4}
= {1,2,3,4} --- B
δ(B,b) = followpos(2) U followpos(4)
= {1,2,3} U {5}
= {1,2,3,5} --- C

δ(C,a) = followpos(1) U followpos(3)
= {1,2,3} U {4}
= {1,2,3,4} --- B
δ(C,b) = followpos(2) U followpos(5)
= {1,2,3} U {6}

= {1,2,3,6} --- D

δ(D,a) = followpos(1) U followpos(3)
= {1,2,3} U {4}
= {1,2,3,4} --- B
δ(D,b) = followpos(2) = {1,2,3}---A

Transition Table		
States	a	b
A	B	A
B	B	C
C	B	D
D	B	A

Table 2.7. Transition table for (a+b)*abb

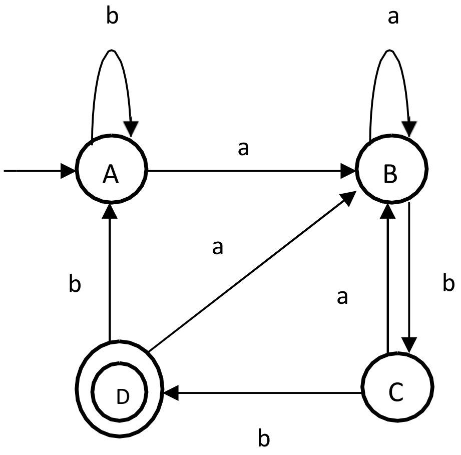


Fig.2.9. DFA for (a+b)*abb

Ex:2 a*b*a(a\b)*b*a#

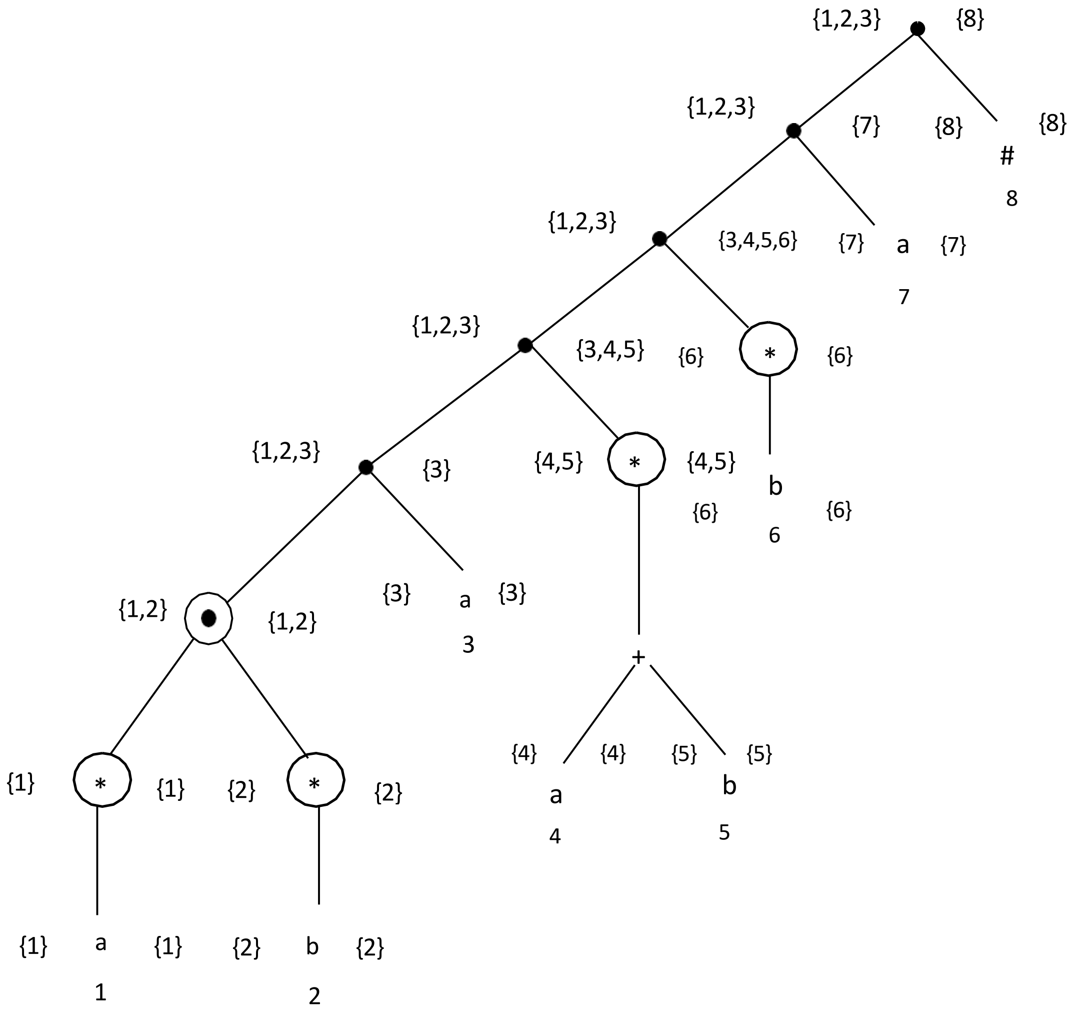


Fig.2.10. Syntax tree for a*b*a(a\b)*b*a#

- To find followpos traverse concatenation and star node in depth first search order

1.
i=lastpos(c1)={7}
firstpos(c2)={8}
followpos(i)=firstpos(c2)
followpos(7)={8}
	2.
	i=lastpos(c1)={3,4,5,6}
	firstpos(c2)={7}
	followpos(i)=firstpos(c2)
	followpos(3)={7}
	followpos(4)={7}
	followpos(5)={7}
	followpos(6)={7}
		3.
		i=lastpos(c1)={3,4,5}
		firstpos(c2)={6}
		followpos(i)=firstpos(c2)
		followpos(3)={6}
		followpos(4)={6}
		followpos(5)={6}

4.
i=lastpos(c1)={3}
firstpos(c2)={4,5}
	5.
	i=lastpos(c1)={1,2}
	firstpos(c2)={3}
		6.
		i=lastpos(c1)={1}
		firstpos(c2)={2}

followpos(i)=firstpos(c2)	followpos(i)=firstpos(c2)	followpos(i)=firstpos(c2)

followpos(3)={4,5}	followpos(1)={3}
	followpos(2)={3}
		followpos(1)={2}

7.
i=lastpos(c1)={1}
firstpos(c1)={1}
followpos(i)=firstpos(c1)
followpos(1)={1}
	8.
	i=lastpos(c1)={2}
	firstpos(c1)={2}
	followpos(i)=firstpos(c1)
	followpos(2)={2}
		9.
		i=lastpos(c1)={4,5}
		firstpos(c1)={4,5}
		followpos(i)=firstpos(c1)
		followpos(4)={4,5}
		followpos(5)={4,5}

9.
i=lastpos(c1)={6}
firstpos(c1)={6}
followpos(i)=firstpos(c1)
followpos(6)={6}		

n	followpos(n)
1	{1,2,3}
2	{2,3}
3	{4,5,6,7}
4	{4,5,6,7}
5	{4,5,6,7}
6	{6,7}
7	{8}

Table 2.8. follow pos table

Construct DFA

Initial node = firstpos (root node)= {1,2,3} -- A

δ (A,a) = followpos(1) U followpos(3)
= {1,2,3} U {4,5,6,7}
= {1,2,3,4,5,6,7} ---B

δ (A,b) = followpos(2)
= {2,3}---- C

δ (B,a) = followpos(1) U followpos(3) U followpos(4) U followpos(7)
= {1,2,3} U {4,5,6,7} U {8} U {4,5,6,7}
= {1,2,3,4,5,6,7,8} --- D

δ (B,b) = followpos(2) U followpos(5) U followpos(6)
={2,3} U {4,5,6,7} U {4,5,6,7}
= {2,3,4,5,6,7}--- E

δ (C,a) = followpos(3)
= {4,5,6,7} ---F

δ (C,b) = followpos(2)
= {2,3} ---C

δ (D,a) = followpos(1) U followpos(3) U followpos(7) U followpos(4)
= {1,2,3} U {4,5,6,7} U {8} U {4,5,6,7}
= {1,2,3,4,5,6,7,8} ---D
δ (D,b) = followpos(2) U followpos(5) U followpos(6)
={2,3} U {4,5,6,7} U {4,5,6,7}
= {2,3,4,5,6,7} ---E

δ (E,a) = followpos(3) U followpos(4) U followpos(7)
={4,5,6,7} U {4,5,6,7} U {8}
= {4,5,6,7,8} ---G
δ (E,b) = followpos(2) U followpos(5) U followpos(6)
={2,3} U {4,5,6,7} U {4,5,6,7}
= {2,3,4,5,6,7} ---E

δ (F,a) = followpos(4) U followpos(7)
= {4,5,6,7} U {8}
= {4,5,6,7,8} ---G
δ (F,b) = followpos(5) U followpos(6)
={4,5,6,7} U {4,5,6,7}
= {4,5,6,7} ---F

δ (G,a) = followpos(4) U followpos(7)
= {4,5,6,7} U {8}
= {4,5,6,7,8} ---G
δ (G,b) = followpos(5) U followpos(6)
={4,5,6,7} U {4,5,6,7}
= {4,5,6,7} ---F

Transition table:

Transition table		
	a	b
A	B	C
B	D	E
C	F	C
D	D	E
E	G	E
F	G	F
G	G	F

Table 2.9. Transition table for a*b*a(a\b)*b*a#

DFA:

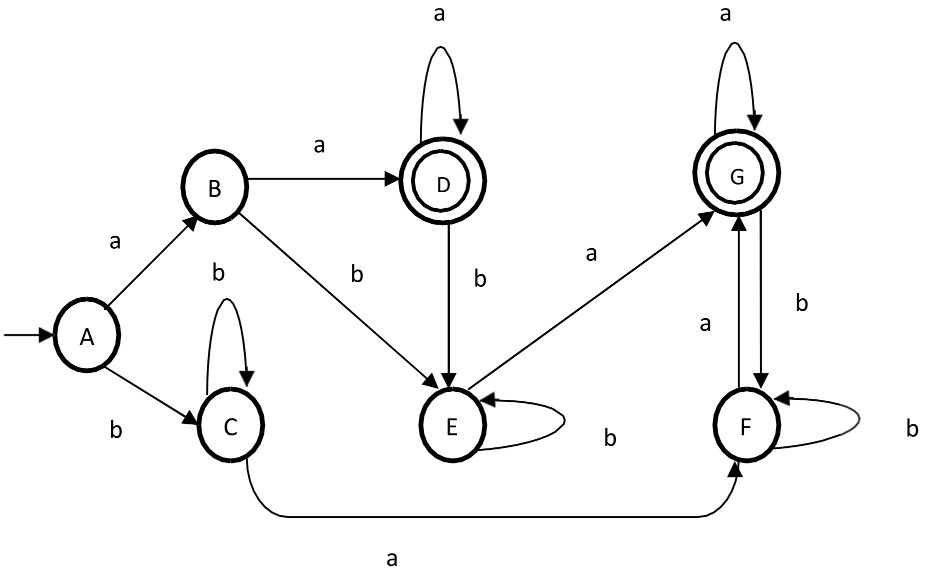


Fig.2.11. DFA for a*b*a(a\b)*b*a#

1. Role of Parser.

- In our compiler model, the parser obtains a string of tokens from lexical analyzer, as shown in fig. 3.1.1.

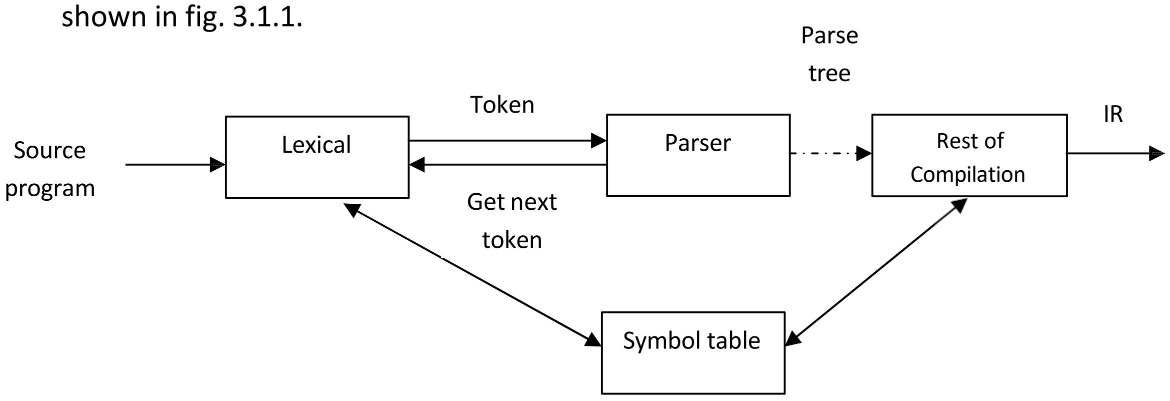


Fig.3.1.1. Position of parser in compiler model

- We expect the parser to report any syntax error. It should also recover from commonly occurring errors.

- The methods commonly used for parsing are classified as a top down or bottom up parsing.
- In top down parsing parser, build parse tree from top to bottom, while bottom up parser starts from leaves and work up to the root.
- In both the cases, the input to the parser is scanned from left to right, one symbol at a time.
- We assume the output of parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer.

2. Difference between syntax tree and Parse tree.

Syntax tree v/s Parse tree

No.	Parse Tree	Syntax Tree
1	Interior nodes are non-terminals, leaves are terminals.	Interior nodes are “operators”, leaves are operands.
2	Rarely constructed as a data structure.	When representing a program in a tree structure usually use a syntax tree.
3	Represents the concrete syntax of a program.	Represents the abstract syntax of a program (the semantics).

Table 3.1.1. Difference between syntax tree & Parse tree

- Example: Consider grammar following grammar,
  E→ E+ E
  E→ E* E

E→ Id

- Figure 3.1.2. Shows the syntax tree and parse tree for string id + id*id.

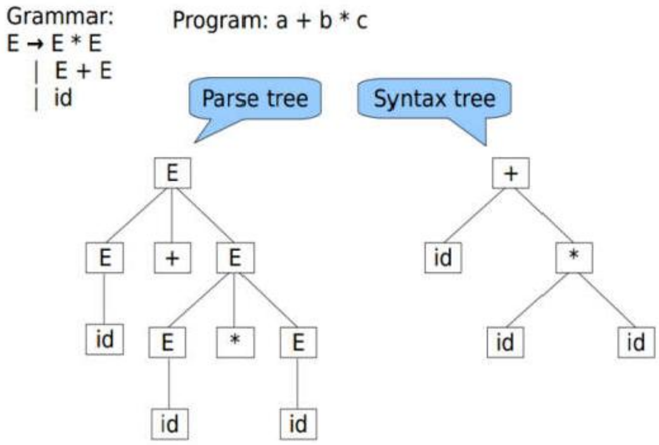


Fig.3.1.2. Syntax tree and parse tree

3. Types of Derivations. (Leftmost & Rightmost)

There are mainly two types of derivations,

1. Leftmost derivation
2. Rightmost derivation

Let Consider the grammar with the production S→S+S | S-S | S*S | S/S |(S)| a

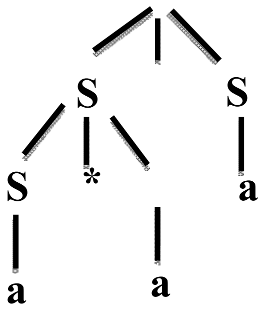
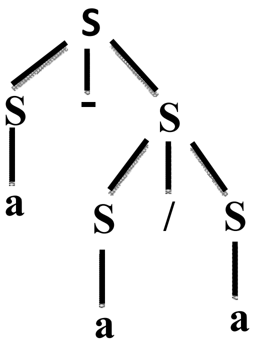
Left Most Derivation	Right Most Derivation
A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.	A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
Consider string a*a-a S→S-S S*S-S a*S-S a*a-S a*a-a	Consider string: a-a/a S→S-S S-S/S S-S/a S-a/a a-a/a
Equivalent left most derivation tree 	Equivalent Right most derivation tree 

Table 3.1.2. Difference between Left most Derivation & Right most Derivation

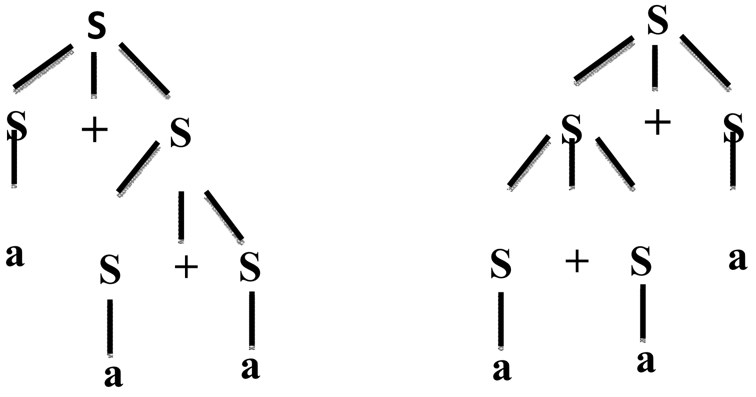
4. Explain Ambiguity with example.

- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

1) Prove that given grammar is ambiguous. S→S+S / S-S / S*S / S/S /(S)/a (IMP)

String : a+a+a

S→S+S	S→S+S
a+S	S+S+S
a+S+S	a+S+S
a+a+S	a+a+S
a+a+a	a+a+a



- Here we have two left most derivation hence, proved that above grammar is ambiguous.

2) Prove that S->a | Sa | bSS | SSb | SbS is ambiguous

String: baaab

S→bSS	S→SSb
baS	bSSSb
baSSb	baSSb
baaSb	baaSb
baaab	baaab

- Here we have two left most derivation hence, proved that above grammar is ambiguous.

5. Elimination of 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 grammar, so a transformation that eliminates left recursion is needed.

Algorithm to eliminate left recursion

1. *Assign an ordering $A_1,…,A_n$ to the nonterminals of the grammar.*

```
2. for i:=1 to n do
   begin
           for j:=1 to i−1 do
           begin
                     replace each production of the form $A_i \rightarrow A_i \gamma$
```

```
                by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ..... \mid \delta_k\gamma$
                where $Aj \rightarrow \delta_1 \mid \delta_2 \mid ..... \mid \delta_k$ are all the current Aj productions;
        end

        eliminate the intermediate left recursion among the $A_i$-productions
    end
```

- Example 1 : Consider the following grammar,
  E→E+T/T
  T→T*F/F
  F→(E)/id
  Eliminate immediate left recursion from above grammar then we obtain,
  E→TE'
  E'→+TE' | ε
  T→FT'
  T'→*FT' | ε
  F→(E) | id
- Example 2 : Consider the following grammar,
  S→Aa | b
  A→Ac | Sd | ε
  Here, non terminal S is left recursive because S→Aa→Sda, but it is not immediately left recursive.
  S→ Aa | b
  A→Ac | Aad | bd | ε
  Now, remove left recursion
  S→ Aa | b
  A→ bdA' | A'
  A→ cA' | adA' | ε

6. Left factoring.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

  Algorithm to left factor a grammar

  Input: Grammar G

  Output: An equivalent left factored grammar.

  1. *For each non terminal A find the longest prefix α common to two or more of its alternatives.*
  2. *If α!= E, i.e., there is a non trivial common prefix, replace all the A productions A→ $\alpha\beta_1$| α $_2$|.............| $\alpha\beta_n$| γ where γ represents all alternatives that do not begin with α by*
     A==> α A'| γ
     A'==>β_1|β_2| ....|β_n

Here A' is new non terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

- EX1: Perform left factoring on following grammar,
  A→ xByA | xByAzA | a
  B→b
  Left factored, the grammar becomes
  A→ xByAA' | a
  A'→zA | Є
  B→ b
- EX2: Perform left factoring on following grammar,
  S→iEtS | iEtSeS | a
  E→b
  Left factored, the grammar becomes
  S→ iEtSS' | a
  S'→eS | Є
  E→b

7. Types of Parsing.

- Parsing or syntactic analysis is the process of analyzing a string of symbols according to the rules of a formal grammar.
- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid sentence of given grammar. Types of parsing are,
  1. **Top down parsing**: In top down parsing parser build parse tree from top to bottom.
  2. **Bottom up parsing**: While bottom up parser starts from leaves and work up to the root.

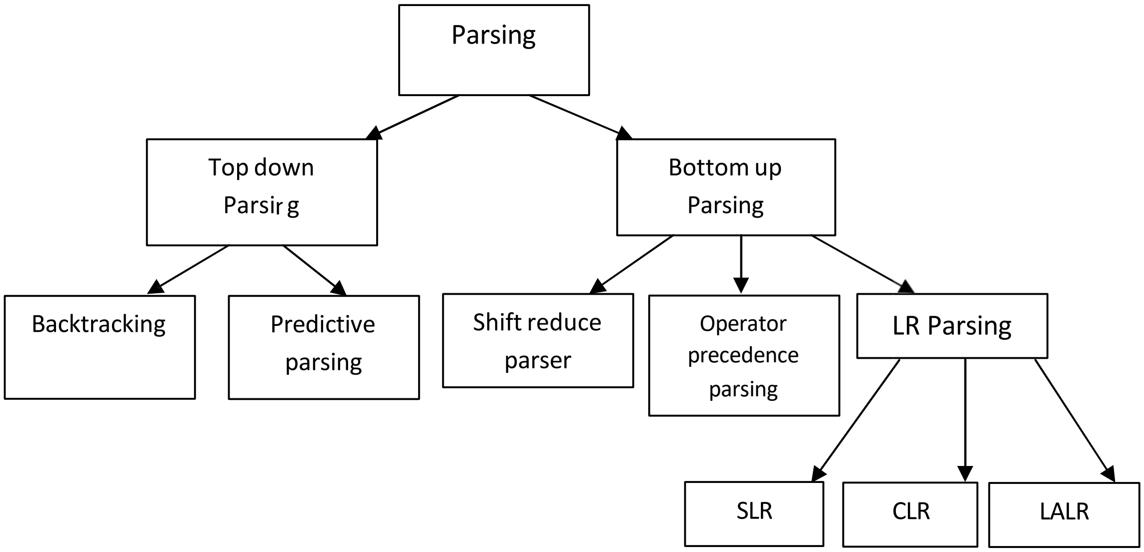


Fig.3.1.3 Parsing Techniques

8. Recursive Decent Parsing.

- A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive decent parser.
- There is a procedure for each non terminal in the grammar.
- Consider RHS of any production rule as definition of the procedure.
- As it reads expected input symbol, it advances input pointer to next position.

Example:

E→ T {+T}*
T→ V{*V}*
V→ id

```
Procedure proc_E: (tree_root);
    var
        a, b : pointer to a tree node;
    begin
        proc_T(a);
        While (nextsymb = '+') do
            nextsymb = next source symbol;
            proc_T(b);
            a= Treebuild ('+', a, b);
        tree_root= a;
        return;
end proc_E;

Procedure proc_T: (tree_root);
    var
        a, b : pointer to a tree node;
    begin
        proc_V(a);
        While (nextsymb = '*') do
            nextsymb = next source symbol;
            proc_V(b);
            a= Treebuild ('*', a, b);
        tree_root= a;
        return;
end proc_T;

Procedure proc_V: (tree_root);
    var
        a : pointer to a tree node;
    begin
        If (nextsymb = 'id') then
            nextsymb = next source symbol;
            tree_root= tree_build(id, , );
```

```
else print "Error";
```

```
            return;
    end proc_V;
```

Advantages

- It is exceptionally simple.
- It can be constructed from recognizers simply by doing some extra work.

Disadvantages

- It is time consuming method.
- It is difficult to provide good error messages.

9. Predictive parsing. OR LL(1) Parsing.

- This top-down parsing is non-recursive. LL (1) – the first L indicates input is scanned from left to right. The second L means it uses leftmost derivation for input string and 1 means it uses only input symbol to predict the parsing process.
- The block diagram for LL(1) parser is given below,

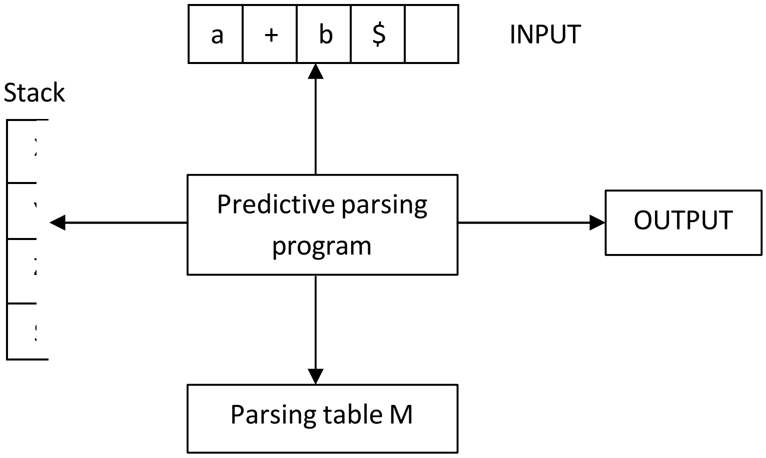


Fig.3.1.4 Model of nonrecursive predictive parser

- The data structure used by LL(1) parser are input buffer, stack and parsing table.
- The parser works as follows,
- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols parsing action can be determined.
- The parser consult the table M[A, a] each time while taking the parsing actions hence this type of parsing method is also called table driven parsing method.
- The input is successfully parsed if the parser reaches the halting configuration. When the stack is empty and next token is $ then it corresponds to successful parsing.

Steps to construct LL(1) parser

1. Remove left recursion / Perform left factoring.
2. Compute FIRST and FOLLOW of nonterminals.
3. Construct predictive parsing table.
4. Parse the input string with the help of parsing table.

Example:

E→E+T/T
T→T*F/F
F→(E)/id

Step1: Remove left recursion

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

Step2: Compute FIRST & FOLLOW

	FIRST	**FOLLOW**
E	{(,id}	{$,)}
E’	{+,ϵ}	{$,)}
T	{(,id}	{+,$,)}
T’	{*,ϵ}	{+,$,)}
F	{(,id}	{*,+,$,)}

Table 3.1.3 first & follow set

Step3: Predictive Parsing Table

	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)		

Table 3.1.4 predictive parsing table

Step4: Parse the string

Stack	Input	Action
$E	id+id*id$	
$E’T	id+id*id$	E→TE’
$ E’T’F	id+id*id$	T→FT’
$ E’T’id	id+id*id$	F→id
$ E’T’	+id*id$	
$ E’	+id*id$	T’→ ϵ
$ E’T+	+id*id$	E’→+TE’
$ E’T	id*id$	
$ E’T’F	id*id$	T→FT’
$ E’T’id	id*id$	F→id
$ E’T’	*id$	
$ E’T’F*	*id$	T’→*FT’
$ E’T’F	id$	
$ E’T’id	id$	F→id

$ E'T'	$	
$ E'	$	T'→ ϵ
$	$	E'→ ϵ

Table 3.1.5. moves made by predictive parse

10. Error recovery in predictive parsing.

- Panic mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing token appears.
- Its effectiveness depends on the choice of synchronizing set.
- Some heuristics are as follows:
  - ✓ Insert 'synch' in FOLLOW symbol for all non terminals. 'synch' indicates resume the parsing. If entry is "synch" then non terminal on the top of the stack is popped in an attempt to resume parsing.
  - ✓ If we add symbol in FIRST (A) to the synchronizing set for a non terminal A, then it may be possible to resume parsing if a symbol in FIRST(A) appears in the input.
  - ✓ If a non terminal can generate the empty string, then the production deriving the ε can be used as a default.
  - ✓ If parser looks entry M[A,a] and finds that it is blank then i/p symbol a is skipped.
  - ✓ If a token on top of the stack does not match i/p symbol then we pop token from the stack.
- Consider the grammar given below:

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

- ✓ Insert 'synch' in FOLLOW symbol for all non terminals.

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

Table 3.1.6. Follow set of non terminals

NT	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E =>TE'			E=>TE'	synch	Synch
E'		E' => +TE'			E' => ε	E' => ε
T	T => FT'	synch		T=>FT'	Synch	synch
T'		T' => ε	T' =>* FT'		T' => ε	T' => ε
F	F => <id>	synch	Synch	F=>(E)	synch	synch

Table 3.1.7. Synchronizing token added to parsing table

Stack	Input	Remarks
$E	)id*+id$	Error, skip)
$E	id*+id$	
$E’ T	id*+id$	
$E’ T’ F	id*+id$	
$E’ T’ id	id*+id$	
$E’ T’	*+id$	
$E’ T’ F*	*+id$	
$E’ T’ F	+id$	Error, M[F,+]=synch
$E’ T’	+id$	F has been popped.
$E’	+id$	
$E’ T+	+id$	
$E’ T	id$	
$E’ T’ F	id$	
$E’ T’ id	id$	
$E’ T’	$	
$E’	$	
$	$	

Table 3.1.8. Parsing and error recovery moves made by predictive parser

11. Explain Handle and handle pruning.

Handle: A “handle” of a string is a substring of the string that matches the right side of a production, and whose reduction to the non terminal of the production is one step along the reverse of rightmost derivation.

Handle pruning: The process of discovering a handle and reducing it to appropriate Left hand side non terminal is known as handle pruning.

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	E→id
E+id2*id3	id2	E→id
E+E*id3	id3	E→id
E+E*E	E*E	E→ E*E
E+E	E+E	E→ E+E
E		

Table 3.1.9. Handles

12. Shift reduce Parsi g.

- The shift reduce parser performs following basic operations,
- Shift: Moving of the symbols from input buffer onto the stack, this action is called shift.
- Reduce: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.

- Accept: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.

- Error: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

  Example: Consider the following grammar,
  E→E + T | T
  T→T * F | F
  F→id
  Perform shift reduce parsing for string id + id * id.

Stack	Input buffer	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce F->id
$F	+id*id$	Reduce T->F
$T	+id*id$	Reduce E->T
$E	+id*id$	Shift
$E+	id*id$	shift
$E+ id	*id$	Reduce F->id
$E+F	*id$	Reduce T->F
$E+T	*id$	Shift
$E+T*	id$	Shift
$E+T*id	$	Reduce F->id
$E+T*F	$	Reduce T->T*F
$E+T	$	Reduce E->E+T
$E	$	Accept

Table 3.1.10. Configuration of shift reduce parser on input id + id*id

13. Operator Precedence Parsing.

- **Operator Grammar**: A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator precedence grammar.
- In operator precedence parsing, we define three disjoint precedence relations <·**,** ·>and = between certain pair of terminals.

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

Table 3.1.11. Precedence between terminal a & b

Leading:-

Leading of a nonterminal is the first terminal or operator in production of that nonterminal.

Trailing:-

Traling of a nonterminal is the last terminal or operator in production of that nonterminal

Example:

E→E+T/T

T→T*F/F
F→id

Step-1: Find leading and trailing of NT.

Leading	Trailing
(E)={+,*,id}	(E)={+,*,id}
(T)={*,id}	(T)={*,id}
(F)={id}	(F)={id}

Step-2:Establish Relation

1. a <· b
   Op · NT → Op <· Leading(NT)
   +T + <· {*,id}
   *F * <· {id}
2. a ·> b
   NT · Op →Traling(NT) ·> Op
   E+ {+,*, id} ·> +
   T* {*, id} ·> *
3. $ <· {+, *,id}
4. {+,*,id} ·> $

Step-3: Creation of table

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

Table 3.1.12. precedence table

Step-4: Parsing of the string using precedence table.

We will follow following steps to parse the given string,

1. Scan the input string until first ·> is encountered.
2. Scan backward until <· is encountered.
3. The handle is string between <· And ·>.

$ <· Id ·> + <· Id ·> * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F+ <· Id ·> * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F + F * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F + F * F $	Perform appropriate reductions of all non terminals.
$ E + T * F$	Remove all non terminal
$ +* $	Place relation between operators

$ <· + <· * >$	The * operator is surrounded by <· ·>. This

	indicates * becomes handle we have to reduce T*F.
$ <· + >$	+ becomes handle. Hence reduce E+T.
$ $	Parsing Done

Table 3.1.13. moves made by operator precedence parser

Operator Precedence Function

Algorithm for Constructing Precedence Functions

1. *Create functions f_a and g_a for each a that is terminal or $.*
2. *Partition the symbols in as many as groups possible, in such a way that f_a and g_b are in the same group if a =· b.*
3. *Create a directed graph whose nodes are in the groups, next for each symbols a and b do:*
   (a) if a <· b, place an edge from the group of g_b to the group of f_a.
   (b) if a ·> b, place an edge from the group of f_a to the group 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.*

- Using the algorithm leads to the following graph:

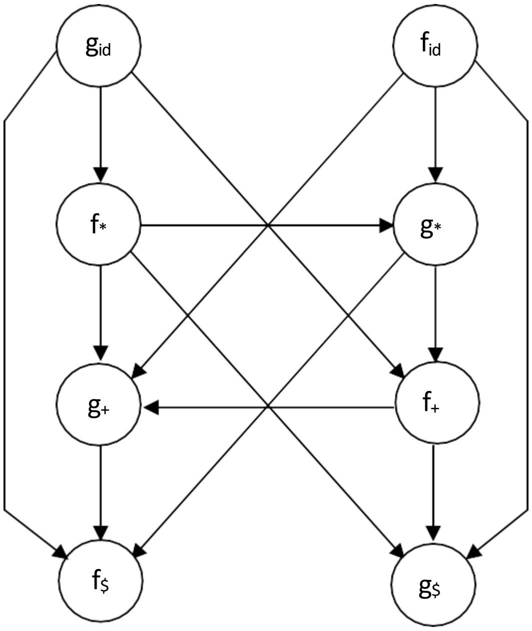


Fig. 3.1.5 Operator precedence graph

- From which we can extract the following precedence functions:

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

Table 3.1.14 precedence function

14. LR parsing.

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR(k) parsing; the "L" is for left to right scanning of input symbol, the "R" for constructing right most derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decision.
- There are three types of LR parsing,
  1. SLR (Simple LR)
  2. CLR (Canonical LR)
  3. LALR (Lookahead LR)
- The schematic form of LR parser is given in figure 3.1.6.
- The structure of input buffer for storing the input string, a stack for storing a grammar symbols, output and a parsing table comprised of two parts, namely action and goto.

Properties of LR parser

- LR parser can be constructed to recognize most of the programming language for which CFG can be written.
- The class of grammars that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- LR parser works using non back tracking shift reduce technique.
- LR parser can detect a syntactic error as soon as possible.

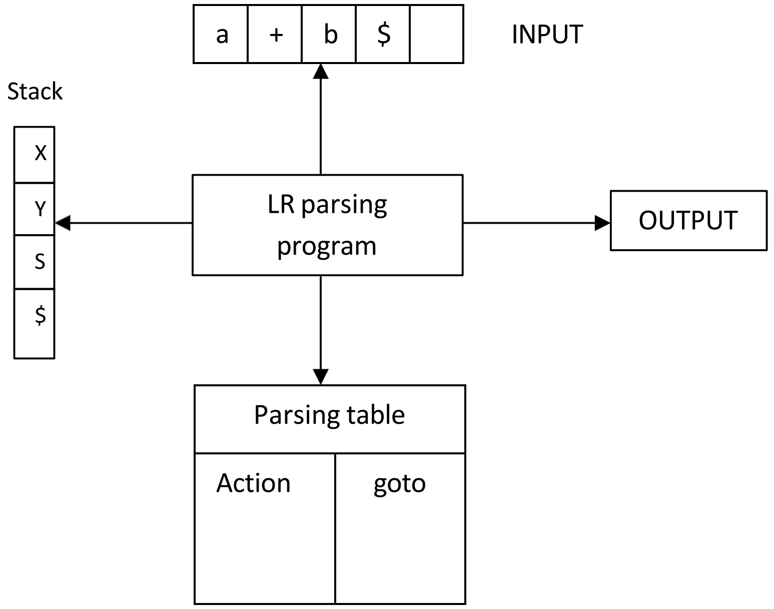


Fig.3.1.6. Model of an LR parser

15. Explain the following terms.

1. **Augmented grammar:** If grammar G having start symbol S then augmented grammar is the new grammar G' in which S' is a new start symbol such that S' -> .S.
2. **Kernel items:** It is a collection of items S'->.S and all the items whose dots are not at the

left most end of the RHS of the rule.

3. **Non-Kernel items**: It is a collection of items in which dots are at the left most end of the RHS of the rule.
4. **Viable prefix:** It is a set of prefix in right sentential form of the production A-> α, this set can appear on the stack during shift reduce action.

16. SLR Parsing.

- SLR means simple LR. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.
- SLR is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. It is quite efficient at finding the single correct bottom up parse in a single left to right scan over the input string, without guesswork or backtracking.
- The parsing table has two states (action, Go to).

  The parsing table has four values:

  ✓ Shift S, where S is a state
  ✓ reduce by a grammar production
  ✓ accept, and
  ✓ error

Example:

E→ E + T | T
T→ TF | F
F→ F * | a | b
Augmented grammar: E' → .E
Closure(I)

I_0 : E' → .E E → .E + T E → .T T→.TF T → .F F → .F * F → .a F → .b	I_1 : Go to (I_0,E) E' → E. E → E.+T	I_2 : Go to (I_0, T) E → T. T→ T.F F→.F * F→.a F→.b
I_3 : Go to (I_0,F) T→ F. F→F.*	I_4 : Go to (I_0,a) F→a.	I_5 : Go to (I_0,b) F→ b.
I_6 : Go to (I_1,+) E→ E+.T T→.TF T→ .F F→.F* F→ .F* F→.a	I_7 : Go to (I_2,F) T→TF. F→F.*	I_8 : Go to (I_3,*) F→F *.

F→.b		
I_9 : Go to(I_6,T) E→ E + T. T→T.F F→.F * F→.a F→.b		

Table 3.1.15. Canonical LR(0) collection

Follow:
Follow (E) : {+,$}
Follow (T) :{+,a,b,$}
Follow (F) : {+,*,a,b,$}

SLR parsing table :

Action						Go to		
state	**+**	*****	**a**	**b**	**$**	**E**	**T**	**F**
0			S_4	S_5		1	2	3
1	S_6				Accept			
2	R_2		S_4	S_5	R_2			7
3	R_4	S_8	R_4	R_4	R_4			
4	R_6	R_6	R_6	R_6	R_6			
5	R_6	R_6	R_6	R_6	R_6			
6			S_4	S_5			9	3
7	R_3	S_8	R_3	R_3	R_3			
8	R_5	R_5	R_5	R_5	R_5			
9	R_1		S_4	S_5	R_1			7