Programming Languages

Hierarchy of Programming Languages based on increasing machine independence includes the following:

1- Machine – level languages.

2- Assembly languages.

3- High – level or user oriented languages.

4- Problem - oriented language.

1- Machine level language: is the lowest form of computer. Each instruction in program is represented by numeric code, and numerical addresses are used throughout the program to refer to memory location in the computer's memory.

2- *Assembly language:* is essentially symbolic version of machine level language, each operation code is given a symbolic code such ADD for addition and MULT for multiplication.

3- *A high level language* such as Pascal, C.

4- *A problem oriented language* provides for the expression of problems in specific application or problem area. Examples of such as languages are SQL for database retrieval application problem oriented language.

Using a high-level language for programming has a large impact on how fast

programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought run on many different machines.

Language processing system

Actually we are trying to convert the high-level language (the source-code we written) to Low-level language (Machine Language). This process involves four stages and utilizes following 'tools':

1. Pre-processor
2. Compiler
3. Assembler
4. Loader/Linker

Compiler

Is a program that reads a program written in one language, (the source language) and translates into an equivalent program in another language (the target language) as shown in figure (1).

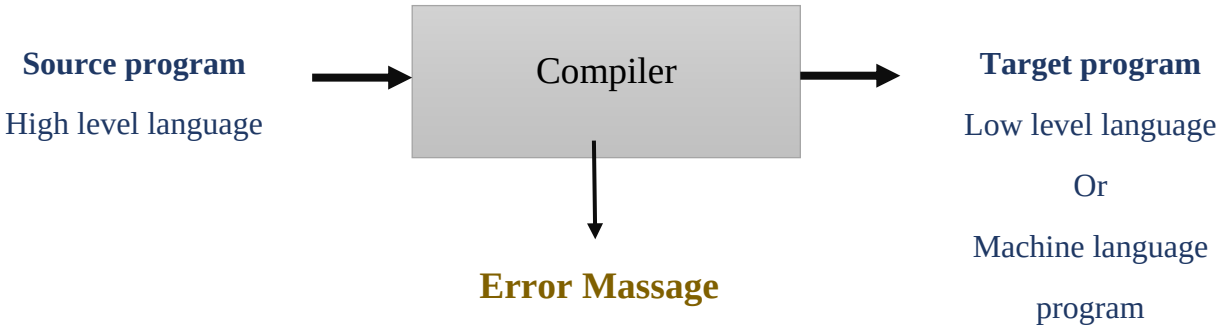


Figure (1): General structure of compiler program

Translator

A translator is program that takes as input a program written in a given programming language (the source program) and produce as output program in another language (the object or target program). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program as shown in figure (2).

If the source language being translated is assembly language, and the object program is machine language, the translator is called **Assembler.**

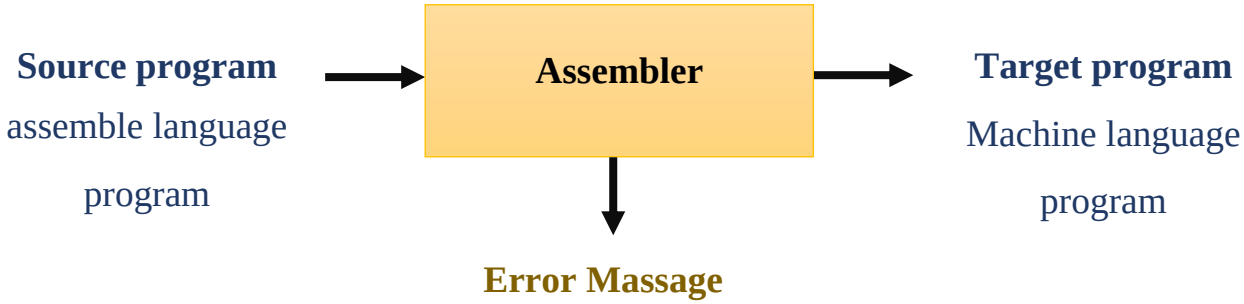


Figure (2) : General structure of Assembler program

A translator, which transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler**.

Another kind of translator called an **Interpreter** . An interpreter converts high level language into low level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. Compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted

programs are usually slower with respect to compiled ones. Figure (3) illustrate the interpretation process.

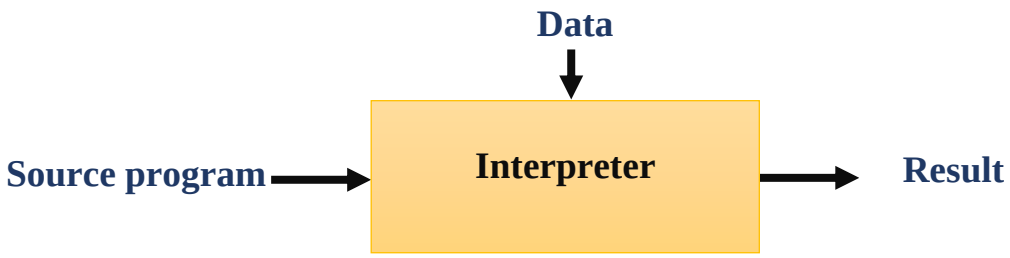


Figure (3) : Interpretation process

The Analysis - Synthesis model of compilation

There are two parts to compilation: analysis and synthesis.

1- Analysis phase (Front- end):- an intermediate representation is created from the give source code

1. Lexical Analyzer (scanner)
2. Syntax Analyzer (parser)
3. Semantic Analyzer
4. Intermediate Code generator

2- Synthesis Phase (Back-end) :- – equivalent target program is created from the intermediate representation. It has two components:

1. Code Optimizer
2. Code Generator

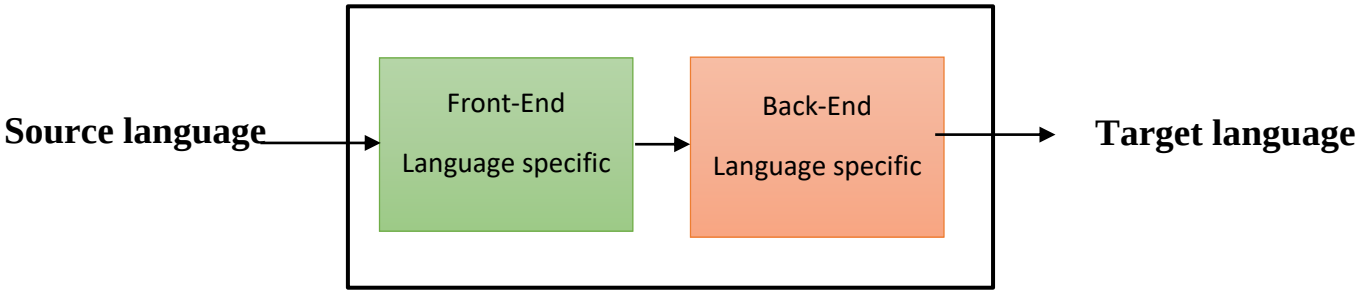


Figure (4) : The Analysis - Synthesis model of compilation

Phases of a Compiler:

A Compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is divided into a series of sub process called ***Phases.*** Figure (5) illustrated the compiler phases

- The different phases of a compiler are as follows

  Analysis Phases:

  1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Intermediate Code generator

  Synthesis Phases:

  5. Code Optimization
  6. Code generation

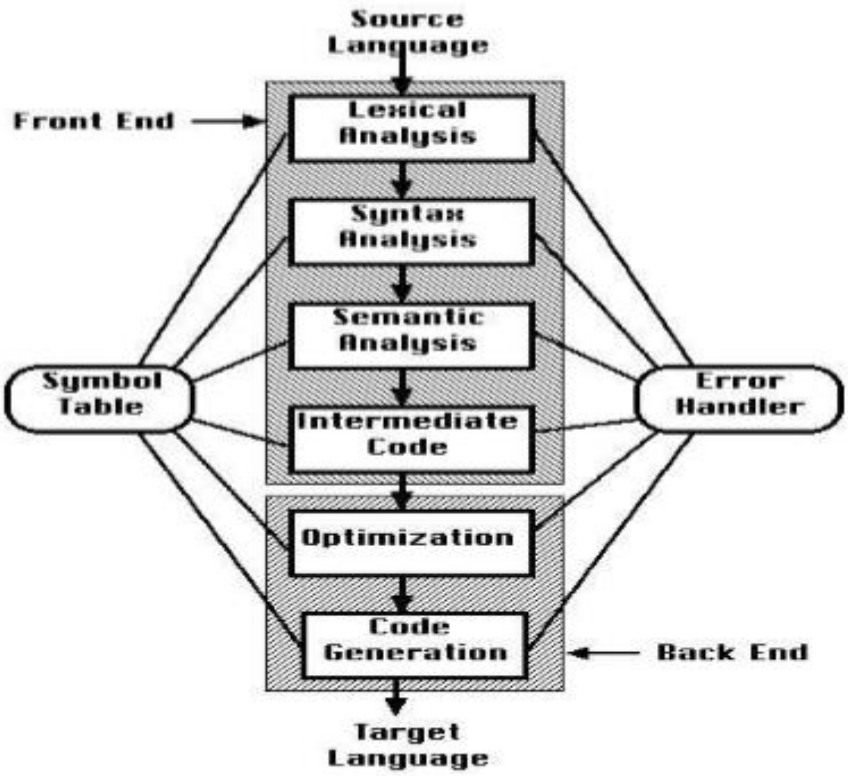


Figure (5) : Phases of Compiler

Compiler structure:

1- lexical analysis

1.1 The Role of lexical analysis

The lexical analyzer is the first stage of a compiler. The main task of lexical analyzer is to read the input characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.

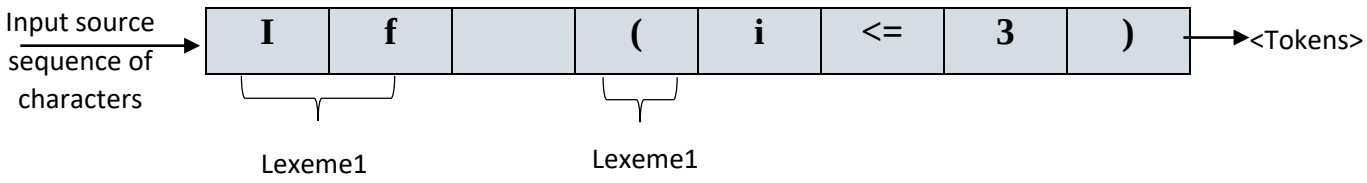


Figure (6): Example of the lexeme

As shown in figure (6) the lexical analyzer scanning the input source characters one by one whenever formatted lexeme then results to this lexeme token that the parser uses for syntax analysis as shown in figure (7).

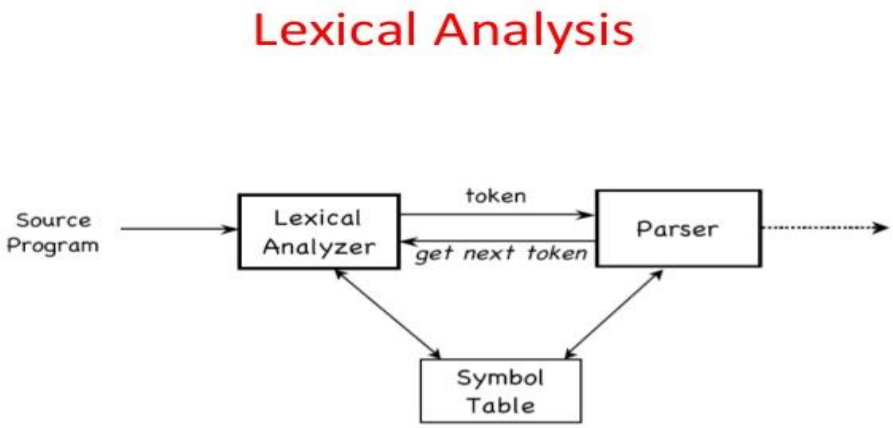


Figure (7): Interaction of lexical analyzer with parser

1.2 lexical analyzer tasks

lexical analyzer tasks are divided into following process:

a) ***Scanning:*** consists of the simple processes that don't require tokenization of the input, such as deletion of comments compaction of consecutive whitespace characters into one.

b) ***Lexical analysis proper***: is more complex portion, where the scanner produces the sequence of tokens as output.

1.3 Tokens, Patterns, Lexemes

When discussing lexical analysis ,we use three related but distinct terms :

Tokens is pair consisting a token name and optional attributed value. Token name is abstract symbol representing kind of token unit which are

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants. The token names are the input symbols that the parser processes.

   Note: The optional attributed means it can be and may not exist.

   For example <If > or < id , pointer symbol-table entry E>

Lexeme is a sequence of characters in the source program that is matched by the pattern for a token. In general, the lexeme is stored in symbol table specially if the lexeme is identifier.

Pattern: is a description of the form that the lexeme of token my take. In case of keyword as a token the pattern is the sequence of characters that form the keyword.

Table (1): Example of the token

Tokens	***Pattern***	***Example of Lexeme***
if	Characters I,f	if
else	Characters e,l,s,e	else
Comparison	<, >, <= , >= , == , !=	<, >, <= , >= , == , !=
id	Letter followed by letter or digit	X ,y3,count
Number	Any numeric constant	3.14159, 0, 6, 02e23
literal	fixed value in source code	String s = "cat" int a=1

In many programming languages, most or all of the tokens are:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the1 operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

1.4 Attributed of token

The attribute of token is a structure that combines several pieces of information the most important example is the token of identifier. The properties of attributed of identifier is a pointer to symbol table entry for that identifier as shown in figure (8).

Example

The token name and associated attribute values of the Fortran statement: E = M * C ** 2

Lexical analyzer is writing Fortran statement as sequence of pair

<id, pointer to symbol –table entry E>

< assign-op>

<id, pointer to symbol –table entry M>

< Multi-op>

<id, pointer to symbol –table entry C>

< Exp -op>

<number, integer value 2>

Figure (8) : Example of the attribute of token

1.5 Input buffer

Lexical analyzer scans the characters of the source program one at a time to discover tokens. It is desirable for the lexical analyzer to input from buffer.

1.5.1 Buffer pairs

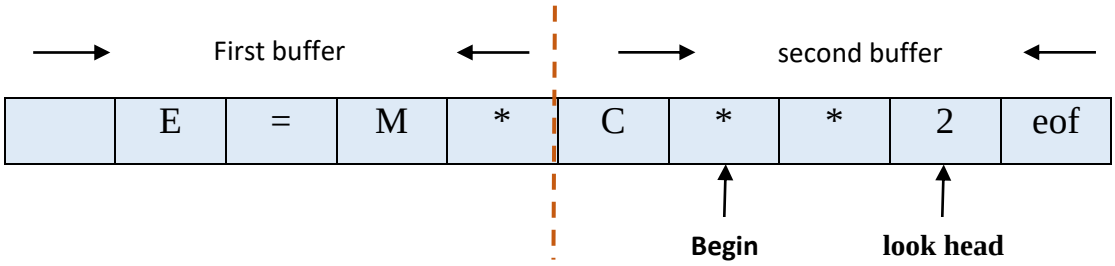


Figure (9) Using a pair of input buffer.

1- Pointer lexeme Begin, marks the beginning of the token being discovered.

2- look head pointer scans ahead of the beginning pointer, until a token is discovered.

Symbol Table

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various source language constructs. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code. We required several capabilities of the symbol table we need to be able to:

1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.

insert(s,t) : this function is to add a new name to the table

Lookup(s) : returns index of the entry for string s, or 0 if s is not found.

2- Access the information associated with a given name, and add new information for a given name.

3- Delete a name or group of names from the tables.

For example, consider tokens **begin,** we can initialize the symbol-table using the function**: insert("begin",1).**

Symbol table management refers to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

1) A symbol table is a data structure, where information about program objects is gathered.

2) Is used in all phases of compiler.

3) The symbol table is built up during the lexical and syntax analysis.

4) Help for other phases during compilation:

1.6 Specification of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

1.6.1 Strings and Languages

The term of ***alphabet*** or ***character class*** denotes any finite set of symbols. Typical examples of symbol are letter and characters. The set {0, 1} is the ***binary alphabet*** ASCII is the examples of ***computer alphabets***.

String: is a finite sequence of symbols taken from that alphabet. The terms ***sentence*** and ***word*** *are* often used as synonyms for term "string".

|S|: is the **Length** of the string S.
Example: |banana| =6

Empty String (∈): special string of length zero.

Exponentiation of Strings

$\mathbf{S^2}$ = SS $\mathbf{S^3}$ = SSS $\mathbf{S^4}$ = SSSS

$\mathbf{S^i}$ is the string **S** repeated **i** times.

By definition S^0 is an empty string.

Languages: A language is any set of string formed some fixed alphabet.

Operations on Languages

There are several important operations that can be applied to languages.

For lexical Analysis the operations are:

1- Union.

2- Concatenation.

3- Closure.

Operation	Definition
Union ***L*** and ***M*** written ***L*∪*M***	***L*∪*M***={s | s is in ***L*** or s 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****	$L^* = \bigcup_{i=0}^{\infty} L^i$ ***L*** denotes "zero or more Concatenation of "L***
Positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ ***L+ denotes "One or more Concatenation of "L***

Example: Let ***L*** and ***M*** be two languages where ***L*** = {a, b, c} and ***D***= {0, 1} then

- Union: ***LUD*** = {a, b, c, 0,1}
- Concatenation: ***LD*** = {a0,a1, b0, b1, c0,c1}
- Expontentiation : L^2 = LL

By definition: $L^0 = \{\in\}$

1.6.2 Regular Definitions

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Example1: The set of C identifiers is the set of strings of letters and digits beginning with a letter.Here is a regular definition for this set:

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 C 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 for the set of all decimal digits.

Example 2: Unsigned numbers in Pascal are strings such as 5280, 39.37, 6.336E4, or 1.894E-4. The following regular definition provides a precise specification for this class of strings:

digit → 0 | 1 | 2 | . . . | 9

digits → digit $^{+}$

optional-fraction → . digits | ∈

optional-exponent → (E (+ | - | ∈**) digits) |**∈

Num → digits optional-fraction optional-exponent

This regular definition says that

- An optional-fraction is either a decimal point followed by one or more digits or it is missing (i.e., an empty string).
- An optional-exponent is either an empty string or it is the letter E followed by an optional + or - sign, followed by one or more digits.

Running Example:

Stmt → if Expr then Stmt
| if Expr then Stmt else Stmt
| e
Expr → Term relop term|Term
Term → id| number

In the above example, the production of the regular definition has two type of tokens (terminal and non -terminal), the lexical analysis interest about terminal tokens such as (if, then, else, and relop (relational operation).

Terminal tokens

{if, else, then} → keywords
relop → operation
Number → number
id → identifier

Now we need regular definition for each terminal tokens which is representing patterns for each token as following.

$digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

$digits \rightarrow digit^{+}$

$Number \rightarrow digit\ (.\ digits \mid \in)(E\ (+ \mid - \mid \in)\ digits) \mid \in$

$letter \rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z$

$id \rightarrow letter\ (letter \mid digit\ )*$

$if \rightarrow if$

$else \rightarrow else$

$then \rightarrow then$

"else "represent direct pattern matching

$relop \rightarrow <|>|<=|>=|=|<>$

In addition , we assign the lexical analyzer the job of stripping out white space ,by recognizing the "token" *ws* defined by

$ws \rightarrow (blank \mid tab \mid newline)^{+}$

Since the {blank ,tab , newline } are terminal tokens then the lexical analyzer can be recognition by single ASCII code for each them.as noted the assign ($^{+}$) means there is a possibility of one or more spaces.

Table (2): Lexeme , Tokens , and attributes for Running example

Lexeme	***Token***	***Attribute value***
Any ws	-	-
If	*If*	-
Then	*Then*	-
Else	*Else*	-
Any id	*id*	*Pointer to table entry*
Any number	*number*	*Pointer to table entry*
<	*Relop*	*LT*
<=	*Relop*	*LE*
=	*Relop*	*EQ*
<>	*Relop*	*NE*
>	*Relop*	*GT*
>=	*Relop*	*GE*

Transition diagrams

A transition diagram is similar to a flowchart for (a part of) the lexical. We draw one for each possible token. It shows the decisions that must be made based on the input seen. The two main components are circles representing states and arrows representing edges.

Example : transition diagram for the relational operation (relop) shown in figure (10).

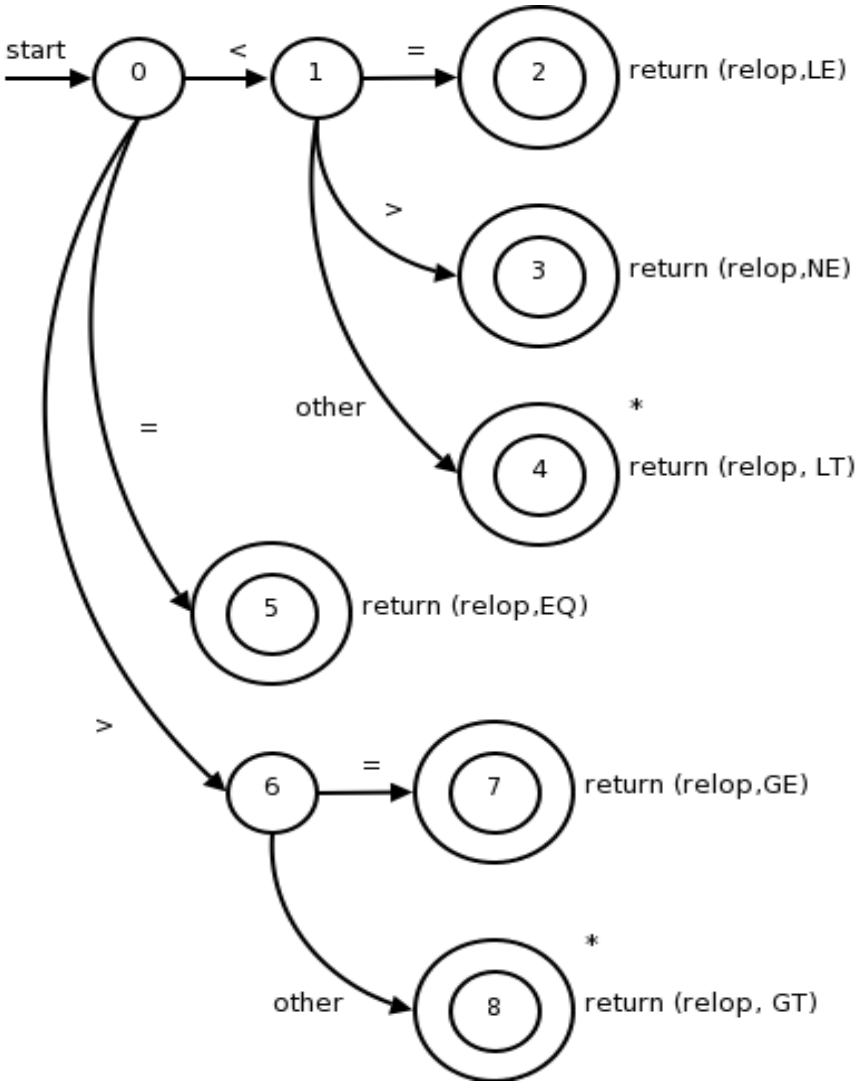


Figure (10): Transition diagram of the relop

1. The double circles represent *accepting* or *final* states at which point a lexeme has been found. There is often an action to be done (e.g., returning the token), which is written to the right of the double circle.
2. Each edge is labeling by symbol or set of symbols.

Recognition of tokens

1. ***Recognition of Reserved Words and Identifiers***

To turn a collection of transition diagrams into a program, we construct a segment of code for each state. The first step to be done in the code for any state is to obtain the next character from the input buffer. For this purpose, we use a function ***GETCHAR***, which returns the next character, advancing the look ahead pointer at each call. The next step is to determine which edge, if any out of the state is labeled by a character, or class of characters that includes the character just read. If no such edge is found, and the state is not one which indicates that a token has been found (indicated by a double circle), we have failed to find this token. The look ahead pointer must be retracted to where the beginning pointer is, and another token must be search for using another token diagram. If all transition diagrams have been tried without success, a lexical error has been detected and an error correction routine must be called as shown in figure (11) which is illustrated transition diagram for identifier.

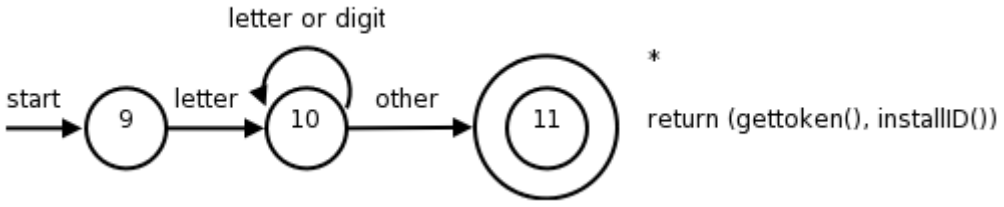


Figure (11): Transition diagrams for identifier.

State 0 : C = GETCHAR ( )
if LETTER(C) then goto state1
else FAIL( )
State1 : C= GETCHAR ( )
if LETTER(C) or DIGIT(C) then goto state1

else if DELIMTER(C) then goto state2

else FAIL ( )

State2: RETRACT ( )

return(id, INSTALL( ))

LETTER(C) is a procedure which return true if and only if C is a letter.

DIGIT(C) is a procedure which return true if and only if C is one of the digit 0,1,…9.

DELIMITER(C) is a procedure which return true whenever C is character that could follow an identifier. The delimiter may be: blank, arithmetic or logical operator, left parenthesis, equals sign, comma,…

Install ( ) checks if the lexeme is already in the symbol table

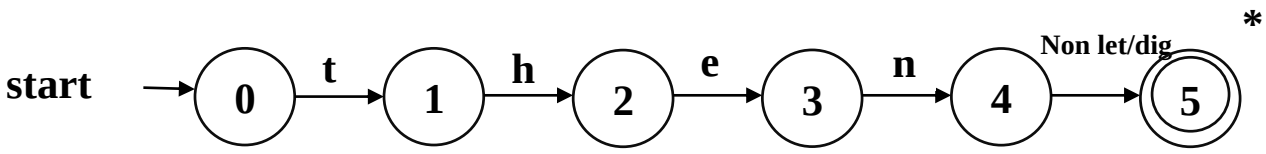


Figure (12) : transition diagram of the keyword then.

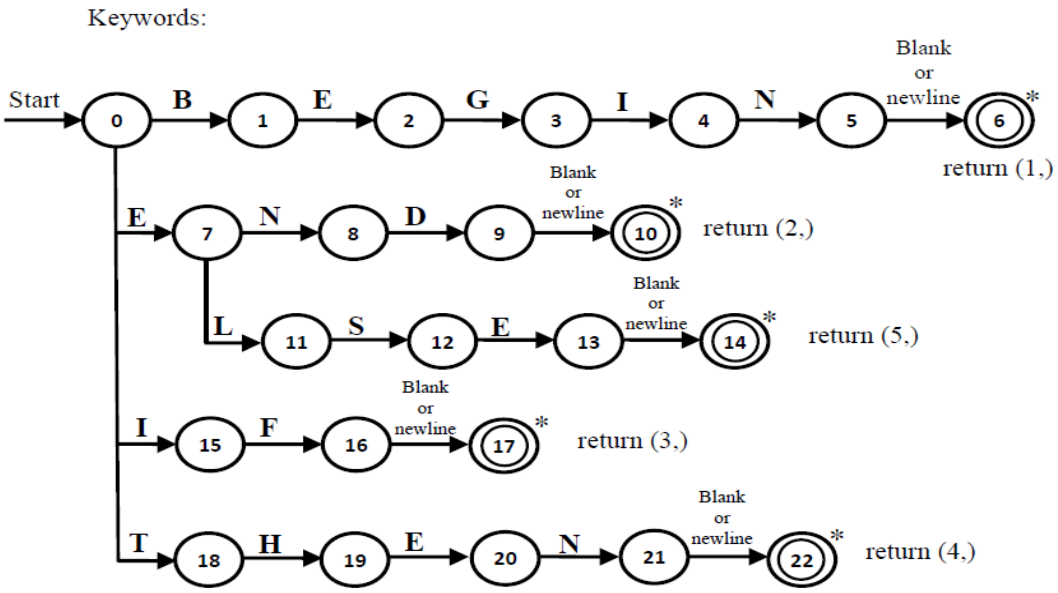


Figure (13) : transition diagram of the keyword then.

Recognizing Numbers

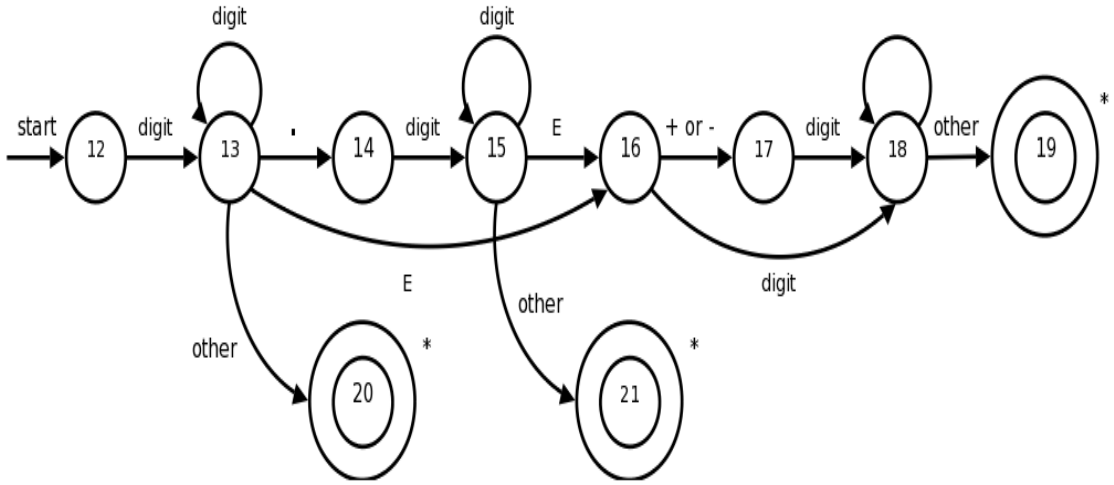


Figure (14) : transition diagram of the number.

The lexical analyzer returns to parser a representation for the token it has found. This representation is:

• An **integer code** if there is a simple construct such as a left parenthesis, comma or colon.

• Or a **pair** consisting of an **integer code** and a **pointer to a table** if the token is more complex element such as an **identifier or constant.**

Recognition of Tokens

Introduction

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Example: assume the following **pattern** to generate **if-stmt** in a specific language:

stmt ⟶ **if (*expr*) *stmt* | if (*expr*) *stmt* else *stmt* |** ∈

expr ⟶ ***term* relop *term* | *term***

term ⟶ **id | num**

Where the terminals ***if***, ***then***, ***else***, ***relop***, ***id***, and ***num*** generate sets of strings given by the following regular definitions:

if ⟶ **if**

else ⟶ **else**

relop ⟶ <| <=| ==| !=| >| >=

id ⟶ **letter(letter|digit)***

num ⟶ **digits optional-fraction optional-exponent**

Where ***letter***([A,Z],[a,z]) and ***digits*** ([0,9]) are as defined previously. For this language fragment the lexical analyzer will recognize the keywords ***if, then, else***, as well as the lexemes denoted by ***relop, id, and num***. To simplify matters, we assume ***keywords are reserved***; that is, they cannot be used as identifiers. The **num** represents the unsigned integer and real numbers of C program.

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws → (blank/tab/newline)$^+$

Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space.

Transition Diagrams (TD)

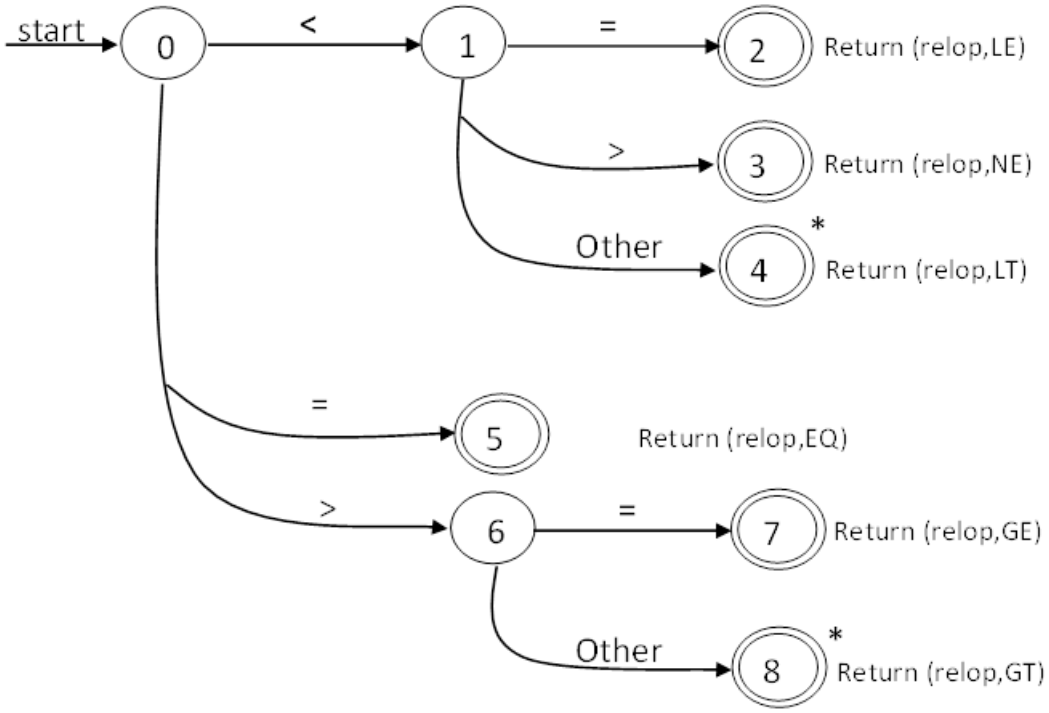
As an intermediate step in the construction of a lexical analyzer, we first produce flowchart, called a ***Transition diagram***. Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

The **TD** uses to keep track of information about characters that are seen as the forward pointer scans the input. It dose that by moving from position to position in the diagram as characters are read.

Components of Transition Diagram

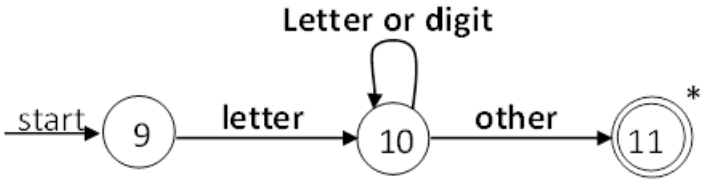
1. One state is labeled the **Start State**; start it is the initial state of the transition diagram where control resides when we begin to recognize a token.
2. **Positions** in a transition diagram are drawn as circles and are called states.
3. The states are connected by **Arrows**, called edges. Labels on edges are indicating the input characters.
4. The **Accepting** states in which the tokens has been found.
5. **Retract** one character use * to indicate states on which this input retraction.

Example: A Transition Diagram for the token relation operators in Pascal language "**relop**" {<=, <, >, >=, =, <>} is shown in Figure:



Transition Diagram for relation operators in **Pascal**.

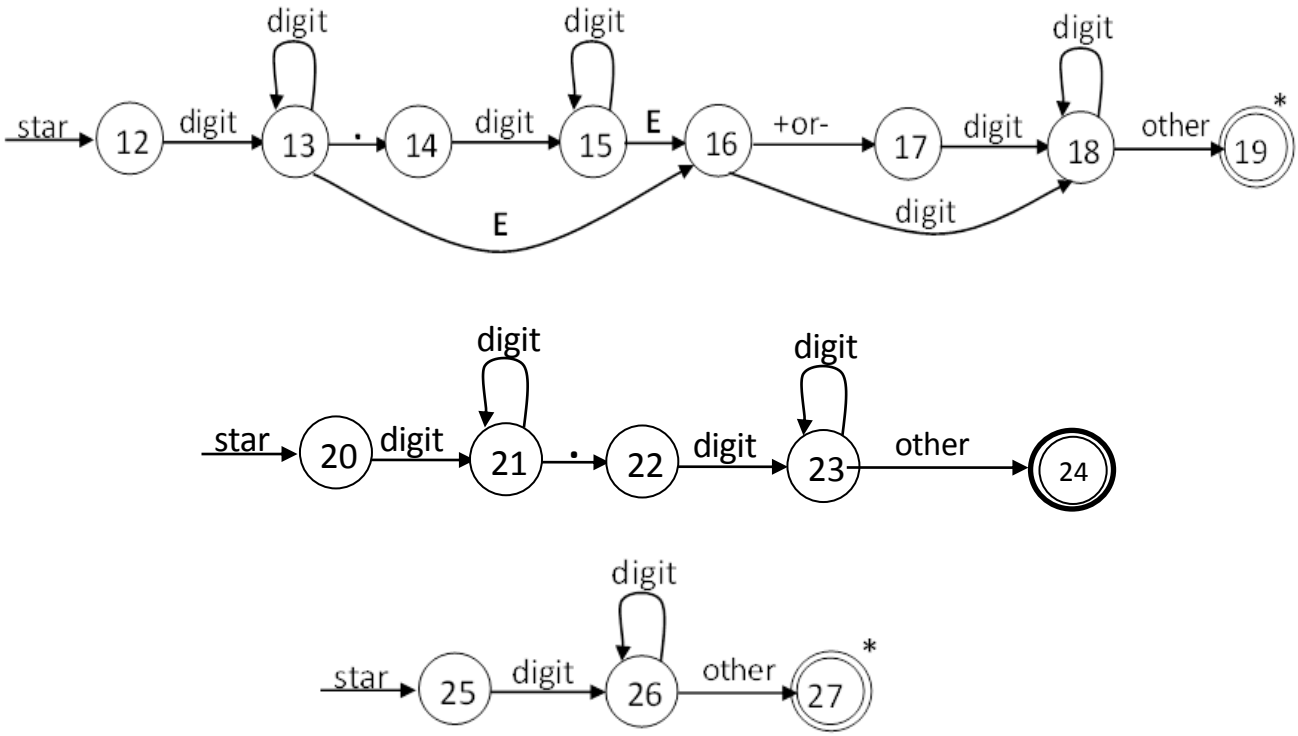
Example: A Transition Diagram for the **identifiers** and **keywords**:



Transition Diagram for identifiers and keywords

The above TD for an identifier, defined to be a letter followed by any no of letters or digits.

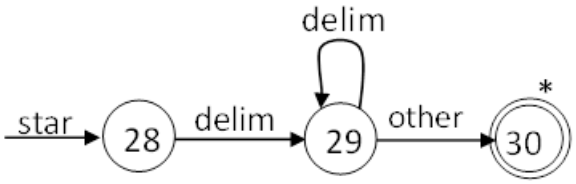
Example: A Transition Diagram for **Unsigned Numbers** in Pascal:



$$num \rightarrow digit^{+}(.\ digit^{+}|\ \in)(E(+|-|\in)digit^{+}|\in)$$

Transition Diagram for unsigned numbers in Pascal

Treatment of White Space (WS):



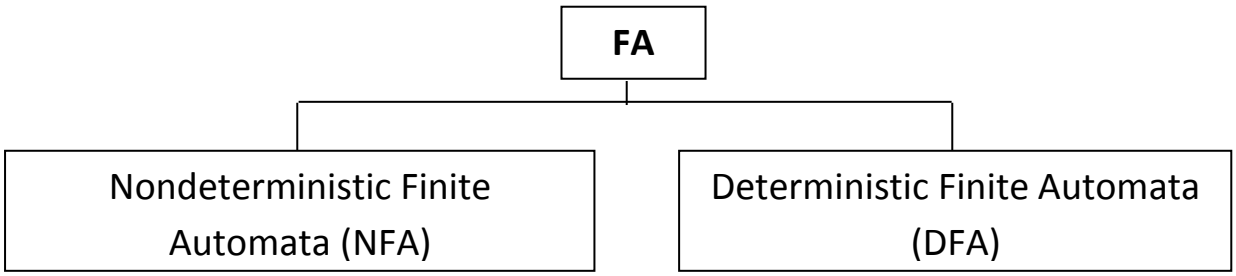
Transition Diagram for White Space

Nothing is returned when the accepting state is reached; we merely go back to the start state of the first transition diagram to look for another pattern.

Finite Automata (FA)

It a generalized transition diagram TD, constructed to compile a regular expression RE into recognizer.

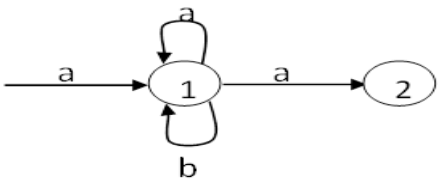
Recognizer for a Language: is a program that takes a string **X** as an input and answers "Yes" if **X** is a sentence of the language and "No" otherwise.



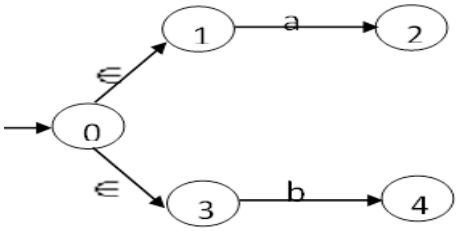
Note: Both **NFA** and **DFA** are capable of recognizing what regular expression can denote.

Nondeterministic Finite Automata (NFA)

NFA: means that more than one transition out of a state may be possible on a same input symbol.



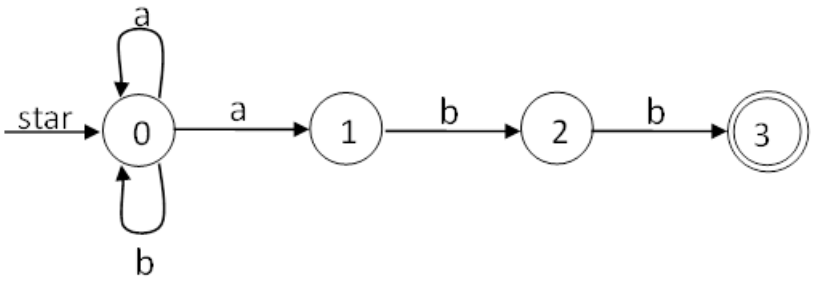
Also a transition on input $\in$ ($\in$-Transition) is possible.



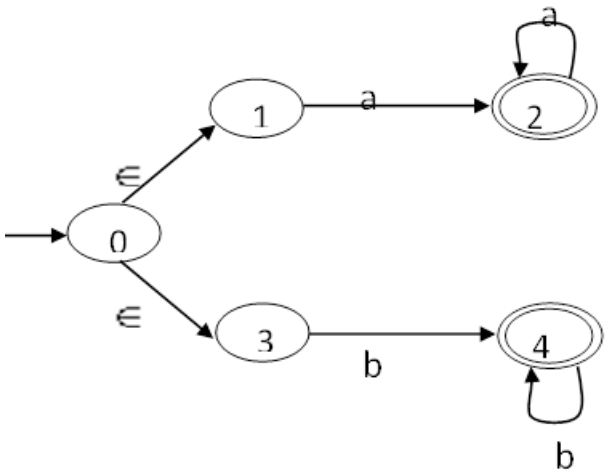
A nondeterministic finite automation **NFA** is a mathematical model consists of

1) A set of states' **S**;
2) A set of input symbol, ∑, called the input symbols alphabet.
3) A set of transition to move the symbol to the sets of states.
4) A state $\mathbf{S_0}$ called the initial or the **start** state.
5) A set of states **F** called the accepting or **final** state.

Example: The NFA that recognizes the language **(a | b)*abb** is shown below:



Example: The NFA that recognizes the language **aa*|bb*** is shown below



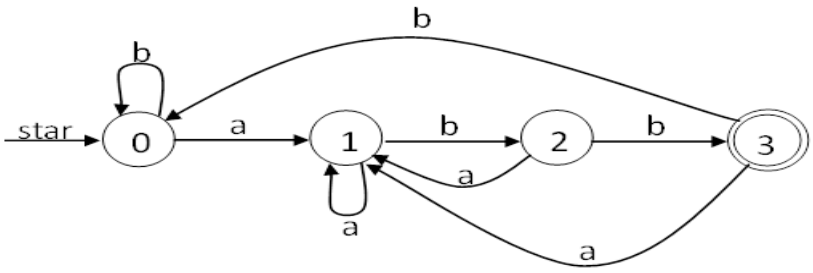
Deterministic Finite Automata (DFA)

A deterministic finite automation (DFA, for short) is a special case of a non-deterministic finite automation (NFA) in which

1. No state has an $\in$-transition, i.e., a transition on input $\in$, and
2. For each state ***S*** and input symbol ***a***, there is at most one edge labeled ***a*** leaving ***S***.

A deterministic finite automation DFA has at most one transition from each state on any input.

Example: The following figure shows a DFA that recognizes the language **(a|b)*abb**.



A DFA is represented by a ***transition table T***, The Transition Table is:

State	a	b
0	1	0
1	1	2
2	1	3
3	1	0

DFAs have the same expressive power as NFAs: A DFA is a special case of NFA and any NFA can be converted to an equivalent DFA. However, this comes at a cost: The resulting DFA can be exponentially larger than the NFA.

Conversion of an NFA into a DFA

It is hard for a computer program to simulate an NFA because the transition function is multivalued. The algorithm that called the **subset construction** will convert an NFA for any language into a DFA that recognizes the same languages.

Algorithm: (Subset construction): constructing DFA from NFA.

Input: ***NFA*** *N*.

Output: ***DFA*** *D* accepting the same language.

Method: this algorithm constructs a transition table ***Dtran*** for ***D***. Each **DFA** state is a set of **NFA** states and we construct ***Dtran*** so that ***D*** will simulate "**in parallel**" all possible moves ***N*** can make on a given input string.

It use the operations in below to keep track of sets of **NFA** states (***s*** represents an NFA state and ***T*** a set of NFA states).

Operations	Description
ε-closure(***s***)	Set of **NFA** states reachable from **NFA** state ***s*** on ε-transitions alone.
ε-closure(***T***)	Set of **NFA** states reachable from some **NFA** state ***s*** in ***T*** on ε-transitions alone.
Move(***T***, a)	Set of **NFA** states to which there is a transition on input symbol ***a*** from some NFA state ***s*** in ***T***.

1) ε-closure (s_0) is the start state of ***D***.
2) A state of ***D*** is accepting if it contains at least one accepting state in ***N***.

Algorithm: (Subset construction):

Initially, ε-closure(s_0) is the only state in ***Dstates*** and it is unmarked;

while there is an unmarked state ***T*** in ***Dstates*** **do begin**

mark T;

For each input symbol ***a*** **do begin**

U: = (ε-closure (move (***T, a***)) ;

if ***U*** is not in ***Dstates*** **then**

add ***U*** as an unmarked state to ***Dstates***;

Dtran [***T, a***]: = ***U***

End

End

We construct ***Dstates***, the set of states of ***D***, and ***Dtran***, the transition table for ***D***, in the following manner. Each state of ***D*** corresponds to a set of **NFA** states that ***N*** could be in after reading some sequence of input symbols including all possible ε-transitions before or after symbols are read.

Algorithm: Computation of ε-closure

Push all states in ***T*** onto slack;
Initialize ε-closure (***T***) to ***T***;
While slack is not empty **do begin**

Pop ***t***, the top clement, of the stack;

For each state ***u*** with an edge from ***t*** to ***u*** labeled ε **do**

If u is not in ε-closure (***T***) **do begin**

Add ***u*** to ε-closure (***T***);

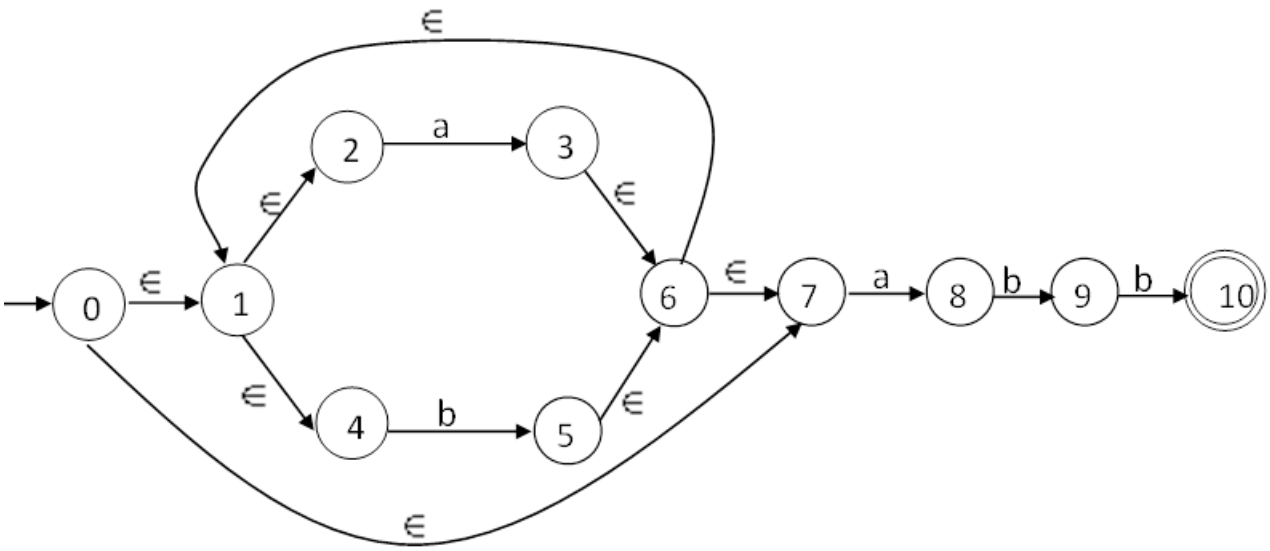
Push ***u*** onto stack

End

End

A simple algorithm to compute ε-closure (***T***) uses a stack to hold states whose edges have not been checked for ε-labeled transitions.

Example: The figure below shows **NFA *N*** accepting the language **(a | b)*abb**.



Sol: apply the Algorithm of Subset construction as follow:

1) Find the **start state** of the equivalent **DFA** is ε-closure (**0**), which is consist of **start state** of **NFA** and the **all states** reachable from **state 0** via a path in which every edge is labeled ε.

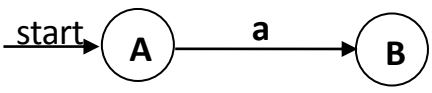
A= {0, 1, 2, 4, 7**}**

2) Compute move (A, a), the set of states of NFA having transitions on **a** from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 have such transitions, to 3 and 8, so

move (A, *a*)={3, 8}

Compute the ε **-closure (move (A, *a*))** = ε **-closure ({3, 8})**,

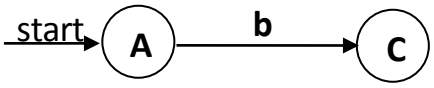
ε **-closure ({3, 8})** = **{1, 2, 3, 4, 6, 7, 8}** Let us call this set **B**.



3) Compute **move (A, *b*)**, the set of states of **NFA** having transitions on ***b*** from members of **A**. Among the states **0**, **1**, **2**, **4** and **7**, only **4** have such transitions, to **5** so: **move (A, *b*)={5}**

Compute the ε **-closure (move (A, *b*))** = ε **-closure ({5})**,

ε **-closure ({5})** = **{1, 2, 4, 5, 6, 7}** Let us call this set **C**. So the **DFA** has a transition on ***b*** from **A** to **C.**



4) We apply the **steps** 2 and 3 on the B and C, this process continues for every new state of the **DFA** until all sets that are states of the **DFA** are marked.

The five different sets of states we actually construct are:

A = {0, 1, 2, 4, 7}

B = {1, 2, 3, 4, 6, 7, 8}

C = {1, 2, 4, 5, 6, 7}

D = {1, 2, 4, 5, 6, 7, 9}

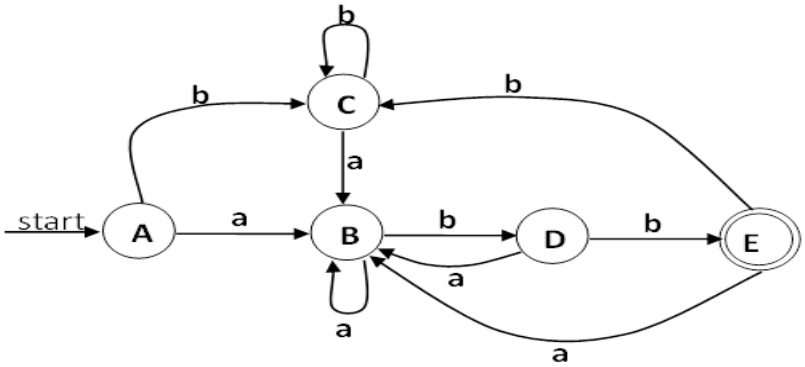
E = {1, 2, 4, 5, 6, 7, 10}

State **A** is the **start state**, and state **E** is the only **accepting state**. The complete **transition table *Dtran*** is shown in below:

STATE	INPUT SYMBOL	
	a	***b***
A	***B***	***C***
B	***B***	***D***
C	***B***	***C***
D	***B***	***E***
E	***B***	***C***

Transition table Dtran for DFA

Also, a transition graph for the resulting **DFA** is shown in below. It should be noted that the **DFA** also accepts **(a | b)*abb**.



From a Regular Expression to an NFA

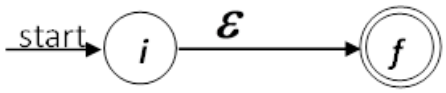
Now give an algorithm to construct an NFA from a regular expression. The algorithm is syntax-directed in that it uses the syntactic structure of the regular expression to guide the construction process.

Algorithm: (Thompson's construction):

Input: a regular expression ***R*** over alphabet ∑.

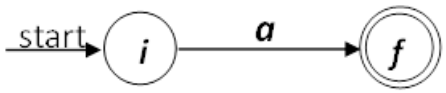
Output: ***NFA*** *N* accepting ***L(R)***.

1- For ε **,** construct the ***NFA***



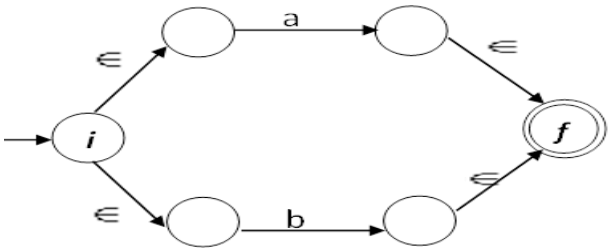
Here ***i*** is a start state and ***f*** a accepting state. Clearly this ***NFA*** recognizes { ε }.

2- For ***a*** in ∑, construct the ***NFA***

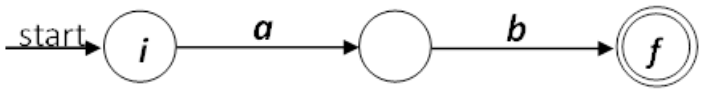


Again ***i*** is a start state and ***f*** a accepting state. This machine recognizes {***a***}.

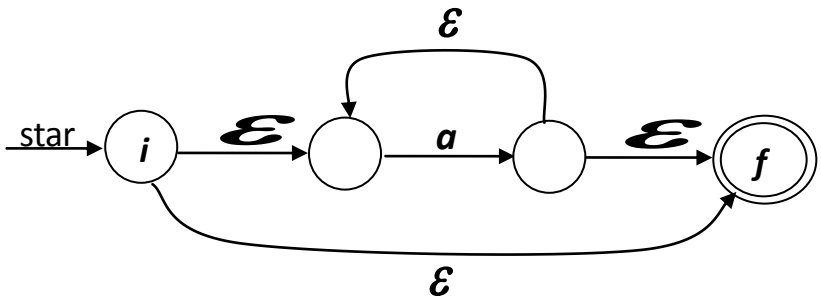
3- For the regular expression a | b construct the following composite NFA N(a | b).



4- For the regular expression ***ab*** construct the following composite NFA ***N(ab)***.

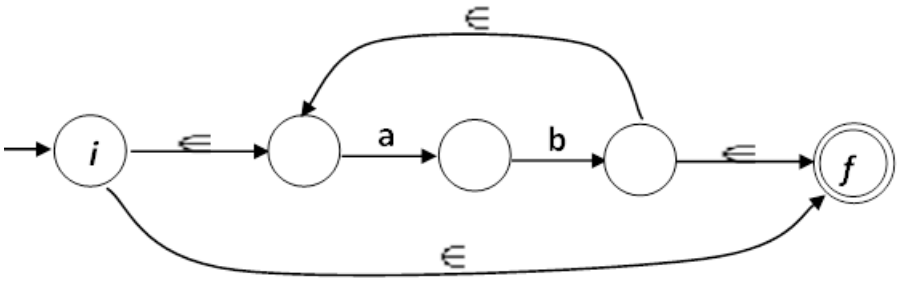


5- For the regular expression ***a**** construct the following composite NFA ***N(a*)***.

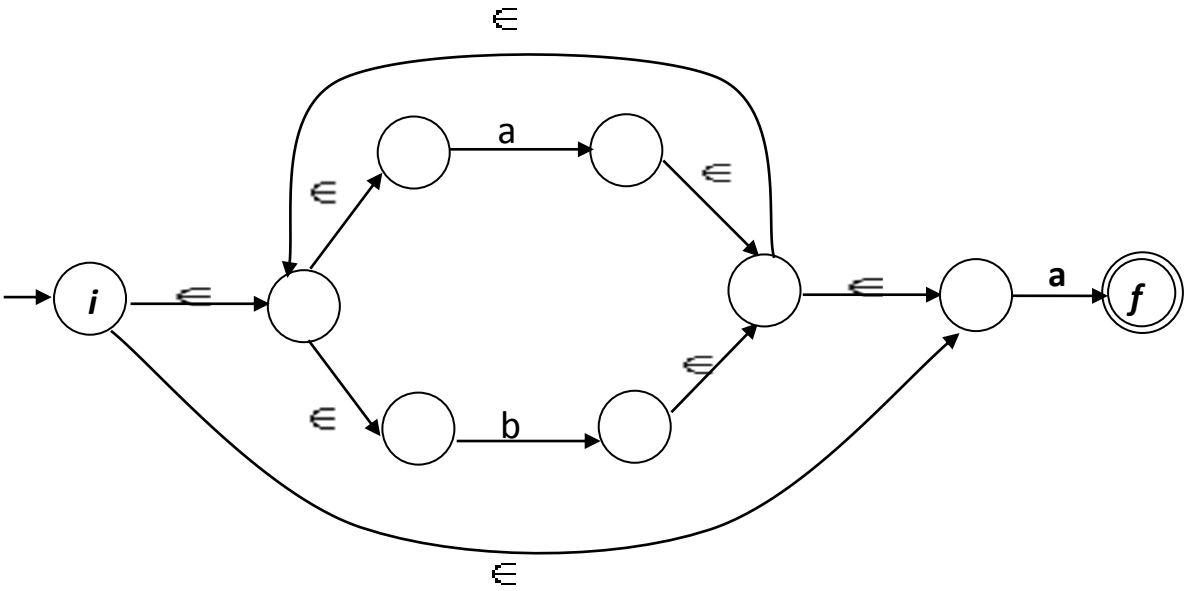


Example: let us use algorithm **Thompson's** construction to construct the following regular expressions:

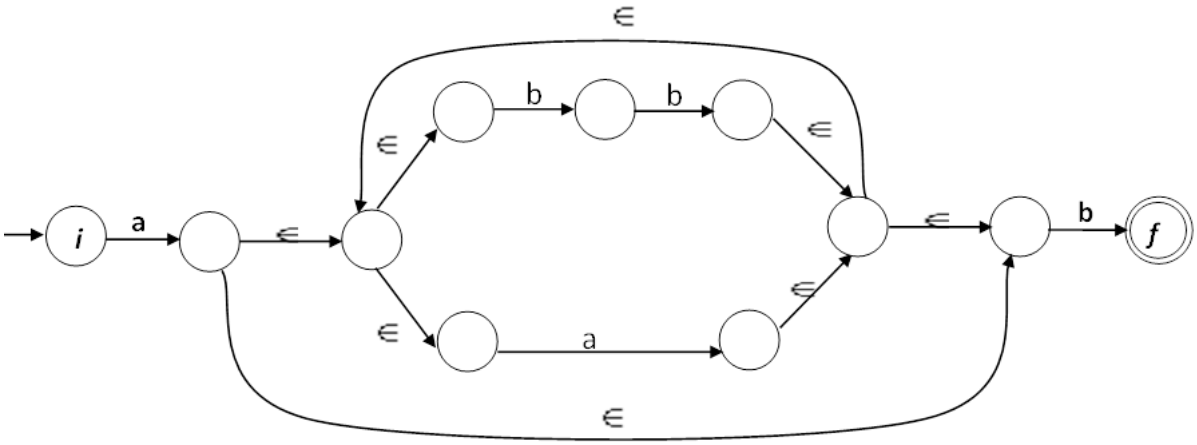
1) RE = (***ab***)*



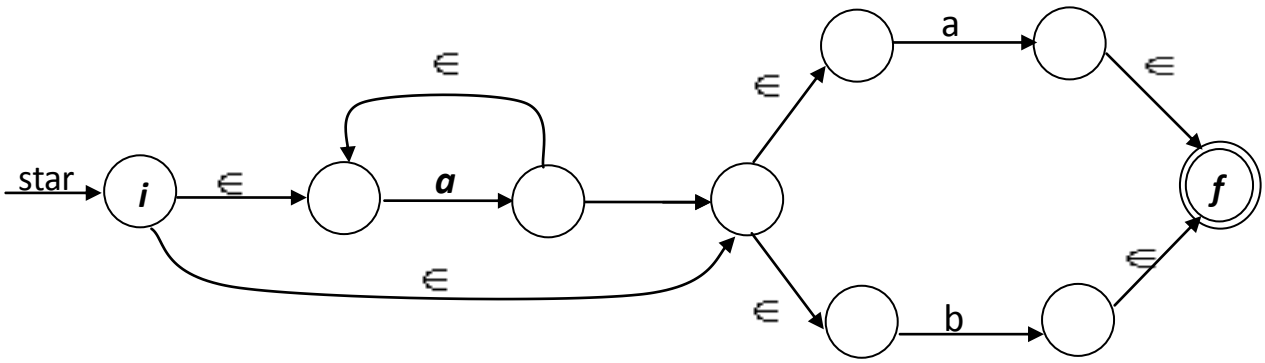
2) RE= (***a*** | ***b***)****a***



3) RE= ***a*** (***bb***| ***a***)****b***



4) RE= ***a**** (***a***



Lexical Errors

What if user omits the space in "Fori"?

No lexical error, single token IDENT ("Fori") is produced instead of sequence For, IDENT ("i").

Typically few lexical error types

1. Delete one character from the input (Keyword Token).

2. Insert an illegal character into the input.

3. Replace a character by another character (Keyword Token).

4. Transpose two adjacent characters (Keyword Token).

5. Misspelling of the keyword.

How is a Scanner Programmed?

1) Describe tokens with regular expressions.

2) Draw transition diagrams.

3) Code the diagram as table/program.

Errors type

We know that programs can contain errors at many different levels. For example, errors can be

1. Lexical errors include misspellings of identifiers, keywords, or operators -e.g., missing quotes around text intended as a string.
2. Syntactic errors include misplaced semicolons or extra or missing braces; that is, '((" or ")."
3. Semantic errors include type mismatches between operators and operands.
4. logical, such as an infinitely recursive call

Syntax Analysis

Parser:

The parser has two functions:

1) It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.
2) It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

Example: if a Pascal program contains the following expression:

A + /B

Then after **lexical analysis** this expression might appear to the syntax analyzer as the token sequence

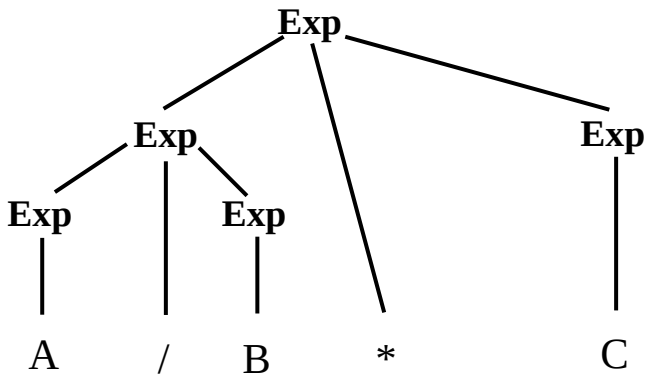
id1 + / id2

On seeing the /, the syntax analyzer should detect an **error** situation, because the presence of these two adjacent operators violates the formation rules of a Pascal expression.

Example: identifying which parts of the token stream should be grouped together:

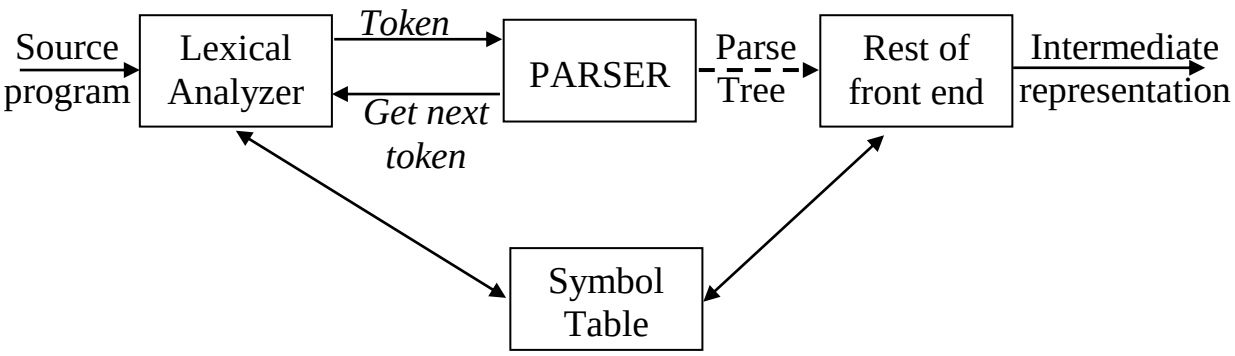
A/B*C

Parse Tree:



The Role of the Parser

In the compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in figure below, and verifies that the string can be generated by the grammar for the source language. The parser must be reported syntax errors clearly fashion.



Position of Parser in Compiler Model

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, an expression out of tokens and so on. The syntax of programming language constructs can be specified by context-free grammars or

BNF (Backus-Naur Form) notation . Grammars offer significant benefits for both language designers and compiler writers

Context-Free Grammars (CFG)

Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars. For example, we might have a conditional statement defined by a rule such as

If $\mathbf{S_1}$ and $\boldsymbol{S_2}$ are statements and E is an expression, then

"If E **then** S_1 **else** S_2" is a statement.

This form of conditional statement cannot be specified using the notation regular expressions.

Could also express as: ***stmt*** ⟶ **if *expr* then *stmt* else *stmt***

Such as a role is called syntactic variables, ***stmt*** to denote the class of statements and ***expr*** the class of expressions.

Components of Context-Free Grammars (CFG)

A context free grammar (CFG for short) consists of terminals, nonterminals, a start symbol, and productions.

1) **Terminals** are the basic symbols from which strings are formed. The word "***token***" is a synonym for "**terminal**" when we are talking about grammars for programming languages.
2) **Nonterminals** are syntactic variables that denote sets of strings.
3) One nonterminal is distinguished as the **Start Symbol**.
4) The set of **Productions** where each production consists of a nonterminal, called the left side followed by an arrow, followed

by a string of nonterminals and/or terminals called the right side.

Example: The grammar with the following productions defines simple arithmetic expressions.

expr ⟶ ***expr op expr***
expr ⟶ **(*expr*)**
expr ⟶ **– *expr***
expr ⟶ **id**
op ⟶ **+**
op ⟶ **–**
op ⟶ *****
op ⟶ **/**
op ⟶ **↑**

In this grammar, the terminal symbols are

id + - * / ↑ ( )

The nonterminal symbols are ***expr*** and ***op,*** and ***expr*** is the start symbol.

The above grammar can be rewriting by using **shorthands** as:

E ⟶ ***EAE*** | **(*E*)** | **-*E*** | **id**
A ⟶ **+** | **-** | ***** | **/** | **↑**

where ***E*** and ***A*** are nonterminals, with ***E*** the start symbol. The remaining symbols are terminals.

Derivations and Parse Trees

How does a context-free grammar define a language? The central idea is that productions may be applied repeatedly to expand the nonterminals in a string of nonterminals and terminals. For example, consider the following grammar for arithmetic expressions:

E ⟶ ***E* + *E* \ *E* * *E*** | **(*E*)** \ **-*E*** \ **id** …… (**1.1**).

The nonterminal ***E*** is an abbreviation for expression. The production ***E*** ⟶ **-*E*** signifies that an expression preceded by minus sign is also an expression. In the simplest case can replace single ***E*** by **-*E***. We can describe this action by writing

$E \longrightarrow -E$ which is read as "***E*** derives ***-E***"

We can take a single ***E*** and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

$E \longrightarrow -E \longrightarrow -(E) \longrightarrow -(\mathbf{id})$

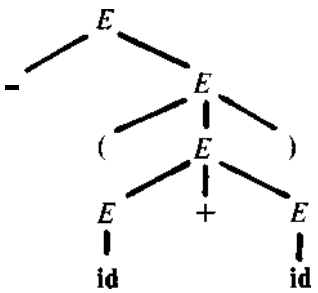
We call such a sequence of replacements a **derivation** of **- (id)** from ***E***. This derivation provides a proof that one particular instance of an expression is the string **- (id)**.

Example: The string **-(id + id)** is a sentence of grammar (1.1) because there is the derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E)$$
$$\Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Parse Tree may be viewed as a graphical representation for derivations that filters out the choice regarding replacement order. Each **interior node** of the parse tree is labeled by some **nonterminal *A***, and the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this ***A*** was replaced in the derivation.

The **leaves** of the parse tree are labeled by **nonterminals** or **terminals** and, read from left to right. For example, the parse tree for **-(id+id)** that implied by the derivation of previous example.

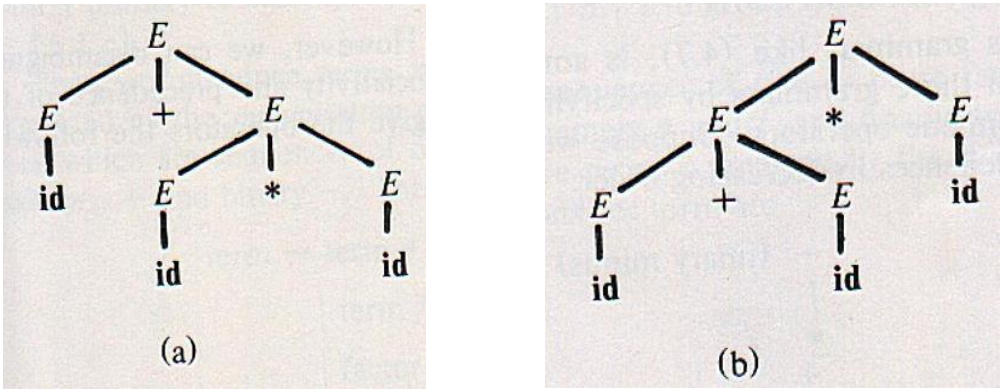


Example: Let us again consider the arithmetic expression grammar (1.1), with which we have been dealing. The sentence **id + id * id** has the two distinct leftmost derivations:

$$E \longrightarrow E + E \longrightarrow \mathbf{id} + E \longrightarrow \mathbf{id} + E * E \longrightarrow \mathbf{id} + \mathbf{id} * E \longrightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

$$E \longrightarrow E * E \longrightarrow \mathbf{E} + E * E \longrightarrow \mathbf{id} + E * E \longrightarrow \mathbf{id} + \mathbf{id} * E \longrightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

With the two corresponding parse tree shown in figure below:



Two parse trees for **id + id * id**

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. In this type, we cannot uniquely determine which parse tree to select for a sentence.

Example: Consider the following grammar for arithmetic expressions involving **+, -, *, /,** and↑ (exponentiation)

$E \longrightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \mathbf{id}$

This grammar is ambiguous.

Eliminate Ambiguity

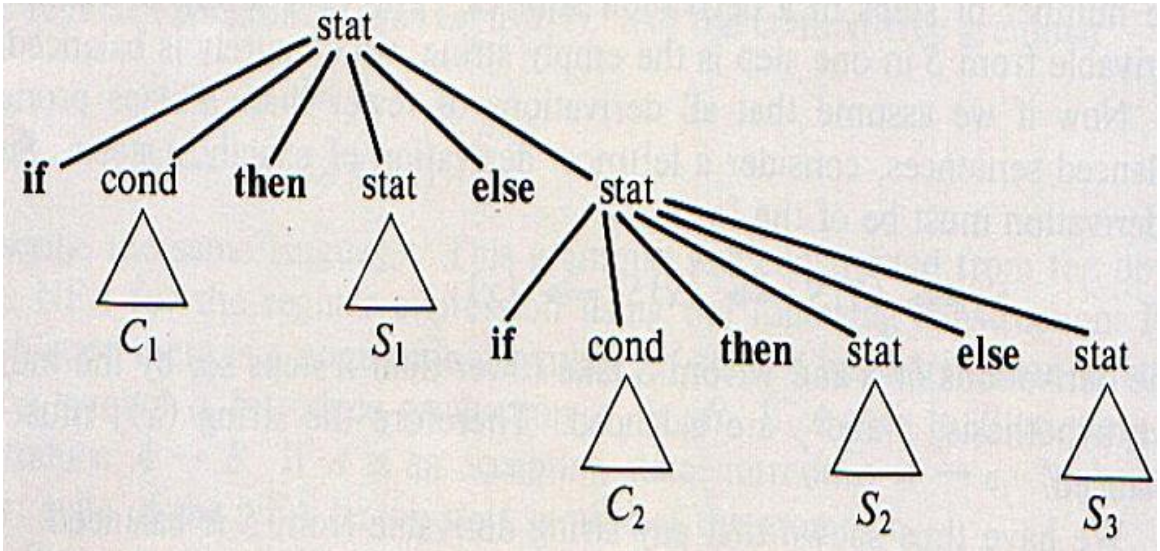
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity and transform the ambiguous grammar to unambiguous grammar. As an example, we shall eliminate the ambiguity from the following "**dangling-else**" grammar:

Stat ⟶ **if *cond* then *stat***
| **if *cond* then *stat* else *stat***
| ***other-stat***

Here "***other-stat***" stands for any other statement. According to this grammar, the compound conditional statement

if C_1 then S_1 else if C_2 then S_2 else S_3

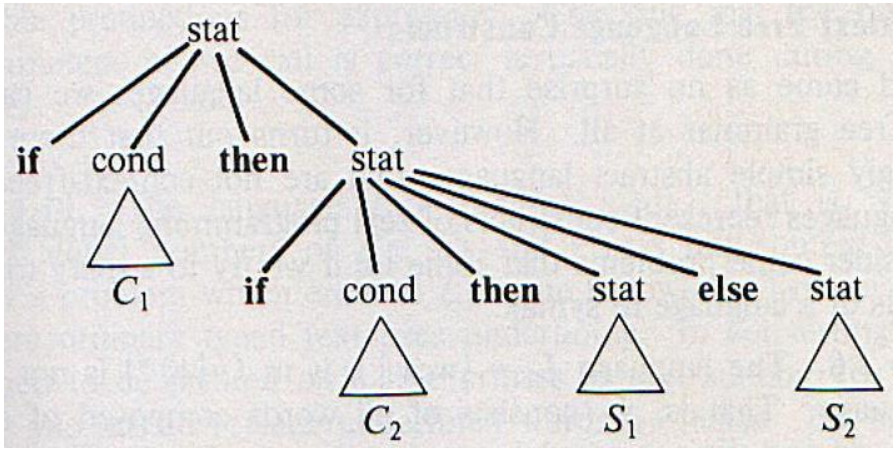
Has the parse tree like the following figure:

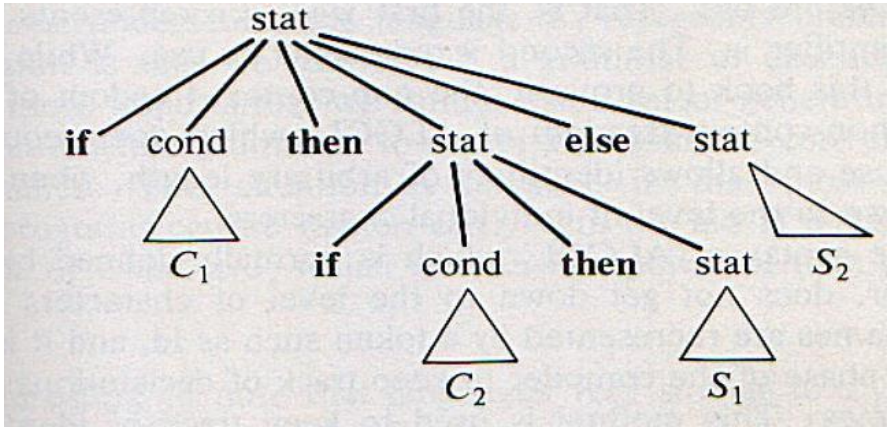


This grammar is ambiguous since the string

if C_1 then if C_2 then S_1 else S_2

Has the two parse tree shown in the figure below:





The above ambiguous grammar transform to the unambiguous grammar:

stat ⟶ ***matched-stat***

| ***unmatched-stat***

matched-stat ⟶ **if *cond* then *matched-stat* else *matched-stat***

| ***other-stat***

unmatched-stat ⟶ **if *cond* then *stat***

| **if *cond* then *matched-stat* else *unmatched-stat***

This grammar generates the same set of string for ambiguous grammar, but it allows only one parsing for above string.

Regular Expression vs. Context-Free Grammar

Every language that can be described by a regular expression can also be described by Context-Free Grammar.

Example: Design CFG that accept the RE = $a\ (a|b)^*b$

$S \longrightarrow aAb$

$A \longrightarrow aA \mid bA \mid \varepsilon$

Example: Design CFG that accept the RE = $(a|b)^*abb$

$S \longrightarrow aS \mid bS \mid aX$

$X \longrightarrow bY$

$Y \longrightarrow bZ$

$Z \longrightarrow \varepsilon$

Example: Design CFG that accept the RE =a^nb^n where $n \geq 1$.

$S \longrightarrow aXb$

$X \longrightarrow aXb \mid \varepsilon$

Example: Design CFG ***Singed Integer*** number.

$S \longrightarrow XD$

$X \longrightarrow + \mid -$

$D \longrightarrow 0D \mid 1D \mid 2D \mid 3D \mid 4D \mid 5D \mid 6D \mid 7D \mid 8D \mid 9D \mid \varepsilon$

Elimination of Left Recursion

A grammar is ***Left Recursive*** if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α. **Top-Down**

Parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed.

In the following example, we show how the left recursion pair of productions $A \longrightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$A \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow \beta A'$$

$$A' \longrightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example: Consider the following grammar for arithmetic expressions.

$$E \longrightarrow E+T \mid T$$

$$T \longrightarrow T*F \mid F$$

$$F \longrightarrow (E) \mid \text{id}$$

Eliminating the immediate ***left recursion*** (productions of the form $A \longrightarrow A\alpha$) to the productions for E and then for T, we obtain

$$E \longrightarrow TE'$$

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

$$T \longrightarrow FT'$$

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

$$F \longrightarrow (E) \mid \text{id}$$

Note: No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \mid \beta_n$$

where no β, begins with an A. Then, we replace the A-productions by:

$$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots \mid \beta_n A'$$
$$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_m A' \mid \varepsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive.

Note: This procedure eliminates all immediate left recursion from the A and A' productions, but it does not eliminate left recursion involving derivations of **two or more** steps.

For **Example**, consider the grammar:

$S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Sd \mid \varepsilon$

The nonterminal S is left- recursive because $S \longrightarrow Aa \longrightarrow Sda$, but is not immediately left recursive. **Algorithm** in below will systematically eliminate left recursion from grammar.

Algorithm: Eliminating left recursion.

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

Output: An equivalent grammar with no left recursion.

Method: Apply the algorithm below to G. Note that the resulting non left-recursive grammar may have ε -productions.

Arrange the nonterminals in some order $A_1, A_2, \ldots\ldots A_n$.

for i: = 1 **to** n **do**

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

    Replace each production of the form $A_i \longrightarrow A_i\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma \mid \delta_2\gamma \mid \delta_3\gamma \mid \ldots\ldots \mid \delta_k\gamma$,

Where $A_j \longrightarrow \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots\ldots \mid \delta_k$ are all the current A_j-productions;
Eliminate the immediate left recursion among A_j-productions
End

Let us apply this procedure to previous grammar. We substitute the ***S***-productions in $A \longrightarrow Sd$ to obtain the following ***A***-productions.

$A \longrightarrow Ac \mid Aad \mid bd \mid \varepsilon$

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

$S \longrightarrow Aa \mid b$

$A \longrightarrow bdA' \mid A'$

$A' \longrightarrow c\,A' \mid adA' \mid \varepsilon$

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive parsing**. The basic idea is that when it is not clear which of **two** alternative productions to use to expand a nonterminal ***A***, we may be able to rewrite the, ***A***-productions to defer the decision until we have seen enough of the input to make the right choice. For example, if we have the two productions

if $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two ***A***-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand ***A*** to $\alpha\beta_1$or to $\alpha\beta_2$. However, we may defer the decision by expanding ***A*** to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is left-factored.

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Longrightarrow \quad \begin{array}{l} A \longrightarrow \alpha A' \\ A' \longrightarrow \beta_1 \mid \beta_2 \end{array}$$

Algorithm : Left factoring a grammar.

Input: Grammar ***G***.

Output: An equivalent left-factored grammar.

Method: For each nonterminal ***A*** find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, replace all the ***A*** productions $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_2 \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \longrightarrow \alpha A' \mid \gamma$$

$$A' \longrightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$$

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

Example:

$$S \longrightarrow iEtS \mid iEtSeS \mid a$$
$$E \longrightarrow b$$

Left-factored, this grammar becomes:

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

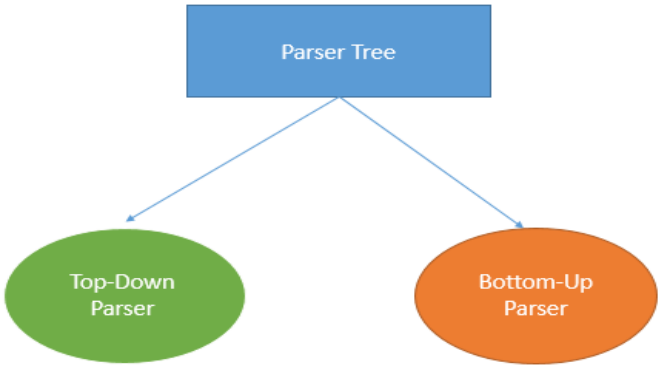
Example:

$$A \longrightarrow aA \mid bB \mid ab \mid a \mid bA$$

Solution:

$$A \longrightarrow aA' \mid bB'$$
$$A' \longrightarrow A \mid b \mid \varepsilon$$
$$B' \longrightarrow B \mid A$$

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.



Top-Down Parser

Top-Down Parser starts constructing a parse tree from the root node gradually moving down to the leaf nodes.

Example: Suppose we have a grammar:

$S \rightarrow E$

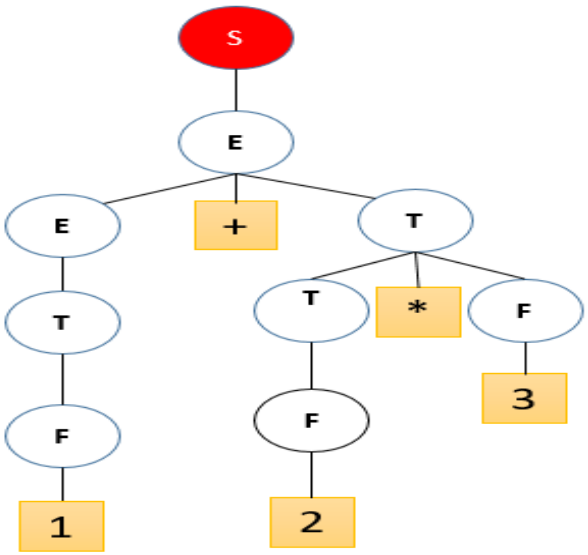
$E \rightarrow T|E + T$

$T \rightarrow F|$T*F

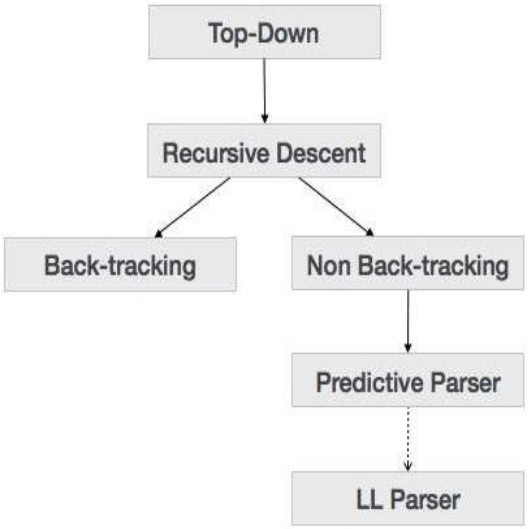
$F \rightarrow$0|1|2|3|4|5|6|7|8|9|∈

Draw parser tree for derivation the following sentence: 1+2*3

Solution:



The types of top-down parsing are shown in figure bellow:



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

Example 1 : Write pseudo-code to parser the following rule or production by recursive descent .

Factor → (exp)|number

Solution:

```
Procedure Factor
Begin
Case token of ( :
match (( );
exp;
match( ));
number;
end case;
end factor;
Procedure match (input)
Begin
```

```
If token =="("|| ")"
accept input
else
reject input
end if
end match
```

Example 2 : Write pseudo-code to parser the following rule or production by recursive descent .

$stmt \rightarrow$ **if** $(exp)statment$

|**if** $(exp)statment$ **else** $statment$

Solution:

Procedure stmt	Procedure match(expected_token)
Begin	Begin
match(if);	If token == expected_token;
match(();	Get Token ;
exp;	else
match());	error;
statement;	end if ;
if token=else then	end match;
match(else);	
match (statement);	
end if;	
end stmt;	

What the Different between Top-Down parser with and without back tracking

In Top-Down Parsing with Backtracking, Parser will attempt multiple rules or production to discover the match for input string by backtracking at every step of derivation.so the different between top-down parser with and without back tracking is shown in table below.

Top-Down Parsing with Backtracking	Top-Down Parsing without Backtracking
The parser can try all alternatives in any order till it successfully parses the string.	The parser has to select correct alternatives at each step.
Backtracking takes a lot of time,	It takes less time.
A Grammar can have left recursion.	Left Recursion will be removed before doing parsing.
t can be difficult to find where actually error has occurred.	It can easy to find the location of the error

Predictive Parsing Method

In many cases, by carefully writing a grammar eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a non backtracking predictive parser. We can build a predictive parser by maintaining a stack. The key problem during predictive parser is that of determining the production to be applied for a nonterminal. The non-recursive parser looks up the production to be applied in a parsing table.

في كثير من الحالات ، من خلال كتابة قواعد نحوية بعناية مع إزالة left recursion منها ، left factoring، يمكننا الحصول على قواعد نحوية يمكن تحليلها بواسطة محلل تنبؤي غير رجعي. يمكننا بناء محلل تنبؤي عن طريق الحفاظ على المكدس. المشكلة الرئيسية أثناء المحلل اللغوي التنبئي هي تحديد production الذي سيتم تطبيقه على a nonterminal.

non-recursive parser يبحث عن الإنتاج ليتم تطبيقه في جدول التحليل.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, (a symbol used as a right endmarker to indicate the end of the input string). The stack contains a sequence of grammar symbols with $ on the bottom,(indicating the bottom of the stack). Initially, the stack contains the start symbol of the grammar on the top of $.

Example:

E →E+T|T

T→T*F|F

F→id|(E)

First step

Removing left recursion for grammar

E → T$\bar{E}$

$\bar{E}$ → +T$\bar{E}$|∈

T→ F $\bar{T}$

$\bar{T}$ →∗ $F\bar{T}$|∈

F→ id|(E)

Second step

Compute First set and Follow set

First set	Follow set
(E) ={ **id,(** }	(E) ={ $,)}
($\bar{E}$) ={+,∈ }	($\bar{E}$) ={ $,)}
(T) ={ **id,(** }	(T) ={ +, $,)}
($\bar{T}$) ={*,∈ }	($\bar{T}$) ={ +, $,)}
(F) ={ **id,(** }	(F) ={ *,+, $,)}

Rules of compute Follow :

1- Put $ in follow set of start symbol which is represents end of file
2- Follow set of Non-terminals (X) is next terminal after it
3- If the next Non-terminals (X) is Non-terminals (Y) then get first this Non-terminals (Y) and put in follow of Non-terminals (X).
4- If the following of the Non-terminals (X) is ∈ then add follow set of current rule left symbol .

Third step

Building parsing or stack table

First set	Follow set
(E) ={ **id,(** }	(E) ={ $,)}
($\bar{E}$) ={+,∈ }	($\bar{E}$) ={ $,)}
(T) ={ **id,(** }	(T) ={ +, $,)}
($\bar{T}$) ={*,∈ }	($\bar{T}$) ={ +, $,)}
(F) ={ **id,(** }	(F) ={ *,+, $,)}

NT \ T	id	(	+	*	)	$
E	E → T$\bar{E}$	E → T$\bar{E}$				
$\bar{E}$			$\bar{E}$ → +T$\bar{E}$		$\bar{E}$ → ∈	$\bar{E}$ → ∈
T	T→ F $\bar{T}$	T→ F $\bar{T}$				
$\bar{T}$			$\bar{T}$ → ∈	$\bar{T}$ →* $F\bar{T}$	$\bar{T}$ → ∈	$\bar{T}$ → ∈
F	F→ id	F→ (E)			F→ (E)	

ملاحظات مهمه لملىء الجدول أعلاه:

1- نبدأ ب first
2- ننتقل الى ال follow في حاله ∈
3- عندما تحتوي القاعدة على ∈ **معناها هي $ end of file or**

Fourth step

Derivation based on stack or parser table with expression =(id)

	id	(	+	*	)	$
E	E → T$\bar{E}$	E → T$\bar{E}$				
$\bar{E}$			$\bar{E}$ → +T$\bar{E}$		$\bar{E}$ → ∈	$\bar{E}$ → ∈
T	T→ F $\bar{T}$	T→ F $\bar{T}$				
$\bar{T}$			$\bar{T}$ → ∈	$\bar{T}$ →* $F\bar{T}$	$\bar{T}$ → ∈	$\bar{T}$ → ∈
F	F→ id	F→ (E)			F→ (E)	

stack	input	output
$E	(id)$	
$$\bar{E}T$$	(id)$	E → T$\bar{E}$
$$\bar{E}\bar{T}F$$	(id)$	T→ F $\bar{T}$
$$\bar{E}\bar{T})E($$	(id)$	F→ (E)
$$\bar{E}\bar{T})E$$	id)$	
$$\bar{E}\bar{T})\bar{E}T$$	id)$	$E \rightarrow T\bar{E}$
$$\bar{E}\bar{T})\bar{E}\bar{T}F$$	id)$	T→ F $\bar{T}$
$$\bar{E}\bar{T})\bar{E}\bar{T}id$$	id)$	$F \rightarrow id$
$$\bar{E}\bar{T})\bar{E}\bar{T}$$	)$	
$$\bar{E}\bar{T})\bar{E}$$	)$	$\bar{T} \rightarrow \in$
$$\bar{E}\bar{T})$$	)$	$\bar{E} \rightarrow \in$
$$\bar{E}\bar{T}$$	$	
$$\bar{E}$$	$	$\bar{T} \rightarrow \in$
$	$	$\bar{E} \rightarrow \in$

بمأنه وصل الاشتقاق الى $ أي نهاية المكدس اذن التعبير او الجملة المدخلة هي مقبولة
ضمن هذه القاعدة بالاعتماد على predicative paring methods