Compiler Lectures

Computer Science

3rd Class

M. Sc. Rajaa Ahmed

2016-2017

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 computers 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 to

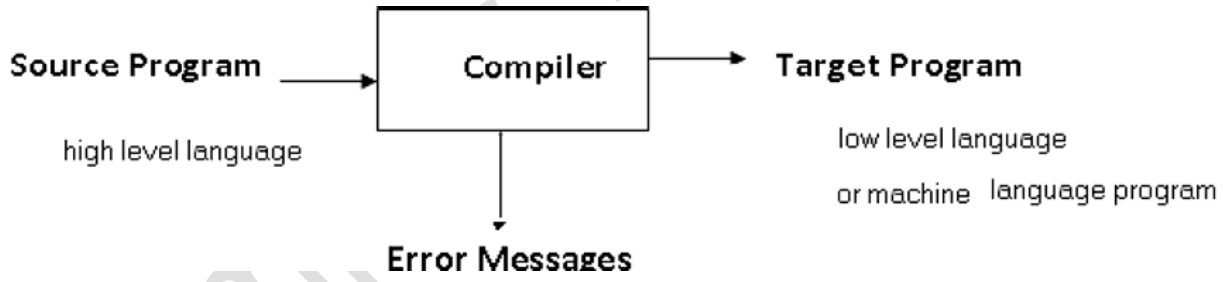
run on many different machines.

- 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 (translator) that reads a program written in one language, (the source language) and translates into an equivalent program in another language (the target language).



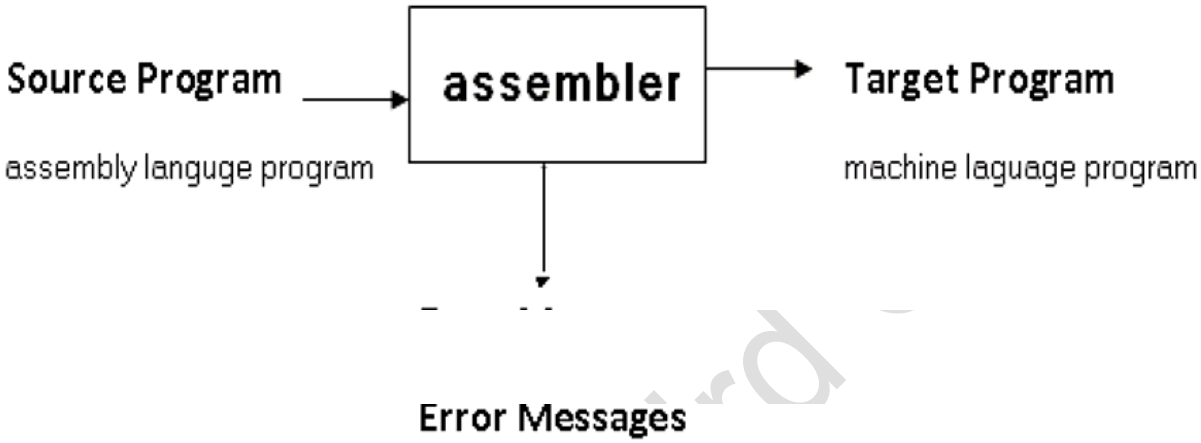
The time at which the conversion of the source program to an object program occurs is called (compile time) the object program is executed at (run time).

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.

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



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** process an internal form of the source program and data at the same time. That is interpretation of the internal source from occurs at run time and an object program is generated Fig (3) which illustrate the interpretation process.

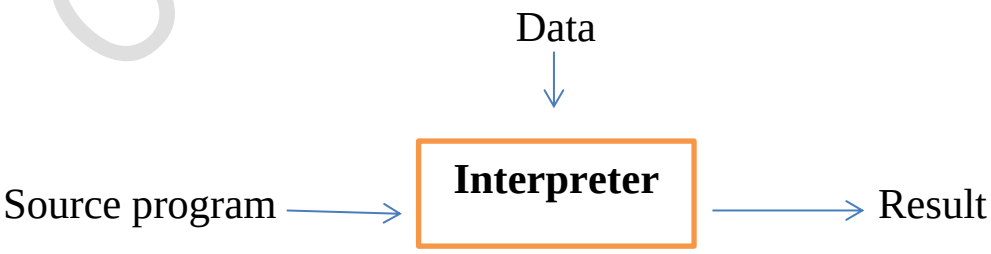
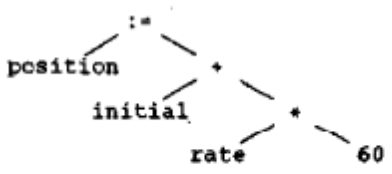


Fig (1)

Compiler	Interpreter
Has Lexing, parsing and type-checking	Has Lexing, parsing and type-checking
generating code from the syntax tree	the syntax tree is processed directly to evaluate expressions and execute statements
typically faster	typically slower
is often complex than writing an interpreter	is often simpler than writing a compiler
is complex than interpreter to move to a different machine	is easier to move to a different machine

The Analysis - Synthesis model of compilation

There are two parts to compilation: analysis and synthesis. Breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called asyntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation. For example, a syntax tree for an assignment statement is shown below:



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

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.***

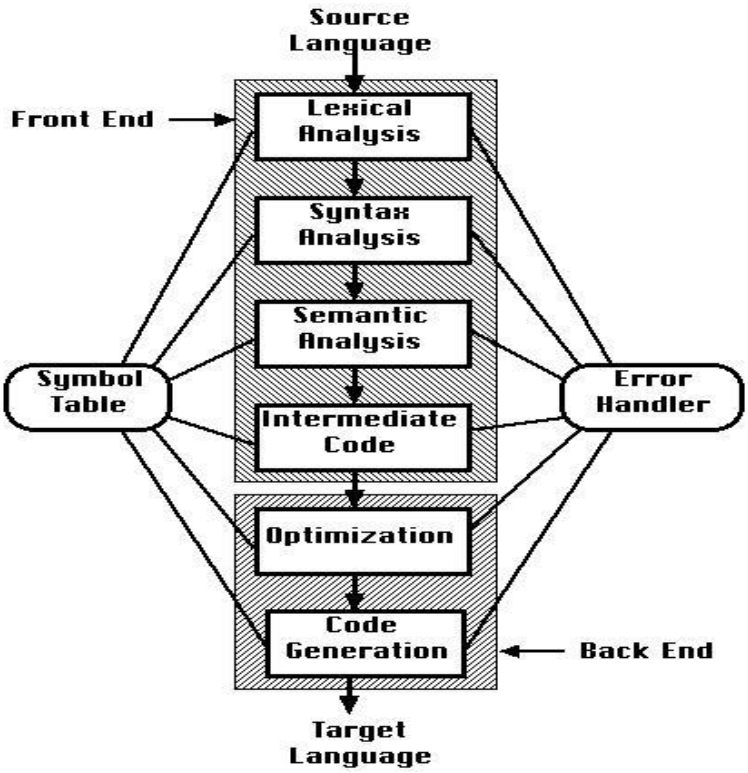
The different phases of a compiler are as follows

Analysis Phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

Synthesis Phases:

4. Intermediate Code generator
5. Code Optimization
6. Code generation.



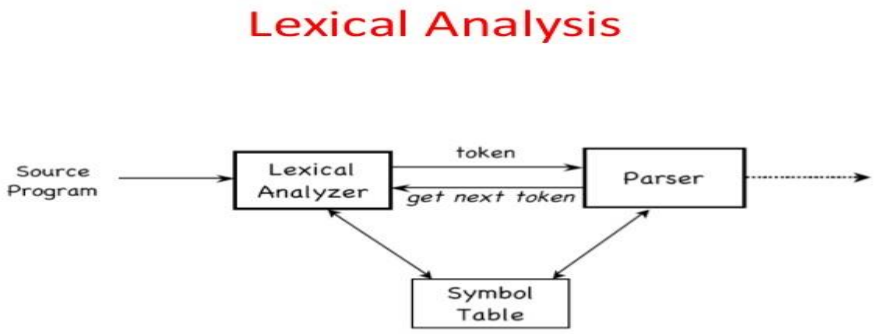
Phases of a Compiler

Compiler structure:

1- lexical analysis

The lexical analyzer is the first stage of a 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 as shown in fig(). The word "lexical" in the traditional sense means "pertaining to words". In terms of programming languages, words are objects like variable names, numbers, keywords …etc. Such words are traditionally called tokens. A lexical analyzer or lexer for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines etc.) and comments.

The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done.



Fig(2):Interaction of lexical analyzer with parser

Preliminary scanning:

- The source code is converted into stream of tokens

- Removes white spaces and comments
- eg: x = a + b * c /* source code */ --> Lexical Analyzer --> id = id + id * id
- This is achieved by using patterns which is known to the lexical analyzer

Tokens, Patterns, Lexemes

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. The pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example: Const pi = 3.1416

The sub string pi is a lexeme for the token "identifier".

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Fig (3): Examples of tokens

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

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

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	lexeme	pattern
Const	const	const
If	if	if
Relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
I	pi	any numeric constant
Nun	3.14	any character b/w “and “except"
Literal	"core"	pattern

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

In many programming languages, the following classes cover most or all of the tokens:

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.

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.**

This integer code gives the token type. The pointer points to the value of the token. For example we may treat “operator" as a token and let the second component of the pair indicate whenever the operator found is +, *, and so on.

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 syntactic analysis.
4) Help for other phases during compilation:

Attributes for tokens

A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept. The token names and associated attribute values for the statement.

E = M * C ** 2

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<number, integer value 2>

Lexical errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string f i is encountered for the first time in a C program in the context:

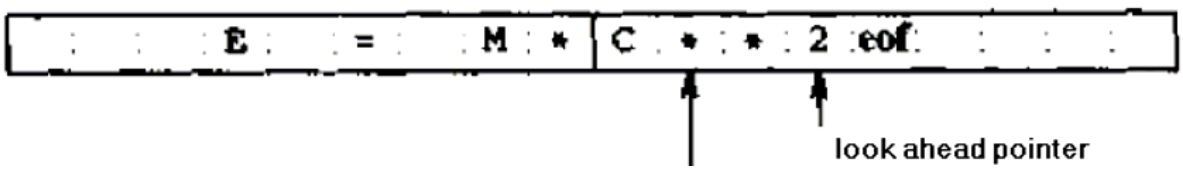
f i (a == f (x)) . ..

a lexical analyzer cannot tell whether f i is a misspelling of the keyword if or an undeclared function identifier. Since f i is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters. However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate. Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

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. One pointer marks the beginning of the token being discovered. A look head pointer scans ahead of the beginning pointer, until a token is discovered.



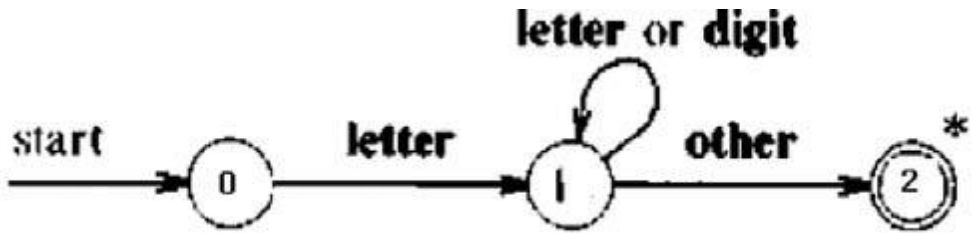
A simple approach to the design of lexical analysis

One way to begin the design of any program is to describe the behavior of the program by a flowchart.

Remembering previous character by the position flowchart is a valuable tool, so that a specialized kind of flowchart for lexical analyzer, called transition diagram has evolved.

In transition diagram, the boxes of the flowchart are drawn as circle and called states. The states are connected by arrows called edge. The labels on the various edges leaving a state indicate the input characters that can appear after that state.

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.



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 DELIM TER(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,…

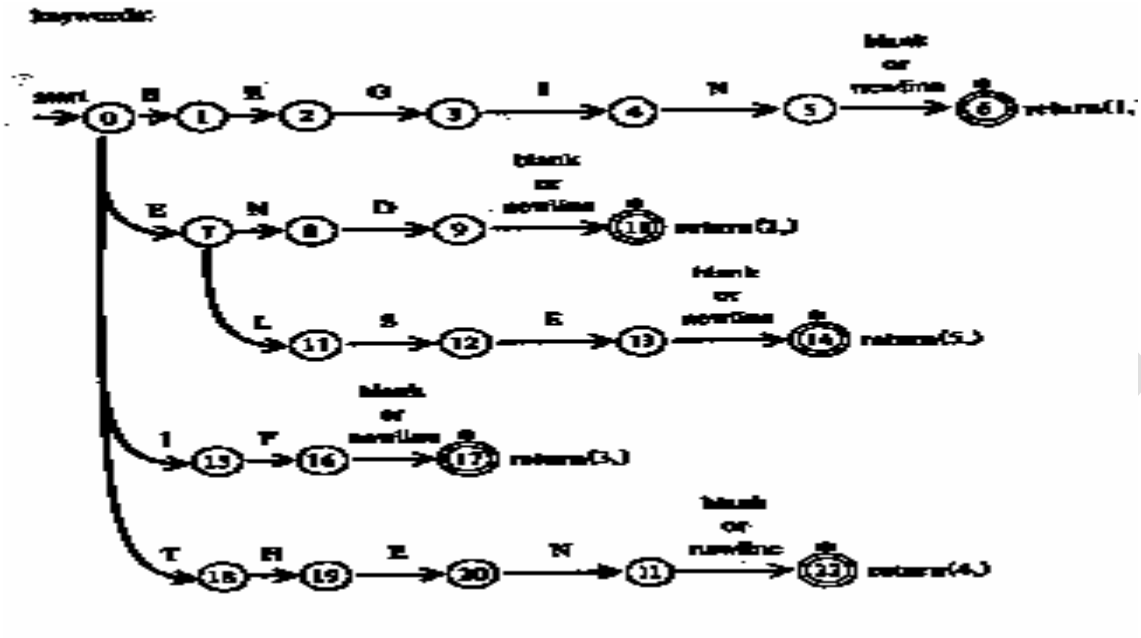
State2 indicates that an identifier has been found.

Since the delimiter is not part of the identifier, we must retract the look ahead pointer one character, for which we use a procedure RETRAC. We must install the newly found identifier in the symbol table if it is not already there , using the procedure INSTALL, In state2 we return to the parser a pair consisting of integer

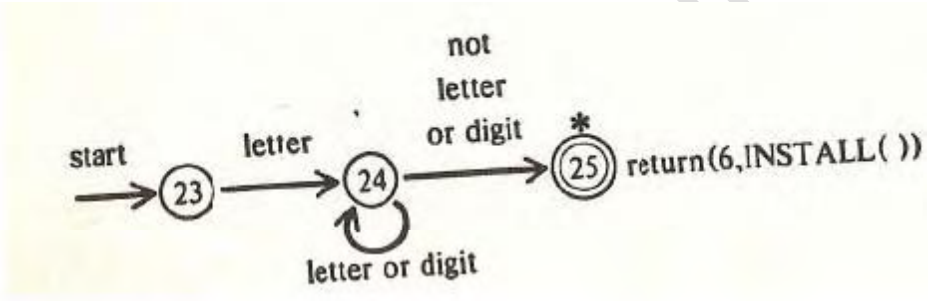
code for an identifier, which we denoted by id, and a value that is a pointer to the symbolic table returned by INSTALL.

Token	Type code	Value
Begin	1	-
End	2	-
If	3	-
Then	4	-
Else	5	-
Identifier	6	Pointer to symbol table
Constant	7	Pointer to symbol table
<	8	1
>	8	2
=	8	3
<>	8	4
+	9	1
-	9	2

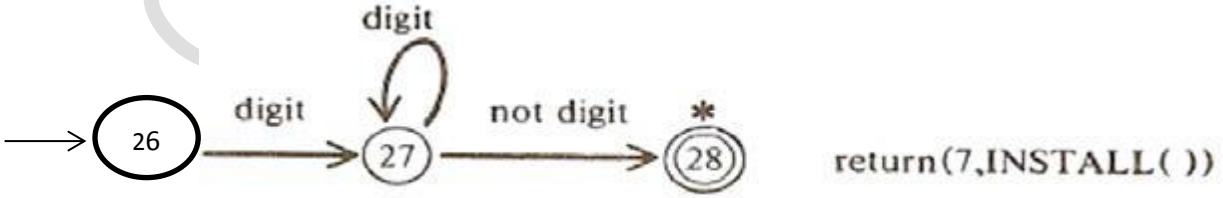
A more efficient program can be constructed from a single transition diagram than from a collection of diagrams, since there is no need to backtrack and rescan using a second transition diagram.



identifier:



Constant:



relops:

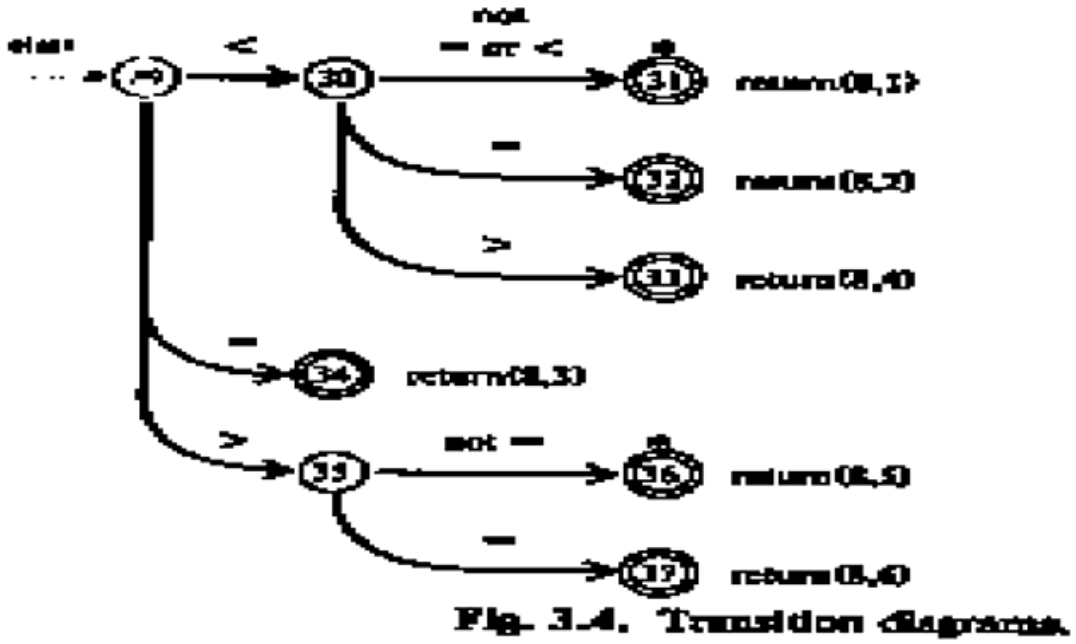


Fig. 3.4. Transition diagrams.

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.

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

S^2 = SS S^3 = SSS S^4 = SSSS

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 $L \cup M$	$L \cup M$={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 concatenations of" ***L***.

Positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations 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 ={∈}

Regular Definitions

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

Example1: The set of Pascal 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 Pascal 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.

Example2: 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 digit*

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

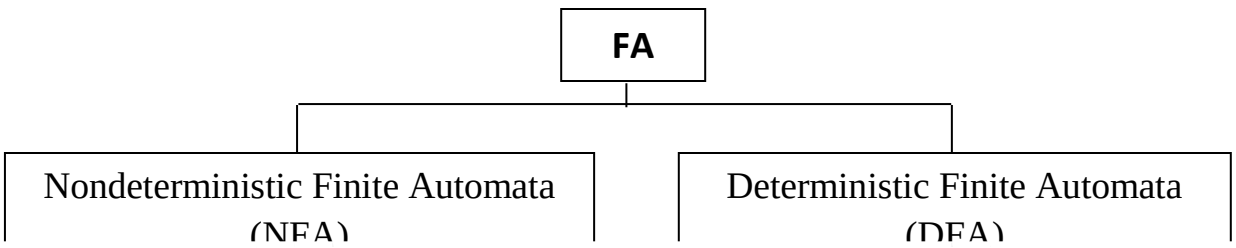
This regular definition says that

- 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.

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.



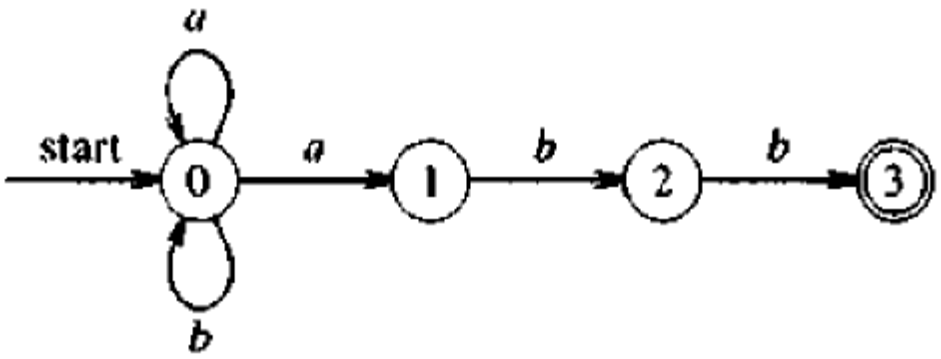
Nondeterministic *Finite Automata (NFA)*

A nondeterministic finite automation is a mathematical model consists of

1. a set of states S;
2. a set of input symbol, Σ, called the input symbols alphabet.
3. a transition function move that maps state-symbol pairs to sets of states.
4. a state so called the initial or the start state.
5. a set of states F called the accepting or final state.

An NFA can be described by a transition graph (labeled graph) where the nodes are states and the edges shows the transition function. The labeled on each edge is either a symbol in the set of alphabet, Σ, or denoting empty string.

Following figure shows an NFA that recognizes the language: (a | b)* a bb.



This automation is nondeterministic because when it is in state-0 and the input symbol is a, it can either go to state-1 or stay in state-0.

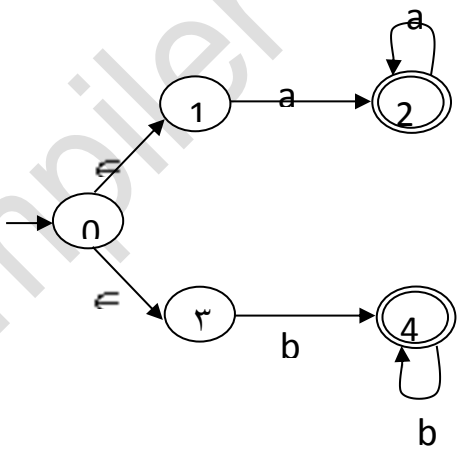
The transition is

State	Input Symbol	
	a	*b*
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

The advantage of transition table is that it provides fast access to the transitions of states and the disadvantage is that it can take up a lot of space.

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.

Algorithm for Simulating a DFA

INPUT:

- string x
- a DFA with start state, so . . .
- a set of accepting state's F.

OUTPUT:

- The answer 'yes' if D accepts x; 'no' otherwise.

The function move (S, C) gives a new state from state S on input character C.

The function 'nextchar' returns the next character in the string.

Initialization:

S := S_0

C := nextchar;

while not end-of-file do

S := move (S, C)

C := nextchar;

end

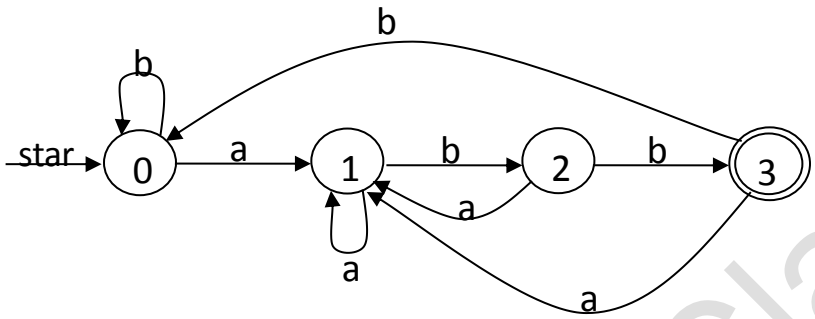
If S is in F then

return "yes"

else

return "No".

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



The Transition Table is:

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

With this DFA and the input string "ababb", above algorithm follows the sequence of states: 0,1,2,1,2,3 and returns "yes"

Typically few lexical error types

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

2. Insert a illegal character into the input.

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

4. Transpose two adjacent characters (Keyword Token).

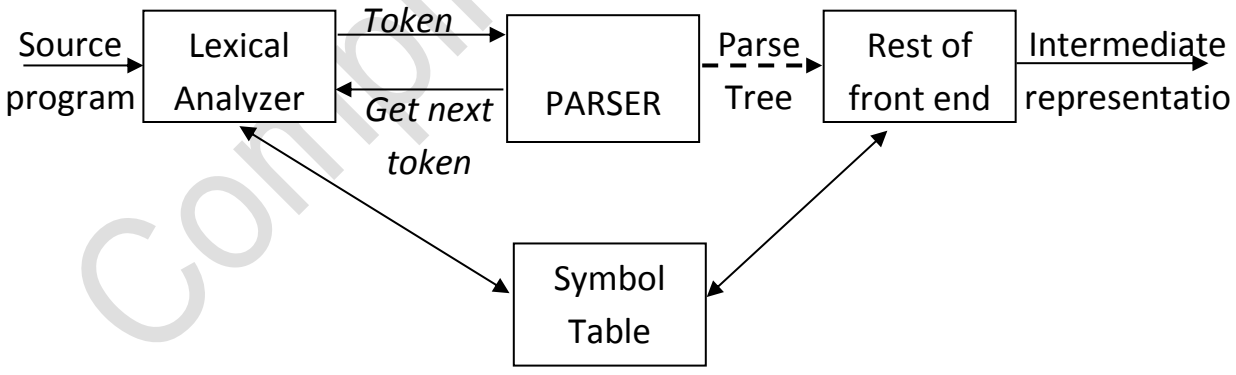
5. Misspelling of the keyword.

How is a Scanner Programmed?

1) Describe tokens with regular expressions.
2) Draw transition diagrams.
3) Code the diagram as table/program.

Syntax Analysis

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source program. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



Position of Parser in Compiler Model

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.

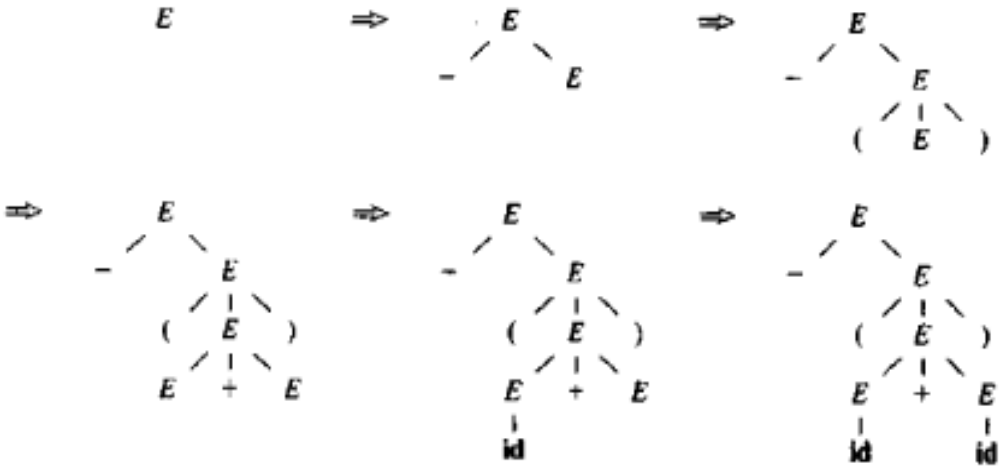
The methods commonly used in compilers are classified as being either Top-down or bottom-up. As indicated by their names, Top-down parsers build parse trees from the top (root) to the bottom (leaves) and work up to the root. In both cases, the input to the parser is scanned from left to right, one symbol at time. We assume the output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

Parse Tree and Derivations

A parse tree may be viewed as a graphical representation for an derivation that fillers out the choice regarding replacement order, that each interior node of parse tree is labeled by some non-terminal A, and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which A was replaced in the derivation, the leaves of the parse tree are labeled by non-terminals or terminals and, read from left to right, they constitute a sentential form, called the yield or frontier of the tree.

For example, the parse tree for -(id+id) implied previously is shown below, For the grammar

E⟶E+E | E-E | E*E | E/E



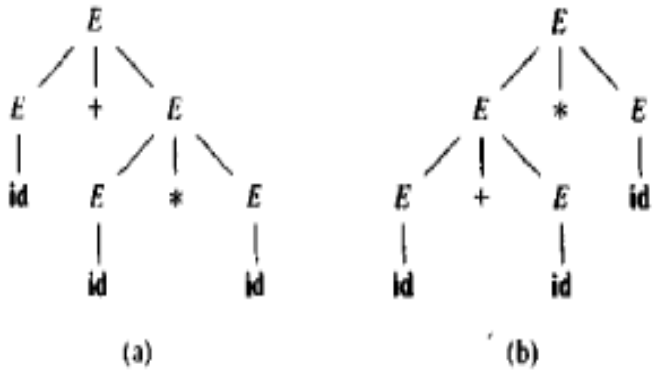
Parse tree ignores variations in the order in which symbols in sentential forms are replaced, these variations in the order in which productions are applied can also be eliminated by considering only left- most or right- most derivations. It is not hard to see that every parse tree has associated with it unique left most and unique right most derivations.

Example

Consider the previous arithmetic expression grammar, the sentence id+id*id has the two distinct left most derivations:

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

With the two corresponding parse trees shown below:



Two parse trees for id+id*id

Writing Grammar

Grammars are capable of describing most, but not all of syntax of programming languages. A limited amount of syntax Analysis is done by produce the sequence of tokens from the input characters, certain constraints on the input, such as the requirement that identifiers be declared before they are used, cannot be described by a context-free-grammar.

Every construct that can be described by a regular expression can also be described by a grammar. For example, the regular expression (a|b)*abb the NFA is:

A0 $\longrightarrow$ **aA0 | bA0 | aA1**

A1 $\longrightarrow$ **bA2**

A2 $\longrightarrow$ **bA3**

A3 $\longrightarrow$ **λ**

The grammar above was constructed from NFA using the following constructed:

• For each state i of NFA, create a non terminal symbol Ai.

• If state i has a transition to state j on symbol a, introduction the production

Ai $\longrightarrow$ aAj

• If state i goes to state j on input λ, introduce the production Ai $\longrightarrow$ Aj

• If state i is on accepting state introduce Ai $\longrightarrow \lambda$

• If state i is the start state, make Ai be symbol of the grammar.

RE' S are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.

Grammars, on the other hand, are most useful in describing nested structures such as balanced parenthesis, matching begin- end's. corresponding if - thenelse's.
These nested structures cannot be described by RE.

1- 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)

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

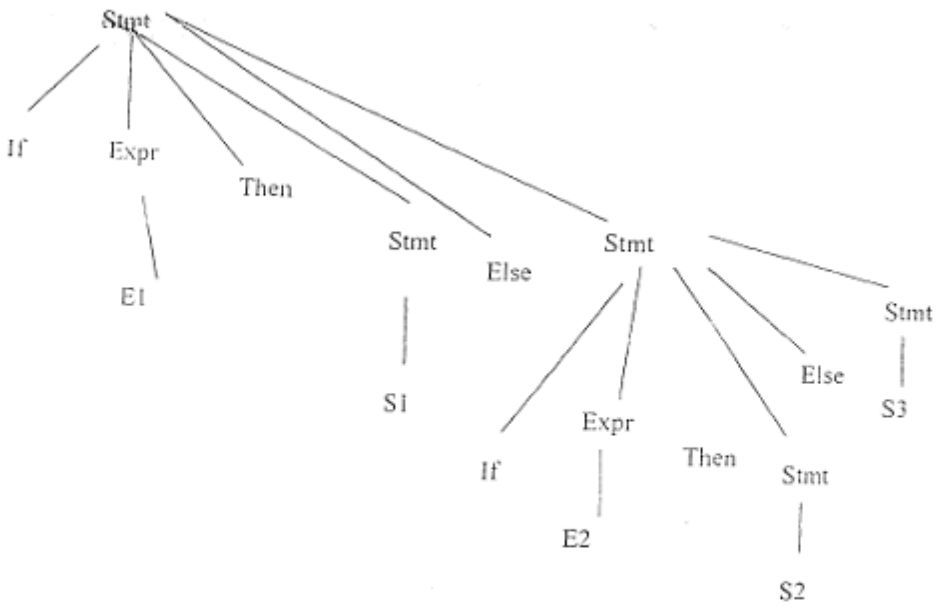
This grammar is ambiguous.

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. **As an example** ambiguo us "else" grammar

Stmt ⟶ Expr then Stmt
| if Expr then Stmt else Stmt
|other

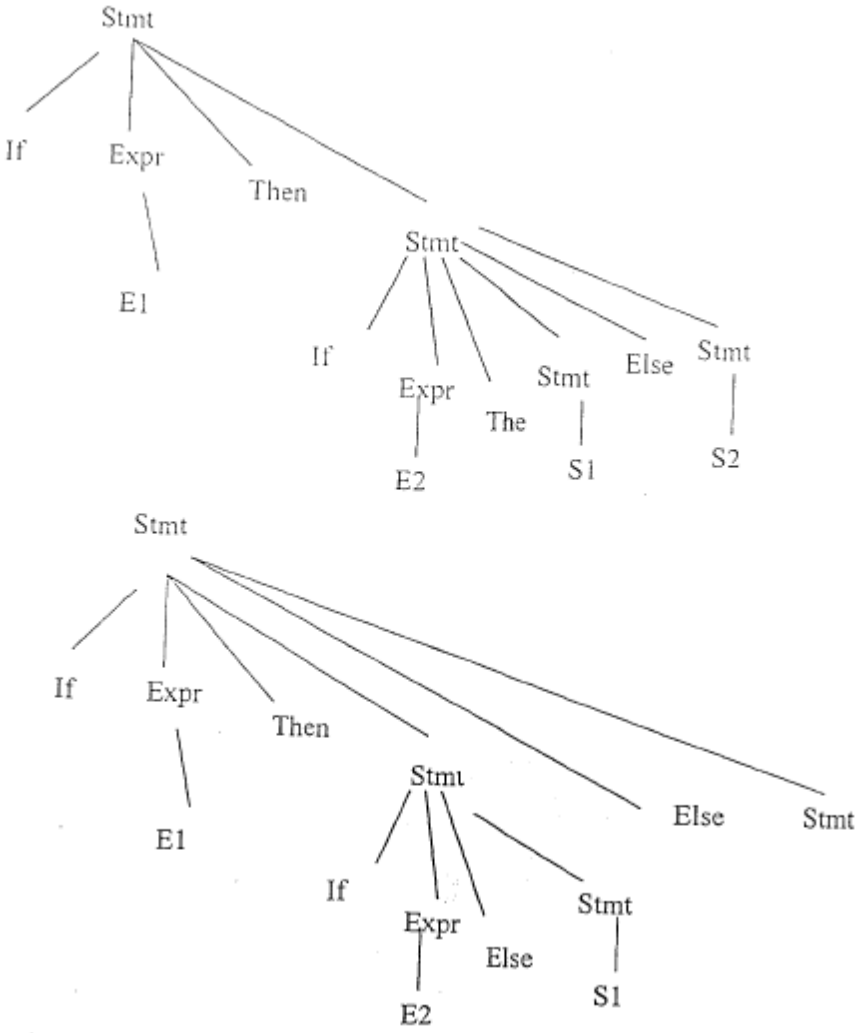
According to this grammar, the compound conditional statement

If El then S1 else if E2 then S2 else S3 has the parse tree link below:



The grammar above is ambiguous since the string:

If E1 then if E2 then S1 else S2. Has the two parse trees shown below:



Generating of Context-Free Grammar

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

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

$S \longrightarrow aAb$

$A \longrightarrow aA \mid bA \mid \in$

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

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

$X \longrightarrow bY$

$Y \longrightarrow bZ$

$Z \longrightarrow \in$

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

$S \longrightarrow aXb$

$X \longrightarrow aXb \mid \in$

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

$S \longrightarrow XD$

$X \longrightarrow + \mid -$

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

2- Left Recursion

A grammar is left recursion if it has a non terminal A, such that there is a derivation A αA Fer some string α. Top-down parsing methods cannot handle left recursion grammars, so a transformation that eliminates left recursion is needed. In the following example, we show how that left recursion pair of production

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

Could be replaced by the non left recursional productions:

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

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \lambda$

Example: Consider the following grammar for arithmetic expressions.

E ⟶ E+T |T

T ⟶ T*F|F

F ⟶ (E) | id

Eliminating the immediate left recursion (productions of the form A ⟶ Aα to the production for E and then for T, we obtain:

E ⟶ TE'

E' ⟶ +T E'| λ

T ⟶ FT'

T' ⟶ *FT' | λ

F ⟶ (E) | id

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 production as

A ⟶ Aα1| Aα2| …| Aαn| β1| β2| … βn

where no βi begins with an A. then, we replace the A -productions by

A ⟶ β1A'| β2A'| …| βnA'

A ⟶ α1A'| α2A'| …| αnA' | λ

This produce eliminates all immediate left recursion from A and A' production. But it does not eliminate left recursion involving derivation of two or more steps.

S ⟶ Aa | b

A ⟶ Ac | Sd | λ

The non terminal S is left recursion because S Aa Sda, but is not immediately left recursion.

3- 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.

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$A \longrightarrow \alpha A'$$

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

Algorithm : Left factoring a grammar.

Input: Grammar ***G***.

Output*:* An equivalent left-factored grammar.

Method: For each nonterminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, replace all 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:

$S \longrightarrow iEtSS' \mid a$

$S' \longrightarrow eS \mid \in$

$E \longrightarrow b$

Example:

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

Solution:

$A \longrightarrow aA' \mid bB'$

$A' \longrightarrow A \mid b \mid \in$

$B' \longrightarrow B \mid A$

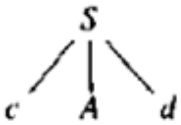
Top down parser

In this section there are basic ideas behind top-down parsing and show how constructs an efficient non- backtracking form of top-down parser called a predictive parser. Top down parsing can be viewed as attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to

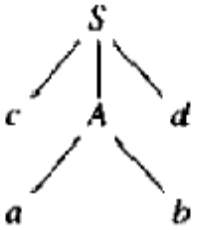
construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder. The following grammar requires **backtracking**:

$S \rightarrow cAd$
$A \rightarrow ab \mid a$

the input string $w = cad$. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w. We then use the first production for S to expand the tree and obtain

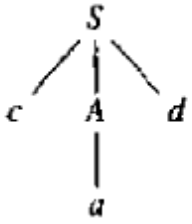


The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree



We now have a match for the second input symbol so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b. Since b does not match d, we report failure and go back to A to see whether there is another alternative for A that we have not tried but that might produce a match.

We now try the second alternative for A to obtain the tree:

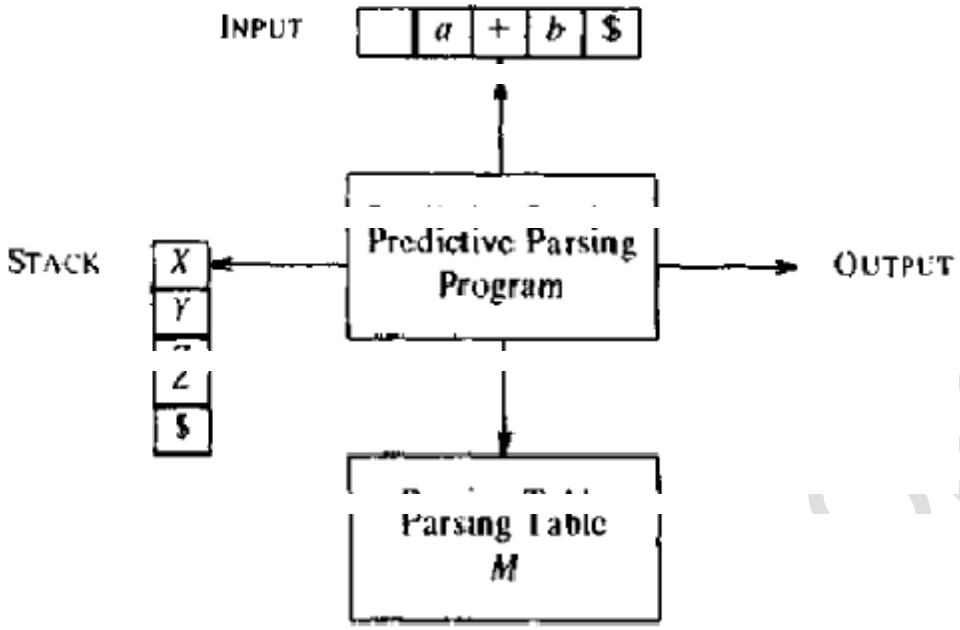


The leaf *a* matches the second symbol of *w* and the leaf *d* matches the third symbol. Since we have produced a parse tree for *w*, we halt and announce successful completion of parsing.

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 nonrecursive parser looks up the production to be applied in a parsing table.

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 $. The parsing table is a two-dimensional array M[A,a], where A is a nonterminal, and **a** is a terminal or the symbol $.



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

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

If M[X,a]= error, the parser calls an error recovery routine.

Algorithm nonrecusive predictive parser

Input. A string w and a parsing table M for grammar G.

Output. If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method. Initially, the parser is in a configuration in which it has \$$S$ on the stack with S, the start symbol of G on top, and w\$ in the input buffer.

```
– set ip to point to the first symbol of w$;
  repeat
       let X be the top stack symbol and a the symbol pointed to by ip;
       if X is a terminal or $ then
            if X = a then
                 pop X from the stack and advance ip
            else error()
       else      /* X is a nonterminal */
            if M[X, a] = X → Y1Y2 · · · Yk then begin
                 pop X from the stack;
                 push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
                 output the production X → Y1Y2 · · · Yk
            end
            else error()
   until X = $    /* stack is empty */
```

Example:

Parse the input **id * id + id** in the grammar:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

The parse table M for the grammar:

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

The moves made by predictive parser on input id+id*id

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \to TE'$
$E'T'F	id + id * id$	$T \to FT'$
$E'T'id	id + id * id$	$F \to id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \to \epsilon$
$E'T+	+ id * id$	$E' \to +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \to FT'$
$E'T'id	id * id$	$F \to id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \to *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \to id$
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

FIRST and FOLLOW

The construction of a predictive parser is aided by two functions associated with a grammar ***G***. These functions, **FIRST** and **FOLLOW**, allow us to fill in the entries of a predictive parsing table for ***G***, whenever possible.

Define the FIRST(α) to be the set of terminals that begin the strings derived from α, and the FOLLOW(A) for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form. To compute FIRST(x) for all grammar symbols x, apply the following rules until no more terminals or **ε** can be added to any first set.

1- If x is terminal, then FIRST(x) is {x}.

2- If X→ a ; is a production, then add a to FIRST(X) and

If X → **ε** ; is a production, then add **ε** to FIRST(X).

3- If X is nonterminal and X→Y1,Y2…Yi ; is a production, then add FIRST(Y1) to FIRST(X).

4- a- for (i = 1; if Y_i can derive epsilon **ε**; i++)

b- add First(Y_{i+1}) to First(X)

If Y1 does not derive **ε** , then we add nothing more to FIRST(X), but if

Y1→ **ε**, then we add FIRST(Y2) and so on .

First function examples

Example1

1- FIRST (terminal) = {terminal }

S → aSb │ba │ ε

FIRST (a) ={a}

FIRST (b) ={b}

2- FIRST(non terminal) = FIRST (first char)

FIRST (S)= {a,b, **ε** }

Example2: Consider the following grammar

$S \longrightarrow TabS \mid X$

$T \longrightarrow cT \mid \in$

$X \longrightarrow b \mid bX$

Sol:

FIRST (S) = {**c**, **a**, **b** }

FIRST (T) = {**c**, $\in$**}**

FIRST(X) = {**b}**

Example3: Consider the following grammar

$S \longrightarrow AB \mid bS$

$A \longrightarrow aB \mid BB$

$B \longrightarrow b \mid cB$

Sol:

FIRST (S) = {a, **b, c}**

FIRST (A) = {a, **b, c}**

FIRST (B) = {**b, c}**

Example4: Consider the following grammar

$S \longrightarrow abS \mid bX$

$X \longrightarrow \mathcal{E} \mid cN$

$N \longrightarrow N\mathbf{b} \mid \mathbf{c}$

There is the left recursion problem, so we must solve this problem before finding the first function

$S \longrightarrow \mathbf{ab}S \mid \mathbf{b}X$

$X \longrightarrow \in \mid \mathbf{c}N$

$N \longrightarrow \mathbf{c}N'$

$N' \longrightarrow \mathbf{b}N' \mid \in$

Sol:

FIRST (S) = {**a, b**}

FIRST (X) = {**c,** $\in$**}**

FIRST (N) = {**c**}

FIRST (N') = {**b,** $\in$**}**

To compute FOLLOW(A) for all non terminals A, is the set of terminals that can appear immediately to the right of A in some sentential form S → aAxB... To compute Follow, apply these rules to all nonterminal in the grammar:

1- Place $ in FOLLOW(S) , where S is the start symbol and $ is the input right end marker.

FOLLOW(START) = {$}

2- If there is a production X→ α Aβ , then everything in FIRST(β) except for **ε** is placed in FOLLOW(A).

i.e. FOLLOW(A) = FIRST(β)

3- If there is a production X→ α A, or a production X→ α Aβ , where FIRST(β) Contains ε (β → **ε**), then everything in FOLLOW(X) is in FOLLOW(A).

i.e. : FOLLOW(A)= FOLLOW(X)

Follow function examples:

Example 1:

S → aSb | X

X → cXb | b

X → bXZ

Z → **n**

First	**Follow**
S a , c , b	$, b
X c , b	b , n , $
Z n	b , n , $

Example 2:

S → bXY

X → **b | c**

Y → **b | ε**

First	**Follow**
S b	$
X b , c	b , $
Y b , ε	$

Example 3:

S → ABb | bc

A → ε | abAB

B → bc | cBS

First	Follow
S b , a , c	$, b , c , a
A ε , a	b , c
B b , c	b , c , a

Example 4:

X → ABC | nX

A → bA | bb | ε

B → bA | CA

C → ccC | CA | cc

First	Follow
X n , b , c	$
A b , ε	b , c , $
B b , c	c
C c	b , $

Construction of predictive parsing tables:

The following algorithm can be used to construct a predictive parsing table for a grammar G. The idea behind the algorithm is the following :

Suppose A→ α is a production with a in FIRST(α). Then the parser will expand A by α when the current input symbol is a. The only complication occurs when α→ . In this case, we should again expand A by α if the current input symbol is in FOLLOW(A).

Algorithm: Construction of a predictive parsing table.

Input: Grammar ***G***.

Output: Parsing table ***M***.

Method:

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

Example:

E→ E+T | T

T → T*F | F

F → (E) | id

Parse the string **id + id * id** by using predictive parser for the following grammar:

1- we must solve the left recursion and left factoring if it founded in the grammar

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

2- we must find the first and follow to the grammar:

FIRST (***E***) = FIRST (***T***) = FIRST (***F***) = {**(, id}**

FIRST (***E'***) = {**+,∈}**

FIRST (***T'***) = {***,∈}**

FOLLOW (***E***) = FOLLOW (***E'***) = {), $}

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

FOLLOW (***F***) = {*, +,), $}

For example, **id** and left parenthesis are added to FIRST(*F*) by rule (3) in the definition of FIRST with $i = 1$ in each case, since FIRST(**id**) = {**id**} and FIRST('(') = { (} by rule (1). Then by rule (3) with $i = 1$, the production $T \rightarrow FT'$ implies that **id** and left parenthesis are in FIRST(*T*) as well. As another example, ϵ is in FIRST(*E'*) by rule (2).

To compute FOLLOW sets, we put $ in FOLLOW(*E*) by rule (1) for FOLLOW. By rule (2) applied to production $F \rightarrow (E)$, the right parenthesis is also in FOLLOW(*E*). By rule (3) applied to production $E \rightarrow TE'$, $ and right parenthesis are in FOLLOW(*E'*). Since $E' \stackrel{*}{\Rightarrow} \epsilon$, they are also in FOLLOW(*T*). For a last example of how the FOLLOW rules are applied, the production $E \rightarrow TE'$ implies, by rule (2), that everything other than ϵ in FIRST(*E'*) must be placed in FOLLOW(*T*). We have already seen that $ is in FOLLOW(*T*). □

3- We must find or construct now the predictive parsing table

Since FIRST(TE') = FIRST(T) = {(,id}, production E→TE' causes M[E,( ] and M[E,id] to acquire the entry E→TE'. Production E'→+TE' causes m[E',+] to acquire E'→+TE'. Production E'→ causes M[E',)] and M[E',$] to acquire E'→€,since FOLLOW(E') = {),$}. So the parsing table produced by the previous algorithm.

NONTER-MINALS	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \longrightarrow TE'$			$E \longrightarrow TE'$		
E'		$E' \longrightarrow +TE'$			$E' \longrightarrow \in$	$E' \longrightarrow \in$
T	$T \longrightarrow FT'$			$T \longrightarrow FT'$		
T'		$T' \longrightarrow \in$	$T' \longrightarrow *FT'$		$T' \longrightarrow \in$	$T' \longrightarrow \in$
F	$F \longrightarrow$ **id**			$F \longrightarrow (E)$		

Predictive Parsing Table *M* For Above Grammar

Blanks are error entries; non-blanks indicate a production with which to expand the top nonterminal on the stack.

Stack	Input	Output
$ ***E***	**id + id * id$**	
$ ***E'T***	**id + id * id$**	$E \longrightarrow TE'$
$ ***E'T'F***	**id + id * id$**	$T \longrightarrow FT'$
$ ***E'T'*** **id**	**id + id * id$**	$F \longrightarrow$ **id**
$ ***E'T'***	**+ id * id$**	

$ *E'*	+ id * id$	$T' \longrightarrow \varepsilon$
$ *E'T*+	+ id * id$	$E' \longrightarrow +TE'$
$ *E'T*	id * id$	
$ *E'T'F*	id * id$	$T \longrightarrow FT'$
$ *E'T'* id	id * id$	$F \longrightarrow$ id
$ *E'T'*	* id$	
$ *E'T'F**	* id$	$T' \longrightarrow *FT'$
$ *E'T'F*	id$	
$ *E'T'* id	id$	$F \longrightarrow$ id
$ *E'T'*	$	
$ *E'*	$	$T' \longrightarrow \varepsilon$
$	$	$E' \longrightarrow \varepsilon$