Compiler

References:

1. Principle of compiler design
Alfred V. Aho & Jeffrey D. Ullman
2. Basics of compiler Design
Torben Egidius Mogensen
3. Compilers : principles, techniques, and tools
Alfred V. Aho & Jeffrey D. Ullman

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.

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.**

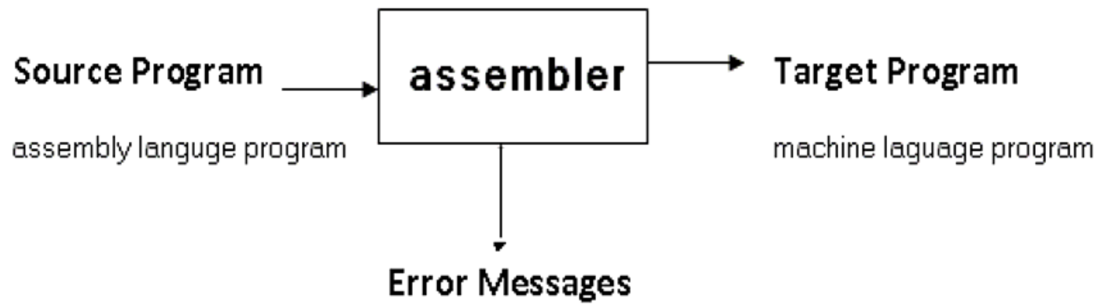


Fig (1)

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 (2) which illustrate the interpretation process.

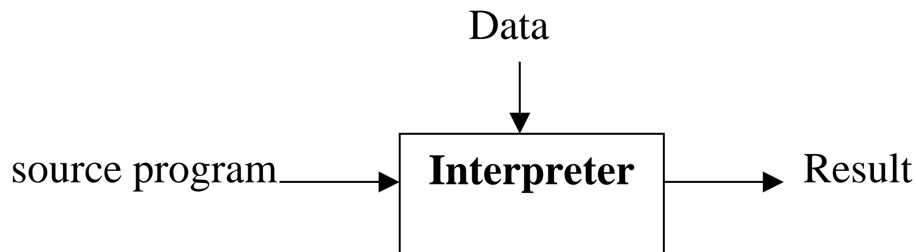


Fig (2)

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).

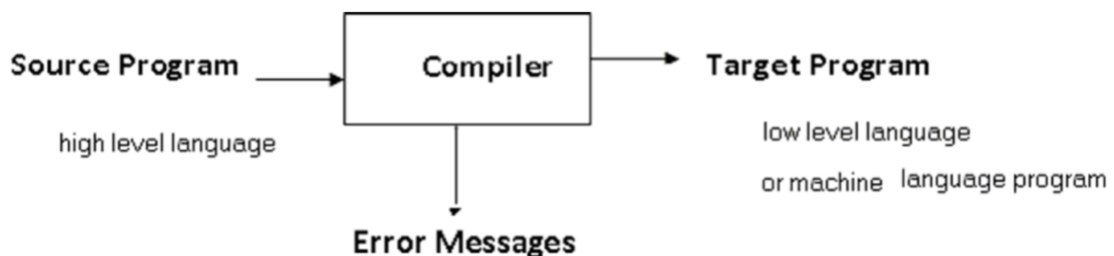


Fig (3)

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).

Fig (4) illustrate the compilation process Note that the program and data are processed at different times, compile time and run time respectively.

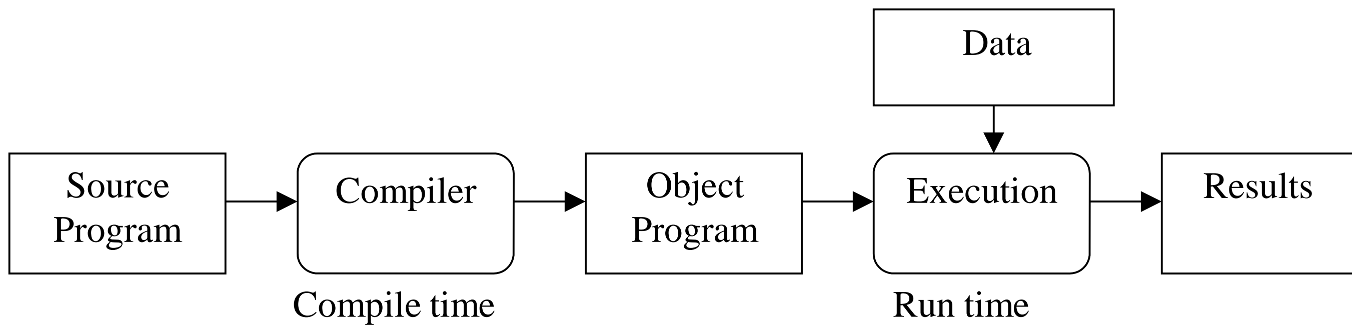
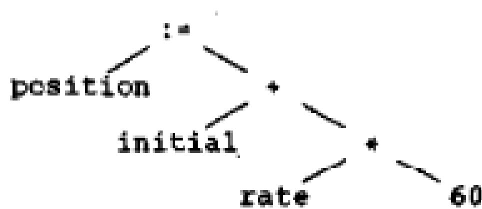


Fig (4) Compilation process

The Analysis - Synthesis model of compilation

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.

During analysis, the operations 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.

Compiler structure :

A compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in fig (5).

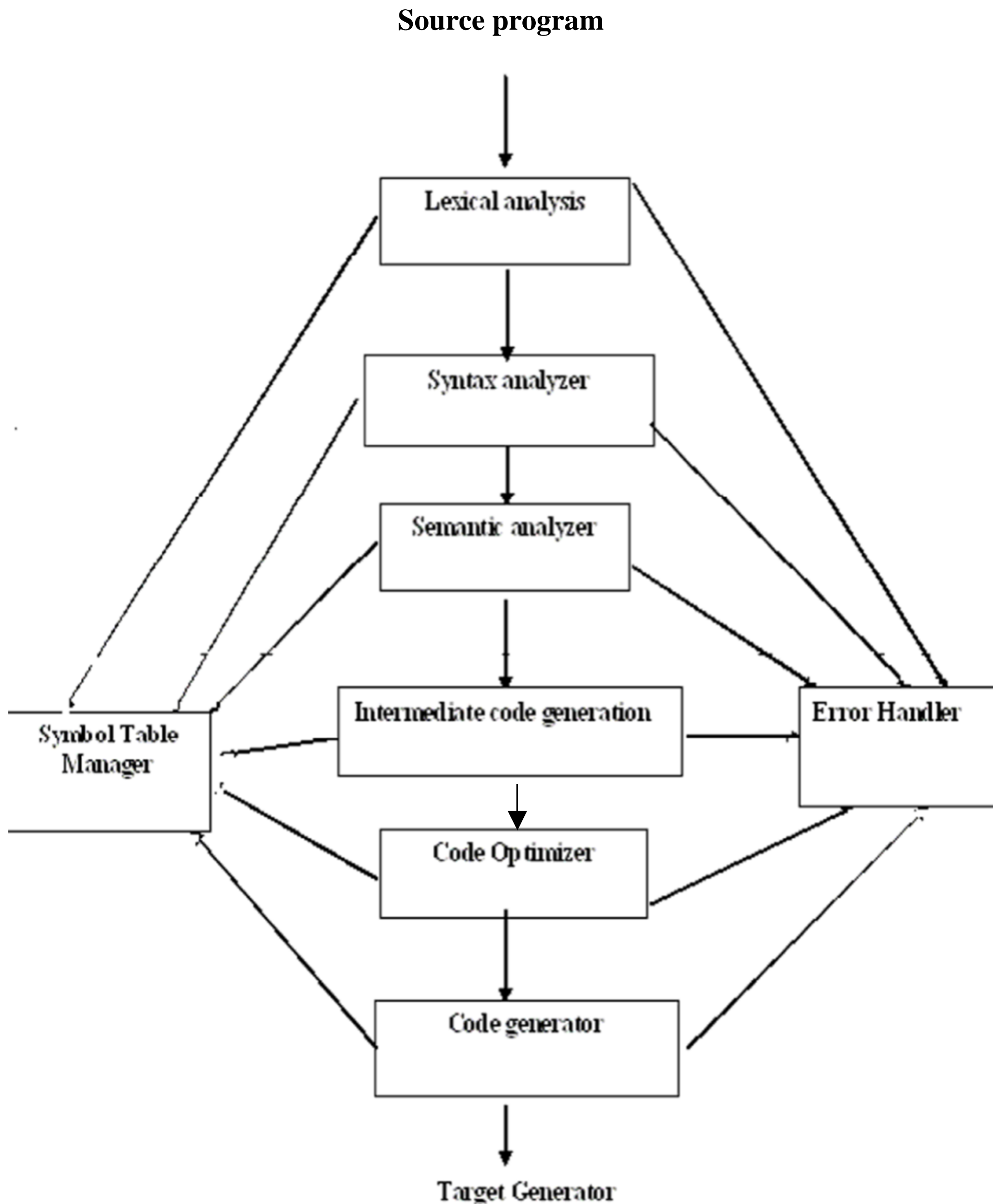


Fig (5) Phases of a Compiler

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.

2- syntax analysis (parsing)
The syntax analysis (or parsing) is the process of determining if a string of tokens can be generated by grammar. Every programming language has rules that prescribe the syntactic structure of well-formed programs.
Syntax Analyzer takes an out of lexical analyzer and produces a large tree

3- Semantic analysis
The semantic analysis phase checks the source program for semantics errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
Semantic analyzer takes the output of syntax analyzer and produces another tree.

4- Intermediate code generation
Generate an explicit intermediate representation of the source program. This representation should have two important properties, it should be easy to produce and easy to translate into the target program.

5- Code Optimization
Attempts to improve the intermediate code so that faster running machine code will result.

6- code generation
Generates a target code consisting normally of machine code or an assemble code. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are each translated in to a sequence of machine instructions that perform the same task.

- Symbol table management :

Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as type (integer, real, etc.). The data structure used to record this information is called symbolic table.

-Error handler:

Is called when an error in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can produced.

Types of errors

The syntax and semantic phases usually handle a large fraction of errors detected by compiler.

1. Lexical error: The lexical phase can detect errors where the characters remaining in the input do not form any token of the language . few errors are discernible at the lexical level alone ,because a lexical analyzer has a very localized view of the source program. Example : If the string fi is encountered in a C program for the first time in context:

fi (a== f(x)….

A lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function name. since **fi** is a valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

2- syntax error: The syntax phase can detect Errors where the token stream violates the structure rules (syntax) of the language.

3- semantic error: During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.

4- runtime error.

Error Detection and Reporting

Each phase can encounter errors after detecting an error, a phase must some how deal with that error, so the compilation can proceed, allowing further error in the source program to be detected. A compiler that stops when it finds the first error is not as helpful as it could be.

Passes:

In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by subsequent pass.

Lexical Analyzer

The lexical analyzer is the first phase of compiler. The main task of lexical Analyzer is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc.. for syntax analyzer. This interaction, summarized in fig.6, is commonly implemented by making the lexical analyzer be a subroutine of the parser. Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

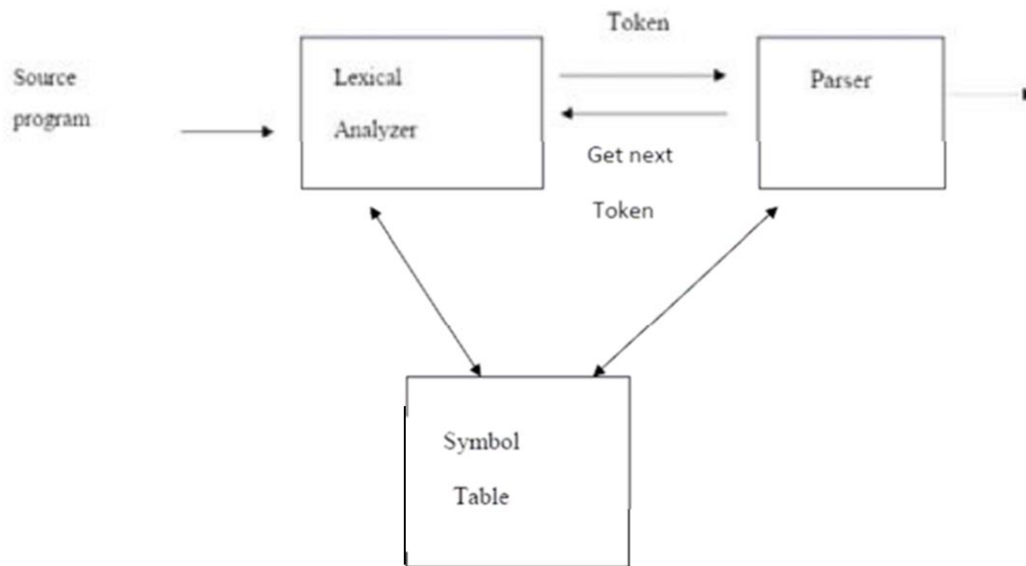


Fig. (6) Interaction of lexical analyzer with parser

Preliminary scanning :

Since the lexical analyzer is the part of compiler that reads the source text; it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, tab, and new line characters. Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of new line characters seen, so that a line number can be associated with an error message.

Some times, lexical analyzers are divided into a cascade of two phases, the first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations. For example, a FORTRAN compiler might use a scanner to eliminate blanks from the input.

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 (7) Examples of tokens

Token

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. In most programming language , the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings (any characters between " and "), and punctuations symbols such as parentheses, commas, and semicolons.
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)**

Attributes for Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern **num** matches both the strings 0 and 1, but it is essential for the code generator to know what string was actually matched.

The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions; the attributes influence the translation of tokens. As a practical matter, a token has usually only a single attribute — a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token. For diagnostic purposes, we may be interested in both the lexeme for an identifier and the line number on which it was first seen. Both these items of information can be stored in the symbol-table entry for the identifier.

Example The tokens and associated attribute-values for the Fortran 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**, >

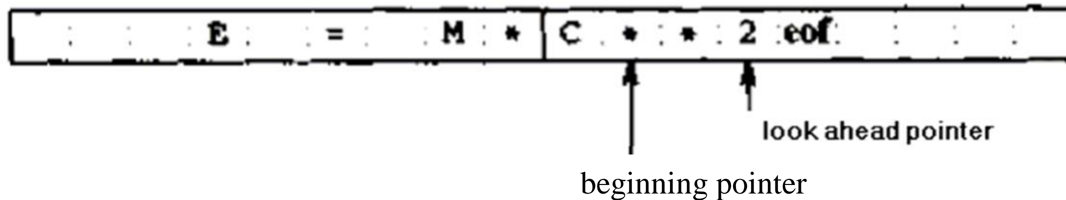
<**id**, pointer to symbol-table entry for C>

<**exp_op**, >

<**num**, integer value 2>

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 ahead pointer scans ahead of the beginning pointer, until a token is discovered.



beginning pointer

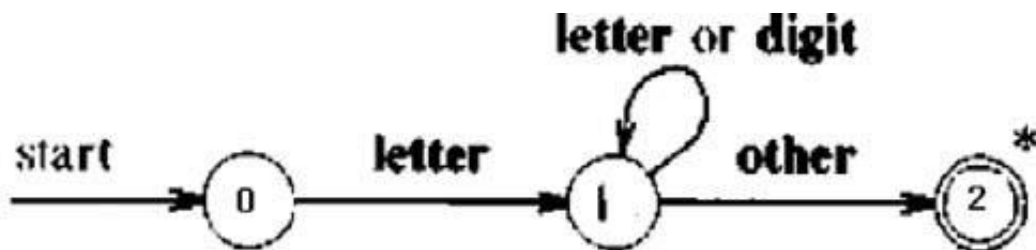
A simple approach to the design of lexical analysis

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.

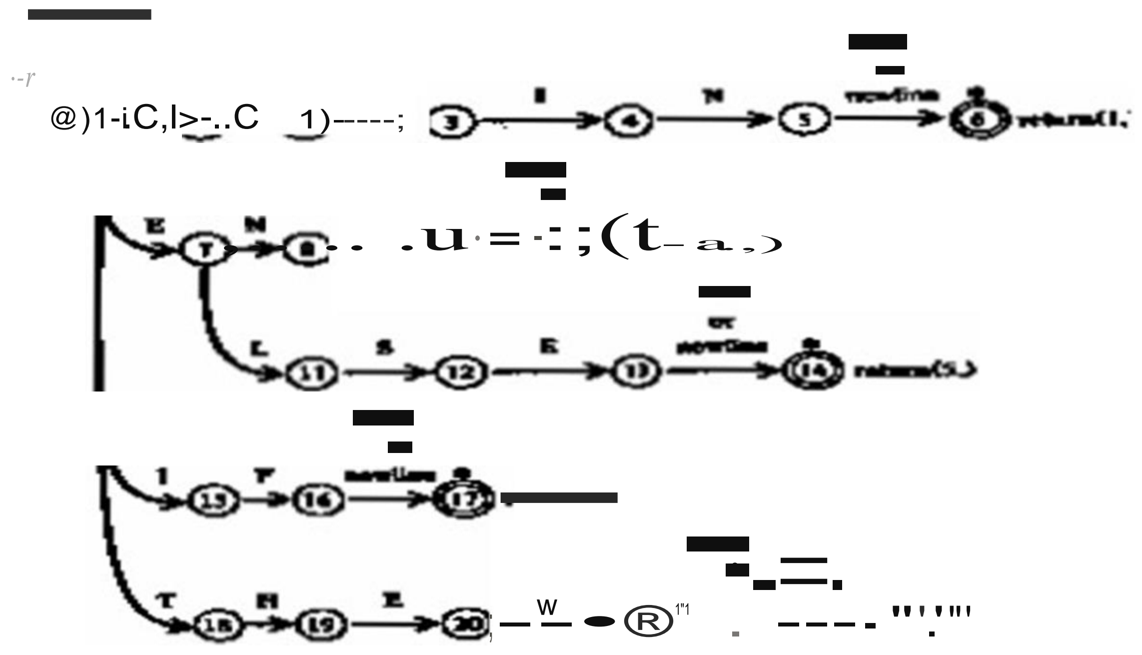


State 0 : C = GETCHAR ( )
if LETTER(C) then goto state1
else FAIL( )
State1 : C= GETCHAR ( )
if LETTER(C) or DIGIT(C) then goto state1
else if DELIMITER(C) then goto state2
else FAIL ( )
State2: RETRACT( )
return(id,INSTALL( ))

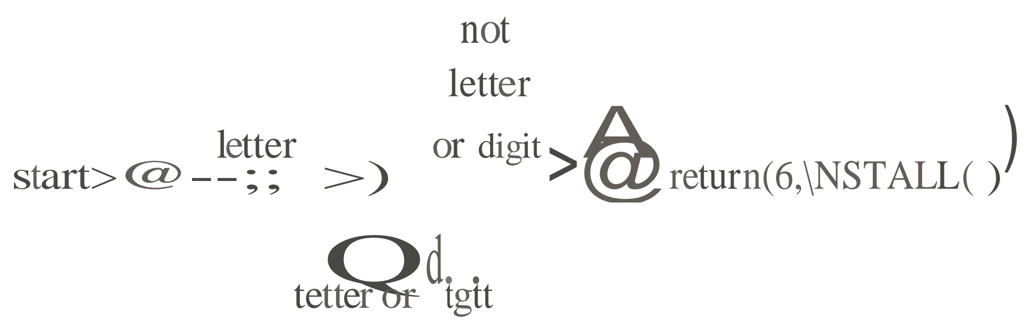
LETTER(C) is a procedure which return true if and only if C is a letter.
DIGIT(C) is a procedure which return true if and only if C is one of the digits 0,1,…,9.
DELIMITER(C) is a procedure which return true whenever C is character that could follow an identifier. The delimiter may be: blank, arithmetic 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:

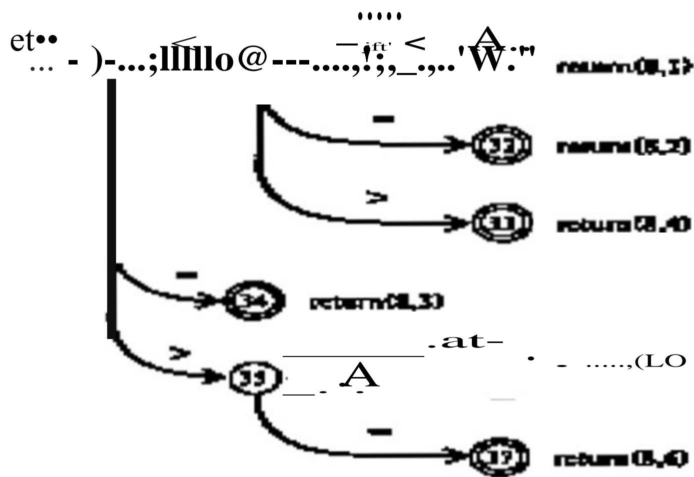


Fig. 3.4. Transition diagrams.

- Specifkation ofTokens

An alphabet or a character class is a finite set of symbols. Typical examples of symbols arletters and charactrs.

Patterns

Thset of strings is described by a mle called a pattem associatcl with the token.

Regular expressions are an imp01tant notation for specifying pattems. For example, the pattem forthPacal identifir token, id, is: id --+ l ttr (lett r | digit)*'.

Strings

A string over some alphabet is a finite sequence of symbol taken from that alphabet.

For xample, banana ia sequ nce of six symbols (i.e., string of l ngth six) tak n from ASCII computr alphabet.

The empty string denotE'd by e, is a special string with zero symbols (i e., string length is 0).

If x and y are two strings, then the concatenation of x andy, written 'f0', is the string fonued by appending y to x.
For example, If x =dog andy= house, then xy =doghouse.

For mpty string, E, we have Sf = ES = S.

String exponentiation concatenates a string with itself a given number of times:

S^2 = SS or S.S
S^3 = SSS or S.S.S
S^4 = SSSS or S.S.S.S and so on

By definition S^0 is an empty string, ϵ, and S` = S. For example, if x =ba and na then xy^2 = banana.

Languages
A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.

Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then

- Union: LUM = {dog, ba, na, house}
- Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}
- Exponentiation: L^2 = LL
- By definition: L^0 ={ ϵ} and L` = L

The kleene closure of language L, denoted by L*, is "zero or more Concatenation of" L.

$L^* = L^0 \cup L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$

For example, If L = {a, b}, then

L* = { ϵ, a, b, aa, ab, ba, bb, aaa, aba, baa, . . . }

The positive closure of Language L, denoted by L^+, is "one or more Concatenation of" L.

$L^+ = L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$

For example, If L = {a, b}, then

L^+ = {a, b, aa, ba, bb, aaa, aba, . . . }

Regular Definitions

Regular expressions is a useful notation suitable for describing tokens . A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Example1 :

keyword = BEGIN|END|IF|THEN|ELSE

Identifier = letter (letter|digit)*

constant = digit+

relop = <|<=|=|<>|>|>=

example2: Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letter and digits beginning with a letters.

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

The regular expression id is the pattern for the 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.

Example3: The pattern for the Pascal unsigned token can be specified as follows:

digit → 0 | 1 | 2 | . . . | 9
digits → digit digit*

Optimal-fraction → **.** digits | ε
Optimal-exponent → (E (+ | - | ϵ) digits) | ε
num → digits optimal-fraction optimal-exponent

This regular definition says that

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

Finite automata

A 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.

One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automation.

Nondeterministic *Finite Automata (NFA)*

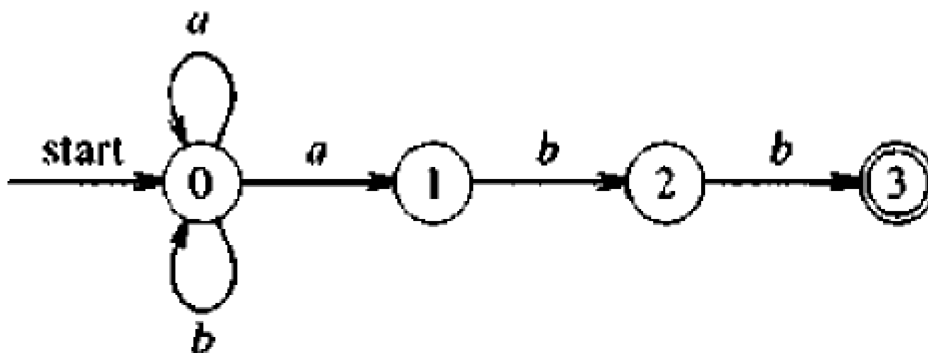
A nondeterministic finite automation is a mathematical model consists of

1. a set of states S;
2. a set of input symbol, ∑, 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, ∑, 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.

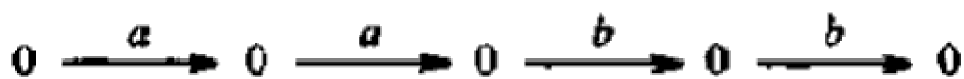
The transition is

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.

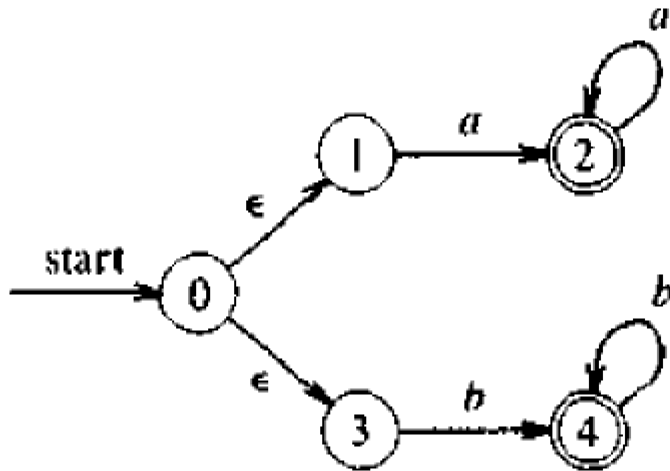
The following diagram shows the move made in accepting the input strings aabb :

In general, more than one sequence of moves can lead to an accepting state. If at least one such move ended up in a final state



The language defined by an NFA is the set of input strings that particular NFA accepts.

Following figure shows an NFA that recognize aa* | bb*.



String aaa is accepted by moving through states 0,1,2,2 and 2

Deterministic Finite Automata (DFA)

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

1. no state has an ϵ-transition
2. for each state s and input symbol a, there is at most one edge labeled a leaving s.

A DFA has at most one transition from each state on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA).

it is very easy to determine whether a DFA accepts an input string, since there is at most one path from the start state labeled by that string.

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.

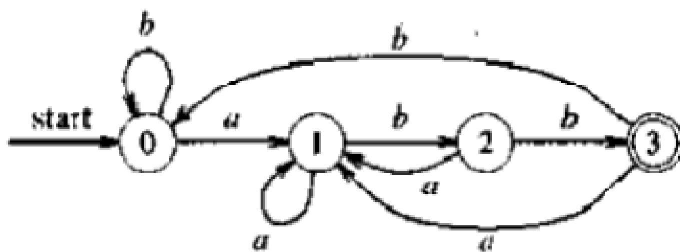
The function 'nextchar' returns the next character in the string.

Initialization:

```
    S := S0
   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 :

Following figure shows a DFA that recognizes the language (a|b)*abb.



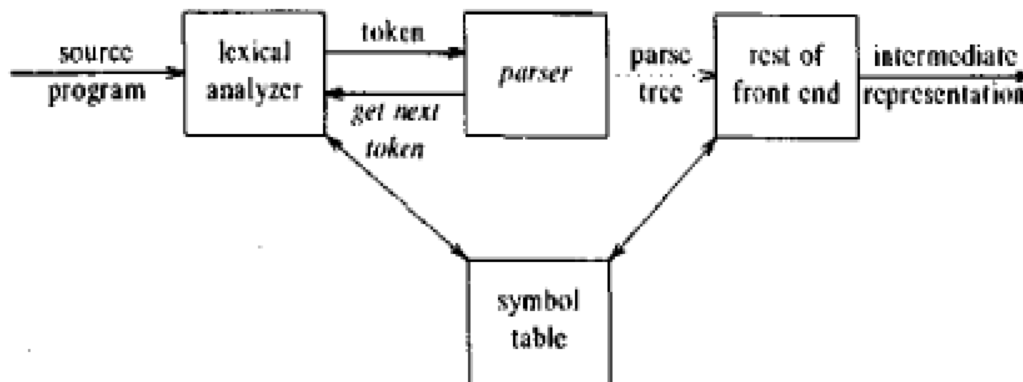
The transition table is

b	a	state
0	1	0
2	1	1
3	1	2
0	1	3

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

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.



Fig(8) Position of parser in Compiler model

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 <u>yield</u> or <u>frontier</u> of the tree. **For example**, the parse tree for -(id+id) implied previously is shown bellow, 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.

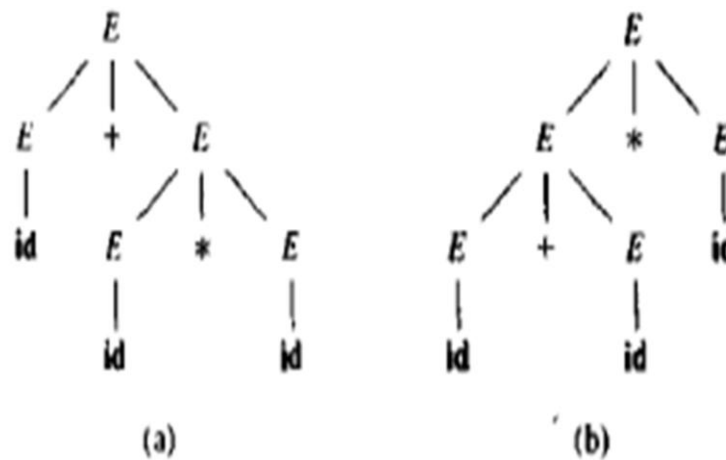
Example

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

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

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

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 a mount of syntax Analysis is done by lexical analyze as it 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, can not 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:

$A0 \longrightarrow aA0 \mid bA0 \mid aA1$

$A1 \longrightarrow bA2$

$A2 \longrightarrow bA3$

$A3 \longrightarrow \lambda$

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 $\longrightarrow$ λ
- 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 - then-else's.
These nested structures cannot be described by RE.

1- Ambiguity: (problems of grammar)

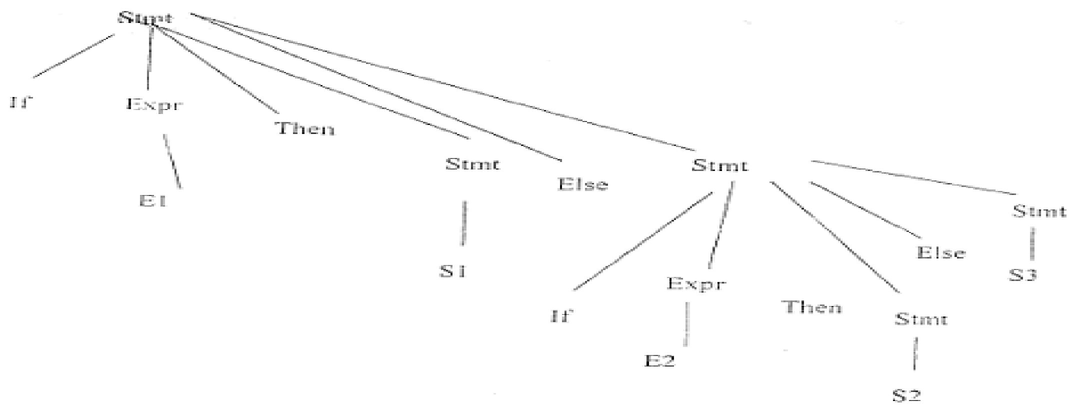
A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar is one that produces more than one leftmost or more than one right most derivation for the same sentence. For certain types of parsers, it is desirable that the grammar be made unambiguous, for if it is not, we can not uniquely determine which parse tree to select for a sentence.

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

Stmt $\longrightarrow$ 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: