

MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution



PG DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: PRINCIPLES OF COMPILER DESIGN

SUBJECT CODE: PSA3G

SEMESTER: III

PREPARED BY: PROF.G.GAYATHRY

UNIT I
Introduction to Compilers - Finite Automata and lexical Analysis.
UNIT II
Syntax Analysis: Context free grammars - Derivations and parse trees – Basic parsing techniques - LR parsing.
UNIT III
Syntax - directed translation, symbol tables.
UNIT IV
Code optimization - More about code optimization.
UNIT V
Code generation - Error detection and recovery.

UNIT-I

1.1 Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

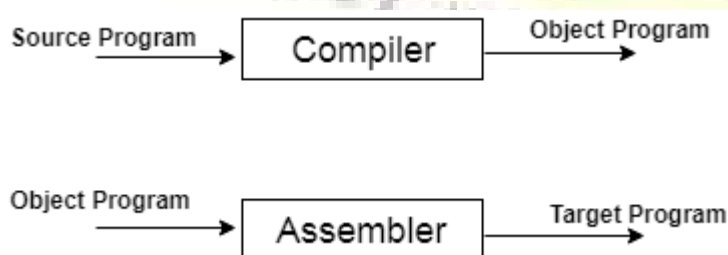


Fig: Execution process of source program in Compiler

1.2 Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

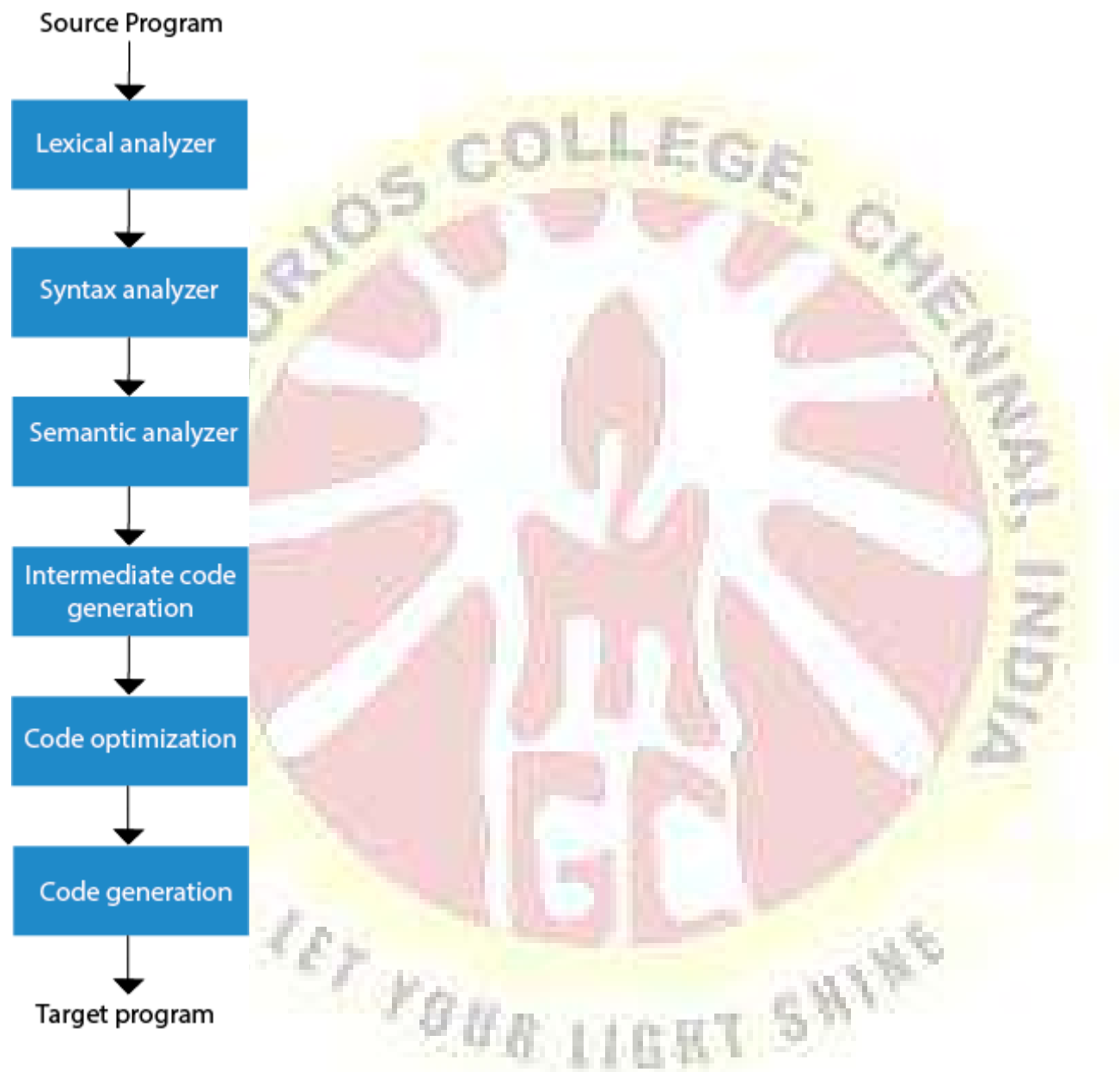


Fig: phases of compiler

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

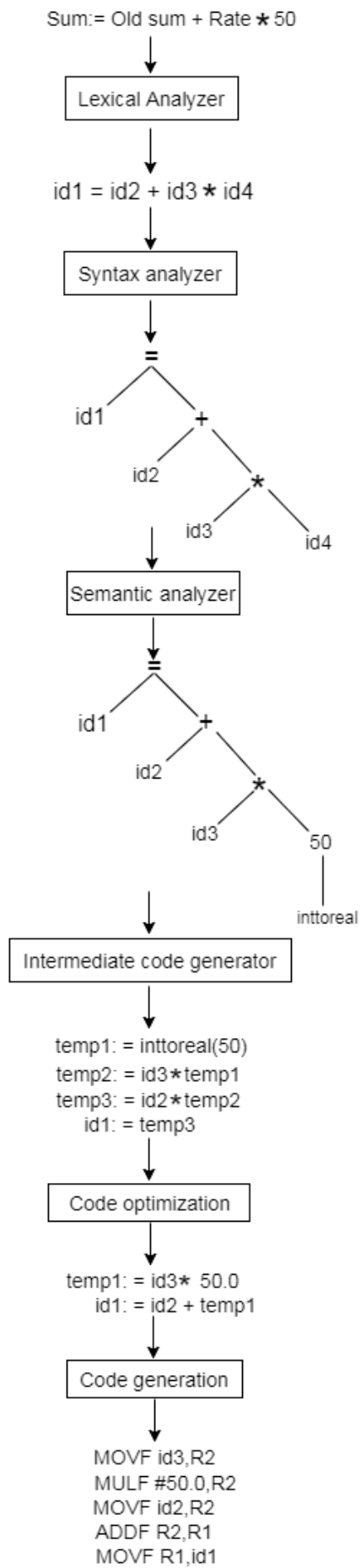
In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Code Optimization

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Example:

1.3 Lexical Analysis Phase

The purpose of the lexical analyzer is to read the source program, one character at time, and to translate it into a sequence of primitive units called tokens. Keywords, identifiers, constants, and operators are examples of tokens.

- **The Role of the Lexical Analyzer**

The lexical Analyzer could be a separate pass, placing its output on an intermediate file from which the parser would then take its input, or the lexical analyzer and parser are together in the same pass where the lexical analyzer acts as a subroutine which is called by the parser whenever it needs a new token. This organization eliminates the need for the intermediate file. In this arrangement, the lexical analyzer returns to the parser a representation for the token it has found. The representation is an integer code if the token is a simple construct such as a left parenthesis, comma or colon. The representation is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant. The integer code gives the token type and the pointer points to the value of that token. Pairs are also returned whenever we wish to distinguish between instances of tokens. For example, we may treat "operator" as a token and let the second component of the pair indicate whether the operator found is +, *, and so on.

- **The Need for Lexical Analyzer**

The purpose of splitting the analysis of the source program into two phases, lexical analysis and syntax analysis is to simplify the overall design of a compiler. It is easier to specify the structures of tokens than the syntactic structure of the source program. Consequently, we can construct a more specialized, and hence more efficient, recognizer for tokens than for the syntactic structure.

Other functions sometimes performed by the lexical analyzer are keeping track of line numbers, stripping out white space (such as redundant blanks and tabs), and deleting comments.

Specification of tokens

String and languages

First we introduce some terms that dealing with languages. We shall use the term

"alphabet" or "character class", to denote any finite set of symbols.

The set $\{0, 1\}$ is an alphabet. It consists of the two symbols 0 and 1, and it is called binary alphabet. Two important examples of programming language alphabets are the ASCII and EBCDIC character sets.

A string is a finite sequence of symbols, such as 0011. Sequence and word are synonyms for string. The length of the string x , usually denoted $|x|$ is the total number of the symbols in x . For example 01101 is a string of length 5. A special string is the empty string, which we shall denote by ϵ . This string is of length zero.

If x and y are string, then the concatenation of x and y ; written $x.y$ or xy , is the string formed by the symbol of x followed by the symbol of y . For example if $x=abc$ and $y=de$, where a, b, c, d, e are symbols, then $xy=abcde$. The condition of the empty string with any string is that string, more formally $\epsilon x = x\epsilon = x$

We may think of condition as a "product". It thus makes sense to talk of exponentiation of string as representing an iterated product. For example, $x^1=x$, $x^2=xx$, $x^3=xxx$ and so on. In general, x^i is the string x repeated i times. As a useful convention, we take x^0 to be ϵ for any string x . Thus, ϵ , is the identity of concatenation.

The term languages to mean any set of string formed from some specific alphabet.

The notation of concatenation can also be applied to languages. If L and M are languages, then $L.M$ is the language consisting of all string xy , which can be found by selecting a string x from L , and a string y from M , and concatenating them in that order. That is,

$LM = \{xy | x \text{ is in } L \text{ and } y \text{ in } M\}$ we call LM the concatenation of L and M .

Example: Let L be $\{0, 01, 110\}$, and let M be $\{10, 110\}$.

Then $LM = \{010, 0110, 01110, 11010, 110110\}$.

We use L^i to stand for $LL \dots L$ (i times). It is logical to define L^0 to be $\{\epsilon\}$.

The union of languages L and M is given by

$$L \cup M = \{x \mid x \text{ is in } L \text{ or } x \text{ is in } M\}.$$

The empty set, \emptyset , is the identity under union, since

$$\emptyset \cup L = L \cup \emptyset = L$$

And $\emptyset L = L \emptyset = \emptyset$

There is another operation on languages which plays an important role in specifying tokens. This is the Kleen closure operator. We use L^* to denote the concatenation of language L with itself any number of times.

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Example

Let D be the language consisting of the string 0, 1... 9, that is, each string is a single decimal digit. Then D^* is all strings of digits, including the empty string. For example, if $L = \{aa\}$, then L^* is all string of an even number of a's, since $L^0 = \{\epsilon\}$, $L^1 = \{aa\}$, $L^2 = \{aaaa\}$, If we wished to exclude ϵ , we could write $L.(L^*)$, to denote that language. That is:-

$$L.(L^*) = L.\bigcup_{i=0}^{\infty} L^i = \bigcup_{i=0}^{\infty} L^{i+1} = \bigcup_{i=1}^{\infty} L^i$$

We shall often use the L^* for $L.(L^*)$. The unary postfix operator $+$ is called positive closure, and denotes "one or more instances of".

1.4 Definition of Regular Expression

After the definition of the string and languages, we are ready to describe regular expressions, the notation we shall use to define the class of languages known as regular sets. Recall that a token is either a single string (such as a punctuation symbol) or one of a collection of string of a certain type (such as an identifier). If we view the set of strings in each token class as a language, we can use the regular-expression notation to describe tokens.

In regular expression notation we could write the definition for identifier as:

Identifier = letter (letter | digit)*

The vertical bar means "or" that is union, the parentheses are used to group sub expressions, and the star is the closure operator meaning "zero or more instances".

What we call the regular expression over alphabet Σ are exactly those expressions that can be constructed from the following rules. Each regular expression denotes a language and we give the rules for construction of the denoted languages along with the regular-expression construction rules.

1- ϵ Is a regular expression denoting $\{\epsilon\}$, that is, the language consisting only the empty string.

2- For each a in Σ , a is a regular expression denoting $\{a\}$, the language with only one string, that string consisting of the single symbol a .

3- If R and S are regular expression denoting language L_R and L_S , respectively, then:-

i) $(R) \mid (S)$ is a regular expression denoting $L_R \cup L_S$

ii) $(R) \cdot (S)$ is a regular expression denoting $L_R \cdot L_S$

iii) $(R)^*$ is a regular expression denoting L_R^*

1- The regular expression a^* denotes the closure of the language $\{a\}$, that is

$$a^* = \bigcup_{i=0}^{\infty} \{a^i\}$$

The set of all strings of zero or more a's. The regular expression aa^* , which by our precedence rules is parsed $a(a)^*$, denote the strings of one or more a's. We may use a^+ for aa^*

2- What does the regular expression $(a \mid b)^*$ denote? We see that $a \mid b$ denotes $\{a, b\}$, the language with two string **a** and **b**. Thus $(a \mid b)^*$ denote

$$\bigcup_{i=0}^{\infty} \{a, b\}^i$$

3- The expression $a \mid ba^*$ is grouped $a \mid (b(a)^*)$, and denotes the set of strings consisting of either a single "a" or "b" followed by zero or more a's.

4- The expression $aa \mid ab \mid ba \mid bb$ denotes all strings of length two, so $(aa \mid ab \mid ba \mid bb)^*$ denotes all strings of even length. Note that ϵ is a string of length zero.

5- $\epsilon \mid a \mid b$ denotes strings of length zero or one.

Example: The token discussed in fig (5), can be described by regular expression as follows:

Keyword=BEGIN \mid END \mid IF \mid THEN \mid ELSE

Identifier=letter (letter \mid digit)*

Constant=digit*

Relops= < \mid <= \mid = \mid <> \mid > \mid >=

Where letter stands for A \mid B \mid ... \mid Z, and digit stands for 0 \mid 1 \mid ... \mid 9.

For any regular expression R, S and T, the following axioms hold:-

- 1- $R \mid S = S \mid R$ (\mid is commutative)
- 2- $R \mid (S \mid T) = (R \mid S) \mid T$ (\mid is associative)
- 3- $R (ST) = (RS) T$ ($.$ is associative)
- 4- $R(S \mid T) = RS \mid RT$
and $(S \mid T) R = SR \mid TR$ ($.$ distributes over \mid)
- 5- $\epsilon R = R \epsilon = R$ (ϵ is the identity for concatenation)

1.5 A simple Approach to the Design of Lexical Analyzers

One way to begin the design of any program is to describe the behavior of the program by a flowchart. This approach is particularly useful when the program is a lexical analyzer, because the action taken is highly dependent on what characters have been seen recently. Remembering previous characters by the position in a flowchart is a valuable tool, so much so that a specialized kind of flowchart for lexical analyzer, called a transition diagram, has evolved. In a transition diagram, the boxes of the flowchart are drawn as circles and called states. The states are connected by arrow, called edges. The labels on the various edges leaving a state indicate the input characters that can appear after that state.

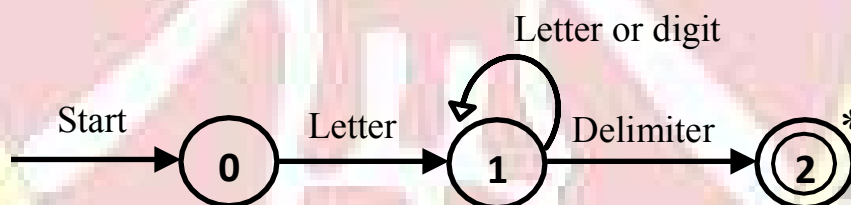


Fig (4) Transition diagram for identifier

Fig (4) shows a transition diagram for an identifier, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state 0, the edge from which indicates that the first input character must be a letter. If this is the case, we enter state 1 and look at the next input character if this is a letter or the digit, we continue this way, reading letters and digits, and making transition from state 1 to itself, until the next input characters is a delimiter for an identifier, which we have assume is any character that is not a letter or a digit. On reading the delimiter, we enter state 2.

To turn a collection of transition diagram 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 such an edge is found, control is transferred to the state pointed to by that edge. If no such edge is found, and the state is not one which indicated that a token has been found (indicated by a double circle), we have fail to find this token. The look ahead pointer must be retracted to where the beginning pointer is, and another token must be searched for, using another transition diagram. If all transition diagrams have been tried without success, a lexical error has been detected, and an error correction routine must be called.

Consider the transition diagram in fig (4), the code for state 0 might be:-

```
State 0:  C: = GETCHAR ();
          If LETTER(C) then goto state 1
          else FAIL ()
```

Here, LETTER is a procedure which returns true if and only if C is a letter. Fail() is a routine which retracts the lookahead pointer and starts up the next transition diagram, if there is one, or calls the error routine. The code for state1 is:

```
State 1   C:=GETCHAR ();
          if LETTER (C) or DIGIT (C) then goto state 1
          else if DELIMITER(C) then goto state 2
          else FAIL ()
```

DIGIT is a procedure which returns true if and only if C is one of the digits 0, 1... 9. DELIMITER is a procedure which returns true whenever C is a character that could follow an identifier. If we define a delimiter to be any character that is not letter or digit, then the clause "if DELIMITER (C) then", need not be presented in state 1. To detect errors more effectively we might define a delimiter precisely (e.g., blank, arithmetic or logical operator, left or right parenthesis, equal sign, colon, semicolon, or comma), depending on the language being compiled.

State 2 indicates that an identifier has been found. Since the delimiter is not part of the identifier, we must retract the lookahead pointer one character, for which we use a procedure RETRACT. We use '*' to indicate states on which input retraction must take place. We must also install the newly-found identifier in the symbol table if it is not already

there, using the procedure INSTALL^* . In state 2 we return a pair consisting of the integer code for an identifier, which we denote by id , and a value that is a pointer to the symbol table returned by INSTALL . The code for state 2 is:

State 2: $\text{RETRACT}()$
 $\text{return}(\text{id}, \text{INSTALL}())$

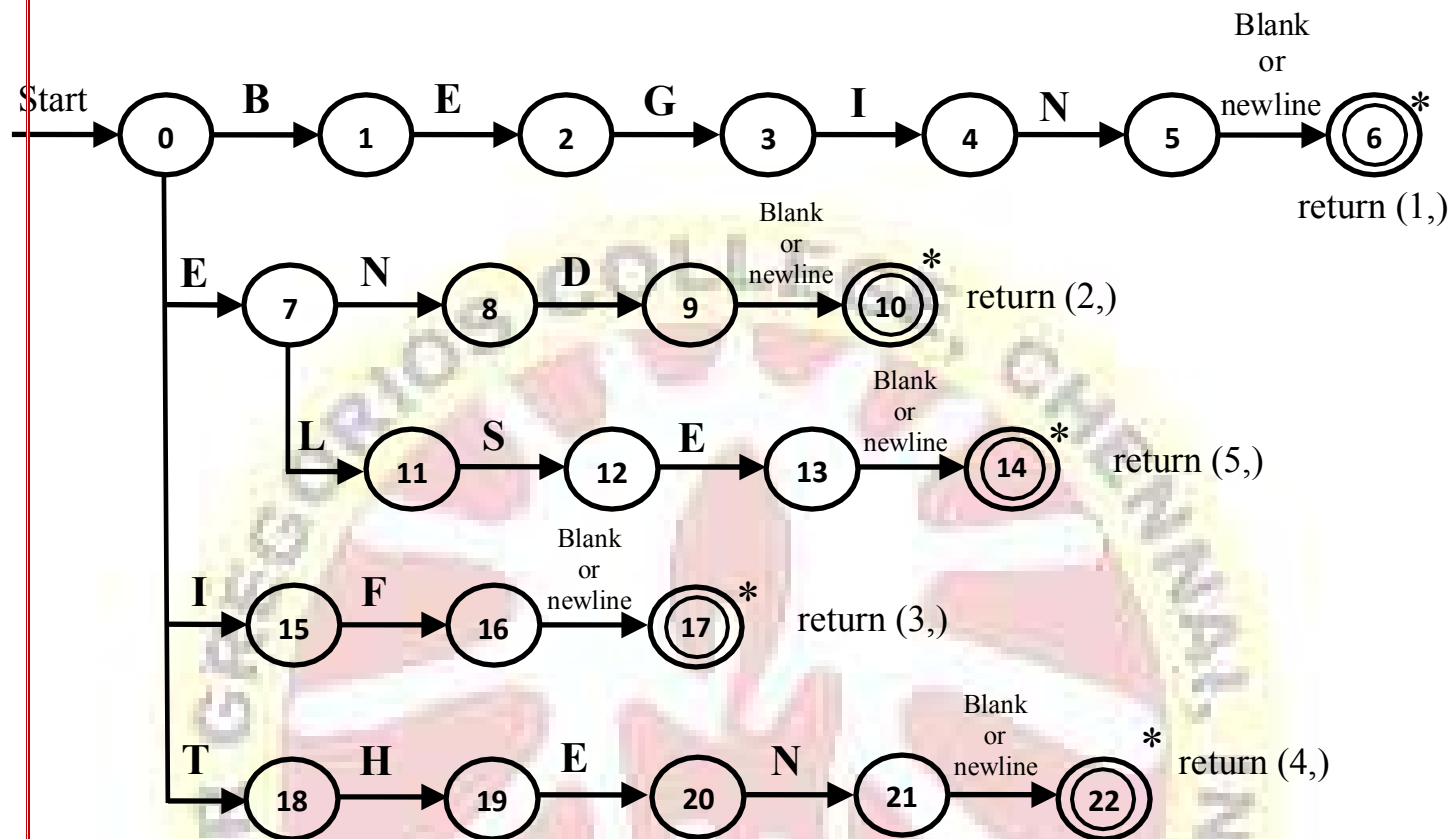
If blank must be skipped in the language at hand, we should include in the code for state 2 a step that moved the beginning pointer to the next non-blank.

Fig (5) shows a list of tokens that we want to recognize using token recognizer that use transition diagram explained in fig (6).

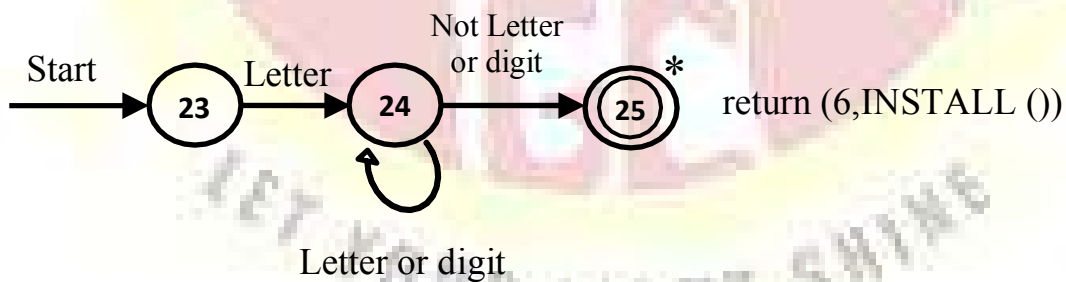
<i>Token</i>	<i>Code</i>	<i>Value</i>
begin	1	-----
end	2	-----
if	3	-----
then	4	-----
else	5	-----
identifier	6	Pointer to Symbol Table
constant	7	Pointer to Symbol Table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

Fig (5) Token Recognizer

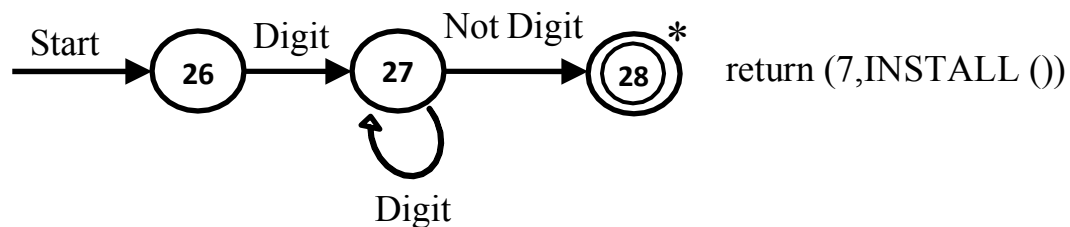
Keywords:



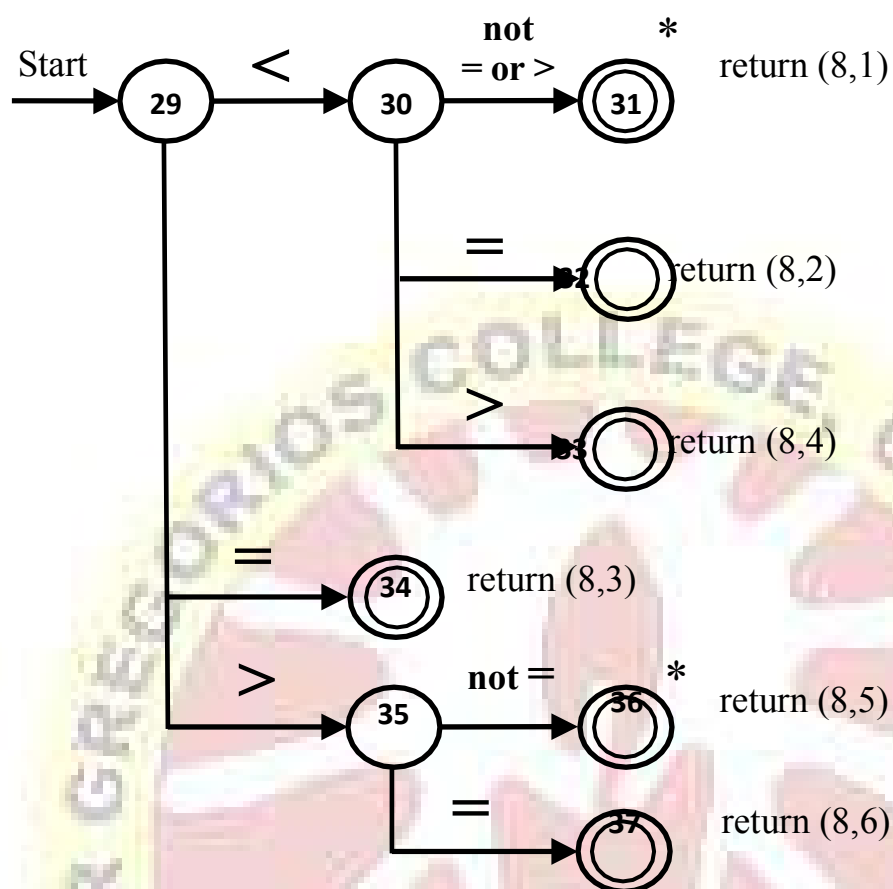
Identifier:



Constant:

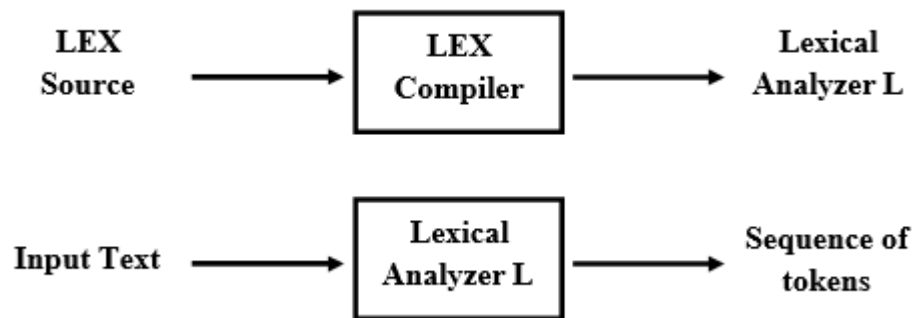


Re lops:



1.6 The Language for Specifying Lexical Analyzer

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code which is to be executed whenever a token specified by the corresponding regular expression is recognized. The output of LEX is a lexical analyzer program constructed from the LEX source specification.



A LEX source program consists of two parts, a sequence of auxiliary definition followed by a sequence of translation rules.

Auxiliary Definitions

The auxiliary definitions are statements of the form:

$$D_1 = R_1$$

$$D_2 = R_2$$

$$D_n = R_n$$

Where each D_i is a distinct name, and each R_i is a regular expression whose symbols are chosen from $\Sigma \cup \{D_1, D_2, D_{i-1}\}$, i.e., characters or previously defined names. The D_i 's are shorthand names for regular expressions. Σ is our input symbol alphabet.

Example:

We can define the class of identifiers for a typical programming language with the sequence of auxiliary definitions.

$$\text{Letter} = A \mid B \mid \dots \mid Z$$

$$\text{Digit} = 0 \mid 1 \mid \dots \mid 9$$

$$\text{Identifier} = \text{Letter} (\text{Letter} \mid \text{Digit})^*$$

Translation Rules

The translation rules of a LEX program are statements of the form:-

$$\begin{array}{l} P_1 \quad \{A_1\} \\ P_2 \quad \{A_2\} \\ \vdots \\ \bullet \\ \vdots \\ P_m \quad \{A_m\} \end{array}$$

Where each P_i is a regular expression called a pattern, over the alphabet consisting of Σ and the auxiliary definition names. The patterns describe the form of the tokens. Each A_i is a program fragment describing what action the lexical analyzer should take when token P_i is found. The A_i 's are written in a conventional programming language, rather than any particular language, we use pseudo language. To create the lexical analyzer L, each of the A_i 's must be compiled into machine code.

The lexical analyzer L created by LEX behaves in the following manner: L read its input, one characters at a time, until it has found the longest prefix of the input which matches one of the regular expressions, P_i . Once L has found that prefix, L removes it from the input and places it in a buffer called TOKEN.

Example: Let us consider the collection of tokens defined in Fig.7, LEX programis shown in Fig.26

AUXILIARY DEFINITION

Letter= A | B | ... | Z

Digit= 0 | 1 | ... | 9

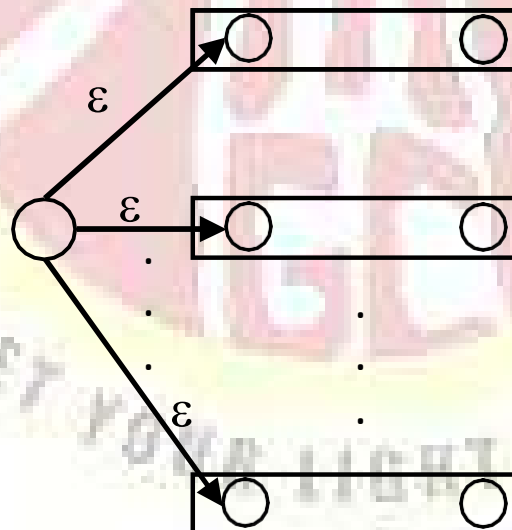
TRANSLATION RULES

BEGIN	{return 1}
END	{return 2}
IF	{return 3}
THEN	{return 4}
ELSE	{return 5}

letter(letter digit)*	{LEX VAL:= INSTALL(); return 6}
digit ⁺	{LEX VAL:= INSTALL(); return 7}
<	{LEX VAL := 1; return 8}
<=	{LEX VAL := 2; return 8}
=	{LEX VAL := 3; return 8}
< >	{LEX VAL := 4; return 8}
>	{LEX VAL := 5; return 8}
>=	{LEX VAL := 6; return 8}

1.7 Implementing the Lexical Analyzer

The LEX can build from its input a lexical analyzer that behaves roughly like a finite automaton. The idea is to construct a NFA N_i for each tokens pattern P_i in the translation rules, then links these NFA's together with a new start states as shown in Fig.27. Next we convert this NFA to a



DFA.

Example: Suppose we have the following LEX program.

AUXILIARY DEFINITION

(none)

TRANSLATION RULES

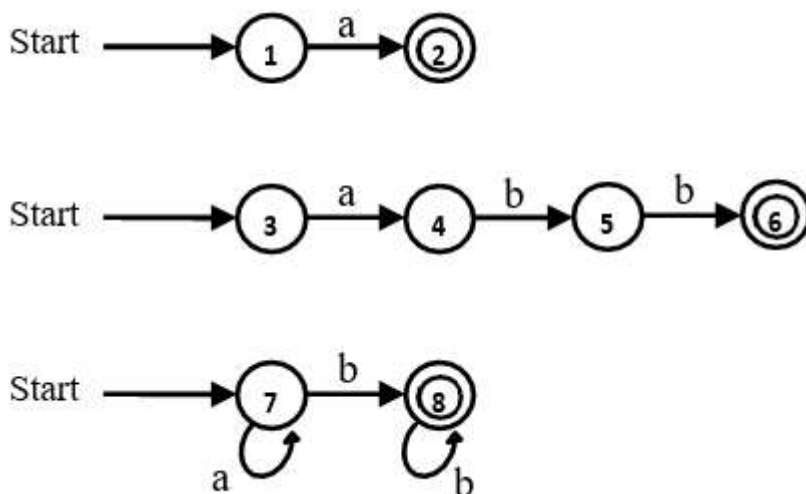
a {} /* actions are omitted here*/

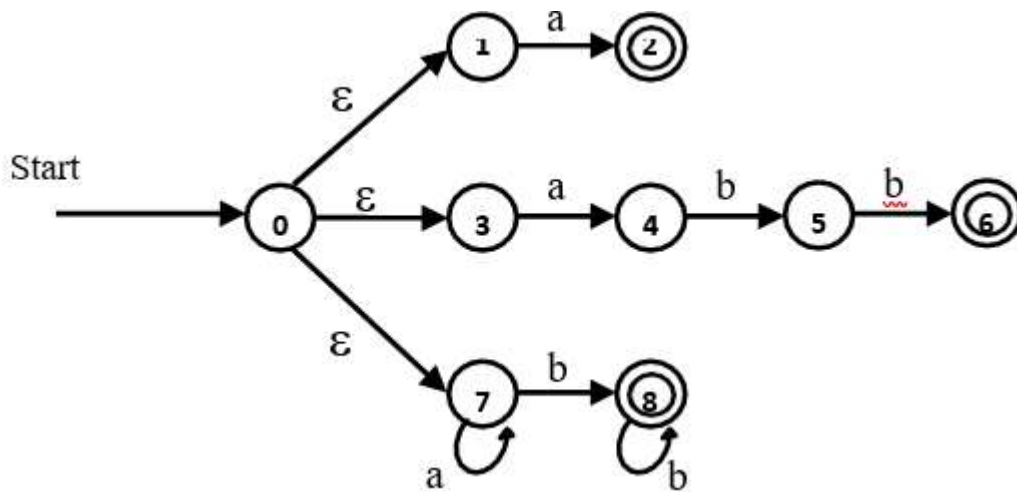
abb {}

a^*b^+ {}

The three tokens above are recognized by the simple automata of Fig.28. We may convert the NFA's of Fig.28 into one NFA as described earlier. The result is shown in Fig.29. Then this NFA may be converted to a DFA using the algorithm and the transition table is shown in Fig.30, where the states of the DFA have been named by lists of the states of the NFA.

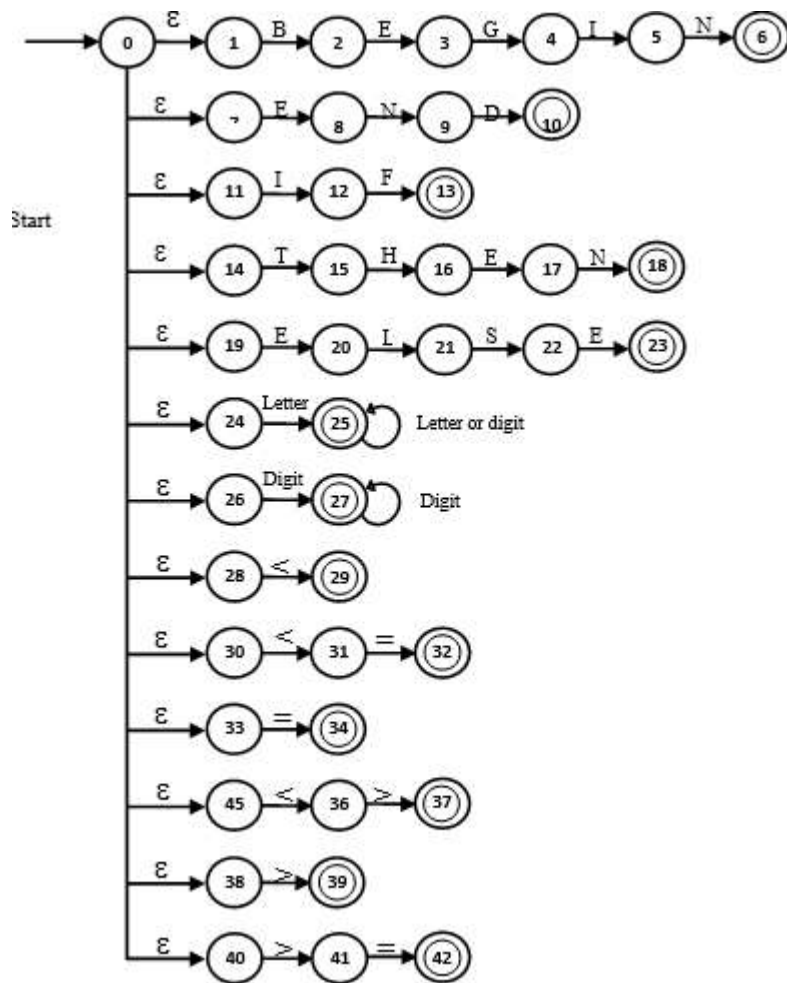
The last column in Fig.30 indicates the token which will be recognized if that state is the last state entered that recognizes any token at all. In all cases but the last line, state 68, the token recognized is the only token whose final state is included among the NFA states forming the DFA state. For example, among NFA states 2, 4, and 7, only 2 is final, and it is the final state of the automaton for regular expression 'a' in Fig.28. Thus, DFA state 247 recognizes token 'a'. In the case of DFA state 68, both 6 and 8 are final states of their respective nondeterministic automata. Since the translation rules of our LEX program mention **abb** before a^*b^+ , NFA state 6 has priority, and we announce that abb has been found in DFA state 68.





State	a	b	Token found
0137	247	8	none
247	7	58	a
8	---	8	a*b ⁺
7	7	8	none
58	---	68	a*b ⁺
68	---	8	abb

In following figure we see NFA's for the patterns. We use here a simple NFA for identifiers and constants.



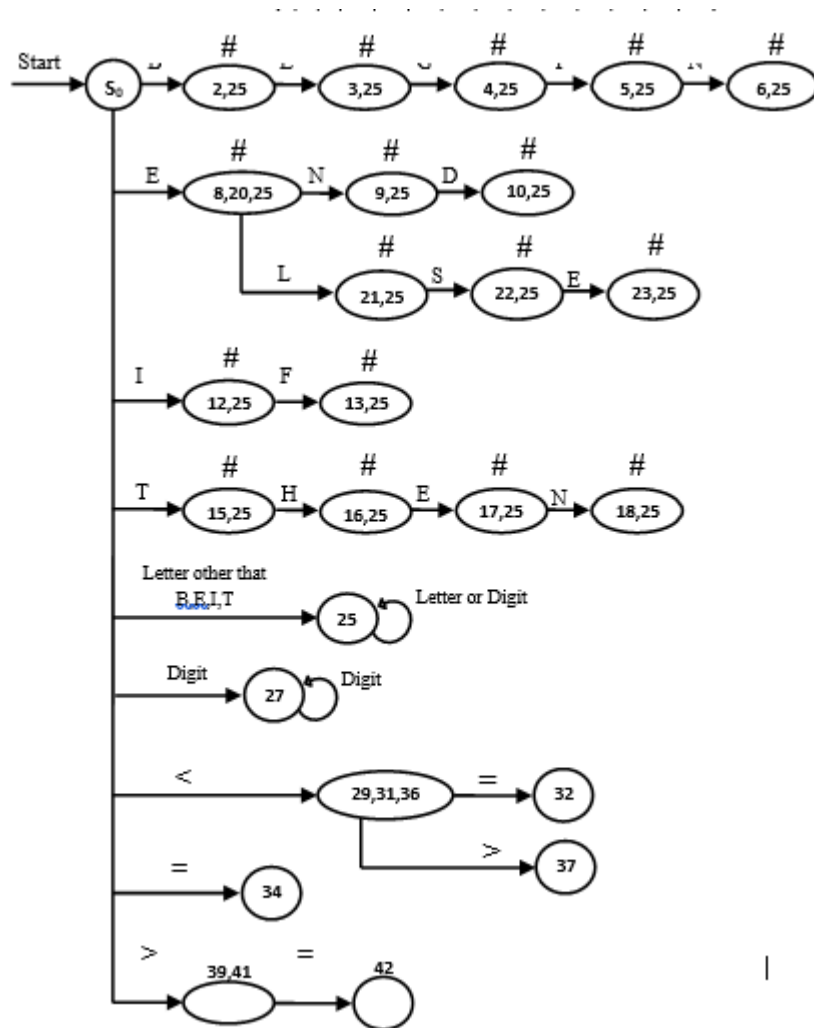
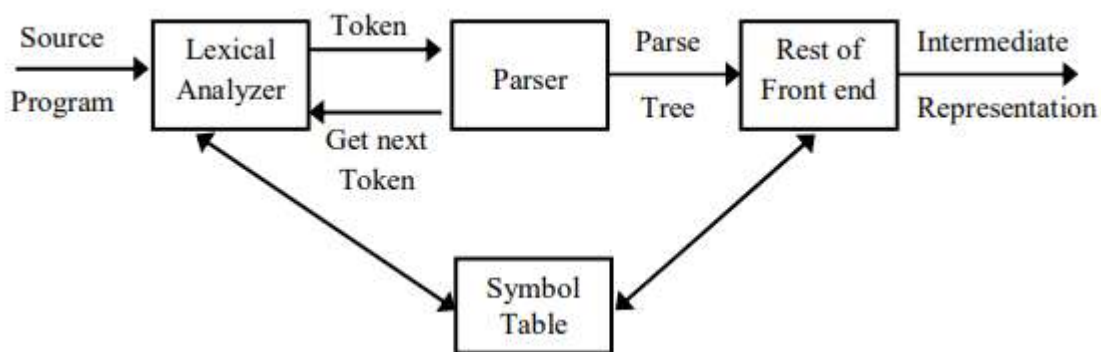


Fig : Combined DFA

UNIT- II

2.1 Syntax Analyzer (Parser)

We show that the lexical structure of tokens could be specified by regular expressions and that from a regular expressions we could automatically construct a lexical analyzer to recognize the tokens denoted by the expression. We shall use a notation called a context-free grammar or grammar only, for the syntactic specification of a programming language which is also called a BNF (Backus-Naur Form) description. 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 language. The parser reports any syntax errors and gives an indication of the type of these errors. Fig.1 shows the position of parser



Context Free Grammars (CFG's) It is natural to define certain programming-language construct recursively. For example, we might state:-

If S1 and S2 are statements and E is an expression, then:-

"If E then S1 else S2" is a statement ----- (1)

Or: If S1, S2,..., Sn are statements then

"begin S1; S2 ;...;Sn end" is a statement -----(2)

As a third example:

If E1 and E2 are expressions, then "E1+E2" is an expression ----- (3)

If we use the syntactic category "statement" to denote the class of the statements and "expression" to denote the class of expressions, then (1) can be expressed by this production:

Statement if expression then statement else statement ----- (4)

And (3) can be written as:

Expression expression + expression ----- (5)

If we want to write (2) in the same way we may have:

Statement \rightarrow begin statement; statement; ... ; statement end

But the use of ellipses (...) would create problems when we attempt to define translations based on this description. For this reason, we require that each rewriting rule (production) have a known number of symbols, with no ellipses permitted.

To express (2) by rewriting rules, we can introduce a new syntactic category "statement-list" denoting any sequence of statements separated by semicolons. Then we can write:

Statement \rightarrow begin statement-list end

Statement-list \rightarrow statement | statement; statement-list ----- (6)

A set of rules such that (6) is an example of a grammar. In general, a grammar involves four quantities: start symbol, terminals, nonterminals, productions .

EX: consider the following grammar for simple arithmetic expressions. The non-terminal symbols are expressions and operator, with expression the start symbol. The terminal symbols are: id, +, -, *, /, (,), \uparrow

The productions are

Expression \rightarrow Expression operator Expression

Expression \rightarrow (Expression)

Expression \rightarrow - Expression

Expression \rightarrow id

Operator \rightarrow +

Operator \rightarrow -

Operator \rightarrow *

Operator \rightarrow /

Operator \rightarrow \uparrow

We can write these productions in this form:-

$$E \longrightarrow EAE \mid (E) \mid -E \mid \text{id}$$

$$A \longrightarrow + \mid - \mid / \mid \uparrow \mid *$$

2.2 Derivation and Parse Trees

The central idea of how context free grammar defines a language is that the productions may be applied repeatedly to expand the non-terminals in a string of non-terminals and terminals. For example, consider the following grammar for arithmetic expressions

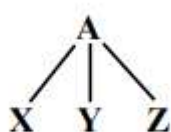
$$E \longrightarrow E+E \mid E * E \mid (E) \mid -E \mid \text{id} \text{ ----- (7)}$$

Parse Trees

We can create a 'graphical representation' for derivations that filter out the choice regarding replacement order. This representation is called the parse tree, and it has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar. Each interior node of the parse tree is labeled by some non-terminal A , and the children of 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.

EX: let us again consider the arithmetic expression grammar (7), with which we have been dealing. The sentence $\text{id}+\text{id}*\text{id}$ has the two distinct left most derivations:

For example if $A \rightarrow XYZ$ is a production used at some step of a derivation, then the parse tree for that derivation will have the sub tree:-



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 $-(\text{id}+\text{id})$ implied by the derivation of the previous example is shown in Fig.2

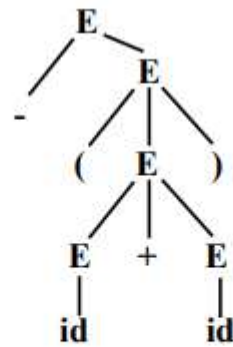


Fig.2: Parse tree

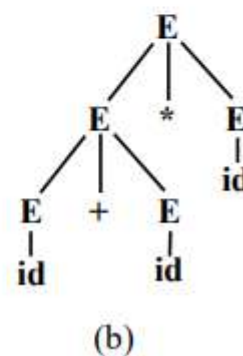
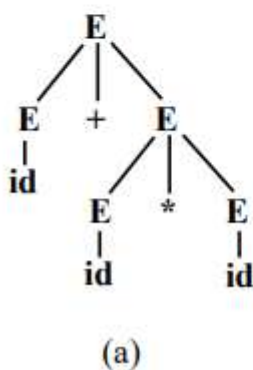
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

EX: let us again consider the arithmetic expression grammar (7), with which we have been dealing. The sentence $id+id*id$ has the two distinct left most derivations:

$E \Rightarrow E+E$	$E \Rightarrow E * E$
$\Rightarrow id+E$	$\Rightarrow E+E * E$
$\Rightarrow id+E * E$	$\Rightarrow id+E * E$
$\Rightarrow id+id * E$	$\Rightarrow id+id * E$
$\Rightarrow id+id * id$	$\Rightarrow id+id * id$

(a)

The two parse tree are shown in Fig.3



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 produced more than one left most or more than one right most derivation for some sentence. For certain types of parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Example Consider the following grammar for arithmetic expression involving $+$, $-$, $*$, $/$ and \uparrow (exponentiation)

$$E \longrightarrow E+E \mid E-E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \text{id} \quad \text{----- (9)}$$

This grammar like (7) is ambiguous. However, we can disambiguate both these grammars by specifying the associativity and precedence of the arithmetic operators.

Suppose we wish to give the operators the following precedence in decreasing order:-

- (unary minus) \uparrow * / + -

We begin by introducing one non-terminal for each precedence level. An 'element' is either a single identifier or a parenthesized expression. We therefore have the productions:-

$$\text{element} \longrightarrow (\text{expression}) \mid \text{id}$$

Next we introduce the category of 'Primaries', which are elements with zero or more of the operator of highest precedence, the unary minus. The rule for primary is:-

$$\text{Primary} \longrightarrow - \text{Primary} \mid \text{element}$$

Then we construct 'factors' as a sequence of one or more primaries connected by exponentiation signs. That is:-

$$\text{factor} \longrightarrow \text{primary} \uparrow \text{factor} \mid \text{primary}$$

Then we introduce 'term',

$$\text{term} \longrightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}.$$

Then the final, unambiguous grammar is:-

$$\begin{aligned} \text{Expression} &\longrightarrow \text{expressions} + \text{term} \\ &\quad \mid \text{expression} - \text{term} \\ &\quad \mid \text{term} \end{aligned}$$

$$\begin{aligned} \text{term} &\longrightarrow \text{term} * \text{factor} \\ &\quad \mid \text{term} / \text{factor} \\ &\quad \mid \text{factor} \end{aligned}$$

$$\text{factor} \longrightarrow \text{primary} \uparrow \text{factor} \mid \text{primary}$$

$$\text{primary} \longrightarrow - \text{primary} \mid \text{element}$$

$$\text{element} \longrightarrow (\text{expression}) \mid \text{id}$$

2.3 Basic Parsing Techniques

We show previously that the CFG can be used to define the syntax of a programming language, but now we show how to check whether an input string is a sentence of a given grammar and how to construct a parse tree for this string. The input to the parse is typically a sentence of tokens. The output of the parser can be of many different forms, and for simplicity we assume that the parser is some representation of the parse tree.

The most common forms of parsers are 'operator precedence' and 'recursive descent'. Operator precedence is especially suitable for parsing expressions, since it can use information about the precedence and associativity of operators to guide the parse. 'Recursive descent' uses a collection of mutually recursive routines to perform the syntax analysis. A common situation is for operator precedence to be used for expressions and recursive descent for the test of the language.

Parsers

A parser for a grammar G is a program that takes as input a string W and produces as output either a parse tree for W , if W is a sentence of G , or an error message indicating that W is not sentence of G .

We discuss the operations of two types of parsers for CFG's:- bottom-up and top-down parsers build, parse tree from the bottom (leaves) to the top (root), while top-down parsers start with the root and work down to the leaves. In both cases the input to the parser is being scanned from the left to right, one symbol at a time.

The bottom-up parsing method is called "Shift-reduce" parsing, because it appears on top of the stack. One type of shift-reduce is "operator precedence parser". The top-down parsing method is called "recursive descent" parsing.

Representation of a Parse Tree

There are two basic types of representations: - implicit and explicit. The sequence of productions used in some derivation is an example of an implicit representation. A linked list structure for the parse tree is an explicit representation. Recall that a derivation in which the left most non-terminal is replaced at every step is said to be left most. If $\alpha \Rightarrow \beta$ by a step in which the left-most non-terminal is α is replaced, we write $\alpha \Rightarrow \beta$. If α derives β by a left most derivation we write $\alpha \Rightarrow^* \beta$. If $S \Rightarrow^* \alpha$, then we say α is a left-sentential form of the grammar. Right most derivations are sometimes called "canonical derivations".

EX:-Consider the grammar:-

- (1) $S \longrightarrow i C t S$
- (2) $S \longrightarrow i C t S e S$ ----- (1)
- (3) $S \longrightarrow a$
- (4) $C \longrightarrow b$

Here i, t and e stand for "if", "then" and "else" respectively and C for "conditional", S for "statement".

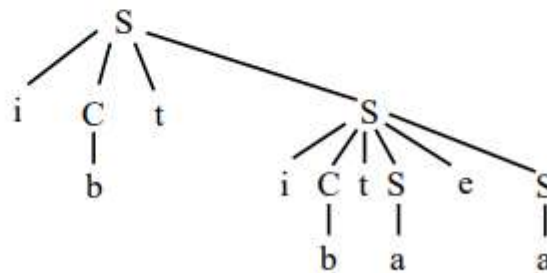


Fig.4: Parse Tree T

The left-most derivation corresponding to this parse tree is given by:

$$\begin{aligned}
 S &\xRightarrow{lm} i C t S \\
 &\xRightarrow{lm} i b t S \\
 &\xRightarrow{lm} i b t i C t S e S \\
 &\xRightarrow{lm} i b t i b t S e S \\
 &\xRightarrow{lm} i b t i b t a e S \\
 &\xRightarrow{lm} i b t i b t a e a
 \end{aligned}$$

A right most derivation can be constructed from a parse tree analogously. At each step we replace the right-most non-terminal by the tables of its children. For example, the first two steps of a right most derivation constructed from Fig.4 would be

$$S \Rightarrow i C t S \Rightarrow i C t i C t S e S$$

2.4 Bottom-Up Parsing

Shift-Reduce Parsing

In this section, we discuss a bottom-up style of parsing called shift-reduce parsing. This parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root. We can think of this process as one of "reducing" a string W

to the start symbol of a grammar. At each step a string matching the right side of the production is replaced by the symbol on the left.

For example, consider the grammar

$$S \longrightarrow a A c B e$$

$$A \longrightarrow Ab \mid b$$

$$B \longrightarrow d$$

and the string "abbcede". We want to reduce this string to S. We scan this string looking for substring that matches the right side of any production. The substring b and d qualify. Let us choose the left most b and replace it by A the left side of the production $A \rightarrow b$. We continue according to this step:

$$\underline{a}bbcede \longrightarrow a\underline{A}bcde \longrightarrow aAcde \longrightarrow aAcBe \longrightarrow S$$

Each replacement of the right side of a production by the left side in the process above is called "reduction". Thus, by a sequence of four reductions we were able to reduce abbcede to S.

Handles

A handle of a right sentential form y is a production $A \rightarrow \beta$ and a position of y , where the string β may be found and replaced by A to produce the previous right-sentential form in the right most derivation of y .

That is, if $s \xRightarrow{*} \alpha A w \xRightarrow{r.m.} \alpha \beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha \beta w$. The string w to the right of the handle contains only terminal symbols.

In the example above, abbcede is a right-sentential form whose handle is $A \rightarrow b$ at the position 2. Likewise, a Abcde is a right-sentential form whose handle is $A \rightarrow Ab$ at position 2.

Sometimes we shall say "the substring β is a handle of $\alpha \beta w$ " if the position of β and the production $A \rightarrow \beta$ we in mind are clear. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

EX: consider the following grammar:-

- (1) $E \longrightarrow E + E$
- (2) $E \longrightarrow E * E$
- (3) $E \longrightarrow (E)$
- (4) $E \longrightarrow id$

And consider the right most derivation.

$$E \xrightarrow{Rm} \underline{E+E} \xrightarrow{Rm} E+\underline{E*E} \xrightarrow{Rm} E+E*\underline{id3} \xrightarrow{Rm} E+\underline{id2}*id3 \xrightarrow{Rm} \underline{id1}+id2*id3$$

We have subscripted the id's for notational convenience and underlined a handle of each right-sentential form for example, id1 is the handle of the right sentential form id1+id2*id3, because id is the right side of the production $E \rightarrow id$, and replacing id1 by E produces the previous right sentential form E+id2*id3.

Handle Pruning

A right most derivation in reverse, often called a "canonical reduction sequence", is obtained by "handle pruning". That is, we start with a string of terminal w which we wish to parse. If w is a sentence of the grammar at hand, then $w=y_n$, where y_n is the nth right-sentential form of some as yet unknown right most derivation:

$$S = y_0 \xrightarrow{Rm} y_1 \xrightarrow{Rm} y_2 \xrightarrow{Rm} y_{n-1} \xrightarrow{Rm} y_n = w.$$

To reconstruct this derivation in reverse order, we locate the handle β_n in y_n and replace β_n by the left side of some production $A_n \rightarrow \beta_n$ to obtain the previous right sentential form y_{n-1} . We then repeat this process. That is, we locate the handle β_{n-1} in y_{n-1} and reduce this handle to obtain the right sentential form y_{n-2} . If by continuing this process we produce a right sentential form consisting only of the start symbol S, then we halt and give successful completion of parsing. The reverse of the sequence of production use in the reduction is right most derivation of the input string.

EX: consider the grammar (2) and input string id1+id2*id3. Then following sequence of reductions reduce id1+id2+id3 to start symbol E:

Right-Sentential Form	Handle	Reducing Production
id1+id2*id3	id1	$E \rightarrow id$
E+id2*id3	id2	$E \rightarrow id$
E+E*id3	id3	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E*E$
E+E	E+E	$E \rightarrow E+E$
E		

Stack Implementation of Shift-Reduce Parsing

A convenient way to implement a shift-reduce parser is to use a stack and an input buffer. We shall use \$ to mark the bottom of the stack and the right of the input.

<u>Stack</u>	<u>Input</u>
\$	W\$

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the

stack. The parser then reduces β to the left side of the appropriate production. The parser then reduces this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

<u>Stack</u>	<u>Input</u>
\$S	\$

In this configuration the parser halts and makes successful parsing.

EX: Let us step through the actions a shift-reduce parser might make in parsing the input string $id_1 + id_2 * id_3$ according to the grammar

<i>Seq</i>	<i>Stack</i>	<i>Input</i>	<i>Action</i>
1	\$	$id_1 + id_2 * id_3 \$$	Shift
2	$\$id_1$	$+ id_2 * id_3 \$$	Reduce by $E \rightarrow id$
3	$\$E$	$+ id_2 * id_3 \$$	Shift
4	$\$E +$	$id_2 * id_3 \$$	Shift
5	$\$E + id_2$	$* id_3 \$$	Reduce by $E \rightarrow id$
6	$\$E + E$	$* id_3 \$$	Shift
7	$\$E + E *$	$id_3 \$$	Shift
8	$\$E + E * id_3$	$\$$	Reduce by $E \rightarrow id$
9	$\$E + E * E$	$\$$	Reduce by $E \rightarrow E * E$
10	$\$E + E$	$\$$	Reduce by $E \rightarrow E + E$
11	$\$E$	$\$$	Accept

Fig.5: Shift-reduce parsing actions

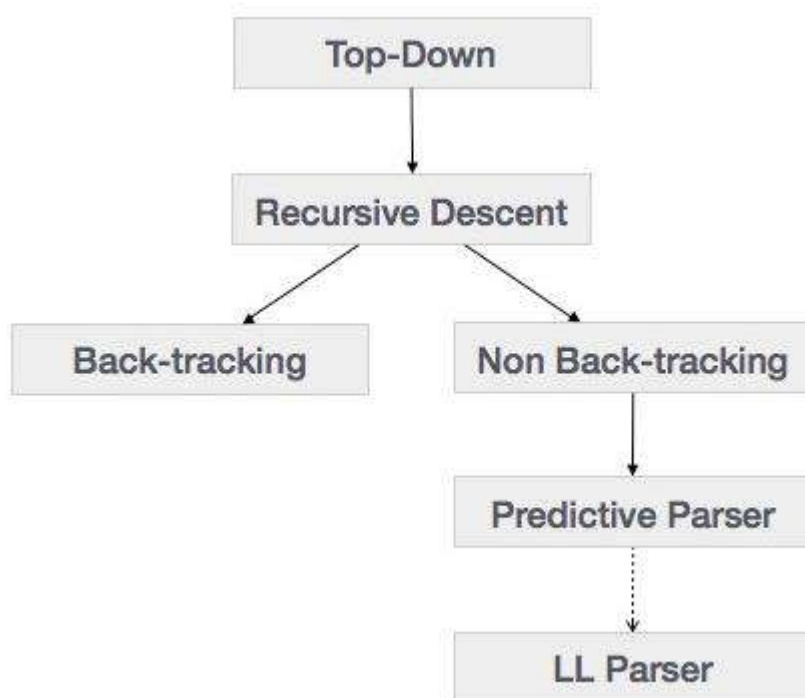
While the primary operations of the parser are shifting and reduce, there are actually four possible actions a shift reduce parser can make : (1) shift, (2) reduce, (3) accept, and (4) error

1. In a shift action, the next input symbol is shifted to the top of the stack.
2. In a reduce action, the parser knows the right end of the handle is at about the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
3. In an accept action, the parser complete parsing successfully.
4. In an error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

2.5 Top-Down Parsing

Top-down parsing can be viewed as attempt to find left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder .

Left-recursive grammar can cause a top-down parser to go into an infinite loop



2.5.1 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 (backtracking)

Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

(Grammar 1)

and the input string $w = cad$

To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled S .

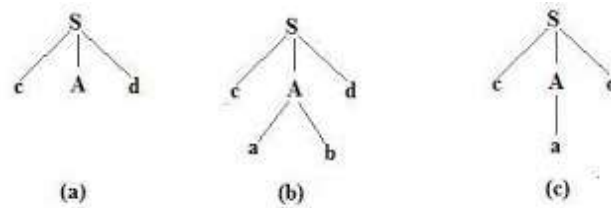


Fig 2

An input pointer points to c, the first symbol of w.

Then use the first production for S to expand the tree and obtain the tree (as in Fig 2(a))

The leftmost leaf, labeled c, matches the first symbol of w.

Next input pointer to a, the second symbol of w.

Consider the next leaf, labeled A.

Expand A using the first alternative for A to obtain the tree (as in Fig 2(b)).

Now have a match for the second input symbol. Then advance to the next input pointer d, the third input symbol and compare d against the next leaf, labeled b. Since b does not match d, report failure and go back to A to see whether there is another alternative. (Backtracking takes place).

If there is another alternative for A, substitute and compare the input symbol with leaf.

Repeat the step for all the alternatives of A to find a match using backtracking. If match found, then the string is accepted by the grammar. Else report failure.

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

For the grammar (**Grammar 1**) above write the recursive procedure for each nonterminal S and A.

Procedure S

procedure S()

$S \rightarrow cAd$

begin

$A \rightarrow ab \mid a$

if input symbol = 'c' then

begin

ADVANCE();

if A() then

if input symbol = 'd' then

begin ADVANCE(); return true end

end;

return false

end

Procedure A

```

procedure A( )
begin
  isave := input-pointer;
  if input symbol = 'a' then
    begin
      ADVANCE( );
      if input symbol = 'b' then
        begin ADVANCE( ); return true end
      end
    end
  input-pointer := isave;
  /* failure to find ab */
  if input symbol = 'a' then
    begin ADVANCE( ); return true end
  else return false
end

```

$S \rightarrow cAd$

$A \rightarrow ab \mid a$ not left factoring

$S \rightarrow cAd$

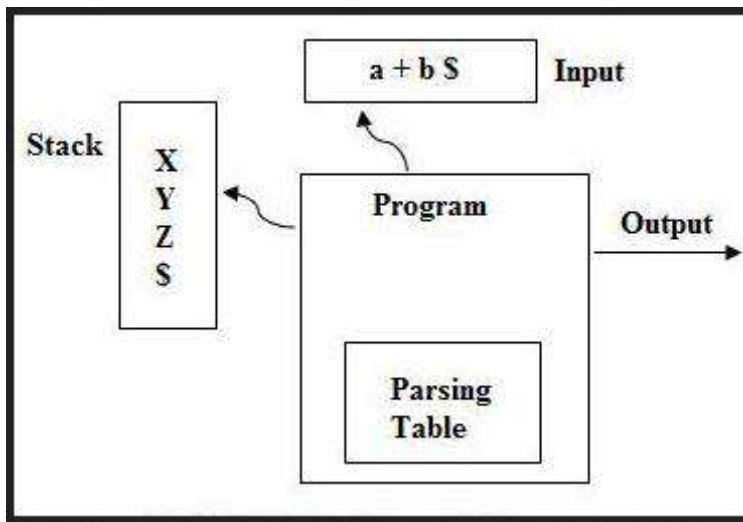
$A \rightarrow aC'$

$C' \rightarrow b \mid \epsilon$ left factoring

2.5.2 Predictive parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(1) grammar.

The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.



predictive parser

Non recursive Predictive Parser

The predictive parser has an input, a stack, a parsing table, and an output.

- The input contains the string to be parsed, followed by \$, the right endmarker.
- The stack contains a sequence of grammar symbols, preceded by \$, the bottom-of-stack marker.
- Initially the stack contains the start symbol of the grammar preceded by \$.
- 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 determines 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 $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top).

If $M[X, a] = \text{error}$, the parser calls an error recovery routine.

Moves by Predictive parser using the input string

Predictive parsing program

repeat

begin

let X be the top stack symbol and a the next input symbol;

if X is a terminal or $\$$ then


```

if  $X = a$  then
    pop  $X$  from the stack and remove  $a$  from the input
else
    ERROR( )
else /*  $X$  is a nonterminal */
    if  $M[X,a] = X \rightarrow Y_1, Y_2, \dots, Y_k$  then
        begin
            pop  $X$  from the stack;
            push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack,  $Y_1$  on top
        end
    else
        ERROR( )
    end
until  $X = \$$  /* stack has emptied */

```

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

Predictive parser table

MATCHED	STACK	INPUT	ACTION
	$E\$$	id + id * id\$	
	$TE'\$$	id + id * id\$	output $E \rightarrow TE'$
	$FT'E'\$$	id + id * id\$	output $T \rightarrow FT'$
	id $T'E'\$$	id + id * id\$	output $F \rightarrow id$
id	$T'E'\$$	+ id * id\$	match id
id	$E'\$$	+ id * id\$	output $T' \rightarrow \epsilon$
id	+ $TE'\$$	+ id * id\$	output $E' \rightarrow + TE'$
id +	$TE'\$$	id * id\$	match +
id +	$FT'E'\$$	id * id\$	output $T \rightarrow FT'$
id +	id $T'E'\$$	id * id\$	output $F \rightarrow id$
id + id	$T'E'\$$	* id\$	match id
id + id	* $FT'E'\$$	* id\$	output $T' \rightarrow * FT'$
id + id *	$FT'E'\$$	id\$	match *
id + id *	id $T'E'\$$	id\$	output $F \rightarrow id$
id + id * id	$T'E'\$$	\$	match id
id + id * id	$E'\$$	\$	output $T' \rightarrow \epsilon$
id + id * id	\$	\$	output $E' \rightarrow \epsilon$

Construction of Parsing Table:

Before constructing the parsing table, two functions are to be performed to fill the entries in the table.

FIRST() and FOLLOW() functions.

These functions will indicate proper entries in the table for a grammar G.

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 is nonterminal and $X \rightarrow \alpha a$ is a production, then add a to FIRST(X). If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then for all i such that all of Y_1, \dots, Y_{i-1} are nonterminals and FIRST(Y_j) contains ϵ for $j = 1, 2, \dots, i-1$ (i.e., $Y_1 Y_2 \dots Y_{i-1} \Rightarrow \epsilon$), add every non- ϵ symbol in FIRST(Y_i) to FIRST(X). If ϵ is in FIRST(Y_j) for all $j = 1, 2, \dots, k$, then add ϵ to FIRST(X).

To **compute FOLLOW(A)** for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. \$ is in FOLLOW(S), where S is the start symbol.
2. If there is a production $A \rightarrow \alpha B \beta$, $\beta \neq \epsilon$, then everything in FIRST(β) but ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \Rightarrow \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Example

Consider the following grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

(Grammar 2)

Compute the FIRST and FOLLOW function for the above grammar.

Solution:

Here the (Grammar 2) is in left-recursion, so eliminate the left recursion for the (Grammar 2) we get;

	FIRST	FOLLOW
$E \rightarrow TE'$	{id, (}	{\$,) }
$E' \rightarrow +TE' \mid \epsilon$	{+, ϵ }	{\$,) }
$T \rightarrow FT'$	{id, (}	{+, \$,) }
$T' \rightarrow *FT' \mid \epsilon$	{*, ϵ }	{+, \$,) }
$F \rightarrow (E) \mid \text{id}$	{id, (}	{+, *, \$,) }

Example

Consider the following grammar. Compute the FIRST and FOLLOW function for the above grammar.

	FIRST	FOLLOW
$S \rightarrow ABCDE$	$\{a,b,c\}$	$\{\$ \}$
$A \rightarrow a \mid \epsilon$	$\{a, \epsilon\}$	$\{b,c\}$
$B \rightarrow b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \rightarrow c$	$\{c\}$	$\{d,e,\$ \}$
$D \rightarrow d \mid \epsilon$	$\{d, \epsilon\}$	$\{e,\$ \}$
$E \rightarrow e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$ \}$

Example

Consider the following grammar. Compute the FIRST and FOLLOW function for the above grammar.

	FIRST	FOLLOW
$S \rightarrow Bb \mid Cd$	$\{a,b,c,d\}$	$\{\$ \}$
$B \rightarrow aB \mid \epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \rightarrow cC \mid \epsilon$	$\{c, \epsilon\}$	$\{d\}$

Exercise

Consider the grammar

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

(Grammar 4)

Compute the FIRST and FOLLOW for the (Grammar 4)

Construction of Predictive Parsing Table:

The following algorithm can be used to construct a predictive parsing table for a grammar G

Algorithm Constructing a predictive parsing table

Input: Grammar G

Output: Parsing table M

Method:

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

Example

Construct the predictive parsing table for the (Grammar 3) using the **Algorithm** of constructing a predictive parsing table

	FIRST	FOLLOW
$E \rightarrow TE'$	$\{id, (\}$	$\{ \$,) \}$
$E' \rightarrow +TE' \mid \epsilon$	$\{ +, \epsilon \}$	$\{ \$,) \}$
$T \rightarrow FT'$	$\{id, (\}$	$\{ +, \$,) \}$
$T' \rightarrow *FT' \mid \epsilon$	$\{ *, \epsilon \}$	$\{ +, \$,) \}$
$F \rightarrow (E) \mid id$	$\{id, (\}$	$\{ +, *, \$,) \}$

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

Example

Construct the predictive parsing table for the below grammar using the **Algorithm** of constructing a predictive parsing table

	FIRST	FOLLOW
$S \rightarrow aABC$	$\{a\}$	$\{ \$ \}$
$A \rightarrow a \mid bb$	$\{a, b\}$	$\{a, b, \$ \}$
$B \rightarrow a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, \$ \}$
$C \rightarrow b \mid \epsilon$	$\{b, \epsilon\}$	$\{ \$ \}$

	A	b	\$
S	$S \rightarrow aABC$		
A	$A \rightarrow a$	$A \rightarrow bb$	
B	$B \rightarrow a$	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$
C		$C \rightarrow b$	$C \rightarrow \epsilon$

predictive parsing table

Example

Construct the predictive parsing table for the (**Grammar 4**)

NON - TERMINAL	INPUT SYMBOL					
	<i>a</i>	<i>b</i>	<i>e</i>	<i>i</i>	<i>t</i>	\$
<i>S</i>	$S \rightarrow a$			$S \rightarrow iEtSS'$		
<i>S'</i>			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
<i>E</i>		$E \rightarrow b$				

2.5.3 LL(1) Grammars:

- o When there is a situation that the parsing table consists of at least one multiply defined entries, then the easiest recourse is to transform the grammar by eliminating all left-recursion and then left-factoring whenever possible, to produce a grammar for which the parsing table has no multiply-defined entries.
- o A grammar whose parsing table has no multiply-defined entries is said to be *LL(1)* grammar.
- o A grammar is *LL(1)* if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of *G* the following conditions hold:
 1. For no terminal *a* do α and β derive strings beginning with *a*.
 2. At most one of α and β can derive the empty string.
 3. If $\beta \rightarrow^* \epsilon$, then α does not derive any strings beginning with a terminal in FOLLOW(*A*).

2.5.4 LR Parsers

LR parsers are used to parse the large class of context free grammars. This technique is called LR(*k*) parsing.

- L is left-to-right scanning of the input.
- R is for constructing a right most derivation in reverse.
- *k* is the number of input symbols of lookahead that are used in making parsing decisions.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

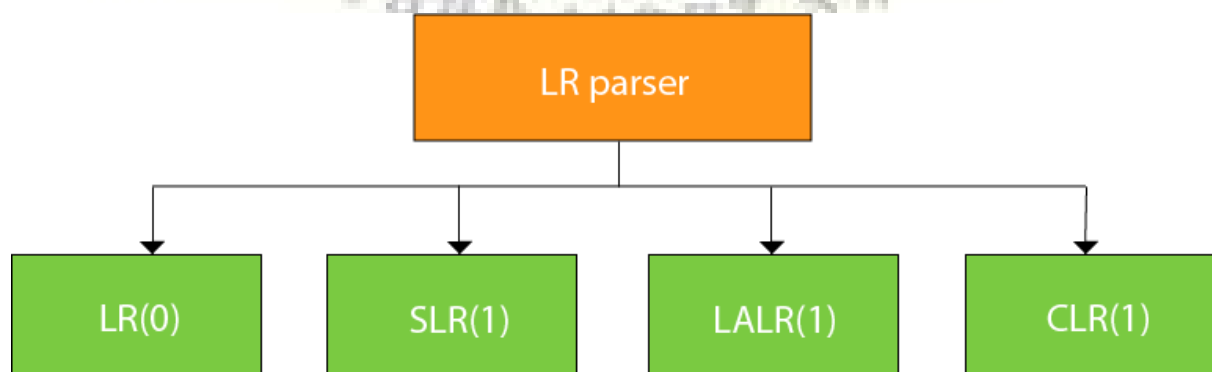


Fig: Types of LR parser

There are three widely used algorithms available for constructing an LR parser:

- SLR(l) - Simple LR
 - o Works on smallest class of grammar.
 - o Few number of states, hence very small table.
 - o Simple and fast construction.
- LR(1) - LR parser
 - o Also called as Canonical LR parser.
 - o Works on complete set of LR(l) Grammar.
 - o Generates large table and large number of states.
 - o Slow construction.
- LALR(l) - Look ahead LR parser
 - o Works on intermediate size of grammar.
 - o Number of states are same as in SLR(l).

Reasons for attractiveness of LR parser

- LR parsers can handle a large class of context-free grammars.
- The LR parsing method is a most general non-back tracking shift-reduce parsing method.
- An LR parser can detect the syntax errors as soon as they can occur.
- LR grammars can describe more languages than LL grammars.

Drawbacks of LR parsers

- It is too much work to construct LR parser by hand. It needs an automated parser generator.
- If the grammar contains ambiguities or other constructs then it is difficult to parse in a left-to-right scan of the input.

Model of LR Parser

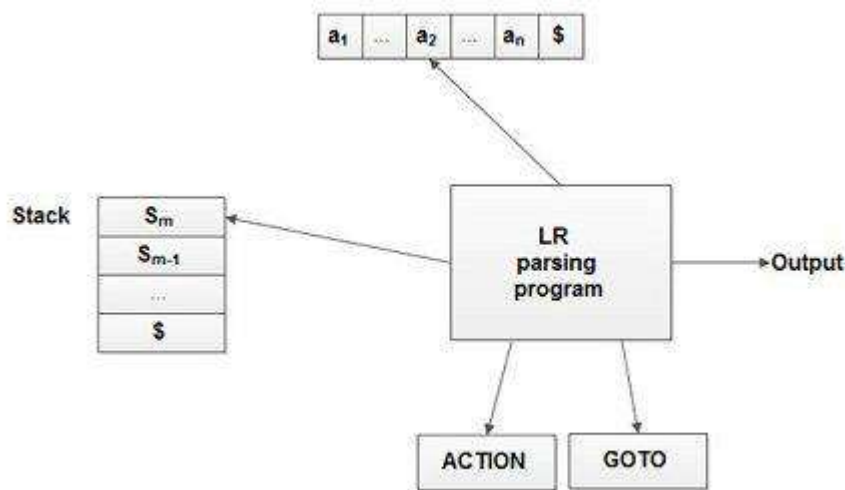
LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions

1. Action
2. Goto

The driver program is same for all LR parsers. Only the parsing table changes from one parser to another.

The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the [information](#) contained in the stack.

The stack holds a sequence of states, s_0, s_1, \dots, s_m , where s_m is on the top.



Action This function takes as arguments a state i and a terminal a (or $\$,$ the input end marker). The value of ACTION $[i, a]$ can have one of the four forms:

- i) Shift j , where j is a state.
- ii) Reduce by a grammar production $A \rightarrow \beta$.
- iii) Accept.
- iv) Error.

Goto This function takes a state and grammar symbol as arguments and produces a state.

If $\text{GOTO}[i, A] = j$, the GOTO also maps a state i and non terminal A to state j .

Behavior of the LR parser

1. If $\text{ACTION}[s_m, a_i] = \text{shift } s$. The parser executes the shift move, it shifts the next state s onto the stack, entering the configuration

a) S_m - the state on top of the stack.

b) a_i - the current input symbol.

2. If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 s_1 \dots s_{(m-r)} S, a_{i+1} \dots a_n \$)$$

a) where r is the length of β and $s = \text{GOTO}[s_m - r, A]$.

b) First popped r state symbols off the stack, exposing state s_{m-r} .

c) Then pushed s , the entry for $\text{GOTO}[s_{m-r}, A]$, onto the stack.

3. If $\text{ACTION}[s_m, a_i] = \text{accept}$, parsing is completed.

4. If $\text{ACTION}[s_m, a_i] = \text{error}$, the parser has discovered an error and calls an error recovery routine.

Example

STATE	Id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				

9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Figure shows the parsing action and goto functions of an LR parser for the grammar

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow \epsilon$

(6) $F \rightarrow id$

The codes for the actions are

1. s_i means shift and stack state i ,
2. r_j means reduce by production numbered j ,
3. acc means accept
4. blank means error

Consider the moves made by parser on input $id*id+id$. The sequence of stack and input contents is shown below

Stack	Input
(1) 0	$id*id+id\$$
(2) 0 id 5	$*id + id \$$
(3) 0 F 3	$*id + id \$$
(4) 0 T 2	$*id + id \$$
(5) 0 T 2 * 7	$id + id \$$
(6) 0 T 2 * 7 id 5	$+ id \$$
(7) 0 T 2 * 7 F 10	$+ id \$$
(8) 0 T 2	$+ id \$$
(9) 0 E 1	$+ id \$$
(10) 0 E 1 + 6	$id \$$
(11) 0 E 1 + 6 id 5	$\$$
(12) 0 E 1 + 6 F 3	$\$$
(13) 0 E 1 + 6 T 9	$\$$
(14) 0 E 1	$\$$

Figure: Moves of LR parser on $id*id+id$

LR Parsing Algorithm

Algorithm LR Parsing Algorithm.

Input Input string w ,

LR-Parsing table with functions ACTION and
GOTO for a grammar G

Output If w is in $L(G)$, the reduction steps of a
bottom-up parse for w ,

otherwise, an error indication.

Method Initially, the parser has S_0 on its stack,
where S_0 is the initial state, and $w \$$ in the
input buffer.

let a be the first symbol of $w \$$

while(1) { //repeat forever

let s be the state on top of the stack;

if(ACTION[s, a] = shift t {

push t onto the stack;

let a be the next input symbol;

} else if (ACTION [s, a] = reduce $A \rightarrow \beta$) {

pop β symbols off the stack;

let state t now be on top of the stack;

push GOTO[t, A] onto the stack;

output the production $A \rightarrow \beta$;

} else if (ACTION [s, a] = accept) break;

//parsing is done

else call error-recovery routine;

}

LR(O) Items

An LR(O) item of a grammar G is a production of G with a dot at some position of the body.

(eg.)

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet eYZ$

$A \rightarrow XY \bullet eZ$

$A \rightarrow XYZ \bullet$

One collection of set of LR(O) items, called the canonical LR(O) collection, provides finite automaton that is used to make parsing decisions. Such an automaton is called an LR(O) automaton.

LR(O) Parser SLR(1) Parser

An LR(O)parser is a shift-reduce parser that uses zero tokens of lookahead to determine what action to take (hence the 0). This means that in any configuration of the parser, the parser must have an unambiguous action to choose-either it shifts a specific symbol or applies a specific reduction. If there are ever two or more choices to make, the parser fails and the grammar is not LR(O).

An LR parser makes shift-reduce decisions by maintaining states to keep track of parsing. States represent a set of *items*.

Closure of item sets

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

- Initially, add every item I to $CLOSURE(I)$.
- If $A \rightarrow \alpha B \beta$ is in $CLOSURE(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to $CLOSURE(I)$, if it is not already there. Apply this rule until no more items can be added to $CLOSURE(I)$.

Construct canonical LR(O) collection

- Augmented grammar is defined with two functions, CLOSURE and GOTO. If G is a grammar with start symbol S , then augmented grammar G' is G with a new start symbol $S' \rightarrow S$.
- The role of augmented production is to stop parsing and notify the acceptance of the input i.e., acceptance occurs when and only when the parser performs reduction by $S' \rightarrow S$.

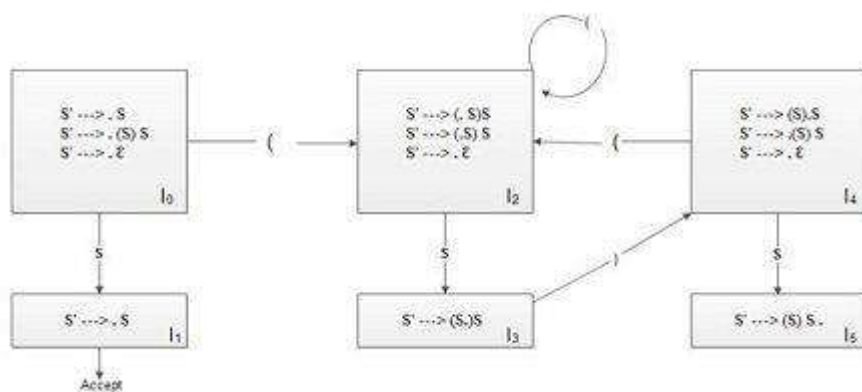
Limitations of the LR(O) parsing method

Consider grammar for matched parentheses

- $S' \rightarrow S$
- $S' \rightarrow (S) S$
- $S' \rightarrow \epsilon$

The LR(O) DFA of grammar G is shown below

In states: 0, 2 and 4 parser can shift (and reduce ϵ to S)



Conflicts

Conflicts are the situations which arise due to more than one option to opt for a particular step of shift or reduce.

- Two kinds of conflicts may arise.

Shift-reduce and *reduce-reduce*.

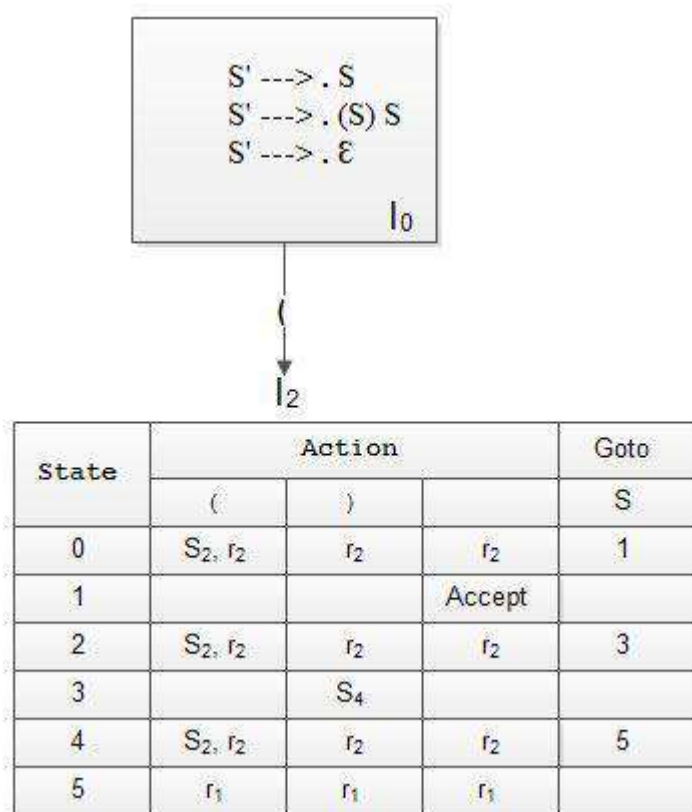
- In state 0 parser encounters a conflict.

It can shift state 2 on stack when next token is (.

It can reduce production 2: $S \rightarrow \epsilon$ on +.

This is called a *shift-reduce* conflict.

This conflict also appears in states 2 and 4.



Shift-reduce conflict parser can shift and can reduce.

Reduce-reduce conflict two (or more) productions can be reduced.

SLR(1) grammars

- SLR(1) parsing increases the power of LR(0) significantly.

Look ahead token is used to make parsing decisions

Reduce action is applied more selectively according to FOLLOW set.

- A grammar is SLR(1) if two conditions are met in every state.

If $A \rightarrow \alpha \cdot x \gamma$ and $B \rightarrow \beta \cdot$, then token $x \notin \text{FOLLOW}(B)$.

If $A \rightarrow \alpha \cdot$ and $B \rightarrow \cdot$ then $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) = \emptyset$.

- Violation of first condition results in *shift-reduce conflict*.

$A \rightarrow \alpha \cdot x \gamma$ and $B \rightarrow \beta \cdot$ and $x \in \text{FOLLOW}(B)$ then ...

Parser can shift x and reduce $B \rightarrow \beta$.

- Violation of second condition results in *reduce-reduce conflict*.

$A \rightarrow \alpha \cdot$ and $B \rightarrow \beta \cdot$ and $x \in \text{FOLLOW}(A) \cap \text{FOLLOW}(B)$.

Parser can reduce $A \rightarrow \alpha$ and $B \rightarrow \beta$.

- SLR(l) grammars are a superset of LR(O) grammars.

LR(1) Parser / Canonical LR (CLR)

- Even more powerful than SLR(l) is the LR(l) parsing method.
- LR(l) includes LR(O) items and a look ahead token in itemsets.
- An LR(l) item consists of,
 - o Grammar production rule.
 - o Right-hand position represented by the dot and.
 - o Lookahead token.
 - o $A \rightarrow X_1 \cdots X_i \cdot X_{i+1} \cdots X_n, l$ where l is a *lookahead token*
- The \cdot represents how much of the right-hand side has been seen,
- o $X_1 \cdots X_i$ appear on top of the stack.
- o $X_{i+1} \cdots X_n$ are expected to appear on input buffer.
- The lookahead token l is expected after $X_1 \cdots X_n$ appears on stack.
- An LR(l) state is a set of LR(l) items.

Introduction to LALR Parser

- LALR stands for *lookahead LR* parser.
- This is the extension of LR(O) items, by introducing the one symbol of lookahead on the input.
- It supports large class of grammars.
- The number of states in LALR parser is lesser than that of LR(1) parser. Hence, LALR is preferable as it can be used with reduced memory.
- Most syntactic constructs of programming language can be stated conveniently.

Steps to construct LALR parsing table

- Generate LR(l) items.
- Find the items that have same set of first components (core) and merge these sets into one.
- Merge the *goto's* of combined itemsets.
- Revise the parsing table of LR(l) parser by replacing states and *goto's* with combined states and combined *goto's* respectively.

2.5.5 Constructing a Parse Tree

The bottom-up tree construction process has two aspects.

- 1- When we shift an input symbol a onto the stack we create a one-node tree labelled a . Both the root and the yield of this tree are a , and the yield truly represents the string of the terminals "reduced" (by zero reduction) to symbol a .

- 2- When we reduce X_1, X_2, \dots, X_n to A , we create a new node labelled A . Its children, from the left to right, are the roots of the trees for X_1, X_2, \dots, X_n . If for all i the tree for X_i has yielded X_i , then the yield for the new tree is X_1, X_2, \dots, X_n . This string has in fact been reduced to A by a series of reductions culminating in the present one. As a special case, if we reduced E to A we create a node labelled A with one child labelled E .

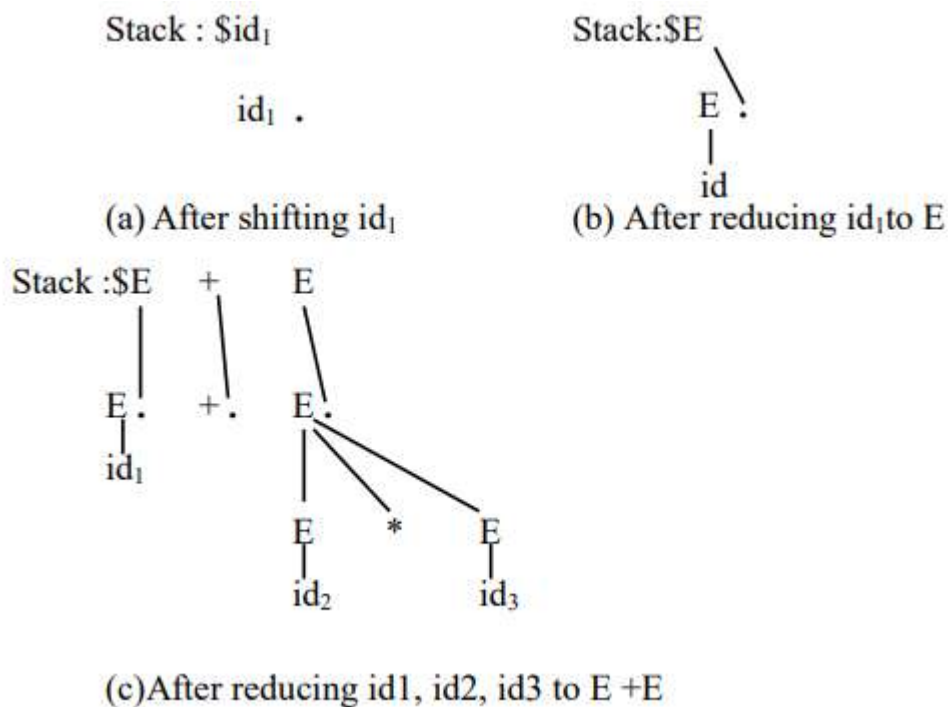


Fig.6: parse tree construction

2.5.6 Operator – Precedence Parsing

For a certain small class of Grammars we can easily construct efficient shiftreduce parsers by hand. These grammars have the property that no production right side is $id E$ or has two adjacent non-terminals. A grammar with the latter property is called an Operator grammar.

EX:- The following grammar for expressions

$E \longrightarrow EAE \mid (E) \mid -E \mid id$

$A \longrightarrow + \mid - \mid * \mid / \mid \uparrow$

Is not operator grammar, because the right side EAE has two (in fact three) consecutive non-terminals.

However, if a substitute for A each of its alternates, we obtain the following operator grammar:

$E \longrightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \text{id} \cdot$

In operator-precedence parsing, we use **three disjoint precedence relations** $<, =, >$, between certain pairs of **terminals**. These precedence relations guide the selection of handle. **If $a < b$, we say a "yields precedence to" b ; if $a = b$, a "has the same precedence as" b ; if $a > b$, a "takes precedence over" b .** We should caution that these relations differ from arithmetic operators. For example we could have $a < b$ and $a > b$ for the same language, or we might have none of $a < b$, $a = b$, and $a > b$ holding for some terminals a and b .

There are **two common ways of determining what precedence relation** should hold **between a pair of terminals**. The **first** method is based on the traditional notations of **associativity and precedence** of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ < . *$ and $* > +$. This approach will be seen to resolve the ambiguous of grammar (3) and to enable us to write an operator-precedence parser for it.

The **second** method of selecting operator-precedence relations are to first construct an **unambiguous grammar** for the language, a grammar which reflect the **correct associativity and precedence** in its parse trees.

Using Operator Precedence Relations

The goal of the precedence is to determine the handle of a right-sentential form, with $<$, marking the left end, $=$ appearing in the interior of the handle, and $>$ marking the right end. Suppose we have a right-sentential form of an operator grammar. The fact that **no adjacent non-terminals** appear on the right side of productions implies that no right-sentential form will have two adjacent non-terminals. Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 \dots a_n \beta_n$, where each β_i is either ϵ or a single non-terminal. Suppose that between a_i and a_{i+1} exactly one of the relations $<, =, >$. Further, **$\$$ marks each end of the string**, and we define $\$ < . b$ and $b > \$$ for terminals b .

Now, **suppose we remove the non-terminals from the string and place the correct relation, $>, =, <$. Between each pair of terminals and between the end most terminals and the $\$$'s marking the ends of the string.** For example, suppose we initially have the right-sentential form **id+id*id** and the precedence relations are given in Fig.7.

	id	+	*	\$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
\$	<.	<.	<.	

Fig.7: operator precedence relations

Then the string with the precedence relations inserted is:

$\$<. \text{Id} .> + <. \text{Id} .> * <. \text{Id} .> \$$ ----- (4)

Now the handle can be found by the following process,

- 1- Scan the string from the left and until the left most .> is encountered. In (4) above, this occurs between the first id and +.
- 2- Then scan backwards (to the left) over any '='s until a <. Is encountered. In (4) above we scan backwards to \$.
- 3- The handle contains everything to the left of the first .> and to the right of the <. Encountered in step (2). In (4) the handle is the first id.

UNIT -III

3.1 Syntax directed translation

In syntax directed translation, is a CFG in which a program fragment called an output action or semantic action or semantic rule is associated with each production.

So we can say that

1. Grammar + semantic rule = SDT (syntax directed translation)
 - In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
 - Value associated with a grammar symbol is called a translation of the symbol
 - In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
 - If we have a production with several instances of the same symbol on the right, we shall distinguish the symbols with superscript

Synthesized translation

It defines the value of the translation of the nonterminal on left side of the production as a function of the translations of the nonterminals on the right side

$$E \rightarrow E^{(1)} + E^{(2)} \{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$$

Inherited translation

The translation of nonterminal on the right side of the production is defined in terms of translation of the nonterminal on the left.

Ex:

$$A \rightarrow XYZ \{Y.VAL := 2 * A.VAL\}$$

Example

Production	Semantic Rules
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (F)$	$F.val := F.val$
$F \rightarrow \text{num}$	$F.val := \text{num.lexval}$

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

3.2 Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
------------	----------------

$S \rightarrow E \$$	$\{ \text{printE.VAL} \}$
$E \rightarrow E + E$	$\{ E.\text{VAL} := E.\text{VAL} + E.\text{VAL} \}$
$E \rightarrow E * E$	$\{ E.\text{VAL} := E.\text{VAL} * E.\text{VAL} \}$
$E \rightarrow (E)$	$\{ E.\text{VAL} := E.\text{VAL} \}$
$E \rightarrow I$	$\{ E.\text{VAL} := I.\text{VAL} \}$
$I \rightarrow I \text{ digit}$	$\{ I.\text{VAL} := 10 * I.\text{VAL} + \text{LEXVAL} \}$
$I \rightarrow \text{digit}$	$\{ I.\text{VAL} := \text{LEXVAL} \}$

3.3 Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parse the input and produce a parse tree as a result.

For Example for the input expression **23*5+4\$**, the program is to produce the value 119.

Example Parse tree for SDT:

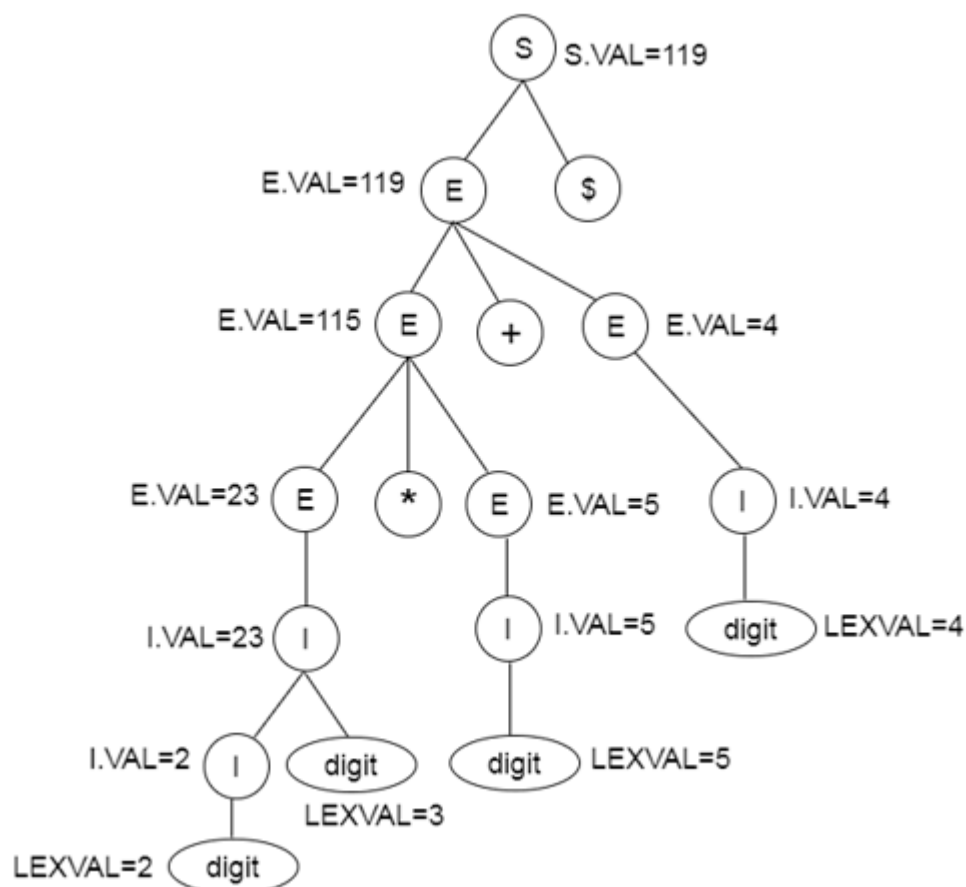


Fig: Parse tree

3.4 Symbol Tables

Introduction

A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a symbol table. The information collected about a name includes:

- 1- The string of characters by which it is denoted.
- 2- It's type (e.g. integer, real, string).
- 3- It's form (e.g. a simple variable, a structure).
- 4- It's location in memory.
- 5- Other attributes depending on the language.

Each entry in the symbol table is a pair of the form (name, information). Each time a name is encountered, the symbol table is searched to see whether that name has been seen previously. If the name is new, it is entered into the table. Information about that name is entered into the table during lexical and syntactic analysis.

The information collected in the symbol table is used during several stages in the compilation process. It is used in semantic analysis, that is, in checking that uses of names are consistent with their implicit or explicit declarations. It is also used during code generation. Then we need to know how much and what kind of run-time storage must be allocated to a name.

There are also a number of ways in which the symbol table can be used to aid in error detection and correction. For example, we can record whether an error message such as "variable A undefined" has been printed out before, and reject from doing so more than once. Additionally, space in the symbol table can be used for code-optimization purposes, such as to flag temporaries that are used more than once.

The primary issues in symbol table design are the format of the entries, the method of access, and the place where they are stored (primary or secondary storage). Block-structured languages impose another problem in that the same identifier can be used to represent distinct names with nested scopes. In compilers for such languages, the symbol table mechanism must make sure that the inner most occurrence of an identifier is always found first, and that names are removed from the active portion of the symbol table when they are no longer active.

3.4.1 The Contents of a Symbol Table

A simple table is a table with two fields, a name field and information field. We require several capabilities of the symbol table. We need to be able to:-

- 1- Determine whether a given name is in the table.
- 2- 2- Add a new name to the table.
- 3- 3- Access the information, associated with a given name.
- 4- 4- Add new information for a given name.
- 5- 5- Delete a name or group of names from the table.

In a compiler, the names in the symbol table denote objects of various sorts. There may be separate tables for variable names, labels, procedure names, constants, and other types of names depending on the language. Depending on how lexical analysis is performed, it may be useful to enter keywords into the symbol table initially. If the languages does not reserve keywords (forbid the use of keywords as identifiers), then it is essential that keywords be entered into the symbol table and that they have associated information warning of their possible use as a keyword.

3.4.2 Basic Implementation Techniques

The first consideration of symbol table implementation is how entering and find, store and search for names. Depending on the number of names we wish to accommodate and the performance we desire, a wide variety of implementations is possible:-

Unordered List

Use of an unordered list is the simplest possible storage mechanism. The only data structure required is an array, with insertions being performed by adding new names in the next available location. Of course, a linked list may be used to avoid the limitations imposed by a fixed array size. Searching is simple using an iterative searching algorithm, but it is impractically slow except for very small tables.

Ordered List

If a list of names in an array is kept ordered, it may be searched using binary searched, which requires $O(\log(n))$ time for a list of n entries. However, each new entry must be inserted in the array in the appropriate location. Insertion in an ordered array is relatively expensive operation. Thus ordered lists are typically used only when the entire set of names in a table is known in advance. They are useful therefore for tables of reserved words.

Binary Search Trees

Binary search trees are a data structure designed to combine the size flexibility and insertion efficiency of a linked data structure with the search speed provided by a binary search. On average, entering or searching for a name in a binary search tree built from random inputs requires $O(\log(n))$ time. One compelling argument in favor of binary search trees is their simple, widely known implementation. This implementation simplicity and the common perception of good average case performance make binary search trees a popular technique for implementing symbol tables.

Hash Tables

Hash tables are probably the most common means of implementing symbol tables in production compilers and other system software. With a large enough table, a good hash function, and the appropriate collision-handling technique, Searching can be done in essentially constant time regardless of the number of entries in the table.

UNIT -IV

4.1 Code Optimization

It is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.

identification

Input : a sequence of 3-address stmt

- Output: A list of After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

4.2 Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
    item = 10;
    value = value + item;
```

```
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
    value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

4.3 Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

4.4 Loop Optimization

Begin

PROD:= 0;

I:=1;

Do

Begin

PROD:= PROD +A[I] * B[I];

I:= I+1;

END

While I<=20

End

- (1) PROD: =0
- (2) I:=1
- (3) T1:=4*I
- (4) T2=addr(a)-4
- (5) T3:=T2[T1] A[I]
- (6) T4 :=addr(b)-4
- (7) T5:=T4[T1] B[I]
- (8) T6:=T3*T5
- (9) PROD:= PROD+T6
- (10) I:=I+1
- (11) IF I<=20 goto(3)

4.5 Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block basic block with each 3-addr stmt in exactly one block

Method

1. We first determine the set of leaders, the first statements of bb.
2. (i) The first stmt is a Leader
(ii) Any stmt which is the target of a conditional or unconditional goto is a header
(iii) Any stmt which immediately follows a conditional goto is a header
3. For each leader construct its bb, which consist of leader and all stmts up to but not including the next leader or end of prg. Any stmt not placed in block can never be executed and may now be removed, if desired.

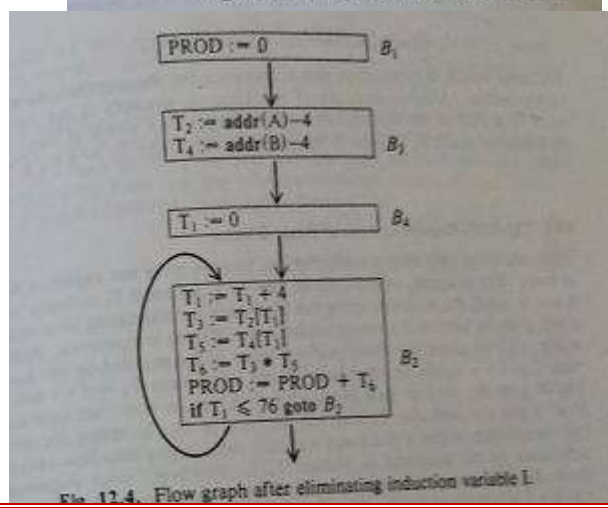
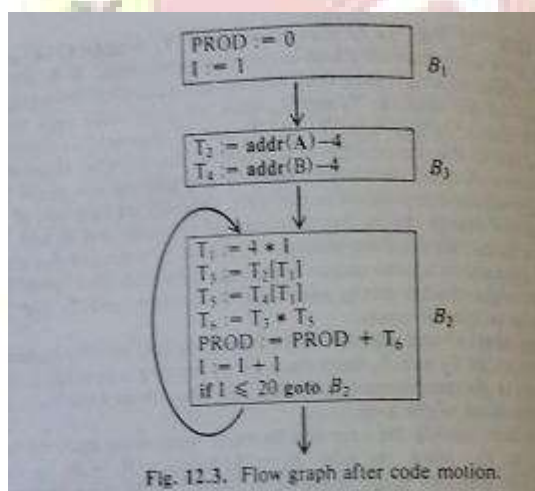
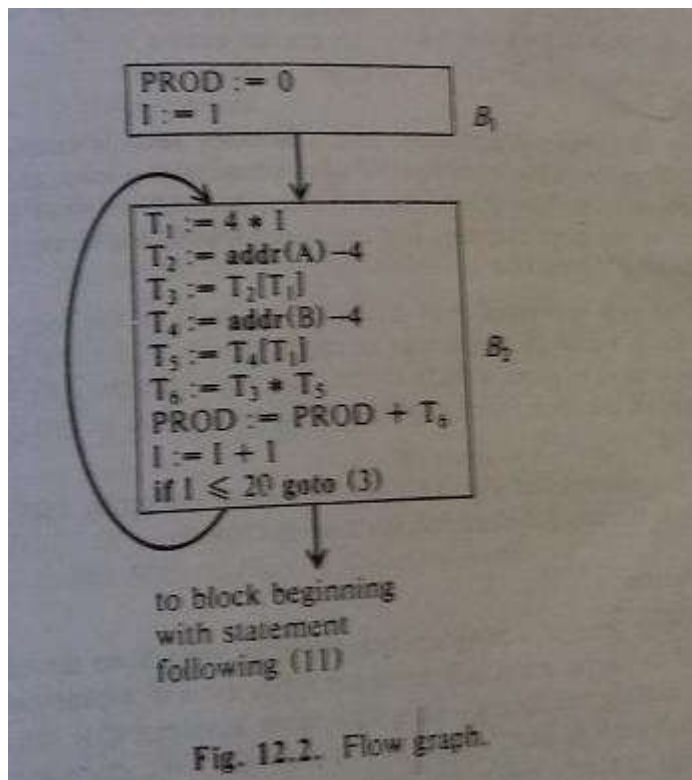
4.6 Flow Graph

It is useful to portray the bb and their successor relationships by a directed graph called flow graph. The nodes of the fg are the basic block. One node is distinguished as initial it is the block whose leader is the first stmt. There is a directed edge from block B1 to block B2, if B2 could immediately follow B1 during execution, that is

1. There is conditional or unconditional jump from last stmt of B1 to first stmt of B2
2. B2 immediately follows B1 in order of prg, and B1 does not end in unconditional jump

B1 is a predecessor of B2 and

B2 is a successor of B1



4.7 Peephole Optimization

A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimizations:

Redundant-instructions elimination

Flow-of-control optimizations

Algebraic simplifications

Use of machine idioms

Unreachable

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a

(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
```

```
....
```

```
If ( debug ) {
```

Print debugging information

}

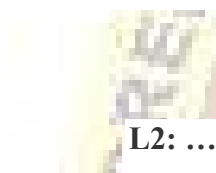
In the intermediate representations the if-statement may be translated as:

If debug =1 goto L1
goto L2

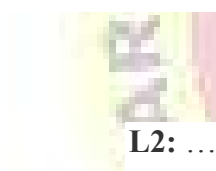
L1: print debugging information

L2: (a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:



If debug ≠1 goto L2
Print debugging information
L2: (b)



If debug ≠0 goto L2
Print debugging information
L2: (c)

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

4.8 Flows-of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

L1: gotoL2 (d)

by the sequence

goto L2

....

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

....

L1: goto L2 (e)

can be replaced by

If a < b goto L2

....

L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2

L3:

may be replaced by

If a < b goto L2

goto L3

.....

L3:

While the number of instructions in (e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e). Thus (f) is superior to (e) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

$$x := x + 0 \text{ or}$$

$$x := x * 1$$

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$$X^2 \rightarrow X * X$$

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

$$i := i + 1 \rightarrow i++$$

$$i := i - 1 \rightarrow i--$$

Production	Semantic action
$E \rightarrow E^{(1)} OP E^{(2)}$	$E.code = E^{(1)}.code \parallel E^{(2)}.code \parallel 'op'$
$E \rightarrow E^{(1)}$	$E.code = E^{(1)}.code$
$E \rightarrow id$	$E.code = id$

4.9 Intermediate representation

Intermediate code can be represented in two ways:

1. High Level intermediate code:

High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code

Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

4.10 Postfix Notation

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the product of x and y is with operator in the middle: $x * y$. But in the postfix notation, we place the operator at the right end as $xy *$.
- In postfix notation, the operator follows the operand.

Example

Consider the postfix expression $ab+c*$. Suppose a,b and c have values 1,3,5 respectively. To evaluate $13+5 *$ the following actions are taken:

1. Stack 1
2. Stack 3
3. Add the two topmost element pop them off the stack and then stack the result, 4
4. Stack 5
5. Multiply the two topmost element pop them off the stack and then stack the result 20.

The value on top of the stack at the end is the value of the entire expression.

Syntax-Directed Translation to Postfix Code

Example

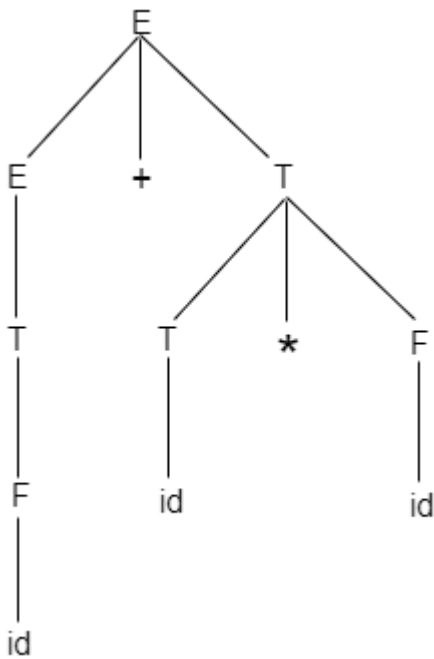
Production

1. $E \rightarrow E^{(1)} \text{ op } E^{(2)}$
2. $E \rightarrow E^{(1)}$
3. $E \rightarrow \text{id}$

Production	Program fragment
$E \rightarrow E^{(1)} \text{ OP } E^{(2)}$	{print op}
$E \rightarrow E^{(1)}$	{ }
$E \rightarrow \text{id}$	{print id}

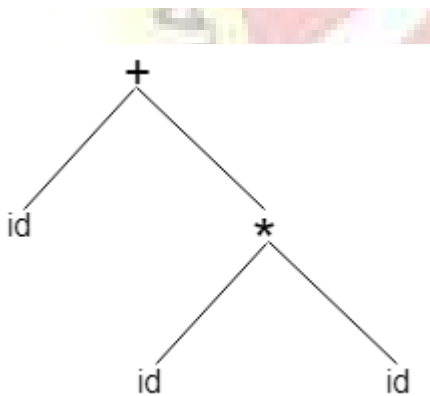
4.11 Parse tree and Syntax tree

When you create a parse tree then it contains more details than actually needed. So, it is very difficult to compiler to parse the parse tree. Take the following parse tree as an example:



- In the parse tree, most of the leaf nodes are single child to their parent nodes.
- In the syntax tree, we can eliminate this extra information.
- Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- Syntax tree is usually used when represent a program in a tree structure.

A sentence **id + id * id** would have the following syntax tree:



Abstract syntax trees are important data structures in a compiler. It contains the least unnecessary information.

Abstract syntax trees are more compact than a parse tree and can be easily used by a compiler.

Syntax-Directed Construction of Syntax Trees

Production	Semantic Action
$E \rightarrow E^{(1)} \text{ op } E^{(2)}$	$\{E.VAL := \text{NODE}(\text{op}, E^{(1)}.VAL, E^{(2)}.VAL)\}$
$E \rightarrow E^{(1)}$	$\{E.VAL := E^{(1)}.VAL\}$
$E \rightarrow -E^{(1)}$	$\{E.VAL := \text{UNARY}(-, E^{(1)}.VAL)\}$
$E \rightarrow \text{id}$	$\{E.VAL := \text{LEAF}(\text{id})\}$

The function $\text{NODE}(\text{OP}, \text{LEFT}, \text{RIGHT})$ takes three arguments. The first is the name of the operator, the second and third are pointers to roots of subtrees. The function $\text{UNARY}(\text{OP}, \text{CHILD})$ creates a new node labelled OP and makes CHILD its child.

4.12 Three address code

- Three-address code is an intermediate code. It is used by the optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example

Given Expression:

$$1. \ a := (-c * b) + (-c * d)$$

Three-address code is as follows:

$$t_1 := -c$$

$$t_2 := b * t_1$$

$$t_3 := -c$$

$$t_4 := d * t_3$$

$$t_5 := t_2 + t_4$$

$$a := t_5$$

t is used as registers in the target program.

The three address code can be represented in two forms: **quadruples** and **triples**.

4.13 Quadruples

The quadruples have four fields to implement the three address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

Operator
Source 1
Source 2
Destination

Fig: Quadruples field

Example

1. $A := -B * C + D$

Three-address code is as follows:

$T_1 := -B$

$T_2 := C + D$

$T_3 := T_1 * T_2$

$A := T_3$

These statements are represented by quadruples as follows:

	Operator	ARG1	ARG2	RESULT
(0)	uminus	B	-	T ₁
(1)	+	C	D	T ₂
(2)	*	T ₁	T ₂	T ₃

(3)	:=	T ₃	-	A
-----	----	----------------	---	---

4.14 Triples

The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

Operator
Source 1
Source 2

Fig: Triples field

Example:

1. $A := -B * C + D$

THREE ADDRESS CODE IS AS FOLLOWS:

$T_1 := -B$

$T_2 := C + D$

$T_3 := T_1 * T_2$

$A := T_3$

These statements are represented by triples as follows:

Operator			
(0)	uminus	B	-
(1)	+	C	D
(2)	*	(0)	(1)

(3)	:=	A	(2)
-----	----	---	-----

4.15 Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

1. $S \rightarrow id := E$
2. $E \rightarrow E1 + E2$
3. $E \rightarrow E1 * E2$
4. $E \rightarrow (E1)$
5. $E \rightarrow id$

The Abstract Translation Scheme

The translation of E have two fields

E.PLACE, the name that hold the value of expression and

E.CODE, a sequence of 3-address statements evaluating the expression.

The translation scheme of above grammar is given below:

Production rule	Semantic actions
$S \rightarrow id := E$	$\{A.CODE := E.CODE \parallel$ $id.PLACE \parallel := \parallel E.PLACE\}$
$E \rightarrow E^{(1)} + E^{(2)}$	$\{T := NEWTEMP();$ $E.PLACE := T;$ $E.CODE := E^{(1)}.CODE \parallel E^{(2)}.CODE \parallel$ $E.PLACE \parallel := \parallel E^{(1)}.PLACE \parallel + \parallel E^{(2)}.PLACE$ $\}$
$E \rightarrow E^{(1)} * E^{(2)}$	$\{T := NEWTEMP();$ $E.PLACE := T;$ $E.CODE := E^{(1)}.CODE \parallel E^{(2)}.CODE \parallel$ $E.PLACE \parallel := \parallel E^{(1)}.PLACE \parallel * \parallel E^{(2)}.PLACE$ $\}$

$E \rightarrow -E^{(1)}$	$T := \text{NEWTEMP}();$ $E.\text{PLACE} := T;$ $E.\text{CODE} := E^{(1)}.\text{CODE} \parallel$ $E.\text{PLACE} \parallel := - \parallel E^{(1)}.\text{PLACE} \}$
$E \rightarrow E^{(1)}$	$\{E.\text{PLACE} := E^{(1)}.\text{PLACE} ;$ $E.\text{CODE} := E^{(1)}.\text{CODE} \}$
$E \rightarrow \text{id}$	$\{E.\text{PLACE} := \text{id}.\text{PLACE} ;$ $E.\text{CODE} := \text{null} \}$

- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

Traces of Syntax Directed Translation

Input	stack	Place	Generated code
A:=-B*(C+D)			
:= -B*(C+D)	id	A	
-B*(C+D)	id:=	A -	
B*(C+D)	id:= -	A - -	
*(C+D)	id:= - id	A - -B	
*(C+D)	id:= - E	A - -B	T ₁ := -B
*(C+D)	id:= E	A - T ₁	
(C+D)	id:= - E *	A - T ₁ -	
C+D)	id:= -E * (A - T ₁ - -	
+D)	id:= -E * (id	A - T ₁ - - C	
+D)	id:= -E * (E	A - T ₁ - - C	
D)	id := -E * (E + A - T ₁ - - C -		

)	$id := -E * (E + id$	$A - T_1 - - C -$	
)	$id := -E * (E + E$	$A - T_1 - - C - D$	$T_2 := C + D$
)	$id := E * (E$	$A - T_1 - - T_2$	
	$id := E * (E)$	$A - T_1 - - T_2 -$	
	$id := E * E$	$A - T_1 - T_2$	$T_3 := T_1 * T_2$
	$id := E$	$A - T_3$	$A := T_3$
	A	A	

Assignment Statements with Mixed Types

Ex:

$E \rightarrow E + E$

$E \rightarrow E + E \quad \{ \text{If } E^{(1)}.MODE = \text{INTEGER AND } E^{(2)}.MODE = \text{INTEGER THEN}$
 $E.MODE := \text{INTEGER}$
 $\text{Else } E.MODE := \text{REAL} \}$

4.16 Boolean expressions

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1. $E \rightarrow E \text{ OR } E$
2. $E \rightarrow E \text{ AND } E$
3. $E \rightarrow \text{NOT } E$
4. $E \rightarrow (E)$
5. $E \rightarrow id \text{ relop } id$
6. $E \rightarrow \text{TRUE}$
7. $E \rightarrow \text{FALSE}$

The relop is denoted by $<, >, <=, >=, =$.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Methods of Translating Boolean Expression

Goto L

If A goto L

If A relop B goto L

Numerical Representations

Ex:

The translation for

A or B and C

Is the three-address sequence

$T_1 := B \text{ and } C$

$T_2 := A \text{ or } T_1$

A relational expression such as $A < B$ is equivalent to conditional statement if $A < B$ then 1 else 0, translated into 3-address sequence

(1) If $A < B$ goto (4)

(2) $T := 0$

(3) Goto (5)

(4) $T := 1$

(5)

Translation of **$A < B \text{ or } C$**

(1) If $A < B$ goto (4)

(2) $T_1 := 0$

(3) Goto (5)

(4) $T_1 := 1$

(5) $T_2 := T_1 \text{ or } C$

(6)

4.17 Control-Flow Representation of Boolean Expression

Ex:

Consider an expression of the form $E^{(1)} \text{ or } E^{(2)}$. If $E^{(1)}$ is true then we know that E itself is true. If $E^{(1)}$ is false then we evaluate $E^{(2)}$, so we make FALSE for $E^{(1)}$ be the first statement in the code for $E^{(2)}$.

The code we generate therefore is a series of branching statements with targets of jumps temporarily left unspecified. Each such quadruples will be on one or another list of quadruples to be filled in when the proper location is found. We call this subsequent filling in of quadruples **backpatching**

To manipulate the list of quadruples we use three functions.

1. MAKELIST(i) creates a new list containing only i , an index into the array of quadruples being generated. MAKELIST returns a pointer to the list it has made.
2. MERGE(p_1, p_2) takes the list pointed to by p_1 and p_2 , concatenates them into one list and returns a pointer to the concatenated list
3. BACKPATCH(p, i) makes each of the quadruples on list pointed to by p take quadruple i as a target.

$M \rightarrow \{ M.QUAD := NEXTQUAD \}$

The revised grammar is

- (1) $E \rightarrow E^{(1)} \text{ or } M E^{(2)}$
- (2) $\quad \mid E^{(1)} \text{ and } M E^{(2)}$
- (3) $\quad \mid \text{Not } E^{(1)}$
- (4) $\quad \mid (E^{(1)})$
- (5) $\quad \mid \text{Id}$
- (6) $\quad \mid \text{Id}^{(1)} \text{ relop id}^{(2)}$
- (7) $M \rightarrow \{ \}$

The syntax-directed translation scheme is as follows:

- (1) $E \rightarrow E^{(1)} \text{ or } M E^{(2)}$

$\{ \text{BACKPATCH}(E^{(1)}.FALSE, M.QUAD);$

$E.TRUE := \text{MERGE}(E^{(1)}.TRUE, E^{(2)}.TRUE);$

$E.FALSE := E^{(2)}.FALSE \}$

(2) $E \rightarrow E^{(1)} \text{ and } M E^{(2)}$

{ BACKPATCH($E^{(1)}$.TRUE, M.QUAD);

E .TRUE := $E^{(2)}$.TRUE}

E .FALSE := MERGE($E^{(1)}$.FALSE, $E^{(2)}$.FALSE);

(3) $E \rightarrow \text{NOT } E^{(1)}$

{ E .TRUE := $E^{(1)}$.FALSE;

E .FALSE := $E^{(1)}$.TRUE}

(4) $E \rightarrow (E^{(1)})$

{ E .TRUE := $E^{(1)}$.TRUE;

E .FALSE := $E^{(1)}$.FALSE}

(5) $E \rightarrow \text{id}$

{ E .TRUE := MAKELIST(NEXTQUAD);

E .FALSE := MAKELIST(NEXTQUAD + 1);

GEN (if id.PLACE goto -);

GEN(goto -) }

(6) $E \rightarrow \text{Id}^{(1)} \text{ relop id}^{(2)}$

{ E .TRUE := MAKELIST(NEXTQUAD);

E .FALSE := MAKELIST(NEXTQUAD + 1);

GEN (if $\text{id}^{(1)}$.PLACE relop $\text{id}^{(2)}$.PLACE goto -);

GEN(goto -) }

(7) $M \rightarrow \text{£}$

{M.QUAD :=NEXTQUAD}

Here is the example which generates the three address code using the above translation scheme:

Consider the expression:

$P < Q$ or $R < S$ and $T < U$

The values of the translations at each node created by assuming that NEXTQUAD has the initial value 100 and is incremented with each call to GEN. Consider the semantic actions occurring as the numbered nodes as “created” in a bottom-up parse.

In response to the reduction corresponding to node 1, the two quadruples

100 : if $P < Q$ goto –

101 : goto –

Are generated. Then node 2, records the value of NEXTQUAD which is 102. Node 3 generates the quadruples

102: if $R < S$ goto-

103: goto-

Node 4 records the current value of NEXTQUAD, which is now 104. Node 5 generates the quadruples

104: if $T < U$ goto-

105 : goto –

Node 6 corresponds to a reduction by $E \rightarrow E^{(1)}$ and $M E^{(2)}$.

Node 7, the root is created by a reduction $E \rightarrow E^{(1)}$ and $M E^{(2)}$. The associated semantic routine calls BACKPATCH({101},102) which leaves the quadruples looking like:

100 : if $P < Q$ goto –

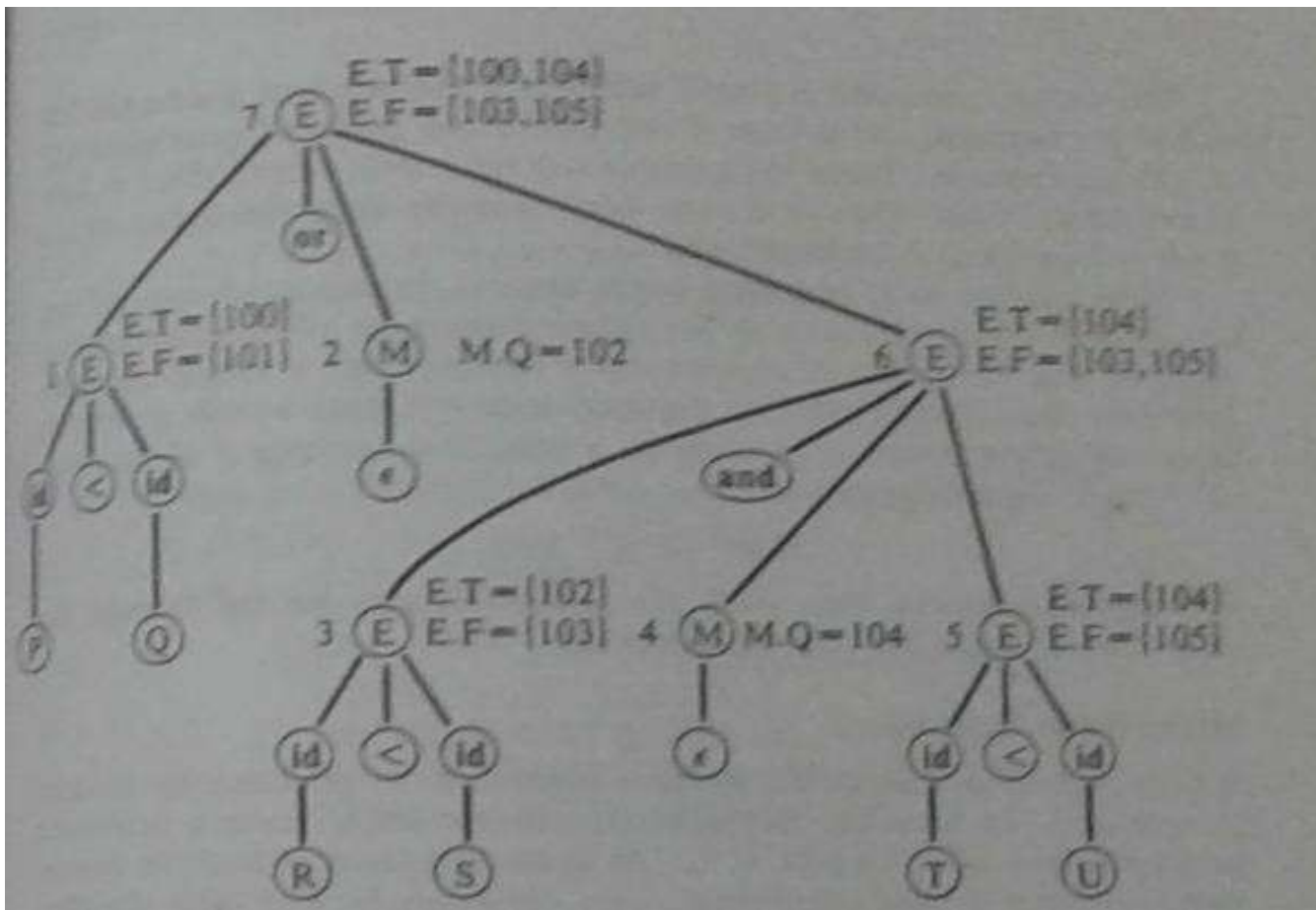
101 : goto 102

102: if $R < S$ goto 104

103: goto-

104: if $T < U$ goto-

105 : goto –



UNIT-V

5.1 Code generation

It can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

5.2 Code Generator

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement $x := y + z$. It can have the following sequence of codes:

```
MOV x, R0
ADD y, R0
```

Register and Address Descriptors:

- A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.

A code-generation algorithm:

- The algorithm takes a sequence of three-address statements as input. For each three address statement of the form $a := b \text{ op } c$ perform the various actions. These are as follows:
- Invoke a function `getreg` to find out the location L where the result of computation $b \text{ op } c$ should be stored.
- Consult the address description for y to determine y' . If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L .
- Generate the instruction **OP z' , L** where z' is used to show the current location of z . if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L . If x is in L then update its descriptor and remove x from all other descriptor.
- If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of $x := y \text{ op } z$ those register will no longer contain y or z .

Generating Code for Assignment Statements:

The assignment statement $d := (a-b) + (a-c) + (a-c)$ can be translated into the following sequence of three address code:

1. $t := a - b$
2. $u := a - c$
3. $v := t + u$
4. $d := v + u$

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
$t := a - b$	MOV a, R0 SUB b, R0	R0 contains t	t in R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
$v := t + u$	ADD R1, R0	R0 contains v	u in R1

		R1 contains u	v in R1
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

5.3 Design Issues

In the code generation phase, various issues can arise:

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to the code generator

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
 - a) Postfix notation
 - b) Syntax tree
 - c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

- a) **Assembly language:** It allows subprogram to be separately compiled.
- b) **Relocatable machine language:** It makes the process of code generation easier.
- c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.

- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

1. $a := b + c$
2. $d := a + e$

Inefficient assembly code is:

1. MOV b, R0 R0 \rightarrow b
2. ADD c, R0 R0 \leftarrow c + R0
3. MOV R0, a a \rightarrow R0
4. MOV a, R0 R0 \rightarrow a
5. ADD e, R0 R0 \leftarrow e + R0
6. MOV R0, d d \rightarrow R0

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

For example:

Consider the following division instruction of the form:

1 D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

5.4 Target Machine

- The target computer is a type of byte-addressable machine. It has 4 bytes to a word.
- The target machine has n general purpose registers, R0, R1,..., Rn-1. It also has two-address instructions of the form:

1. op source, destination

Where, op is used as an op-code and source and destination are used as a data field.

- It has the following op-codes:
 ADD (add source to destination)
 SUB (subtract source from destination)
 MOV (move source to destination)
- The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
absolute	M	M	Add R0, R1	1
register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1
indirect register	*R	contents(R)	ADD * 100	0
indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

- Here, cost 1 means that it occupies only one word of memory.
- Each instruction has a cost of 1 plus added costs for the source and destination.
- Instruction cost = 1 + cost is used for source and destination mode.

5.5 Error Detection and Recovery

Error can be classified into mainly two categories

1. Compile time error
2. Runtime error

5.5.1 Lexical Error

This type of errors can be detected during lexical analysis phase. Typical lexical phase errors are:

1. Spelling errors. Hence get incorrect tokens.
2. Exceeding length of identifier or numeric constants.
3. Appearance of illegal charactersEx:

```
fi ( )
{
}
```

In above code 'fi' cannot be recognized as a misspelling of keyword if rather lexical analyzer will understand that it is an identifier and will return it as valid identifier. Thus misspelling causes errors in token formation.

5.5.2 Syntax error

This type of error appear during syntax analysis phase of compiler Typical errors are:

- ✓ Errors in structure
- ✓ Missing operators
- ✓ Unbalanced parenthesis

The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.

5.5.3 Semantic error

This type of error detected during semantic analysis phase. Typical errors are:

- Incompatible types of operands
- Undeclared variable
- Not matching of actual argument with formal argument

5.6 Error recovery strategies OR Ad-hoc and systematic methods

1. Panic mode

- ✓ This strategy is used by most parsing methods. This is simple to implement.
- ✓ In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon or end. These tokens indicate an end of input statement.
- ✓ Thus in panic mode recovery a considerable amount of input is skipped without checking it for additional errors.
- ✓ This method guarantees not to go in infinite loop.
- ✓ If there is less number of errors in the same statement then this strategy is best choice.

2. Phrase level recovery

- ✓ In this method, on discovering error parser performs local correction on remaining input.
- ✓ It can replace a prefix of remaining input by some string. This actually helps parser to continue its job.
- ✓ The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolon; this type of local correction is decided by compiler designer.
- ✓ While doing the replacement a care should be taken for not going in an infinite loop.
- ✓ This method is used in many error-repairing compilers.

3. Error production

- ✓ If we have knowledge of common errors that can be encountered then we can incorporate these errors by augmenting the grammar of the corresponding language with error productions that generate the erroneous constructs.
- ✓ If error production is used then during parsing, we can generate appropriate error message and parsing can be continued.
- ✓ This method is extremely difficult to maintain. Because if we change grammar then it becomes necessary to change the corresponding productions.

4. Global production

- ✓ We often want such a compiler that makes very few changes in processing an incorrect input string.
- ✓ We expect less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- ✓ Such methods increase time and space requirements at parsing time.

Global production is thus simply a theoretical concept.

5.7 Error recovery in predictive parsing

Consider the grammar given below:

$$E ::= TE'$$

$$E' ::= +TE' \mid \varepsilon$$

$$T ::= FT'$$

$$T' ::= *FT' \mid \varepsilon$$

$$F ::= (E)id$$

Insert 'synch' in FOLLOW symbol for all non terminals. 'synch' indicates resume the parsing

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

NT	Input Symbol					
	id	+	*	()	\$
E	$E \Rightarrow TE'$			$E \Rightarrow TE'$	synch	Synch
E'		$E' \Rightarrow +TE'$			$E' \Rightarrow \varepsilon$	$E' \Rightarrow \varepsilon$
T	$T \Rightarrow FT'$	synch		$T \Rightarrow FT'$	Synch	synch
T'		$T' \Rightarrow \varepsilon$	$T' \Rightarrow *FT'$		$T' \Rightarrow \varepsilon$	$T' \Rightarrow \varepsilon$
F	$F \Rightarrow \langle id \rangle$	synch	synch	$F \Rightarrow (E)$	synch	synch

- ✓ If parser looks entry $M[A, a]$ and finds that it is blank then i/p symbol a is skipped.
- ✓ If entry is "synch" then non terminal on the top of the stack is popped in an attempt to resume parsing.
- ✓ If a token on top of the stack does not match i/p symbol then we pop token from the stack.

Stack	Input	Remarks
\$E)id*+id\$	Error, skip)
\$E	id*+id\$	
\$E' T	id*+id\$	
\$E' T' F	id*+id\$	
\$E' T' id	id*+id\$	
\$E' T'	*+id\$	
\$E' T' F*	*+id\$	
\$E' T' F	+id\$	Error, $M[F, +] = \text{synch}$
\$E' T'	+id\$	F has been popped.
\$E'	+id\$	
\$E' T+	+id\$	

\$E' T	id\$	
\$E' T' F	id\$	
\$E' T' id	id\$	
\$E' T'	\$	
\$E'	\$	
\$	\$	

5.8 Error recovery in LR parsing

- ✓ An LR parser will detect an error when it consults the parsing action table and finds an error entry.
- ✓ Consider expression grammar $E \rightarrow E+E \mid E * E \mid (E) \mid id$

$I_0:$ $E' \rightarrow .E$ $E \rightarrow .E+E$ $E \rightarrow .E * E$ $E \rightarrow .(E)$ $E \rightarrow .id$	$I_1:$ $E' \rightarrow E.$ $E \rightarrow E. + E$ $E \rightarrow E. * E$	$I_2:$ $E \rightarrow (E.)E-$ $>.E+E \quad E-$ $>.E * E \quad E-$ $>.(E)$ $E \rightarrow .id$	$I_3:$ $E \rightarrow id.$	$I_4:$ $E \rightarrow E + .E$ $E \rightarrow .E + E$ $E \rightarrow .E * E \quad E-$ $>.(E)$ $E \rightarrow .id$
$I_5:$ $E \rightarrow E * .E$ $E \rightarrow .E + E E-$ $>.E * E \quad E-$ $>.(E)$ $E \rightarrow .id$	$I_6:$ $E \rightarrow (E.)E-$ $>.E + E \quad E-$ $>.E * E$	$I_7:$ $E \rightarrow E + .E$ $E \rightarrow .E + E$ $E \rightarrow .E * E$	$I_8:$ $E \rightarrow E * .E$ $E \rightarrow .E + E$ $E \rightarrow .E * E$	$I_9:$ $E \rightarrow (E.)$

Set of LR(0) items for given grammar

- ✓ Parsing table given below shows error detection and recovery.

States	action						goto
	id	+	*	()	\$	
0	S3	E1	E1	S2	E2	E1	1
1	E3	S4	S5	E3	E2	Acc	
2	S3	E1	E1	S2	E2	E1	6
3	R4	R4	R4	R4	R4	R4	
4	S3	E1	E1	S2	E2	E1	7
5	S3	E1	E1	S2	E2	E1	8
6	E3	S4	S5	E3	S9	E4	
7	R1	R1	S5	R1	R1	R1	
8	R2	R2	R2	R2	R2	R2	
9	R3	R3	R3	R3	R3	R3	

- ✓ The error routines are as follow:
 - E1: push an imaginary id onto the stack and cover it with state 3.

Issue diagnostics “missing operands”. This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an id or left parenthesis. Instead, an operator + or *, or the end of the input found.

- E2: remove the right parenthesis from the input. Issue diagnostics “unbalanced right parenthesis”. This routine is called from states 0, 1, 2,4,5 on finding right parenthesis.
- E3: push + on to the stack and cover it with state 4
Issue diagnostics “missing operator”. This routine is called from states 1 or 6 when expecting an operator and an id or right parenthesis is found.
- E4: push right parenthesis onto the stack and cover it with state 9. Issue diagnostics “missing right parenthesis”. This routine is called from states 6 when the end of the input is found. State 6 expects an operator or right parenthesis.

Stack	Input	Error message and action
0	id+)\$	
0id3	+) \$	
0E1	+) \$	
0E1+4) \$	
0E1+4	\$	“unbalanced right parenthesis” e2 removes right parenthesis
0E1+4id3	\$	“missing operands” e1 pushes id 3 on stack
0E1+4E7	\$	
0E1	\$	
