SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-I Compiler Design – SCSA1604

UNIT 1-- LEXICAL ANALYSIS

Structure of compiler – Functions and Roles of lexical phase – Input buffering – Representation of tokens using regular expression – LEX – Properties of regular expression – Finite Automata – Regular Expression to Finite Automata – NFA to Minimized DFA.

STRUCTURE OF COMPILER:

Compiler is a translator program that reads a program written in one language -the source language- and translates it into an equivalent program in another language-the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

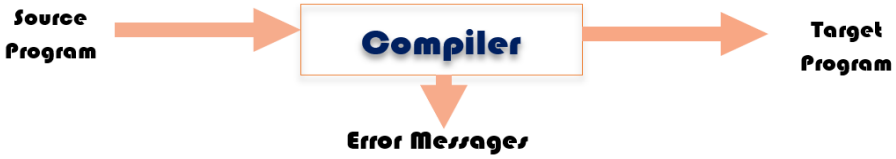


Fig. 1.1 A Compiler

A LANGUAGE-PROCESSING SYSTEM:

The input to a compiler may be produced by one or more preprocessor and further processing of the compiler's output may be needed before running machine code is obtained.

Preprocessors:

Preprocessors produce input to compilers. They may perform the following functions:

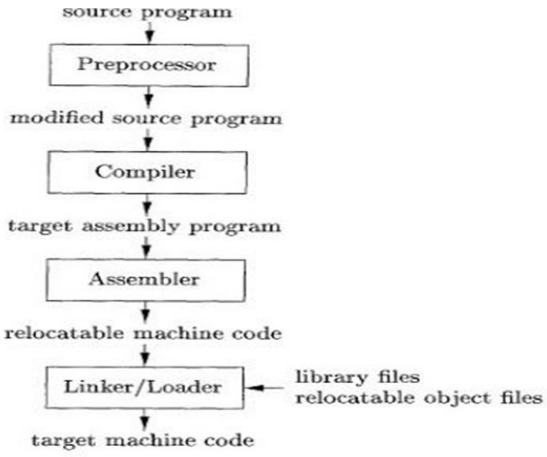


Fig. 1.2. A language-processing system

1. Macro processing: A preprocessor may allow a user to define macros that are shorthand for longer constructs.
2. File inclusion: A preprocessor may include header files into the program text. For example, the C
3. Preprocessor causes the contents of the file <*stdio.h*> to replace the statement
4. *#include* <*stdio.h*>when it processes a file containing this statement.
5. "Rational" preprocessors. These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.
6. Language extensions. These processors attempt to add capabilities to the language by what amounts to built-in-macros.

Assemblers:

Some compilers produce assembly code, which is passed to an assembler for producing a relocatable machine code that is passed directly to the loader/linker editor. The assembly code is the mnemonic version of machine code. A typical sequence of assembly code is:

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

Loaders and Linker-Editors:

- A loader program performs two functions namely, loading and link-editing. The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the alters instructions and data in memory at the proper locations.
- The link-editor allows making a single program from several files of relocatable machine code.

THE PHASES OF A COMPILER

Analysis-Synthesis Model of Compilation:

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

The analysis consists of three phases:

1. Linear analysis, in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of character having a collective meaning.
2. Hierarchical analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

A compiler operates in phases, each of which transforms the source program from one representation to another. The structure of compiler is shown in Fig.1.3.The first three phases form the analysis portion of the compiler and rest of the phases form the synthesis phase.

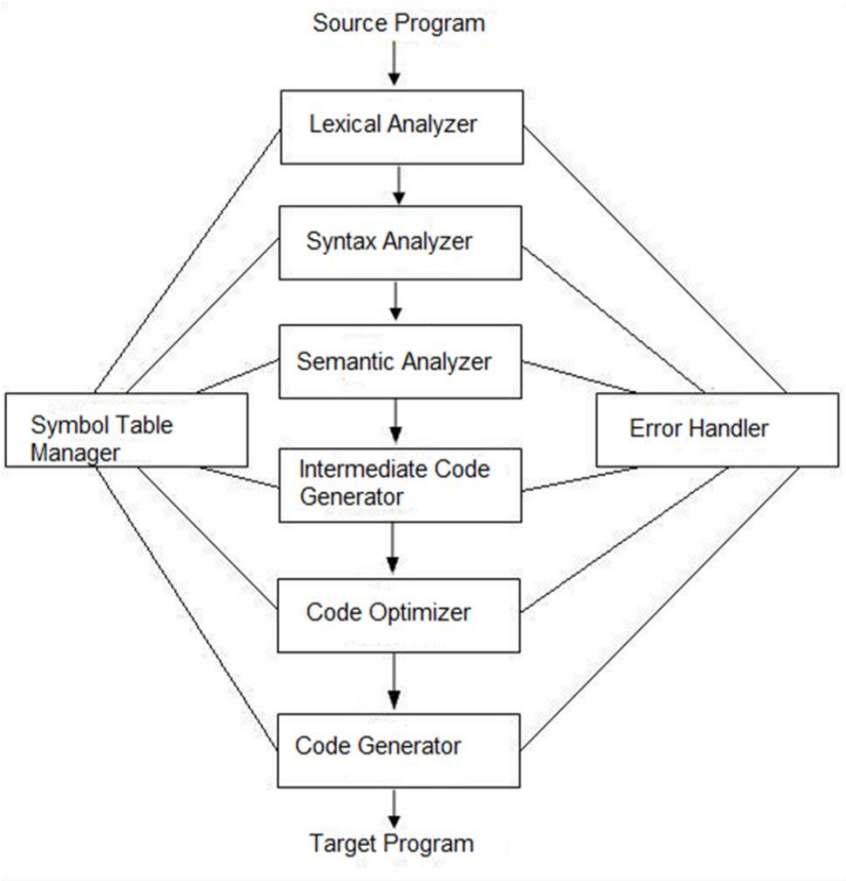


Fig. 1.3. Phases of a compiler

Lexical Analysis:

In a compiler, linear analysis is called lexical analysis or scanning.

- It is a first phase of a compiler
- Lexical Analyzer is also known as scanner
- Reads the characters in the source program from left to right and groups the characters into stream of Tokens.
- Such as Identifier, Keyword, Punctuation character, Operator.
- **Pattern:** Rule for a set of string in the input for which a token is produced as output.
- A **Lexeme** is a sequence of characters in the source code that is matched by the Pattern for a Token.

For example, in lexical analysis the characters in the assignment statement,

position = initial + rate * 60

would be grouped into the following tokens:

1. The identifier *position*.
2. The assignment symbol=.
3. The identifier *initial*.
4. The plus sign.
5. The identifier *rate*.
6. The multiplication sign.
7. The number60.

The blanks separating the characters of these tokens would be eliminated during this phase.

Syntax Analysis:

Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize the output.

The source program is represented by a parse tree as one shown in Fig. 1.5.

- This phase is called the syntax analysis or **Parsing**.
- It takes the token produced by lexical analysis as input and generates a parse tree.
- In this phase, token arrangements are checked against the source code grammar.
- i.e. the parser checks if the expression made by the tokens is syntactically correct.

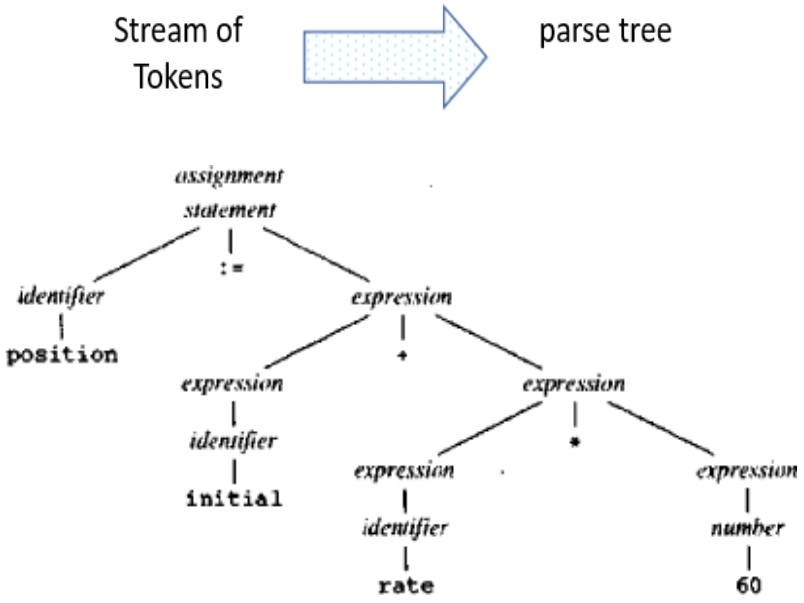


Fig.1.5. Parse tree for position = initial + rate * 60

The hierarchical structure of a program is expressed by recursive rules .i.e by context-free grammars. The following rules define an expression:

1. Any identifier is an expression. i.e. E →id
2. Any number is an expression. i.e. E →num
3. If expression1 (E^1) and expression2 (E^2)are expressions ,i.e. E → E + E | E * E | (E)

Rules (1) and (2) are basic rules which are non-recursive, while rule(3) define expression in terms of operators applied to other expressions.

Semantic Analysis:

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

- An important component of semantic analysis is type checking. i.e .whether the operands are type compatible.
- For example, a real number used to index an array.

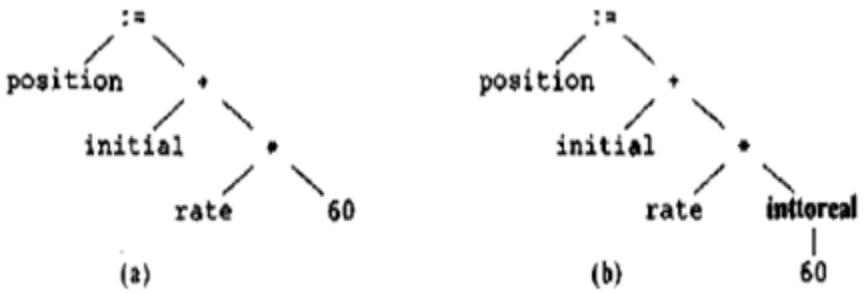


Fig. 1.6. Semantic analysis inserts a conversion from integer to real

Intermediate Code Generation:

After semantic analysis, some compilers generate an explicit intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program. There are varieties of forms.

- Three address code
- Postfix notation
- Syntax Tree

The commonly used representation is three address formats. The format consists of a sequence of instructions, each of which has at most three operands. The IR code for the given input is as follows:

temp1 = inttoreal (60)

temp2 = id3 * temp1

temp3 = id2 + temp2

id1 = temp3

Code Optimization:

- This phase attempts to improve the intermediate code, so that faster running machine code will result.

- There is a better way to perform the same calculation for the above three address code ,which is given as follows:

  temp1 = id3 * 60.0

  id1 = id2 + temp1

- There are various techniques used by most of the optimizing compilers, such as:
  1. Common sub-expression elimination
  2. Dead Code elimination
  3. Constant folding
  4. Copy propagation
  5. Induction variable elimination
  6. Code motion
  7. Reduction in strength. ..... etc..

Code Generation:

- The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
- The intermediate instructions are each translated into sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.
- Using registers R1 and R2,the translation of the given example is:

  MOV id3 ,R2

  MUL #60.0 , R2

  MOV id2 , R1

  ADD R2 , R1

  MOV R1 , id1

Symbol-Table Management:

- An essential function of a compiler is to record the identifiers used in the source program and collect its information.
- A symbol table is a data structure containing a record for each identifier with fields for attributes.(such as, its type, its scope, if procedure names then the number and type of arguments etc.,)
- The data structure allows finding the record for each identifier and store or retrieving

data from that record quickly.

Error Handling and Reporting:

- Each phase can encounter errors. After detecting an error, a phase must deal that error, so that compilation can proceed, allowing further errors to be detected.
- Lexical phase can detect error where the characters remaining in the input do not form any token.
- The syntax and semantic phases handle large fraction of errors. The stream of tokens violates the syntax rules are determined by syntax phase.
- During semantic, the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. e.g. if we try to add two identifiers ,one is an array name and the other a procedure name.

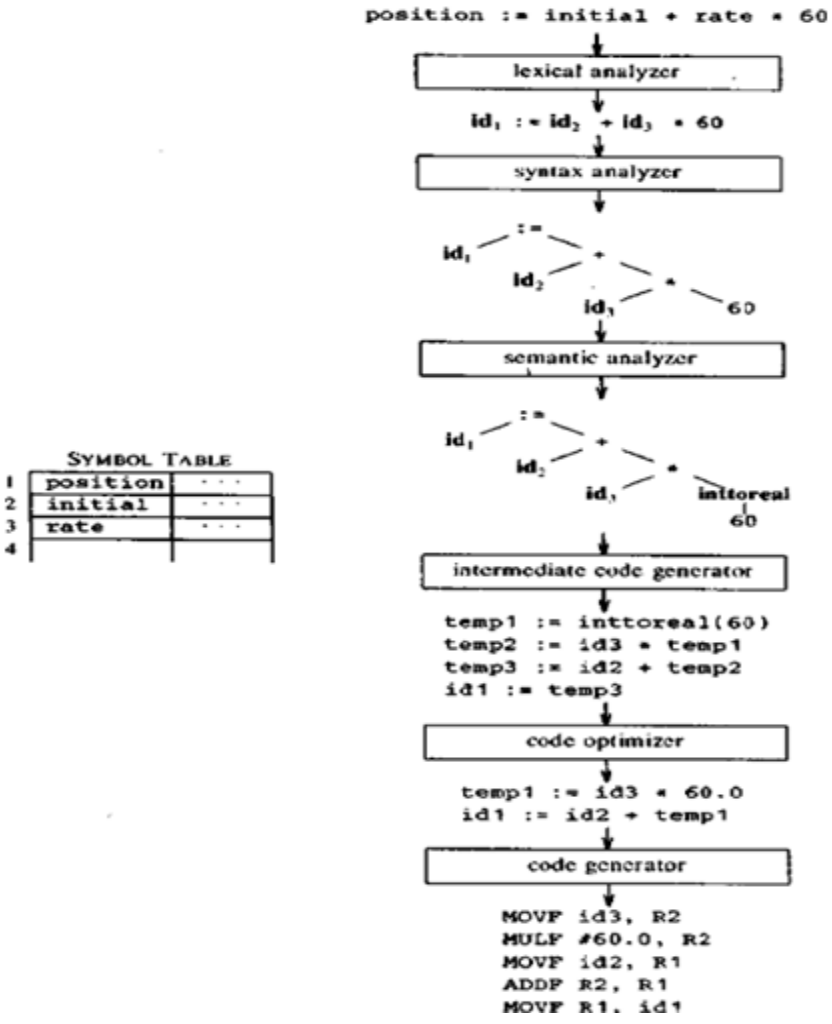


Fig.1.4. Translation of statement

FUNCTIONS AND ROLES OF LEXICAL PHASE:

- It is the first phase of a compiler.
- Its main task is to read input characters and produce tokens.
- "get next token "is a command sent from the parser to the lexical analyzer(LA).
- On receipt of the command, the LA scans the input until it determines the next token and returns it.
- It skips white spaces and comments while creating these tokens.
- If any error is present the LA will correlate that error with source file and the line number.

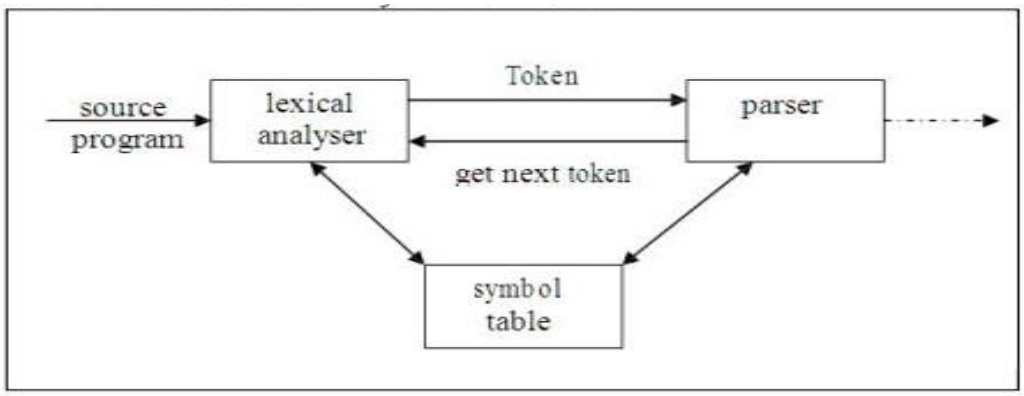


Fig.1.5.Interaction of lexical analyzer with parser

Issues in Lexical Analysis:

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

1. Simpler design is the most important consideration.
2. Compiler efficiency is improved. A large amount of time is spent reading the source program and partitioning into tokens. Buffering techniques are used for reading input characters and processing tokens that speed up the performance of the compiler.
3. Compiler portability is enhanced.

Tokens, Patterns, Lexemes:

- Token is a sequence of characters in the input that form a meaningful word. In most languages, the tokens fall into these categories: Keywords, Operators, Identifiers, Constants, Literal strings and Punctuation.

- There is a set of strings in the input for which a token is produced as output. This set is described a rule called pattern.
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Fig.1.6. Examples of tokens

Attributes of tokens:

- The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions and attributes influence the translation of tokens.
- Usually a token has a single attribute i.e. pointer to the symbol table entry in which the information about the token is kept.

Example: The tokens and associated attribute values for the statement given,

Lexeme	<token, token attribute>
E	<id, pointer to symbol table entry for E>
=	<assign-op,>
M	<id, pointer to symbol table entry for M>
*	<mult-op,>
C	<id, pointer to symbol table entry for C>
**	<exp-op,>
2	<number, integer value 2>
;	<separator, >

Lexical errors:

Few errors are discernible at the lexical level alone, because a LA has a much localized view of the source program. A LA may be unable to proceed because none of the patterns for tokens matches a prefix of the remaining input.

Error-recovery actions are:

1. Deleting an extraneous character.
2. Inserting a missing character.
3. Replacing an incorrect character by a correct character.
4. Transposing two adjacent characters.

INPUT BUFFERING

A two-buffer input scheme that is useful when look ahead on the input is necessary to identify tokens is discussed. Later, other techniques for speeding up the LA, such as the use of "sentinels" to mark the buffer end are also discussed.

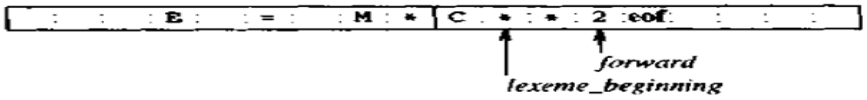
Buffer Pairs:

A large amount of time is consumed in scanning characters, specialized buffering techniques are developed to reduce the amount of overhead required to process an input character. A buffer divided into two N-character halves is shown in Fig. 1.7.Typically, N is the number of characters on one disk block, e.g., 1024 or 4096.

Fig. 1.7. An input buffer in two halves.

- **N, input** characters are read into each half of the buffer with one system read command, instead of invoking a read command for each input character.
- If fewer than **N** characters remain in the input, then a special character **eof** is read into buffer after the input characters.(**eof---end of file)**
- Two pointers, forward and lexeme_beginning are maintained. The string of characters between the two pointers is the current lexeme.
- If the forward pointer has moved halfway mark, then the right half is filled with **N** new input characters. If the forward pointer is about to move the right end of the buffer, then

the left half is filled with **N** new input characters and the wraps to the beginning of the buffer. Code for advancing forward pointer is shown in Fig.1.8.




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

Fig. 1.8. Code to advance forward pointer

Sentinels:

- With the previous algorithm, we need to check each time we move the forward pointer that we have not moved off one half of the buffer. If so, then we must reload the other half.
- This can be reduced, if we extend each buffer half to hold a sentinel character at the end.
- The new arrangement and code is shown in Fig. 1.9 and 1.10.This code performs only one test to see whether *forward* points to an **eof.**

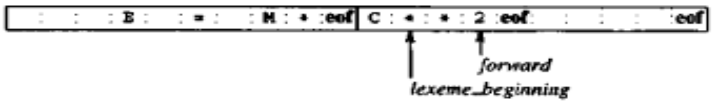


Fig.1.9. Sentinels at end of each buffer half.

```
forward := forward + 1;
if forward↑ = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward + 1
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else  /* eof within a buffer signifying end of input */
                terminate lexical analysis
end
```

Fig. 1.10. Lookahead code with sentinels**.**

REPRESENTATION OF TOKENS USING REGULAR EXPRESSION:

Regular expression is an important notation for specifying patterns. The term alphabet or character class denotes any finite set of symbols. Symbols are letters and characters. The set {0,1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.

- A s*tring* over some alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string *s* is written as **|s|,** is the number of occurrences of symbols in *s*.
- The *empty string*, *denoted* by ε, is a special string of length zero.
- The term *language* denotes any set of strings over some fixed alphabet.
- The *empty set* {ε}, the set containing only the empty string.
- If *x* and *y* are strings, then the concatenation of *x* and *y* ,written as *xy* ,is the string formed by appending y to x.

Some common terms associated with the parts of the string are shown in Fig. 1.11.

TERM	*DEFINITION*
prefix of s	A string obtained by removing zero or more trailing symbols of string s; ban is a prefix of banana.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; nana is a suffix of banana.
substring of s	A string obtained by deleting a prefix and a suffix from s; nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and e are prefixes, suffixes, and substrings of s.
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that s ≠ x.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; baaa is a subsequence of banana.

Fig. 1.11 Terms for parts of a string

Operations on Languages

There are several important operations that can be applied to languages. The various operations and their definition are shown in Fig.1.12.

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Fig. 1.12 Definitions of operations on languages.

Let L be theset {A,B,.....,Z,a,b,. .....,z} consisting upper and lowercase alphabets, and D the set {0,1,2,.....,9} consisting the set of ten decimal digits. Here are some examples of new languages created from L and D.

1. $L \cup D$ is the set of letters and digits.
2. LD is the set of strings consisting of a letter followed by a digit.
3. L^4 is the set of all four-letter strings.
4. L^* is the set of all strings of letters, including ϵ, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Regular Language

- A regular language over an alphabet □ is the one that can be obtained from the basic languages using the operations Union, Concatenation and Kleene *.
- A language is said to be a regular language if there exists a Deterministic Finite Automata (DFA) for that language.
- The language accepted by DFA is a regular language.
- A regular language can be converted into a regular expression by leaving out {} or by replacing {} with () and by replacing U by +.

LEX:

- Lex is a lexical analyzer tool mostly used with yacc parse generator.
- Tool for recognizing tokens in a program.
- Tokens are the terminals of a language,

  English:

  words, punctuation marks, …

  Programming language:

  Identifiers, operators, keywords, …
- Regular expressions define terminals/tokens.
- It lexically analyses (i.e. matches) the patterns (regular expressions) given as input string or as a file.

Steps involved in processing a LEX program:

Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.

Step 2: The C complier compile lex.yy.c file into an executable file called a.out.

Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

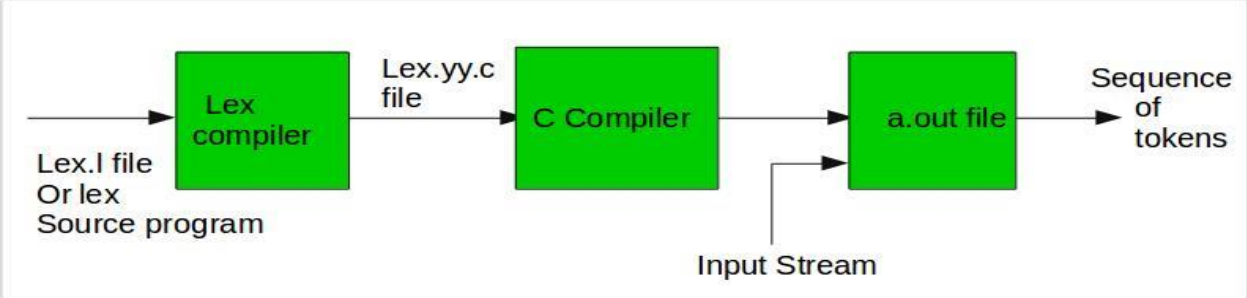


Fig. 1.13 Processing steps involved

```
definitions section
%%
rules section
%%
auxiliary function/user code section
```

Fig. 1.14 Structure of a LEX program

Definition Section:

- Generally used to declare functions, include header files, or define global variables and constants.
- Text is enclosed in **%{ … %}** brackets. Anything written in this bracket is copied directly to the file **lex.yy.c**

eg.:

```
%{
   #include<stdio.h>
   int global_variable;
 }%
```

Rules Section:

- Each rule has the form

pattern action

where:

- pattern describes a pattern to be matched on the input.
- action must begin on the same line.
- If action is empty, the matched token is discarded.

eg.:

```
 %%
{digit}+            {printf(" number");}
{letter}*           {printf(" name");
 %%
```

Auxiliary Functions:

LEX generates C code for the rules specified in the Rules section and places this code into a single function called ***yylex()***.

```
/* Declarations */
%%
/* Rules */
%%
int main()
```

```
{
yylex();
return 1;
}
```

This section contain C statements and additional functions.

Regular definitions, has of the form,

name definition

e.g.:

digit [0-9]

letter [a-zA-Z]

Regular Expressions

Regular expression is a formula that describes a possible set of string.

Component of regular expression:

X	**the character x**
.	**any character, usually accept a new line**
[xyz]	**any of the characters x, y, z,…..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences……**
R_1R_2	**an R_1 followed by anR_2**
R_1 | R_2	**either an R_1 or an R_2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as a language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the Regular Expression (RE) over alphabet.

Definition of RE:

1. ε is a regular expression, L(ε) = {ε}
2. If ***a*** is a symbol in ∑ then ***a*** is a regular expression, L(a) = {a}
3. Suppose r and s are regular expressions denoting the language L(r) and L(s)
    a. (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
    b. (r)(s) is a regular expression denoting the language L(r)L(s)
    c. (r)* is a regular expression denoting (L(r))*
    d. (r) is a regular expression denoting L(r)

Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1. The unary operator * has the highest precedence and is left associative.
2. Concatenation has the second highest precedence and is left associative.
3. **|,** the alternate operator has the lowest precedence and is left associative.

PROPERTIES OF REGULAR EXPRESSION:

There are a number of algebraic laws obeyed by regular expressions and these can be used to manipulate regular expressions into equivalent forms. Algebraic properties are shown in Fig. 1.13.

Axiom	Description
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = rs | rt (s | t) r = sr | tr	concatenation distributes over | (| = alternation)
rϵ = r ϵr = r	ϵ is the identity for concatenation
(r | ϵ)* = r*	
r** = r*	* is idempotent

Fig. 1.15 Algebraic laws of regular expression

Regular Definitions

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|

Digits - 0 | 1 | 2 | …. | 9

id - letter (letter / digit)*

Transition Diagrams for Relational operators and identifier is shown below.

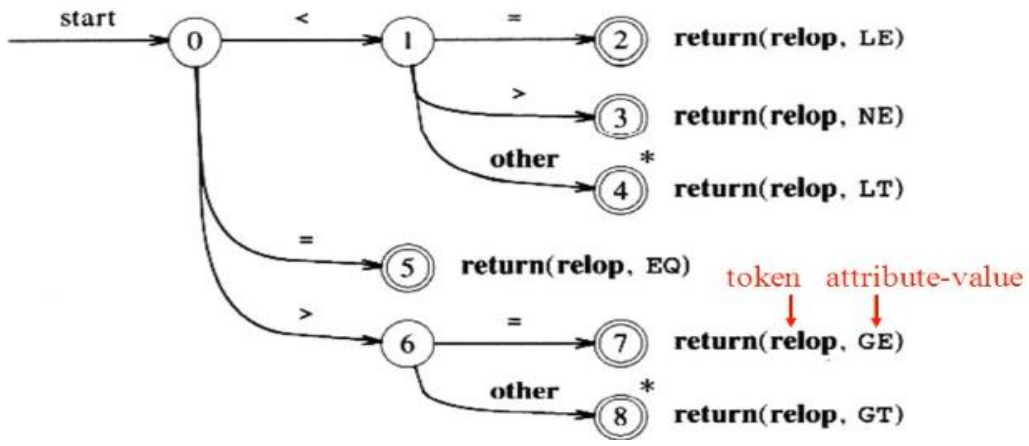


Fig. 1.16 Transition Diagrams for Relational operators

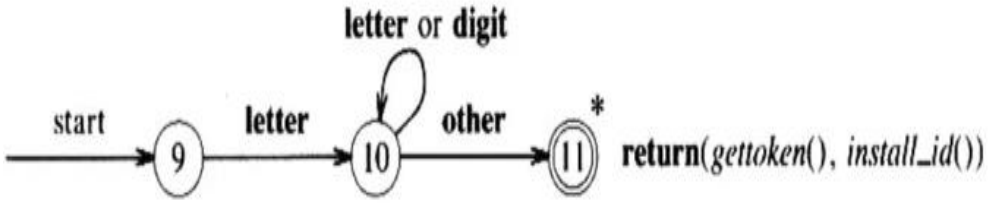


Fig. 1.17 Transition Diagrams for identifier

FINITE AUTOMATA:

A recognizer for a language is a program that takes as input a string x and answer "yes" if x is a sentence of the language and "no" otherwise. The regular expression is compiled into a recognizer by constructing a generalized transition diagram called a finite automaton. A finite automaton can be deterministic or non deterministic, where "nondeterministic" means more than one transition out of a state may be possible on the same input symbol.

Deterministic Finite Automata (NFA)

A Finite Automata (FA) or Finite State Machine (FSM) is a 5- tuple $(Q,\sum,q0,A,\delta)$ where,

- Q is the set of finite states
- $\sum$ is the set of input symbols (Finite alphabet)
- q0 is the initial state
- A is the set of all accepting states or final states.
- δ is the transition function, $Q\times\sum\rightarrow Q$

For any element q of Q and any symbol a in $\sum$, we interpret $\delta(q,a)$ as the state to which the Finite Automata moves, if it is in state q and receives the input 'a'.

How to draw a DFA?

1. Start scanning the string from its left end.
2. Go over the string one symbol at a time.
3. To be ready with the answer as soon as the string is entirely scanned.

Non-Deterministic Finite Automata (NFA) Definition:

A NFA is defined as a 5-tuple, $M=(Q, \sum,q0,A, \delta)$ where,

- Q is the set of finite states
- $\sum$ is the set of input symbols (Finite alphabet)
- q0 is the initial state
- A is the set of all accepting states or final states.
- δ is the transition function, $Q\times\sum\rightarrow 2^Q$

DFA	NFA
DFA can be understood as one machine.	Multiple small machines computing at the same time.
Difficult to construct - Complex structure.	Easier to construct.
Rejects the string if not terminated at the accepting state.	Rejects only after multiple checks.
Less execution time on input string.	More execution time.
All DFA are derived from NFA.	Not all NFA are DFA.
DFA requires more space.	Requires less Space.
The next possible state is clearly set.	Ambiguity occurs.

Fig. 1.18 DFA vs. NFA

Regular expression to NFA (Thompson's construction method):

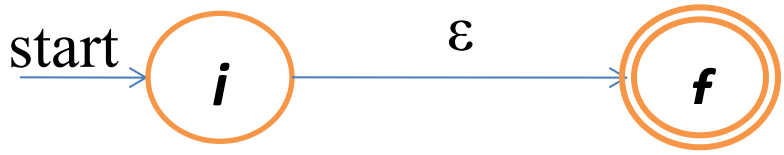
For each kind of RE, define an NFA.

Input: A Regular Expression R over an alphabet ∑

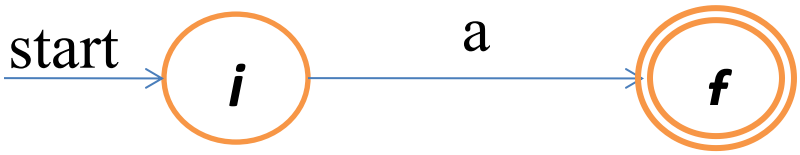
Output: An NFA N accepting L(R)

Method:

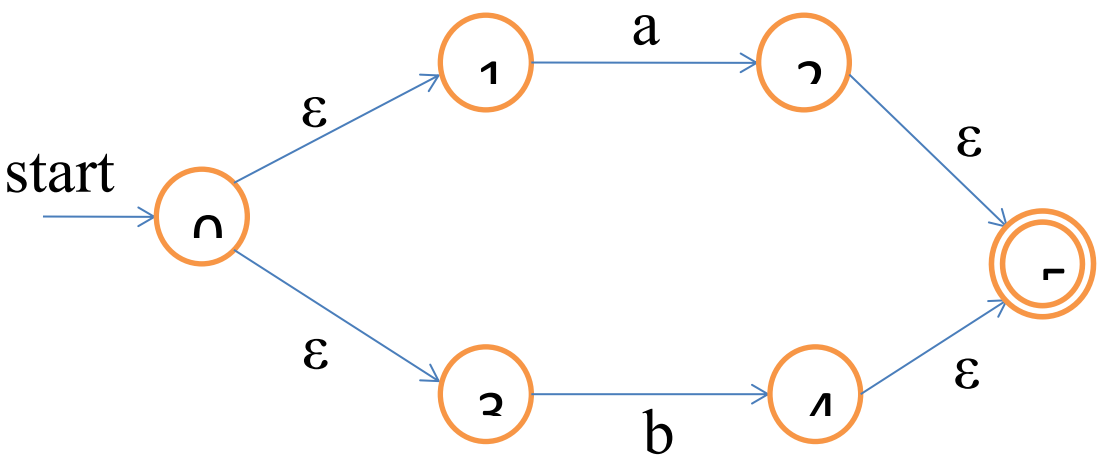
1. R = ε



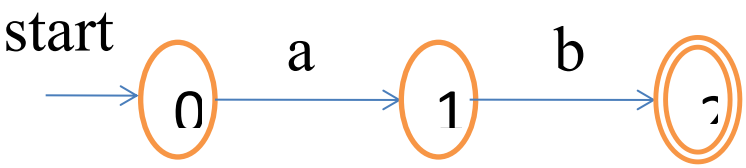
2. R = a



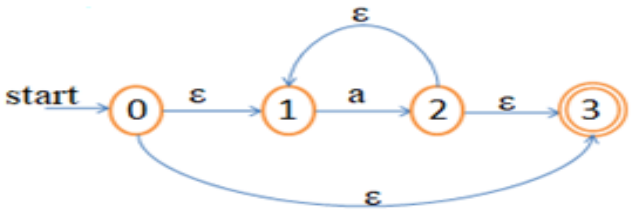
3. R= a | b



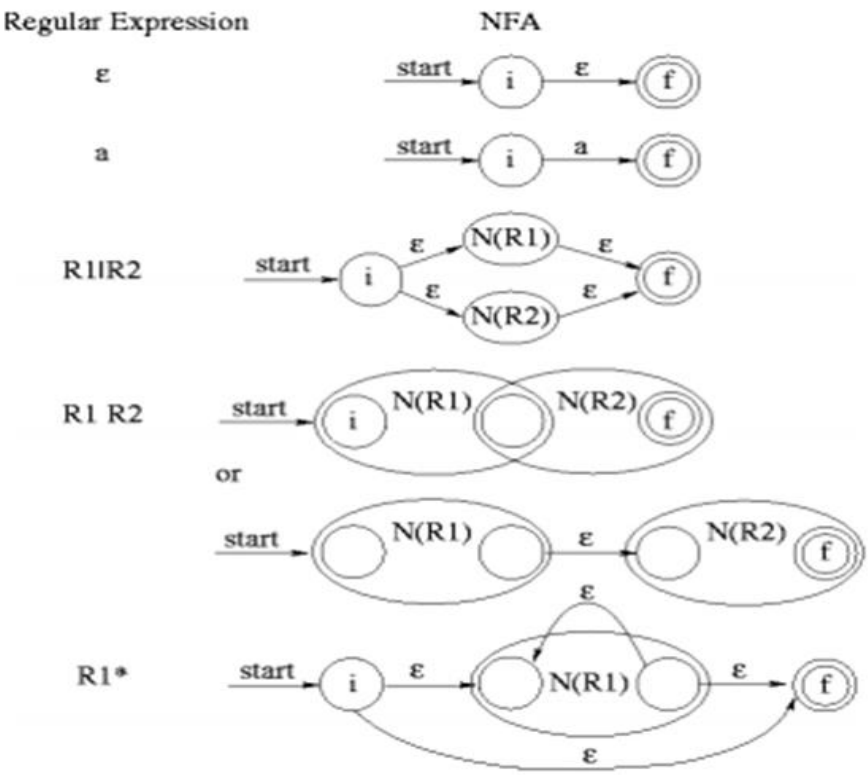
4. R=ab



5. R= a*

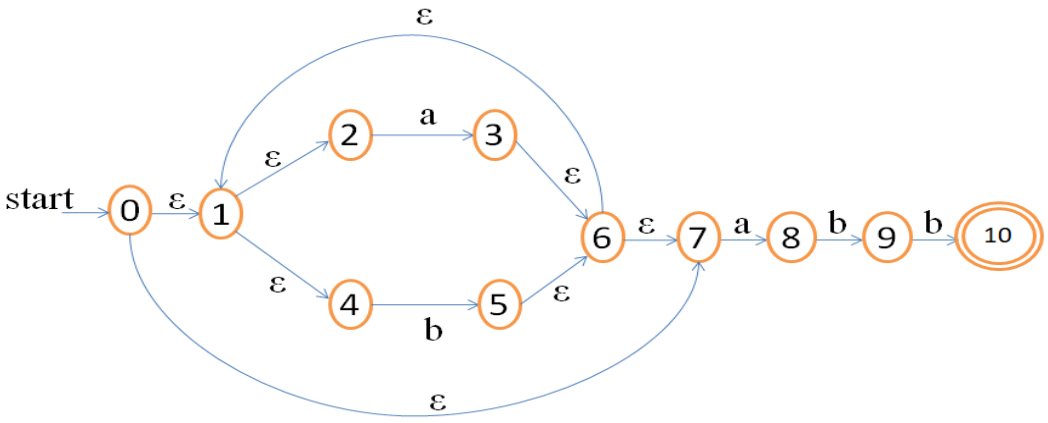


In general the rules if Thompson's construction method is given below:



Problems:

1. Construct NFA for: $(a+b)^*abb$



REGULAR EXPRESSION TO FINITE AUTOMATA – NFA TO MINIMIZED DFA

Converting NFA to DFA: The Subset Construction Algorithm

The algorithm for constructing a DFA from a given NFA such that it recognizes the same language is called subset construction. The reason is that each state of the DFA machine corresponds to a set of states of the NFA. The DFA keeps in a particular state all possible states to which the NFA makes a transition on the given input symbol. In other words, after processing

a sequence of input symbols the DFA is in a state that actually corresponds to a set of states from the NFA reachable from the starting symbol on the same inputs.

There are three operations that can be applied on NFA states:

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

The staring state of the automaton is assumed to be s0. The ε-*closure*(*s*) operation computes exactly all the states reachable from a particular state on seeing an input symbol. When such operations are defined the states to which our automaton can make a transition from set *T* on input *a* can be simply specified as: ε-*closure*(*move*(*T*, *a*))

Subset Construction Algorithm:

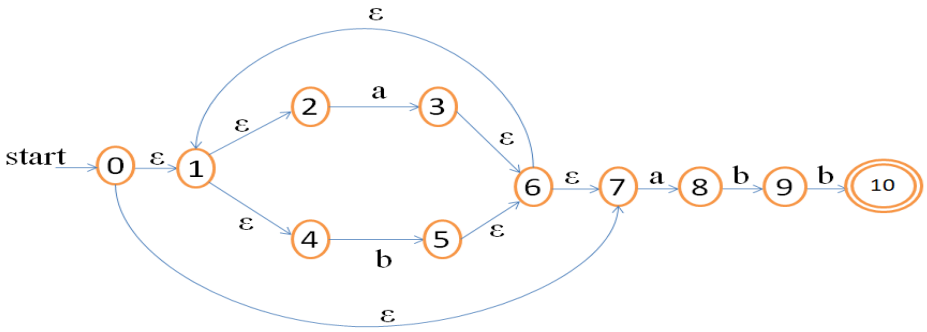
```
initially, ε-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
      mark T;
      for each input symbol a do begin
            U := ε-closure(move(T, a));
            if U is not in Dstates then
                  add U as an unmarked state to Dstates;
            Dtran[T, a] := U
      end
end
```

Algorithm for Computation of ε-*closure*:

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ε do
              if u is not in ε-closure(T) do begin
                    add u to ε-closure(T);
                    push u onto stack
              end
end
```

Example: Convert the NFA for the expression: (a|b)*abb into a DFA using the subset construction algorithm.

Step 1: Convert the above expression in to NFA using Thompson rule constructions.



Step 2: Start state of equivalent DFA is **ε-closure(0)**

ε-closure(0) ={ 0,1,2,4,7}

Step 2.1: Compute **ε-closure(move(A,a))**

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

ε-closure(move(A,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(A,a)) ={1,2,3,4,6,7,8}

Dtran[A,a]=B

Step 2.2: Compute **ε-closure(move(A,b))**

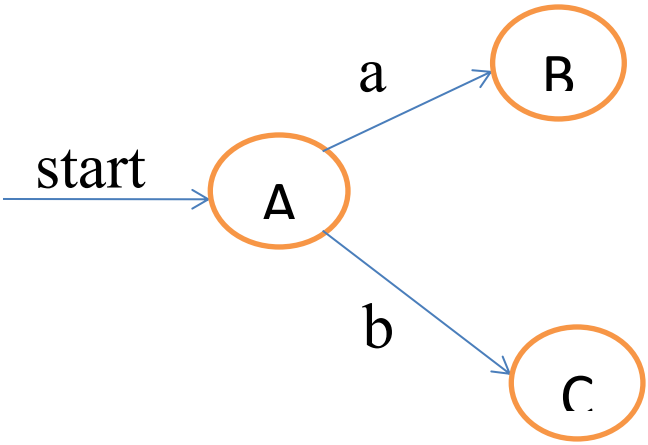
move(A,b) ={5}

ε-closure(move(A,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(A,a)) ={1,2,4,5,6,7}

Dtran[A,b]=C

DFA and Transition table after step 2 is shown below.



DFA states	Input Symbols	
	a	b
A	B	C
B		
C		

Step 3: Compute Transition from **state B** on input symbol {a,b}

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

Step 3.1: Compute **ε-closure(move(B,a))**

move(B,a) ={3,8}

ε-closure(move(B,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(B,a)) ={1,2,3,4,6,7,8}

Dtran[B,a]=B

Step 3.2: Compute **ε-closure(move(B,b))**

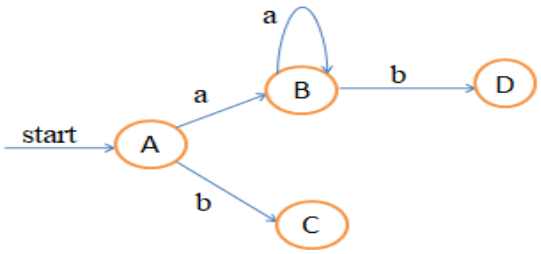
move(B,b) ={5,9}

ε-closure(move(B,b)) = ε-closure(5,9) = {5,9,6,7,1,2,4}

ε-closure(move(B,b)) ={1,2,4,5,6,7,9}

Dtran[B,b]=D

DFA and Transition table after step 3 is shown below.



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C		
D		

Step 4: Compute Transition from **state C** on input symbol {a,b}

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

Step 4.1: Compute **ε-closure(move(C,a))**

move(C,a) ={3,8}

ε-closure(move(C,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(C,a)) ={1,2,3,4,6,7,8}

Dtran[C,a]= B

Step 4.2: Compute **ε-closure(move(C,b))**

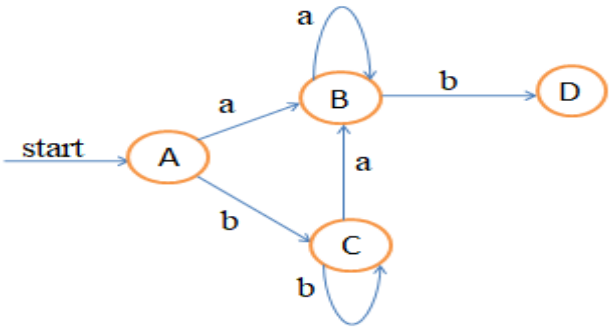
move(C,b) ={5}

ε-closure(move(C,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(C,b)) ={1,2,4,5,6,7}

Dtran[C,b]= C

DFA and Transition table after step 4 is shown below.



Transition Table **Dtran**

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D		

Step 5: Compute Transition from **state D** on input symbol {a,b}

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

Step 5.1: Compute **ε-closure(move(D,a))**

move(D,a) ={3,8}

ε-closure(move(D,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(D,a)) ={1,2,3,4,6,7,8}

Dtran[D,a]= B

Step 5.2: Compute **ε-closure(move(D,b))**

move(D,b) ={5,10}

ε-closure(move(D,b)) = ε-closure(5,10) = {5,10,6,7,1,2,4}

ε-closure(move(C,b)) ={1,2,4,5,6,7,10}

Dtran[D,b]= E

Step 6: Compute Transition from **state E** on input symbol {a,b}

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

Step 6.1: Compute **ε-closure(move(E,a))**

move(E,a) ={3,8}

ε-closure(move(E,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

ε-closure(move(E,a)) ={1,2,3,4,6,7,8}

Dtran[E,a]= B

Step 6.2: Compute **ε-closure(move(E,b))**

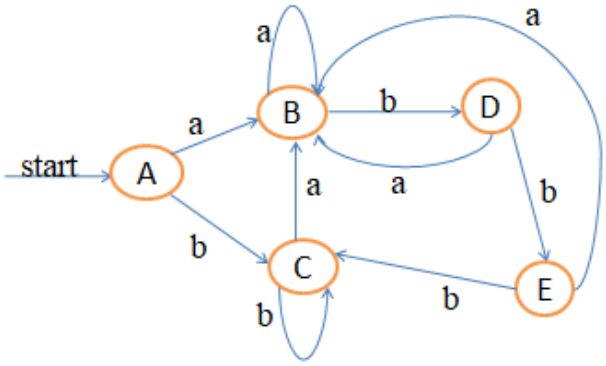
move(E,b) ={5}

ε-closure(move(E,b)) = ε-closure(5) = {5,6,7,1,2,4}

ε-closure(move(E,b)) ={1,2,4,5,6,7}

Dtran[E,b]= C

DFA and Transition table after step 6 is shown below.

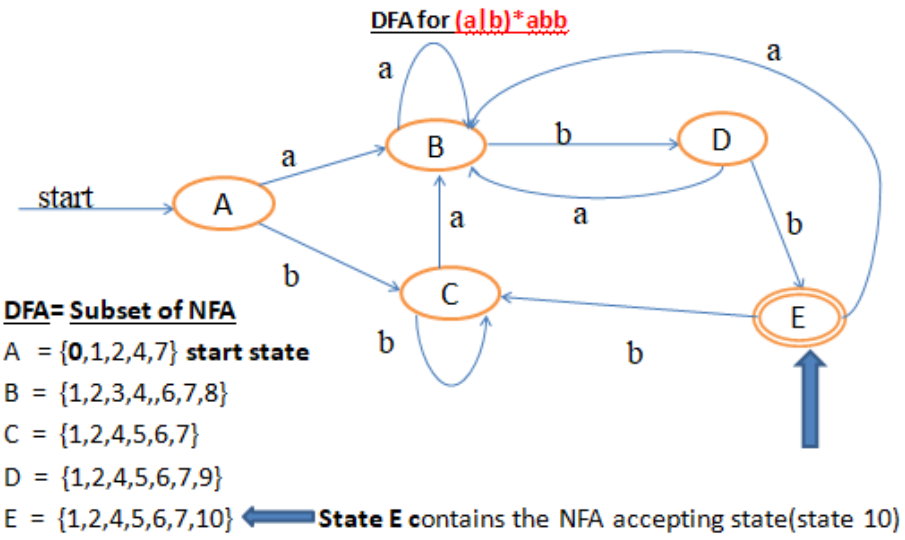


DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Step 7: No more new DFA states are formed.

Stop the subset construction method.

The start state and accepting states are marked the DFA.



Minimized DFA:

Convert the above DFA in to minimized DFA by applying the following algorithm.

Minimized DFA algorithm:

Input: DFA with 's' no of states

Output: Minimized DFA with reduced no of states.

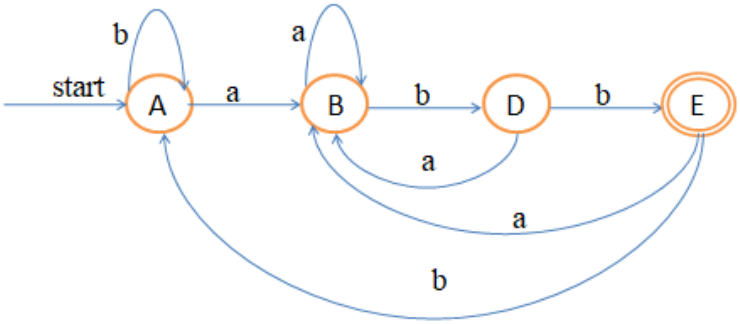
Steps:

1. Partition the set of states in to two groups. They are set of accepting states and non accepting states.
2. For each group G of π do the following steps until π=π new .
3. Divide G in to as many groups as possible, such that two states s and t are in the same group only when for all states s and t have transitions for all input symbols 's' are in the same group itself. Place newly formed group in π new.
4. Choose representative state for each group.
5. Remove any dead state from the group.

After applying minimized DFA algorithm for the regular expression (a|b)*abb , the transition table for the minimized DFA becomes Transition table for Minimized state DFA :

DFA states	Input Symbols	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Minimized DFA:



Exercises:

Convert the following regular expression in to minimized state DFA,

1. (a|b)*

2.(b|a)*abb(b|a)*

3.((a|c)*)ac(ba)*

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II - Compiler Design – SCSA1604

II. PARSER

Role of Parser-Context-free Grammar – Derivations and Parse Tree - Types of Parser – Bottom Up: Shift Reduce Parsing - Operator Precedence Parsing, SLR parser- Top Down: Recursive Decent Parser - Non-Recursive Decent Parser-Error handling and Recovery in Syntax Analyzer-YACC.

SYNTAX ANALYSIS:

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant advantages to both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand. Syntactic specification of a programming language.
- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.
- A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors. Tools are available for converting grammar-based descriptions of translations into working pro-grams.

Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

ROLE OF THE PARSER:

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.

The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

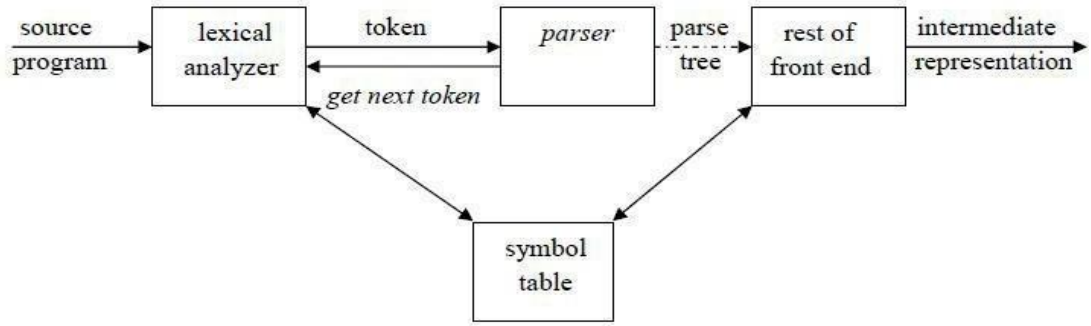


Fig 2.1 Role of Parser

CONTEXT FREE GRAMMARS

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

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
3. In a grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
4. The productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings. Each production consists of a non terminal, followed by an arrow, followed by a string of non terminals and terminals.

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S). Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic- variables).P is a finite set of productions rules in the following form A → α where

A is a non- terminal and α is a string of terminals and non-terminals (including the empty string).S is a start symbol (one of the non-terminal symbol).
L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff S ω where ω is a string of terminals of G. If G is a context-free grammar (G) is a context-free language. Two grammars G_1 and G_2 are equivalent, if they produce same grammar.
Consider the production of the form S α, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.
Example: Consider the grammar for simple arithmetic expressions:

expr → expr op expr
expr → (expr)
expr → - expr
expr → **id**
op → +
op → -
op → *
op → /
op → ^

Terminals : id + - * / ^ ( )
Non-terminals : expr , op
Start symbol : expr

Notational Conventions:

1. These symbols are terminals:
   i. Lower-case letters early in the alphabet such as a, b, c.
   ii. Operator symbols such as +, -, etc.
   iii. Punctuation symbols such as parentheses, comma etc.
   iv. Digits 0,1,…,9.
   v. Boldface strings such as id or if (keywords)
2. These symbols are non-terminals:
   i. Upper-case letters early in the alphabet such as A, B, C..
   ii. The letter *S*, when it appears is usually the start symbol.

iii. Lower-case italic names such as *expr or stmt*.

3. Upper-case letters late in the alphabet, such as X,Y,Z, represent grammar symbols, that is either terminals or non-terminals.
4. Greek letters α , β , γ represent strings of grammar symbols.
   e.g a generic production could be written as A → α.
5. If A → α_1 , A → α_2 , ... , A → α_n are all productions with A , then we can write A → α_1 | α_2 |. . . . | α_n , (alternatives for A).
6. Unless otherwise stated, the left side of the first production is the start symbol.

Using the shorthand, the grammar can be written as:

E → E A E | (E) | - E | **id**

A → + | - | * | / | ^

Derivations:

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

$\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in ≥ one step

$\overset{*}{\underset{G}{\Rightarrow}}$ indicates that the derivation utilizes the rules of grammar G

To create a string from a context-free grammar:

- Begin the string with a start symbol.
- Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

In general a derivation step is αAβ αγβ is sentential form and if there is a production rule A→γ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols α1 α2... αn (αn derives from α1 or α1 derives αn). There are two types of derivation:

1. Leftmost Derivation (LMD):

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

- The sentential form derived by the left-most derivation is called the left-sentential form.

2. Rightmost Derivation (RMD):

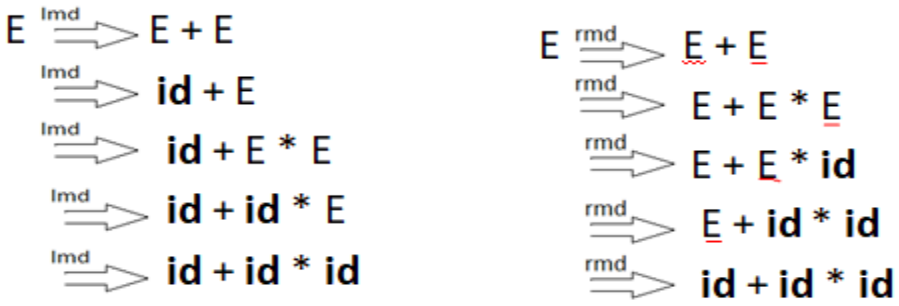
- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
- The sentential form derived from the right-most derivation is called the right-sentential form.

Example:

Consider the G,

E → E + E | E * E | (E) | - E | **id**

Derive the string **id + id * id** using leftmost derivation and rightmost derivation.



(a) **(b)**

Fig 2.2 a) Leftmost derivation b) Rightmost derivation

Strings that appear in leftmost derivation are called left sentential forms. Strings that appear in rightmost derivation are called right sentential forms.

Sentential Forms:

Given a grammar G with start symbol S, if S => α , where α may contain non-terminals or terminals, then α is called the sentential form of G.

Parse Tree:

A parse tree is a graphical representation of a derivation sequence of a sentential form.

In a parse tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

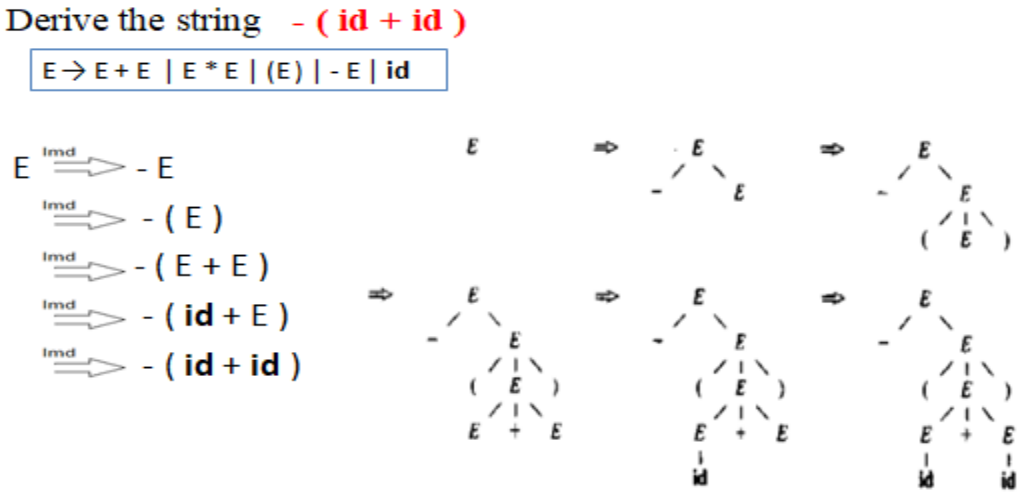


Fig 2.3 Build the parse tree for string **–(id+id)** from the derivation

Yield or frontier of tree:

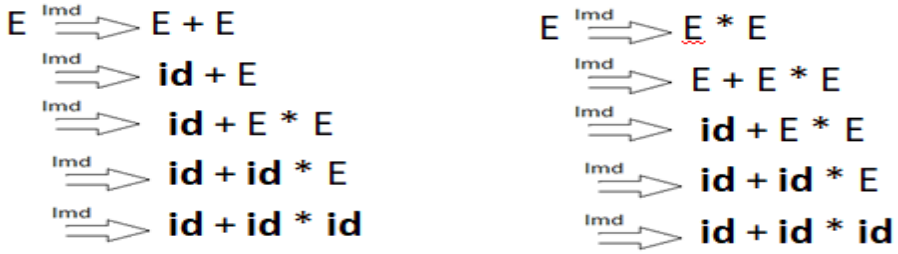
Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentential forms that are read from left to right. The sentential form in the parse tree is called yield or frontier of the tree.

Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar. i.e. An ambiguous grammar is one that produce more than one leftmost or more than one rightmost derivation for the same sentence.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:



The two corresponding parse trees are:

Fig 2.4 Two Parse tree for id+id*id

Consider another example,

stmt → if expr then stmt | if expr then stmt else stmt | other

This grammar is ambiguous since the string if E1 then if E2 then S1 else S2 has the following Two parse trees for leftmost derivation :

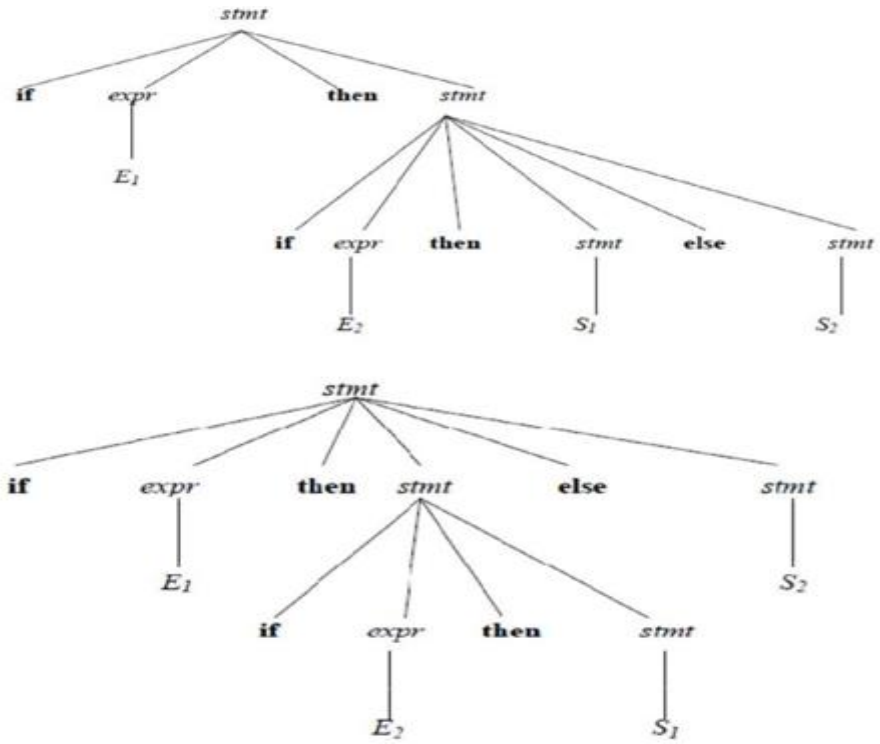


Fig 2.5 Two Parse tree for **if E1 then if E2 then S1 else S2**

Eliminating Ambiguity:

An ambiguous grammar can be rewritten to eliminate the ambiguity. e.g. Eliminate the ambiguity from **"dangling-else"** grammar,

stmt → **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| **other**

Match each else with the closest previous unmatched then. This disambiguity rule can be incorporated into the grammar.

stmt → matched_stmt | unmatched_stmt

matched_stmt →if expr then matched_stmt else matched_stmt

| other

unmatched_stmt → if expr then stmt

| if expr then matched_stmt else unmatched_stmt

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

Table 2.1 Ambiguous grammar vs. Unambiguous grammar

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.

Removing Ambiguity by Precedence & Associativity Rules:

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules:

Rule-1:

- The level at which the production is present defines the priority of the operator contained in it.
  - The higher the level of the production, the lower the priority of operator.
  - The lower the level of the production, the higher the priority of operator.

Rule-2:

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

Example: Consider the ambiguous Grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \textbf{id}$$

Introduce new variable / non-terminals at each level of precedence,

- an expression **E** for our example is a sum of one or more terms. (+,-)
- a term **T** is a product of one or more factors. (*, /)
- a factor **F** is an identifier or parenthesised expression.

The resultant unambiguous grammar is:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

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

Trying to derive the string **id+id*id** using the above grammar will yield one unique derivation.

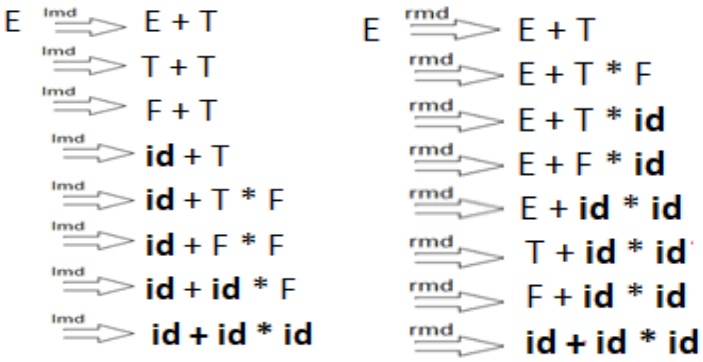


Fig 2.6 Distinct Leftmost and Rightmost derivation

Regular Expression vs. Context Free Grammar:

- Every construct that can be described by a regular expression can be described by a grammar.
- NFA can be converted to a grammar that generates the same language as recognized by the NFA.
- Rules:
  - For each state *i* of the NFA, create a non-terminal symbol A_i .
  - If state *i* has a transition to state *j* on symbol ***a***, introduce the production $A_i \rightarrow$ a A_j
  - If state *i* goes to state *j* on symbol ε, introduce the production $A_i \rightarrow A_j$
  - If *i* is an accepting state, introduce $A_i \rightarrow \varepsilon$
  - If *i* is the start state make A_i the start symbol of the grammar.

Example: The regular expression (a|b)*abb, consider the NFA

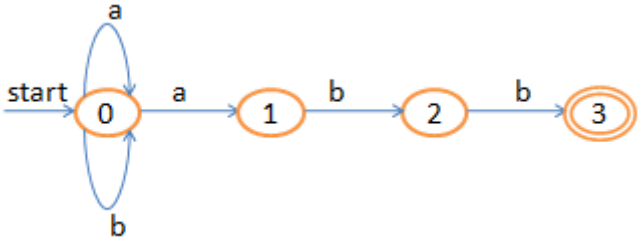


Fig 2.7 NFA for (a|b)*abb

Equivalent grammar is given by:

$A_0 \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1$

$A_1 \rightarrow b\ A_2$

$A_2 \rightarrow b\ A_3$

$A_3 \rightarrow \varepsilon$

Types of Parser:

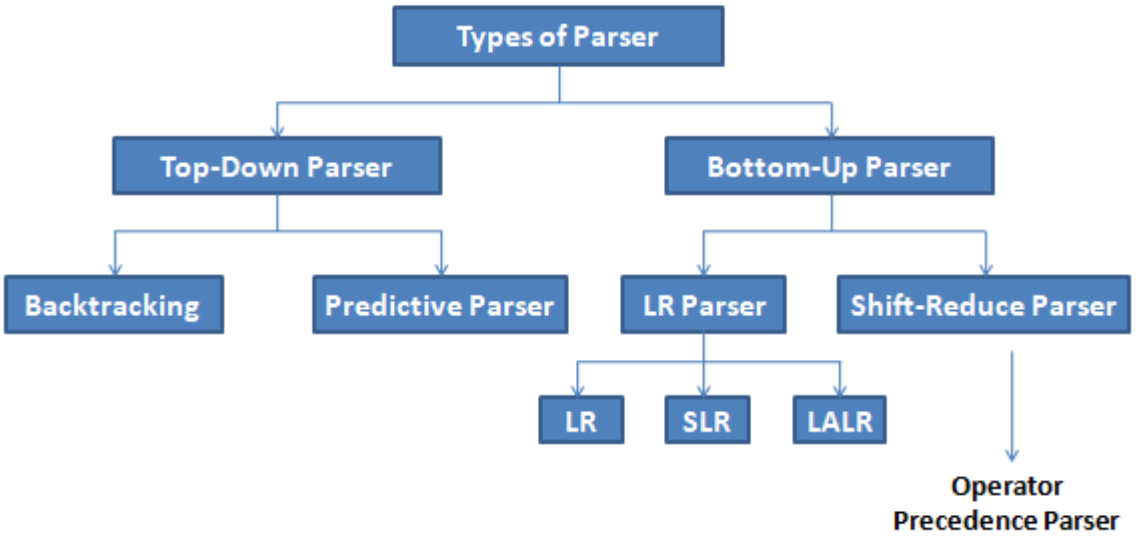


Fig 2.8 Types of Parser

LR Parsing:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

Why LR parsing:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

- The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

Bottom-Up Parsing:

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

Shift-Reduce Parsing:

Shift-reduce parsing is a type of bottom -up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The string to be recognized is abbcde. We want to reduce the string to S.

Steps of reduction:

abbcde (b,d can be reduced)

aAbcde (leftmost b is reduced)

aAde (now Abc,b,d qualified for reduction)

aABe (d can be reduced)

S

Each replacement of the right side of a production by the left side in the above example is called reduction, which is equivalent to rightmost derivation in reverse.

$$S \overset{rmd}{\Rightarrow} aABe \overset{rmd}{\Rightarrow} aAde \overset{rmd}{\Rightarrow} aAbcde \overset{rmd}{\Rightarrow} abbcde$$

Handle:

A substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a handle.

Stack Implementation of Shift-Reduce Parsing:

There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. We use **$** to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string ***w*** is on the input, as follows:

STACK	INPUT
$	w$

The parser operates by shifting zero or more input symbols onto the stack until a handle is on top of the stack. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Example: The actions a shift-reduce parser in parsing the input string $id_1+id_2*id_3$, according to the ambiguous grammar for arithmetic expression.

	STACK	INPUT	ACTION
(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 2.9 Configuration of Shift Reduce Parser on input $id_1+id_2*id_3$

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Fig 2.10 Reductions made by Shift Reduce Parser

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

(1) shift, (2) reduce,(3) accept, and (4) error.

- In a ***shift*** action, the next input symbol is shifted unto the top of the stack.
- In a ***reduce*** action, the parser knows the right end of the handle is at 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.
- In an ***accept*** action, the parser announces successful completion of parsing.
- In an ***error*** action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Figure 2.11 represents the stack implementation of shift reduce parser using unambiguous grammar.

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

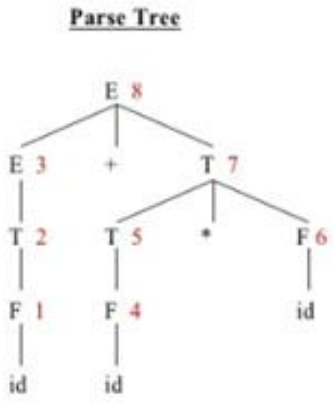


Fig 2.11 A stack implementation of a Shift-Reduce parser

Operator Precedence Parsing:

Operator grammars have the property that no production right side is ε (empty) or has two

adjacent non terminals. This property enables the implementation of efficient operator-precedence parsers.

Example: The following grammar for expressions:

E→E A E | (E) | -E | id

A→ + | - | * | / | ^

This is not an operator grammar, because the right side EAE has two consecutive non-terminals. However, if we substitute for A each of its alternate, we obtain the following operator grammar:

E→E + E |E – E |E * E | E / E | (E) | E ^ E | - E | id

In operator-precedence parsing, we define three disjoint precedence relations between pair of terminals. This parser relies on the following three precedence relations.

Relation	Meaning
$a <\cdot b$	*a* yields precedence to *b*
$a \doteq b$	*a* has the same precedence as *b*
$a \cdot> b$	*a* takes precedence over *b*

Fig 2.12 Precedence Relations

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

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

Fig 2.13 Operator Precedence Relation Table

Example: The input string: $id_1 + id_2 * id_3$

After inserting precedence relations the string becomes:

$\$ <\cdot\ id_1 \cdot> + <\cdot\ id_2 \cdot> * <\cdot\ id_3 \cdot> \$$

Having precedence relations allows identifying handles as follows:

1. Scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any =’s until a <· is encountered.
3. Everything between the two relations <· and ·> forms the handle.

Stack	Rule	Input	Comments
$ < id > + < id > * < id > $	E → id	$ id + id * id $	Here the first "id" is looked as the handle and since we were able to reduce, we reduce it in the input
$ < + < id > * < id > $	E → id	$ E + id * id $	The second handle is also "id" since that is available between a pair of lesser than and greater than precedences
$ < + < * < id > $	E → id	$ E + E * id $	The third handle is also "id".
$ < + < * > $	E → E*E	$ E + E * E $	The fourth handle is E *E, and is popped in the stack and we push the greater than symbol.
$ < + > $	E → E+E	$ E + E $	The last handle is E+E and that is also reduced.
$$			The stack is empty and has only the $ symbol, we say the string is accepted.

Defining Precedence Relations:

The precedence relations are defined using the following rules:

Rule-01:

- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

1. ↑ is of highest precedence and right-associative,
2. * and / are of next highest precedence and left-associative, and
3. + and − are of lowest precedence and left-associative,

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig 2.14 Operator Precedence Relation Table

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Fig 2.15 Stack Implementation

Implementation of Operator-Precedence Parser:

- An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
- More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Steps involved in Parsing:

1. Ensure the grammar satisfies the pre-requisite.
2. Computation of the function LEADING()
3. Computation of the function TRAILING()
4. Using the computed leading and trailing ,construct the operator Precedence Table
5. Parse the given input string based on the algorithm
6. Compute Precedence Function and graph.

Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.
- LEADING(A)={ a| $A =>^{+} \gamma a \delta$ },where γ is ε or any non-terminal, $=>^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for LEADING(A):

{

1. ‘a’ is in LEADING(A) is A→ γaδ where γ is ε or any non-terminal.

2.If ‘a’ is in LEADING(B) and A→B, then ‘a’ is in LEADING(A).

}

Computation of TRAILING:

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- TRAILING(A)={ a| A=>$^{+}$ γaδ },where δ is ε or any non-terminal, =>$^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for TRAILING(A):

{

1. ‘a’ is in TRAILING(A) is A→ γaδ where δ is ε or any non-terminal.

2.If ‘a’ is in TRAILING(B) and A→B, then ‘a’ is in TRAILING(A).

}

Example 1: Consider the unambiguous grammar,

E→E + T

E→T

T→T * F

T→F

F→(E)

F→**id**

Step 1: Compute LEADING and TRAILING:

LEADING(E)= { +,LEADING(T)} ={+ , * , (, id}

LEADING(T)= { *,LEADING(F)} ={* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= { +, TRAILING(T)} ={+ , * ,) , id}

TRAILING(T)= { *, TRAILING(F)} ={* ,) , id}

TRAILING(F)= {) , id}

Step 2: After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the ‘$’ symbol.

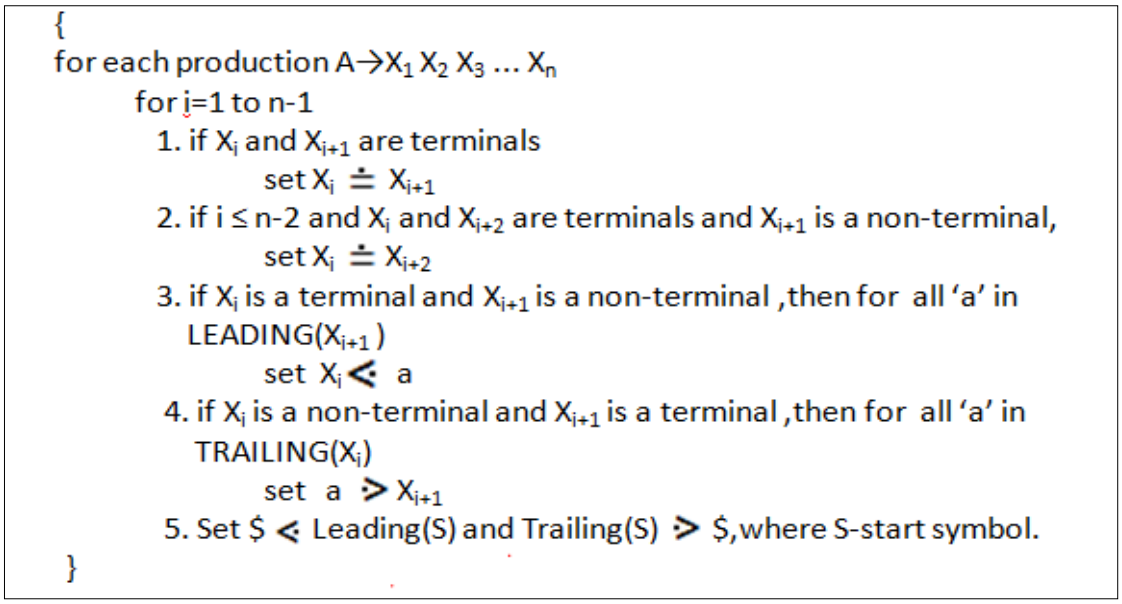


Fig 2.16 Algorithm for constructing Precedence Relation Table

	+	*	id	(	)	$
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
$	<	<	<	<	e	Accept

Fig 2.17 Precedence Relation Table * All undefined entries are error (e).

Rough work:

LEADING(E)= {+ , * , (, id}

LEADING(T)= {* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= {+ , * ,) , id}

TRAILING(T)= {* ,) , id}

TRAILING(F)= {) , id}

Terminal followed by NonTerminal

Rule-1.

+ T => + < leading(T)

Rule-3.

* F => * < leading(F)

Rule-4.

(E => (< leading(E)

Non-Terminal followed by Terminal

Rule-1. E + => Trailing(E) > +

Rule-3. T * => Trailing(T) > *

Rule-4. E) => Trailing(E) >)

Step 3: Parse the given input string **(id+id)*id$**

```
Set ip to point to the first symbol of w$
Repeat forever
    if $ is on the top of the stack and ip points to $ then return
    else begin
        Let a be the top terminal on the stack, and b the symbol pointed to by ip
            if a <· b or a =· b then
                push b onto the stack
                advance ip to the next input symbol
            end
            else if a ·> b then
                repeat
                    pop the stack
                until the top stack terminal is related by <·
                    to the terminal most recently popped
            else error()
    end
```

Fig 2.18 Parsing Algorithm

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(< +	+id)*id$	Shift +

$(+	+ < id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* < id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Fig 2.19 Parse the input string (id+id)*id$

Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. f (a) < g(b) whenever a<·b.
2. f (a) = g(b) whenever a = b. and
3. f(a) > g(b) whenever a ·> b.

Algorithm for Constructing Precedence Functions:

1. Create functions f_a for each grammar terminal *a* and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a = b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

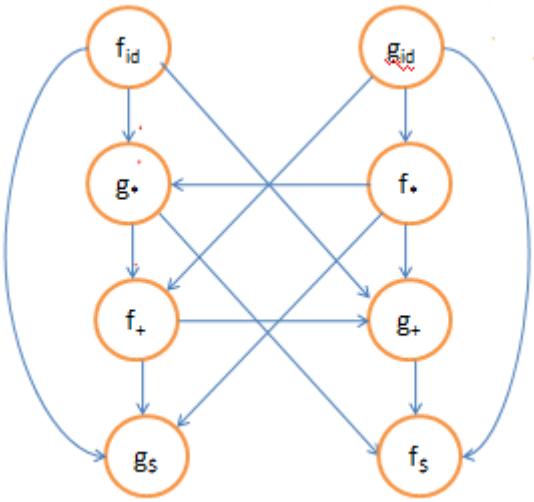


Fig 2.20 Precedence Graph

There are no cycles,so precedence function exist. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1,so g(+)=1.There is a path from g_{id} to f* to g* to f+ to g+ to f$,so g(id)=5.The resulting precedence functions are:

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

Example 2:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R

S→R

L→*R

L→id

R→L

Solution:

1. **Computation of LEADING:**

LEADING(S) = {=, * , id}

LEADING(L) = {* , id}

LEADING(R) = {* , id}

2. **Computation of TRAILING:**

TRAILING(S) = {= , * , id}

TRAILING(L)= {* , id}

TRAILING(R)= {* , id}

3.Precedence Table:

	=	*****	**id**	**$**
=	e	<·	<·	·>
*****	·>	<·	<·	·>
id	·>	e	e	·>
$	<·	<·	<·	accept

* All undefined entries are error (e).

4.Parsing the given input string:

1. ***id = id**

STACK	**INPUT STRING**	**ACTION**
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

2. **id*id=id**

STACK	**INPUT STRING**	**ACTION**
$	id*id=id$	$<·idPush
$id	*id=id$	Error

Example 3: Check whether the following Grammar is an operator precedence grammar or not.

E→E+E

E→E*E

E→id

Solution:

1. **Computation of LEADING:**

LEADING(E) = {+, * , id}

2. **Computation of TRAILING:**

TRAILING(E) = {+, * , id}

3. **Precedence Table:**

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

All undefined entries are error. Since the precedence table has multiple defined entries, the grammar is not an operator precedence grammar.

LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions.. When (k) is omitted, it is assumed to be 1. Table 2.2 shows the comparison between LL and LR parsers.

Table 2.2 LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Types of LR parsing method:

1. SLR- Simple LR
    - Easiest to implement, least powerful.
2. CLR- Canonical LR
    - Most powerful, most expensive.
3. LALR- Look -Ahead LR
    - Intermediate in size and cost between the other two methods

The LR Parsing Algorithm:

The schematic form of an LR parser is shown in Fig 2.25. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).The driver program is the same for all LR parser. The parsing table alone changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a state.

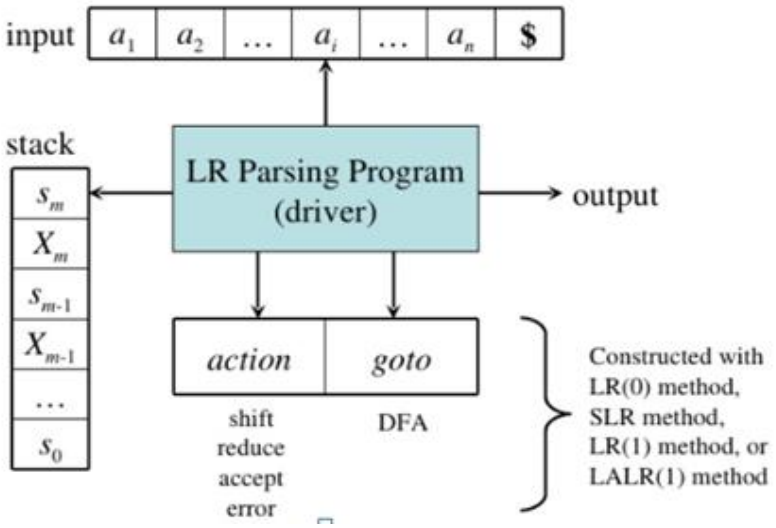


Fig 2.25 Model of an LR Parser

The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → •XYZ

A → X•YZ

A → XY•Z

A → XYZ•

Closure operation:

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:

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

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta]$ is in I.Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G`
2. Construct the canonical collection of set of items C for G"
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G"

Output : The SLR parsing table functions *action* and *goto* for G'

Method :

1. Construct $C = \{I_0, I_1, \ldots I_n\}$, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If $[A \rightarrow \alpha \bullet a\beta]$ is in I_i and $goto(I_i,a) = I_j$, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If$[A \rightarrow \alpha \bullet]$ is in I_i , then set *action*[*i*,*a*] to "reduce $A \rightarrow \alpha$" for all *a* in FOLLOW(A).
   (c) If [S"→S•] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*$(I_i,A) = I_j$, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→•S].

SLR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

set *ip* to point to the first input symbol of *w*$;

repeat forever begin

let *s* be the state on top of the stack and *a* the symbol pointed to by *ip*;

Example: Implement SLR Parser for the given grammar:

1.E→E + T

2.E→T

3.T→T * F

4.T→F

5.F→(E)

6.F→**id**

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

E'→E

E→E + T

E→T

T→T * F

T→F

F→(E)

F→id

Step 2 : Find LR (0) items.

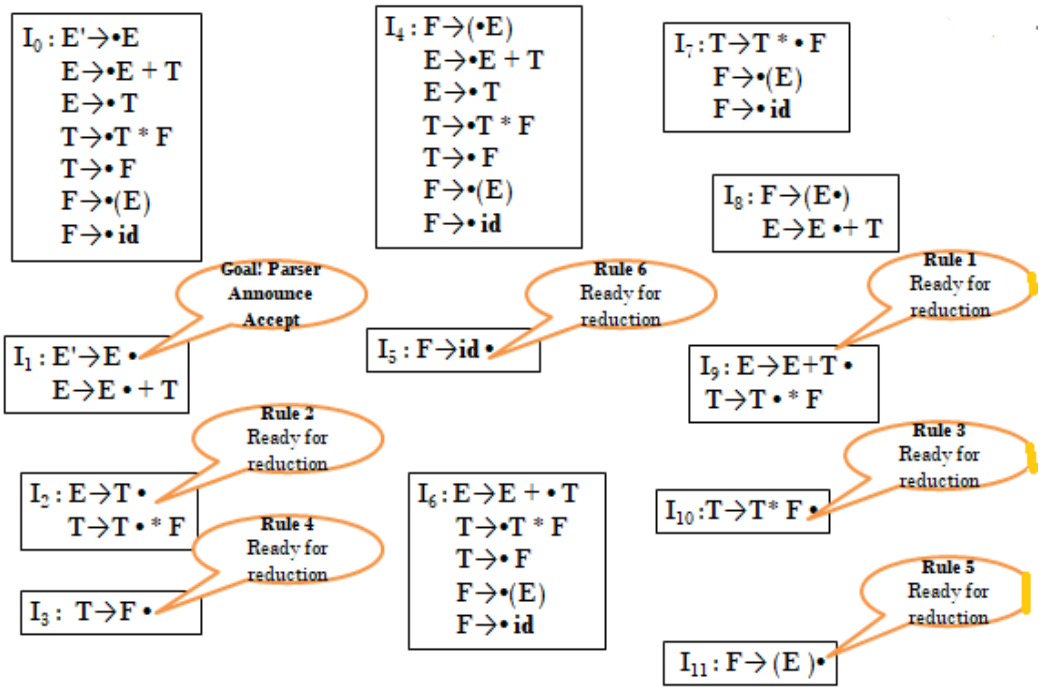


Fig 2.26 Canonical LR(0) collections

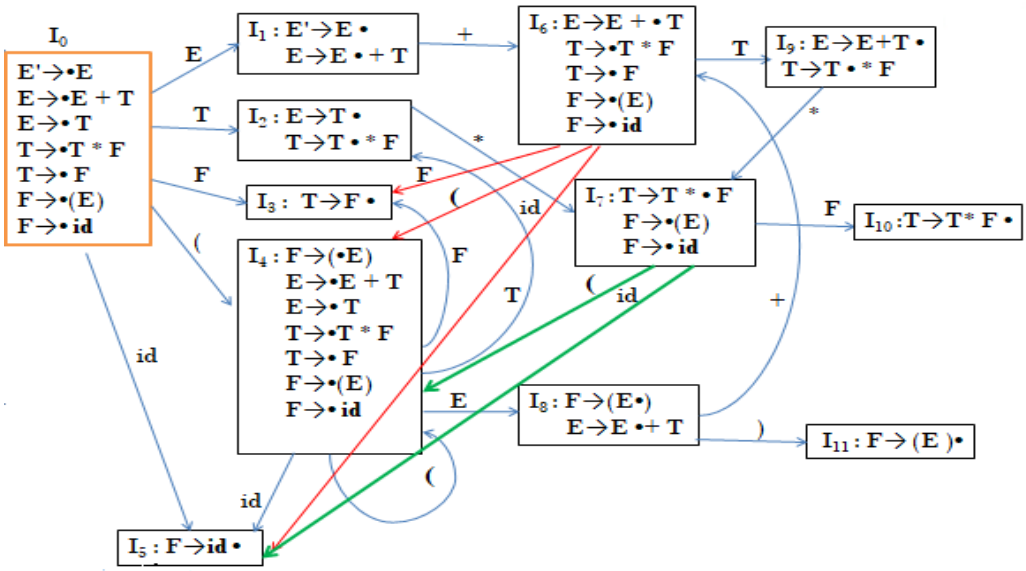


Fig 2.27 DFA representing the GOTO on symbols

Step 3 : Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

   FOLLOW(E) = {+,),$ }

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

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

State	ACTION						GOTO		
	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			

Fig 2.28 Parsing Table for the expression grammar

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.

Step 4: Parse the given input. The Fig 2.29 shows the parsing the string id*id+id using stack implementation.

Stack	Input	Action
0	id*id+id$	s5 Shift 5
0 id 5	*id+id$	r6 Reduce by F→**id**
0 **F** 3	*id+id$	r4 Reduce by T→**F**
0 **T** 2	*id+id$	s7 Shift 7
0 **T** 2 * 7	id+id$	s5 Shift 5
0 **T** 2 * 7 id 5	+id$	r6 Reduce by F→**id**
0 **T** 2 * 7 **F** 10	+id$	r3 Reduce by T→**T*F**
0 **T** 2	+id$	r2 Reduce by E→**T**
0 **E** 1	+id$	s6 Shift 6
0 **E** 1+ 6	id$	s5 Shift 5
0 **E** 1+ 6 id 5	$	r6 Reduce by F→**id**
0 **E** 1+ 6 **F** 3	$	r4 Reduce by T→**F**
0 **E** 1+ 6 **T** 9	$	r1 Reduce by E→**E+T**
0 **E** 1	$	Accept

Fig 2.29 Moves of LR parser on **id*id+id**

Top-Down Parsing- Recursive Descent Parsing:

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

A general form top-down parsing called recursive descent parsing, involves backtracking, that is making repeated scans of the input. A special case of recursive descent parsing called predictive parsing, where no backtracking is required.

Consider the grammar

S → cAd

A → ab | a

and the input string w=cad. Construction of parse is shown in fig 2.21.

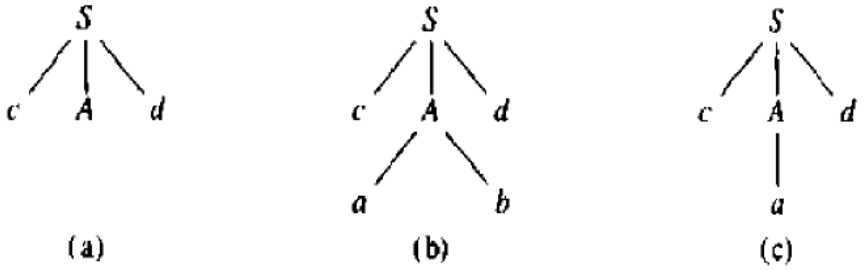


Fig 2.21 Steps in Top-down Parse

The leftmost leaf, labeled c, matches the first symbol of w, hence advance the input pointer to a, the second symbol of w. Fig 2.21(b) and (c) shows the backtracking required to match the input string.

Predictive Parser:

A grammar after eliminating left recursion and left factoring can be parsed by a recursive descent parser that needs no backtracking is a called a predictive parser. Let us understand how to eliminate left recursion and left factoring.

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA'

A' → αA' | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT '

T'→ *FT ' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. **for** i := 1 to n do begin

   for j := 1 to i-1 do begin

   replace each production of the form Ai → Aj γ

   by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$.

   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current Aj-productions;

   end

   eliminate the immediate left recursion among the A_i- productions

   end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

A → αA'

A' → $\alpha\beta_1 \mid \alpha\beta_2$

Consider the grammar,

S → iEtS | iEtSeS | a

E → b

Here,***i,t,e*** stand for **if ,the**,and **else** and **E** and **S** for "expression" and "statement".

After Left factored, the grammar becomes

S → iEtSS' | a

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

Non-recursive Predictive Parsing:

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser in Fig 2.22 looks up the production to be applied in a parsing table.

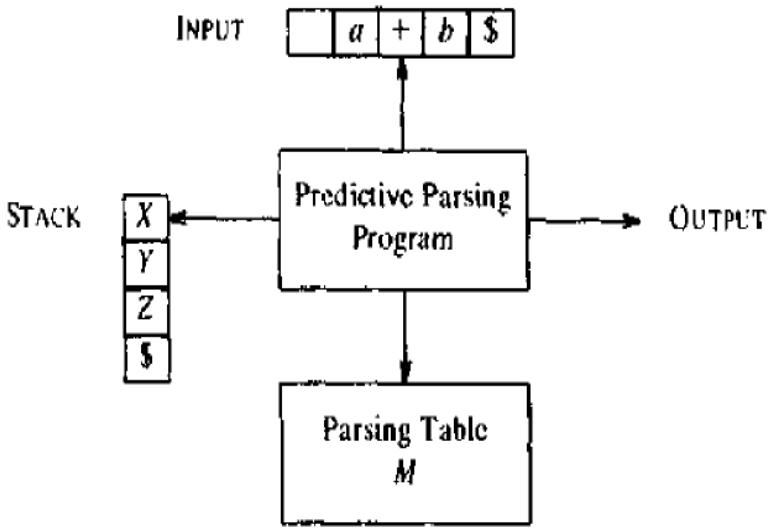


Fig 2.22 Model of a Non-recursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of S. The parsing table is a two-dimensional array *M[A,a]*,where *A* is a non-terminal, and *a* is a terminal or the symbol $.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If X = a =$,the parser halts and announces successful completion of parsing.
2. If X =a ≠$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example,

M[X,a] = {X→UVW}, the parser replaces X on top of the stack by WVU (with U on top). If M[X, a] = error, the parser calls an error recovery routine.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G .These functions are FIRST and FOLLOW.

Rules for FIRST():

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → $Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1),…,FIRST(Y_{i-1}); that is, $Y_1,\ldots Y_{i-1}$ => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW():

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε,then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

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

Algorithm : Non-recursive predictive parsing.

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

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

Method: Initially, the parser is in a configuration in which it has $$ on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

set *ip* to point to the first symbol of *w$;*
repeat
let X be the top stack symbol and **a** the symbol pointed to by i_p;
if X is a terminal or $ ***then***
If X=a ***then***

Example:

Consider the following grammar:

E → E+T | T

T → T*F | F

F → (E) | id

Step 1: After eliminating left recursion the grammar is

E → TE'

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

Step 2: Computation of FIRST( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Step 3: Computation of FOLLOW( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

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

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

Step 4: Construction of Predictive parsing table

M[X,a]	id	+	*	(	)	$
E	E →TE′			E →TE′		
E′		E′ →+TE′			E′ →ε	E′ →ε
T	T →FT′			T →FT′		
T′		T′ →ε	T′ →*FT′		T′ →ε	T′ →ε
F	F →id			F → (E)		

Fig 2.23 Parsing table

Step 5: Parsing the given string

With input **id+id*id** the predictive parser makes the sequence of moves shown in Fig 2,24.

STACK	INPUT	OUTPUT
$E	id+id*id$	E →TE’
$E’T	id+id*id$	T →FT’
$E’T’F	id+id*id$	F →id
$E’T’id	id+id*id$	pop
$E’T’	+id*id$	T’ →ε
$E’	+id*id$	E’ →+TE’
$E’T+	+id*id$	pop
$E’T	id*id$	T →FT’
$E’T’F	id*id$	F →id
$E’T’id	id*id$	Pop
$E’T’	*id$	T’ →*FT’
$E’T’F*	*id$	Pop
$E’T’F	id$	F →id
$E’T’id	id$	Pop
$E’T’	$	T’ →ε
$E’	$	E’ →ε
$	$	Accept

Fig 2.24 Moves made by predictive parser on input **id+id*id**

LL(1) Grammars:

For some grammars the parsing table may have some entries that are multiply-defined. For example, if G is left recursive or ambiguous , then the table will have at least one multiply-defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

Example: Consider this following grammar:

S→ iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S→ iEtSS’ | a S’→ eS | ε

E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) ={ i, a }

FIRST(S’) = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S’) = { $,e }

FOLLOW(E) = {t}

Parsing Table for the grammar:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.

Error detection and Recovery in Syntax Analyzer:

In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting them to users is called the **Error Handling process**.

Functions of an Error handler.

- Detection
- Reporting
- Recovery

Classification of Errors

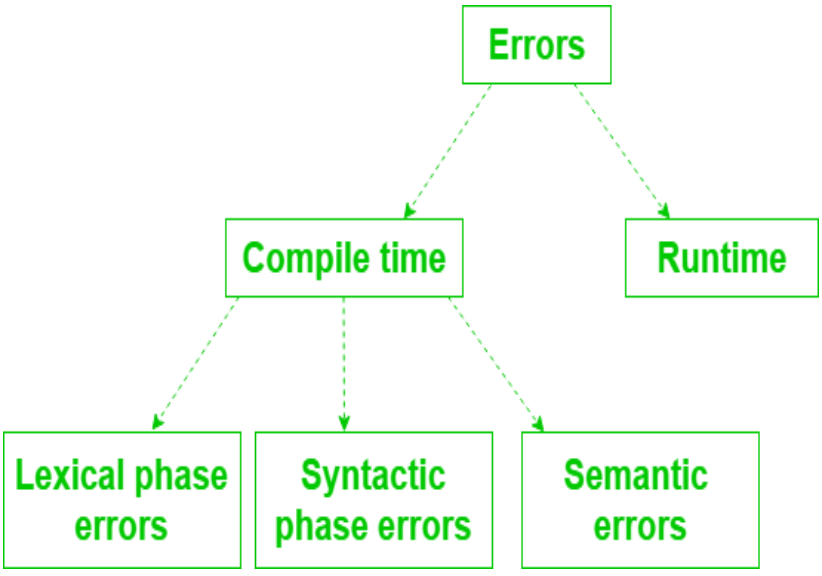


Fig 2.25 Classification of Errors

Compile-time errors:

Compile-time errors are of three types:-

1.Lexical phase errors

These errors are detected during the lexical analysis phase. Typical lexical errors are:

- Exceeding length of identifier or numeric constants.
- The appearance of illegal characters
- Unmatched string

2.Syntactic phase errors:

These errors are detected during the syntax analysis phase. Typical syntax errors are:

- Errors in structure
- Missing operator
- Misspelled keywords
- Unbalanced parenthesis

Error recovery for syntactic phase recovery:

1. Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimeters such as ; or }
- The advantage is that it's easy to implement and guarantees not to go into an infinite loop
- The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2. Statement Mode recovery

- In this method, when a parser encounters an error, it performs the necessary correction on the remaining input so that the rest of the input statement allows the parser to parse ahead.
- The correction can be deletion of extra semicolons, replacing the comma with semicolons, or inserting a missing semicolon.
- While performing correction, utmost care should be taken for not going in an infinite loop.
- A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection.

3. Error production

- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.

- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- The disadvantage is that it's difficult to maintain.

4. Global Correction

- The parser examines the whole program and tries to find out the closest match for it which is error-free.
- The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

3.Semantic errors

These errors are detected during the semantic analysis phase. Typical semantic errors are

- Incompatible type of operands
- Undeclared variables
- Not matching of actual arguments with a formal one

Error recovery for Semantic errors

- If the error **"Undeclared Identifier"** is encountered then, to recover from this a symbol table entry for the corresponding identifier is made.
- If data types of two operands are incompatible then, automatic type conversion is done by the compiler.

YACC-Yet Another Compiler Compiler

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing.

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

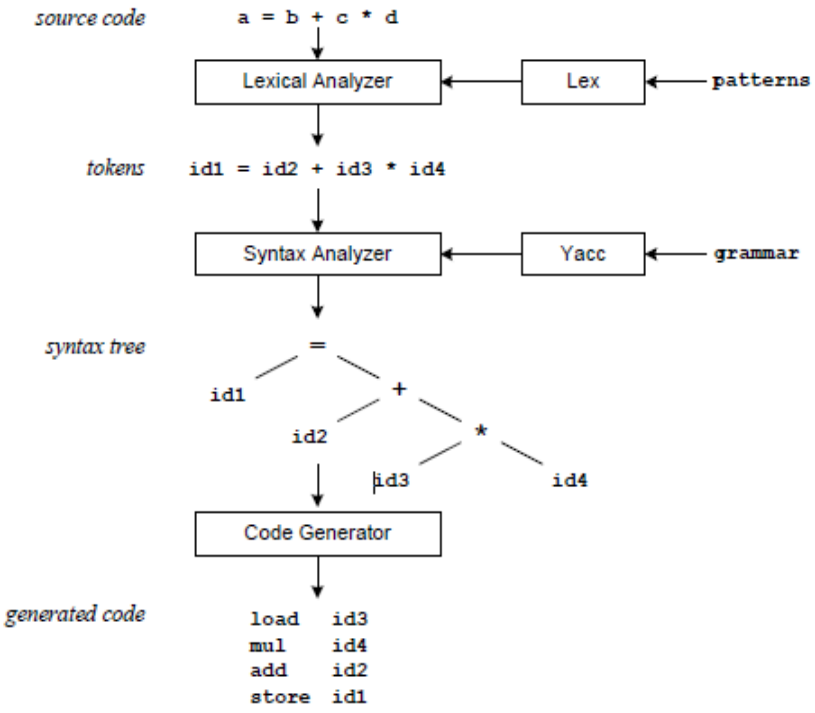


Fig 2.26 Compilation Sequence

The **patterns** in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The **grammar** in the above diagram is a text file you create with a text edtior. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

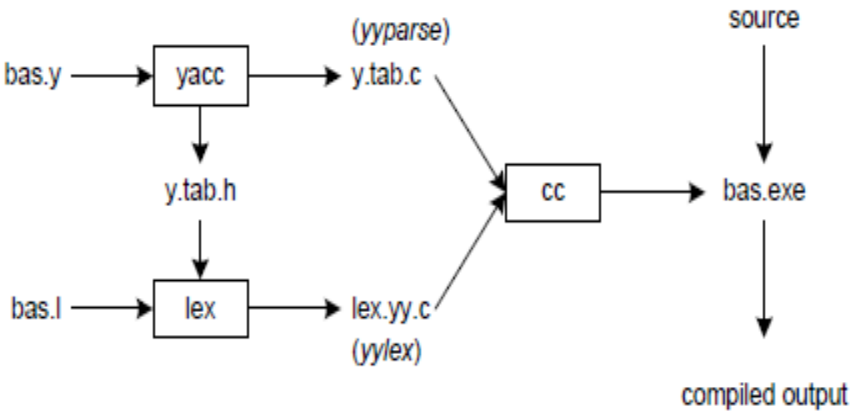


Fig. 2.27 Building a Compiler with Lex/Yacc

```
yacc -d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe    # compile/link
```

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser) that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h.

Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.

Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

Input File:

YACC input file is divided into three parts.

```
/* definitions */
  ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Definition Part:

The definition part includes information about the tokens used in the syntax definition:

```
%token NUMBER
%token ID
```

The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.

Rules Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Example Program:

Evaluation of Arithmetic expression using Unmbiguous Grammar(Use Lex and Yacc Tool)

E-> E+T | E-T|T

T->T*F | T/F|F

F-> (E) | id

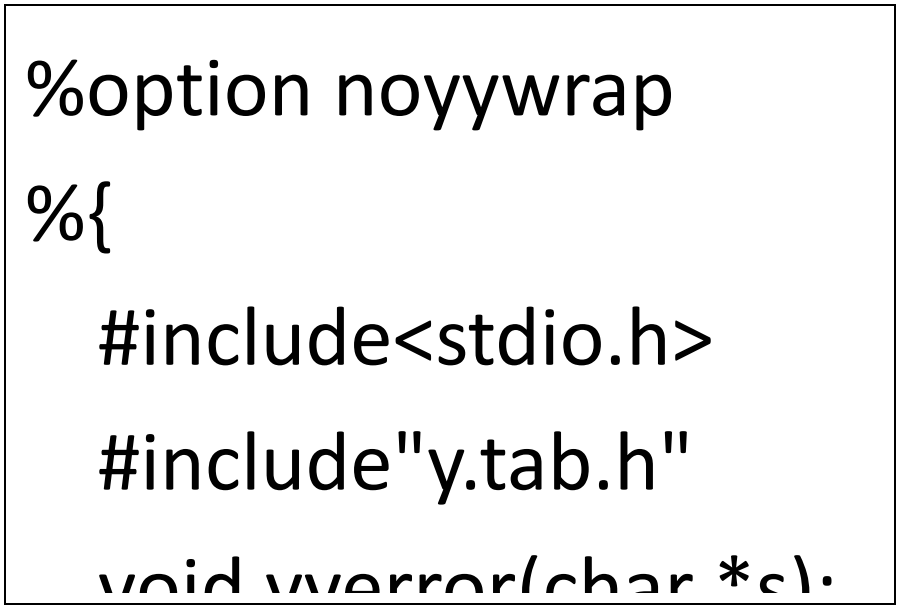


Fig 2.28 Lex Program

```
%{
        #include<stdio.h>
        void yyerror(char*);
        extern int yylex(void);
%}
%token NUM
%%
S:
S E '\n'        {printf("%d\n",$2);}
|
;
E:
E '+' T  {$$=$1+$3;}
|E '-' T {$$=$1-$3;}
|T       {$$=$1;}
T:
T '*' F  {$$=$1*$3;}
|T '/' F {$$=$1/$3;}
|F       {$$=$1;}
F:
'(' E ')'  {$$=$2;}
|NUM  {$$=$1;}
```

Fig 2.29 YACC Program

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-III Compiler Design – SCSA1604

III. INTERMEDIATE CODE GENERATION

Types of Intermediate Code – Representation of three address code - Syntax Directed Translation scheme- Intermediate code generation for: Assignment statements - Boolean statements - Switch-case statement –Procedure call - Symbol Table Generation.

Syntax Directed Translation:

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known. Semantic analysis involves adding information to the symbol table and performing type checking. It needs both representation and implementation mechanism.

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.

We associate Attributes to the grammar symbols representing the language constructs. Values for attributes are computed by Semantic Rules associated with grammar productions

Evaluation of Semantic Rules may:

- Generate Code;
- Insert information into the Symbol Table;
- Perform Semantic Check;
- Issue error messages;

There are two notations for attaching semantic rules:

1. Syntax Directed Definitions.High-level specificationhiding many implementation details (also called Attribute Grammars).

2.Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

There are two kinds of attributes:

1. Synthesized Attributes: They are computed from the values of the attributes of the children nodes.
2. Inherited Attributes: They are computed from the values of the attributes of both the siblings and the parent nodes

Example: Let us consider the Grammar for arithmetic expressions (DESK CALCULATOR)
The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Definition: An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.
Evaluation Order: Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or Post Order, traversal of the parse-tree.
The annotated parse-tree for the input 3*5+4n is:

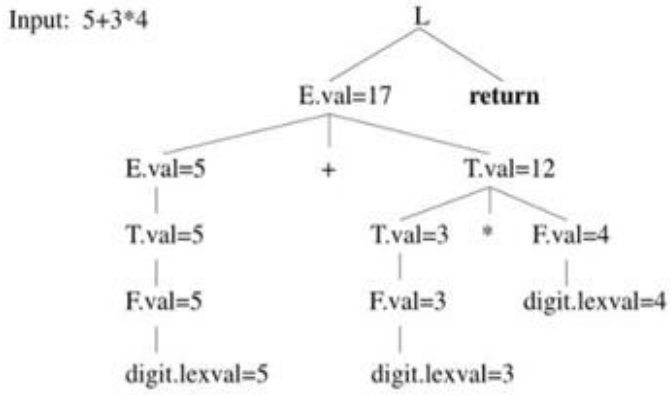
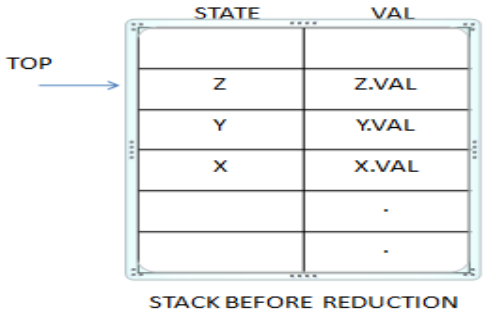


Fig 3.1 Annotated Parse Tree

Implementation of SDT for Desk Calculator:

Use an extra fields in the parser stack entries corresponding to the grammar symbols. These fields hold the values of the corresponding translations.



The grammar and corresponding Semantic action is shown below:

Production	Semantic Action
$S \rightarrow E\ \$$	{ print E.VAL }
$E \rightarrow E_1 + E_2$	{E.VAL := E_1.VAL + E_2.VAL }
$E \rightarrow E_1 * E_2$	{E.VAL := E_1.VAL * E_2.VAL }
$E \rightarrow (E_1)$	{E.VAL := E_1.VAL }
$E \rightarrow I$	{E.VAL := I.VAL }
$I \rightarrow I_1$ digit	{I.VAL := 10 * I_1.VAL + LEXVAL }
$I \rightarrow$ digit	{ I.VAL:= LEXVAL}

The Annotated parse tree for the expression 23*4+5 is depicted in Fig 3.2.

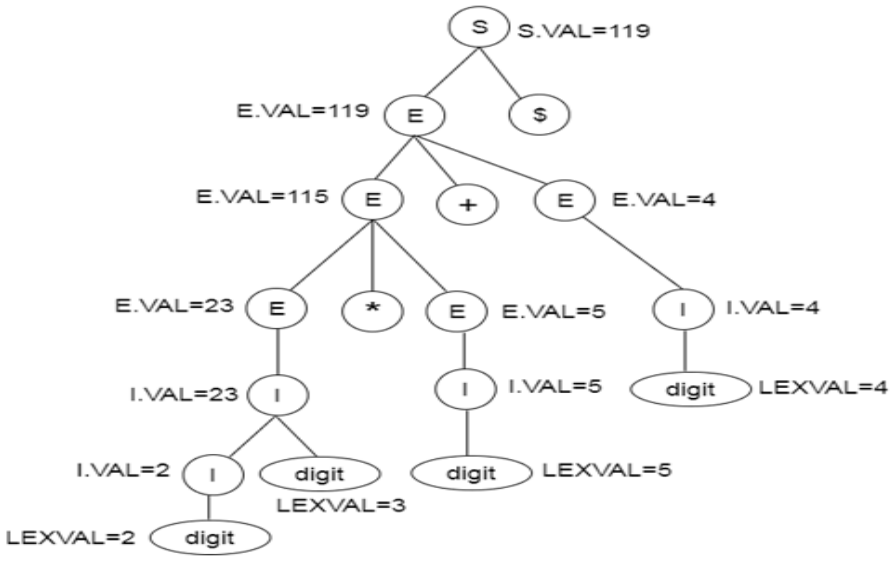


Fig 3.1 Annotated Parse Tree for 23*4+5$

The method of implementing the desk calculator is given using the program fragment represents the stack implementation to evaluate a given expression.

Production	Program Fragment
S → E $	print VAL[TOP]
E → E + E	VAL[TOP] := VAL[TOP] + VAL[TOP-2]
E → E * E	VAL[TOP] := VAL[TOP] * VAL[TOP-2]
E → (E)	VAL[TOP] := VAL[TOP-1]
E → I	none
I → I digit	VAL[TOP] := 10 * VAL[TOP] + LEXVAL
I → digit	VAL[TOP] := LEXVAL}

The sequences of moves to implement the evaluation of an expression are shown in Fig 3.3.

Moves	Input	STATE	VAL	Production used
(1)	23*5+4$	_	_	
(2)	3*5+4$	2	_	
(3)	3*5+4$	I	2	I → digit
(4)	*5+4$	I3	2_	
(5)	*5+4$	I	(23)	I → I digit
(6)	*5+4$	E	(23)	E → I
(7)	5+4$	E*	(23) _	
(8)	+4$	E*5	(23) _ _	
(9)	+4$	E*I	(23) _ 5	I → digit
(10)	+4$	E*E	(23) _ 5	E → I
(11)	+4$	E	(115)	E → E * E
(12)	4$	E+	(115) _	
(13)	$	E+4	(115) _4	
(14)	$	E+I	(115) _4	I → digit
(15)	$	E+E	(115) _4	E → I
(16)	$	E	(119)	E → E + E
(17)	_	E$	(119) _	
(18)	_	S	PRINT 119	S → E $

Fig 3.3 Sequence of moves

Intermediate Code Representation:

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

1. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
2. The second part of compiler, synthesis, is changed according to the target machine.
3. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate codes can be represented in a variety of ways and they have their own benefits.

1. High Level IR - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
2. Low Level IR - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

- During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text.
- The complexity of this code lies between the source language code and the object code.
- The intermediate code can be represented in the form of:
  - Postfix notation,
  - Syntax tree,
  - Three-address code.

Postfix Notation:

The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b. The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the postfix notation is:

e1e2 +

No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.

The postfix representation of the expressions:

1. (a+b)*c → **ab+c***
2. a*(b+c) → **abc+***
3. (a+b)*(c+d) → **ab+cd+***
4. (a – b) * (c + d) + (a – b) → **ab - cd + *ab - +**

Let us introduce a 3-ary (ternary) operator ,the conditional expression. Consider **if e then x else y** ,whose value is

$$\begin{cases} x, if\ e \neq 0 \\ y, if\ e = 0 \end{cases}$$

Using ? as a ternary postfix operator, we can represent this expression as: **exy?**

Evaluation of Postfix Expression:

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

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

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

Syntax Directed Translation for Postfix:

E.Code is a string-valued translation. The value of the translation **E.code** for the first production is the concatenation of $\mathbf{E^{(1)}.code}$, $\mathbf{E^{(2)}.code}$ and the symbol **op**

Production	Semantic Rule	Program Fragment
E → E1 op E2	E.code = E1.code ‖E2.code‖op	Print op
E → (E1)	E.code = E1.code	
E → id	E.code = id	Print id

Fig 3.3 SDT for postfix

Processing the input a+b*c ,a syntax-directed translator based on an LR parser, has the following

sequence of moves and generates the postfix abc*+.

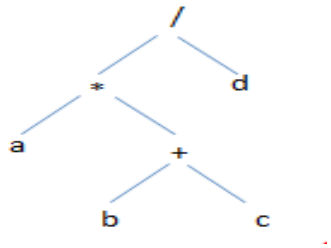
1. Shift a
2. Reduce by E→ id and print a
3. Shift +
4. Shift b
5. Reduce by E→ id and print b
6. Shift *
7. Shift c
8. Reduce by E→ id and print c
9. Reduce by E→ E op E and print *
10. Reduce by E→ E op E and print +

Syntax tree:

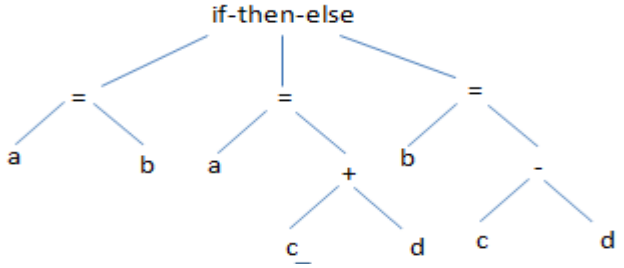
Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example:

The syntax tree for the expression a*(b+c)/d is as shown in the figure.



The syntax tree for the statement if a=b then a=c+d else b=c-d



Syntax Directed Translation for syntax tree:

Production	Semantic Rule
E → E1 op E2	E.val = NODE (op, E1.val, E2.val)
E → (E1)	E.val = E1.val
E → -E1	E.val = Unary (-, E1.val
E → id	E.val = (LEAF (id))

Fig 3.4 SDT for syntax tree

Three Address Code:

Three address code is a sequence of statements of the general form x =y op z where x, y, and z are names, constants, or compiler-generated temporaries;

- op stands for any operator such as a fixed- or floating-point arithmetic operator or a logical operator on Boolean valued data.

Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement.Thus a source language expression like x+ y * z might be translated into a sequence t1= y * z t2=x + t1 where t1, and t2 are compiler-generated temporary names.

Types of Three Address Statements:

Three-address Statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of three-address statement in the array holding intermediate code.

Here are the common three address statements used :

1. Assignment statements of the form x=y op z ,where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form x = op y. where op is a unary operation. Essential unary operations include unary minus. Logical negation,shift operators and conversion operators that, for example. convert fixed-point number to a floating-pointnumber.
3. Copy statement of the form x=y where the value of y is assigned tox.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as If x relop y goto L. This instruction applies a relational

operator(,>=,etc.) to x and y. and executes, the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next,, as is the usual sequence.

6. Param x and call p, n for procedure calls and return y. where y representing a returned value is optional Their typical use it as the sequence of three.address statements.

```
param x1
param x2
….
Param xn
call p,n
```

generated as part of a call of the procedure p(x1,x2,….xn)

7. Indexed assignments of the form x=y[i] and x[i]=y. The first of these sets x to the value in the location i memory units beyond location y. The stat[i]=y sets the contents of the location I units beyond x to the value of y. In both these instructions, x, y. and i refer to data objects.
8. Address and pointer assignments of the form x=&y, x=*y and*x=y

Implementation of Three Address Statements:

A three-address statement is an abstract form of intermediate code.In a compiler, these statements can be implemented as records with fields for the operator and the operands.Three such representations are quadruples, triples, and indirect triples.

Quadruples:

A quadruple is a record structure with four fields, which we call op,. arg1, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x =y op z is represented by placing y in arg1, z in arg2, and x in result. Statements with unary operators like x = -y or x= y do not use arg2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result.

Example : Consider expression a = b * – c + b * – c.

The three address code is:

```
t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
```

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples:

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. Doing so ,the three address statements can be represented by records with only three fields :op,arg1,arg2.

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

The contents of fields arg1,arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created. Triples for conditional and unconditional statements are shown in Fig 3.5.

Unconditional jump:

goto L // for label Triples

op	arg1	arg2
goto	L	--

Conditional jump:

if x < y goto L

	op	arg1	arg2
(0)	<	x	y
(1)	if	(0)	L

Fig 3.5 Triples

Triples for indexed assignment statements are shown in Fig 3.6.

x := y[i]

	op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

x[i] := y

	op	arg1	arg2
(0)	[]=	x	i
(1)	=	(0)	y

Fig 3.6 Triples for indexed assignment statements

Indirect Triples:

Another implementation of three address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples. A triple and their indirect triple representation ins shown below.

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Syntax Directed Translation for Assignment Statements :

In the syntax directed translation, assignment statement is mainly deals with expressions.

The expression can be of type real, integer, array…

Consider the grammar:

S → id := E

E → E_1 + E_2

E → E_1 * E_2

E → (E_1)

E → id

The m=non-terminal E has two attributes:

- E.place, a name that will hold the value of E, and

- E.code, the sequence of three-address statements evaluating E.

A function gen(…) to produce sequence of three address statements .The statements themselves are kept in some data structure, e.g. list – SDD operations described using pseudo code. The syntax directed translation for assignment statement is given as below;

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place ':=' E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place ':=' E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place ':=' E_1.place '*' E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place ':=' 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

The sequence of moves in generating the three address code for an assignment statement is given by the following example.

Example: Generate the three address code for the assignment statement **A= − B*(C+D)**

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 – D	
)	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
	S	S	

Assignment Statement with mixed types:

The constants and variables would be of different types, hence the compiler must either reject certain mixed-mode operations or generate appropriate coercion (mode conversion) instructions.

Consider two modes:

REAL

INTEGER, with integer converted to real when necessary.

An additional field in translation for E is E.MODE whose value is either REAL or INTEGER.

The semantic rule for E.MODE associated with the production **E → E_1 + E_2** is:

E → E_1 + E_2 {if E_1.MODE = INTEGER and E_2.MODE = INTEGER then E.MODE = INTEGER else E.MODE = REAL }

When necessary the three address code

A=inttoreal B is generated,

whose effect is to convert integer B to a real of equal value called A.

Example: Consider the assignment statement

X=Y+I*J

Assume X and Y to be REAL and I and J have mode INTEGER.

Three-addresss code:

T_1=I int * J

T_2= inttoreal T_1

T_3=Y real +T_2

X=T_3

The semantic rule uses two translation fields E.PLACE and E.MODE for the non-terminal E

The semantic rule for E → E_1 op E_2 is given as below:

```
T= NEWTEMP()
if E1.MODE=INTEGER and E2.MODE=INTEGER then
    begin
        GEN(T= E1.PLACE int op E2.PLACE)
        E.MODE=INTEGER
    end

else if E1.MODE=REAL and E2.MODE=REAL then
    begin
        GEN(T= E1.PLACE real op E2.PLACE)
        E.MODE=REAL
    end

else if E1.MODE=INTEGER and E2.MODE=REAL then
    begin
        U=NEWTEMP()
        GEN(U= inttoreal E1.PLACE)
        GEN(T= U real op E2.PLACE)
        E.MODE=REAL
    end

else if E1.MODE=REAL and E2.MODE=INTEGER then
    begin
        U=NEWTEMP()
        GEN(U= inttoreal E2.PLACE)
        GEN(T= E1.PLACE real op U)
        E.MODE=REAL
    end
```

SDT for Boolean Expression:

Boolean expressions have two primary purposes.

1. They are used for computing the **logical values**.
2. They are also used as **conditional expression** that alter the flow of control, such as if-then-else or while-do.

Boolean expression are composed of the boolean operators(**and** ,**or**, and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 relop E_2 , where E_1 and E_2 are arithmetic expressions.

Consider the grammar

E → E **or** E

E → E **and** E

E → **not** E

E → (E)

E → id **relop** id

E → TRUE E → id

E → FALSE

The relop is any of <,≤,=, ≠, >, or ≥.. We assume that **or** and **and** are left associative. **or** has lowest precedence ,then **and** ,then **not**.

Methods of Translating Boolean Expressions:

There are two methods of representing the value of a Boolean.

1.To encode true or false numerically

- 1 is used to denote true.
- 0 is used to denote false.

2.To evaluate a boolean expression analogously to an arithmetic expression.

- Flow of control –representing the value of a boolean expression by a position reached in a program.
- Used in flow of control statements such as if-then-else or while-do statements.

Numerical Representation of Boolean Expressions:

First consider the implementation of boolean expressions using 1 to denote TRUE and 0 to denote FALSE. Expressions will be evaluated from left to right.

Example 1:

A or B and C

Three address sequence:

T_1 = B and C

T_2 = A or T_1

Example 2:

A relational expression A< B is equivalent to the conditional statement.

if A < B then 1 else 0

Three address sequence:

(1) if A < B goto (4)

(2) T=0

(3) goto (5)

(4) T=1

(5) ---

Semantic Rules for Boolean Expression:

The Semantic rules for Boolean expression are given as below:

$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen (E.place = E_1.place **or** E_2.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); gen (E.place = E_1.place **and** E_2.place)}
$E \rightarrow$ NOT E_1	{E.place = newtemp(); gen (E.place = **not** E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{ E.place = newtemp(); gen (if id_1.place **relop.op** id_2.place goto nextquad + 3); gen(E.place =0) gen(goto nextquad + 2) gen(E.place=1) }
E → TRUE	{E.place := newtemp(); gen(E.place=1)}
E → FALSE	{E.place := newtemp(); gen(E.place =0)}

Example: Code for a < b or c < d and e <f

100: if a < b goto 103

101: t1 = 0

102: goto 104

103: t1 = 1

104: if c < d goto 107

105: t2 =0

106: goto 108

107: t2 = 1

108: if e < f goto 111

109: t3 =0

110: goto 112

111: t3 = 1

112: t4 = t2 and t3

113: t5 = t1 or t4

Control Flow Representation:

Short Circuit Evaluation of Boolean Expressions:

Flow of control statements with the jump method.

Consider the following : (short circuit code)

S → if E then S | if E then S1 else S2

S → while E do S1

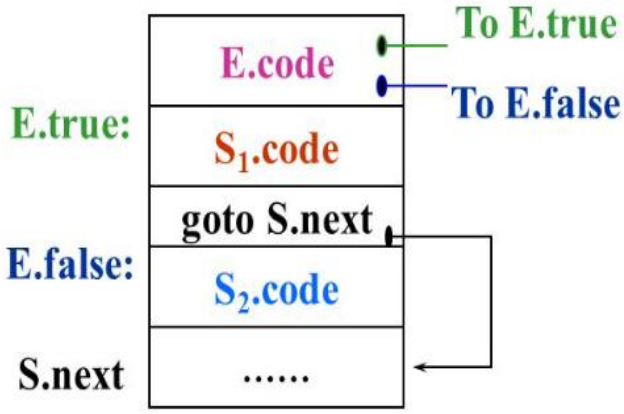


Fig 3.7 Attributes used for if E then S_1 else S_2

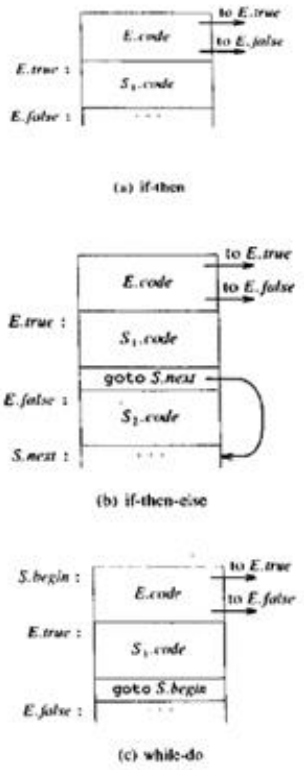
E is associated with two attributes:

1. E.true: label which controls if E is true
2. E.false :label which controls if E is false

S.next - is a label that is attached to the first three address instruction to be executed after the code for S (S_1 or S_2)

The Semantic rule for flow of control statements is given by:

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel$ $gen(E.true ':') \parallel S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel$ $gen(E.true ':') \parallel S_1.code \parallel$ $gen('goto' S.next) \parallel$ $gen(E.false ':') \parallel S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin ':') \parallel E.code \parallel$ $gen(E.true ':') \parallel S_1.code \parallel$ $gen('goto' S.begin)$



SDT for Switch Case statement:

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows: Syntax for switch case statement

switch E

begin

case V1: S1

case V2: S2

…

case Vn-1: Sn-1

default: Sn

end

When switch keyword is seen then a new temporary t and two new labels test and next are generated. When the case keyword occurs then for each case keyword, a new label L_i is created and entered into the symbol table. The value of V_i of each case constant and a pointer

to this symbol-table entry are placed on a stack.

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Translation of a switch-statement

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

Another translation of a switch statement

SDT for Procedure Call:

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record
- Evaluate the argument of the called procedure.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

Consider the grammar for a simple procedure call statement,

(1) S→ call id(Elist)

(2) Elist → Elist,E

(3) Elist → E

Syntax-Directed Translation for Procedure call is given as:

Production Rule	Semantic Action
S → call id(Elist)	{for each item p on QUEUE do GEN (param p) GEN (call id.PLACE) }
Elist → Elist, E	{append E.PLACE to the end of QUEUE }
Elist → E	{initialize QUEUE to contain only E.PLACE}

QUEUE is used to store the list of parameters in the procedure call.

Symbol Table Generation:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces,etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. It is built in lexical and syntax analysis phases.

The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code. It is used by compiler to achieve compile time efficiency. It is used by various phases of compiler as follows:-

Lexical Analysis: Creates new table entries in the table, example like entries about token.

Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and updates it accordingly.

Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

Code Optimization: Uses information present in symbol table for machine dependent optimization.

Target Code generation: Generates code by using address information of identifier present in the table.

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry

for each name in the following format:

<Symbol Name, Type, Attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

Then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table includes:

Operations	Functions
Allocate	To allocate a new empty symbol table
Free	To remove all entries and free the storage of symbol table
Lookup	To search for a name and return pointer to its entry
Insert	To insert name in a symbol table and return a pointer to its entry
Set_Attribute	To associate an attribute with a given entry
Get_Attribute	To get an attribute associated with a given entry

Implementation of Symbol table – Following are commonly used data structure for implementing symbol table:-

List:

- In this method, an array is used to store names and associated information.
- A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from beginning of list till available pointer and if not found we get an error "use of undeclared name"
- While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. "Multiple defined name"
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- Advantage is that it takes minimum amount of space.

Linked List:

- This implementation is using linked list. A link field is added to each record.
- Searching of names is done in order pointed by link of link field.
- A pointer "First" is maintained to point to first record of symbol table.
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Hash Table:

- In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
- A hash table is an array with index range: 0 to table size – 1.These entries are pointer pointing to names of symbol table.
- To search for a name we use hash function that will result in any integer between 0 to table size –1.
- Insertion and lookup can be made very fast –O(1).
- Advantage is that search is possible and disadvantage is that hashing is complicated to implement.

Binary Search Tree:

- Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follows the property of

binary search tree.

- ❖ Insertion and lookup are O(log2 n) on average.

Scope Management:

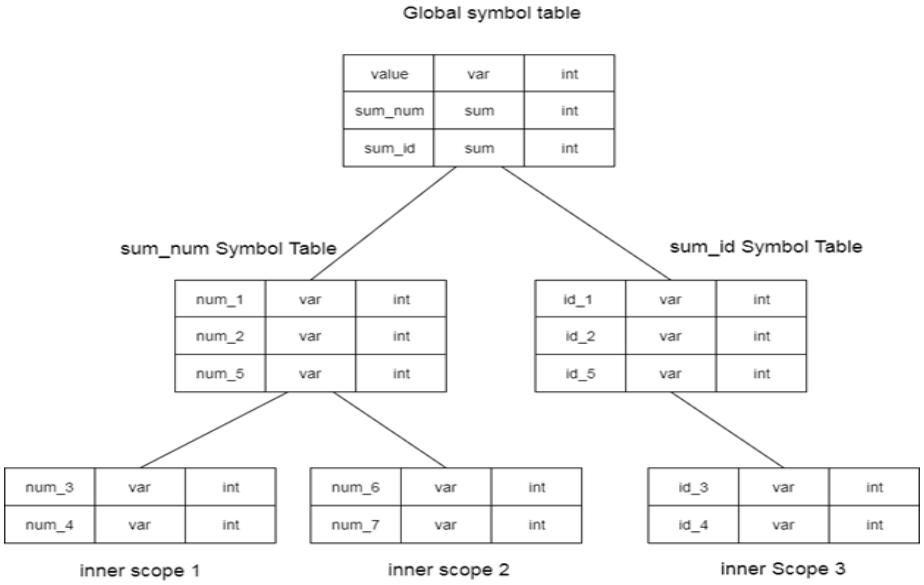
❖ A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

❖ To determine the scope of a name, symbol tables are arranged in hierarchical structure consider the example as shown below:

```
int value=10;
 int sum_num()
   {
    int num_1;
    int num_2;
        {
         int num_3;
         int num_4;
        }
     int num_5;
        {
         int_num 6;
        int_num 7;
        }
   }
```

```
int sum_id
   {
    int id_1;
    int id_2;
       {
        int id_3;
         int id_4;
        }
     int num_5;
   }
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-IV Compiler Design – SCSA1604

IV. CODE OPTIMIZATION

Principle sources of Optimization - Basic Blocks and Flow Graphs - Loop Optimization & its types – DAG - Peephole optimization - Dominators - Global Data Flow Analysis.

4.1 Optimization:

Principles of source optimization:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (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.

• 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.

Types of optimization:

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

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.

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.

Organization of the code optimizer:

The techniques used are a combination of Control-Flow and Data-Flow analysis as shown in Fig 4.1.

Control-Flow Analysis: Identifies loops in the flow graph of a program since such loops are usually good candidates for improvement.

Data-Flow Analysis: Collects information about the way variables are used in a program.

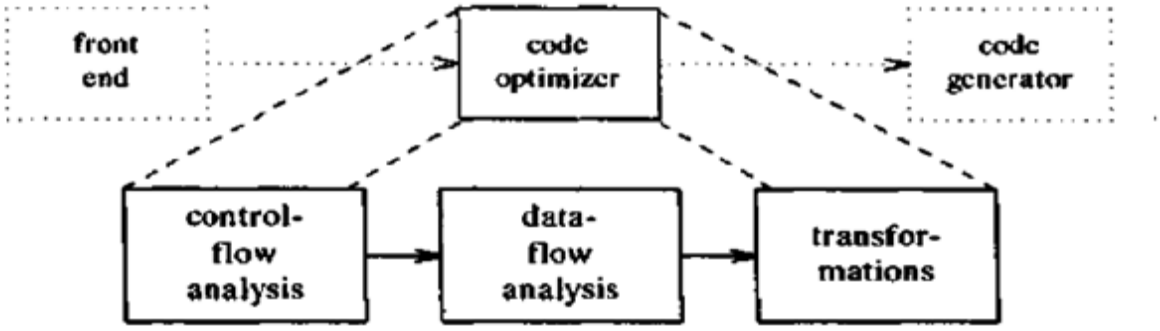


Fig 4.1. Organization of the code optimizer

Steps before optimization:

1) Source program should be converted to Intermediate code
2) Basic blocks construction
3) Generating flow graph
4) Apply optimization

Basic Block and Flowgraphs:

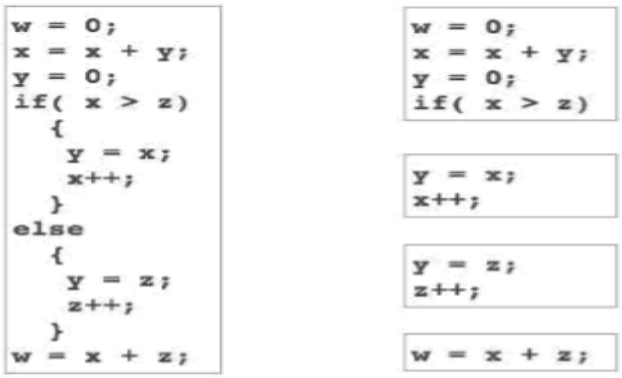
Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the

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, SWITCHCASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Characteristics of Basic Blocks:

1. They do not contain any kind of jump statements in them.
2. There is no possibility of branching or getting halt in the middle.
3. All the statements execute in the same order they appear.
4. They do not lose the flow control of the program.



Source Program Basic Blocks

Fig 4.2. Basic Blocks for the sample source program

We may use the following **algorithm to find the basic blocks in a program:**

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the **set of leaders**, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example:

Consider the following source code for dot product of two vectors:

```
begin
        prod :=0;
        i:=1;
        do begin
        prod :=prod+ a[i] * b[i];
        i :=i+1;
        end
        while i <= 20
end
```

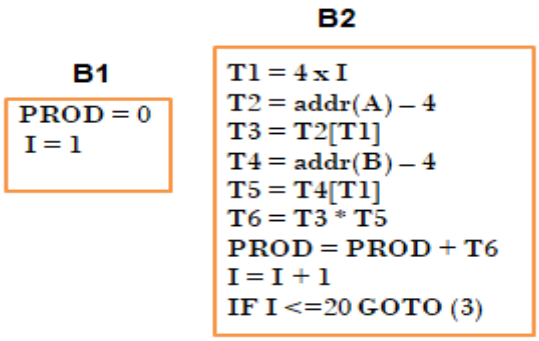
The three address code sequence for the above source is given as follows:

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

Solution: For partitioning the three address code to basic blocks.

- PROD = 0 is a leader since first statement of the code is a leader.
- T1 = 4 * I is a leader since target of the conditional goto statement is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

Now, the given code can be partitioned into two basic blocks as:



4.2 Flow graphs:

A flow graph is a directed graph with flow control information added to the basic blocks.The basic blocks serves as nodes of the flow graph. The nodes of the flow graph are basic blocks. It has a distinguished initial node.

There is a directed edge from block B1 to block B2 if B2 appears immediately after B1 in the code.

E.g. Flow graph for the vector dot product is shown in Fig 4.3:

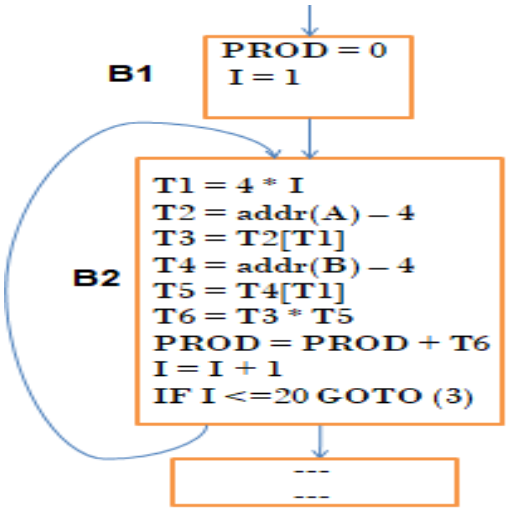


Fig 4.3. Flow graph

In Fig 4.3 B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B2 is the first statement B2, so there is an edge from B2(last statement) to B2 (first statement).B1 is the predecessor of B2, and B2 is a successor of B1.

A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program. Another example is shown in Fig 4.4.

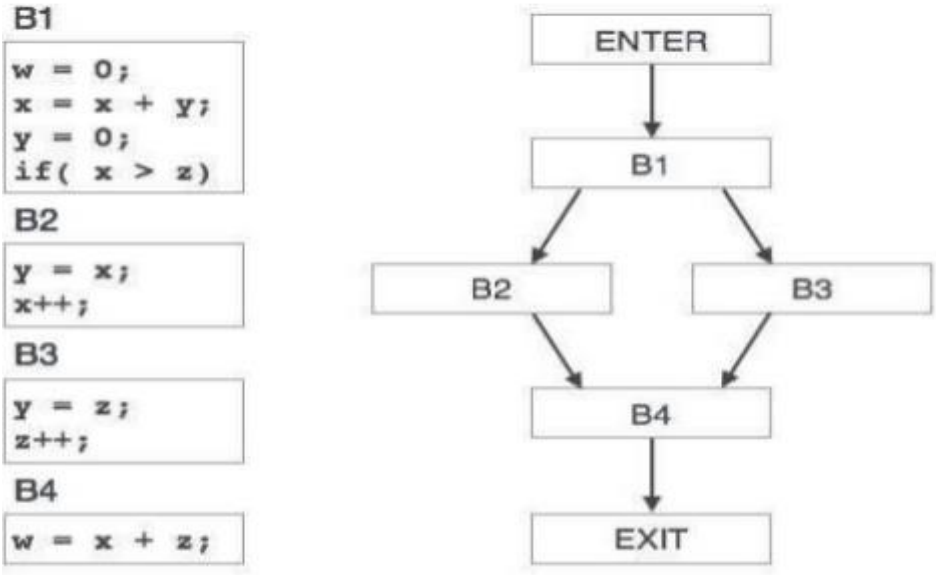


Fig 4.4. Flow graph for the basic blocks

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without modifying expressions computed by the block. Two important classes of transformation are :

1. Structure Preserving Transformations
    - Common sub-expression elimination
    - Dead code elimination
    - Renaming of temporary variables
    - Interchange of two independent adjacent statements
2. Algebraic transformation

Structure preserving transformations:

a)Common sub-expression elimination: Consider the sequence of statements:

1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d

Since the second and fourth expressions compute the same expression, the code can be transformed as:

1) a = b + c
2) b = a - d

3) c = b + c

4) d = b

b) Dead-code elimination: Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables: A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block. Such a block is called a normal-form block.

d) Interchange of statements: Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

iii) x/1 = x , x/2=x*0.5

Loop optimization & its types :

A loop is a collection of nodes in a flow graph such that:

1. All nodes in the collection are strongly connected**,** i.e, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
2. The collection of nodes has a unique entry, i.e, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is through the entry point.

A loop that contains no other loops is called an **inner loop**.

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques

(a) Invariant code :

- If a computation produces the same value in every loop iteration, move it out of the loop.
- An expression can be moved out of the loop if all its operands are invariant in the loop
- Constants are loop invariants.

(b) Induction analysis:

A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.

Eg:

```
extern int sum;
int foo(int n)
{
int i, j; j = 5;
for (i = 0; i < n; ++i)
{
j += 2; sum += j;
}
return sum;
}
```

This can be re-written as,

```
extern int sum;
int foo(int n)
{
int i; sum = 5;
for (i = 0; i < n; ++i)
{
sum += 2 * (i + 1);
}
return sum;
```

}

(c) Reduction in Strength:

There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

```
c = 7;
for (i = 0; i < N; i++)
{
y[i] = c * i;
}
```

can be replaced with successive weaker additions

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
y[i] = k;
k = k + c;
}
```

(d) Constant folding and constant propagation

Constant folding:

It is the process of recognizing and evaluating statements with constant expressions (i=22+222+2222 to i=2466), string concatenation (“abc”+”def” to “abcdef”) and expressions with arithmetic identities (x=0; z=x*y[i]+x*2 to x=0; z=0;) at compile time rather than at execution time.

Constant propagation:

It is the process of substituting values of known constants in an expression at compile time.

Eg:

```
int x=14;
int y=7+x/2;
return y*(28/x+2);
```

Applying constant folding and constant propagation,

```
int x=14;
int y=14;
```

return 56;

4.3 Direct Acyclic Graph (DAG):

DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

• Leaf nodes represent identifiers, names or constants.

• Interior nodes represent operators.

• Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

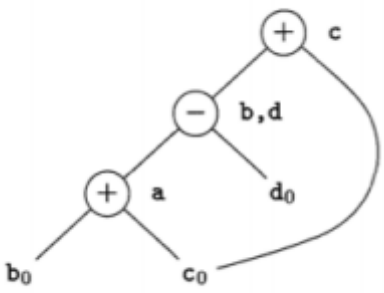


Fig. 4.5. Directed Acyclic Graph

Peephole Optimization

• Optimizing a small portion of the code.

• These methods can be applied on intermediate codes as well as on target codes.

• A bunch of statements is analyzed and are checked for the following possible optimization

(1) Redundant instruction elimination

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

For example:

MOV x, R0

MOV R0, R1

First instruction can be rewritten as

MOV x,R1

(2) Unreachable Code

It 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.

(3) Flow of control optimization

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 (f)

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

(4) 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

or

x := x * 1

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

(5) 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.

X2 → X*X

(6) 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 → i++

i:=i-1 → i- -

4.4 Dominators

Loop:

Loop is any cycle in the flow graph of a program

Properties of a loop:

1. It should have a single entry node, such that al paths from outside the loop to any node in the loop go through the entry.
2. It should be strongly connected (ie) it should be possible to go from any node of the loop to any other, staying within the loop.

Dominators

The method of detecting loops in the flow graphs is described using the notion of Dominators.

In order to deduct the control flow within basic blocks, it's required to compute dominators. In the flow graph consisting a node D and E, if path leaves from D to E, then node D is said as dominating E. Dominator Tree: Dominator tree represents the hierarchy of the blocks and

its execution flow. Here, the initial node is taken as the root and every further parent is said to be intermediate dominator of child node.

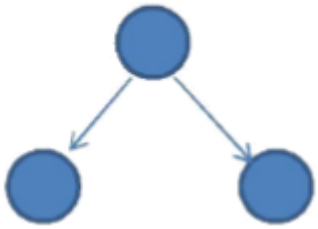


Fig. 4.6. Dominator Tree

In above tree, node 1 dominates 2 and 3.

Dominator Tree:

A useful way of presenting dominator information is in a tree called dominator tree, in which the initial node is the root and the parent of each other node is its immediate dominator

Immediate Dominator

The immediate dominator of ‘n’ is the closest dominator of ‘n’ on any path from the initial node to ‘n’.

Properties of Dominators

The algebraic properties regarding the dominance relation are:

(1) Dominance is a reflexive partial order. It is

Reflexive { a DOM [a for all a]}

Antisymmetric { a DOM b & b DOM a}

Transitive { a DOM b & b DOM c => a DOM c}

(2) The dominators of each node ‘n’ are linearly ordered by the DOM relation

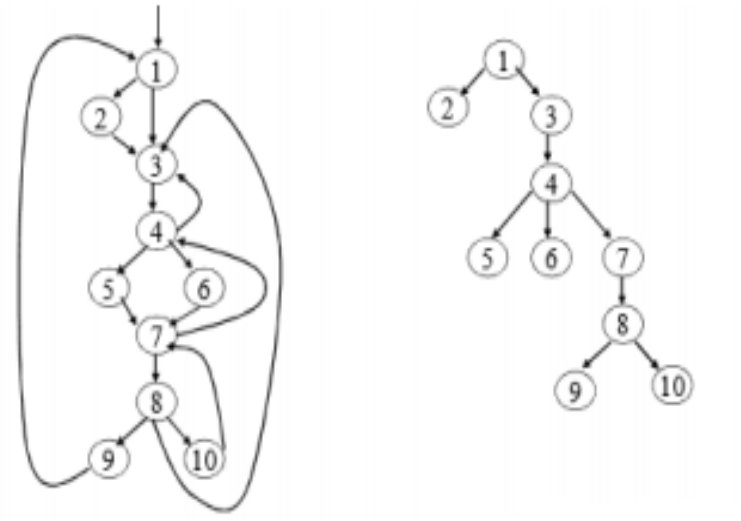
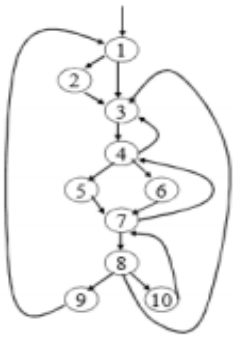


Fig. 4.7. Dominator Tree for the given flow graph

Dominator Computing Algorithm

```
begin
   D(n0)={n0}
   for n in N - {n0} do D(n)=N
   CHANGE=true
while CHANGE do
   begin
        CHANGE=false
        for n in N - {n0} do
            begin
                NEWD = {n} ⋃ ⋂ D(n)
                       Pa pred –
                       ecessor of n
                if D(n) ≠ NEWD then CHANGE=true
                D(n) = NEWD
            end
      end
```



Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8

Fig.4.8 Example for Dominators

4.5 Data flow analysis:

To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.

Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.In order to find out the value of any variable (A) of a program at any point (P) of the program, Global data flow analysis is required.

use-definition (or ud-) chaining is one particular problem.

- Given that variable A is used at point p,at which points could the value of A used at p have been defined?

- By a use of variable A means any occurrence of A as an operand.
- By a definition of A means either an assignment to A or the reading of a value for a.
- By a point in a program means the position before or after any intermediate language statement.
    - Control reaches a point just before a statement when that statement is about to be executed.
    - Control reaches a point after a statement when that statement has just been executed.

A definition of a variable **A** reaches a point **p ,**if there is a path in the flow graph from that definition to **p**, such that no other definitions of **A** appear on the path.To determine the definitions that can reach a given point in a program requires assigning distinct number to each definition.To compute two sets for each basic block **B:**

- **GEN[B]**-set of generated definitions within block **B.**
- **KILL[B]**- set of definitions outside of **B** that define identifiers that also have definition within **B**.

To compute the sets IN[B] and OUT[B]:

IN[B]-set of all definition reaching the point just before the first statement of block B.

OUT[B]-set of definitions reaching the point just after the last statement of B

Data Flow equations:

1. $OUT[B] = IN[B] - KILL[B] \bigcup GEN[B]$
2. $IN[B] = \bigcup_{P \text{ a pred} - \text{ecessor of } B} OUT[P]$

The rule (1) says that a definition d reaches the end of the block B if and only if either

i. d is in IN[B], i.e d reaches the beginning of B and is not killed by B ,or

ii. d is generated within B i.e., it appears in B and its identifier is not subsequently redefined within B.

The rule (2) says that a definition reaches the beginning of the block B if and only if it reaches the end of one of its predecessors.

Algorithm for reaching definition:

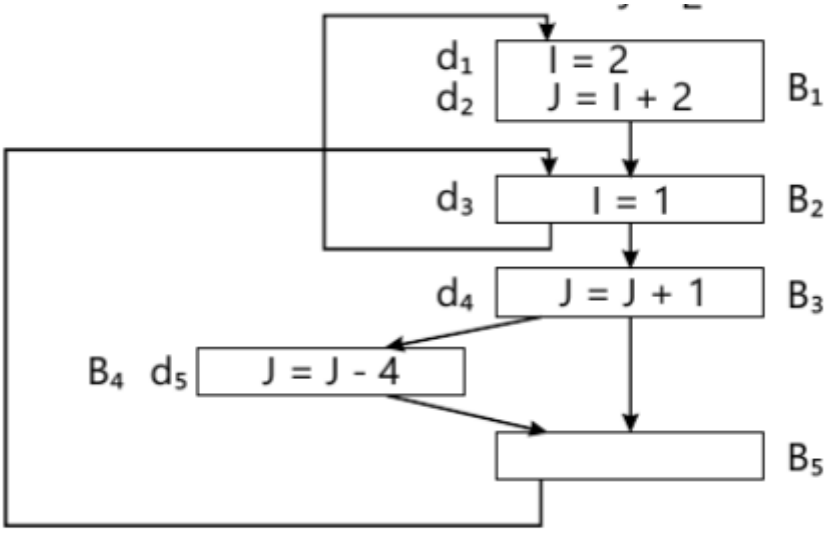
Input: A flow graph for which GEN[B] and KILL[B] have been computed for each block B.

Output: IN[B] and OUT[B] for each block B.

Method: An iterative approach, initializing with IN[B]=Ø for all B and converging to the desired values of IN and OUT.

```
begin
    for each block B do
    begin
            IN[B]=Ø
            OUT[B]=GEN[B]
    end
CHANGE=true
while CHANGE do
    begin
            CHANGE=false
            for each block B do
                begin
                NEWIN  =  ⋃ OUT [P]
                          P a pred - ecessor of B
                if NEWIN≠IN[B] then CHANGE=true
                IN[B]=NEWIN
                OUT[B]  = IN[B]  – KILL[B]  ⋃ GEN[B]
                end
```

Example: Consider the Flow graph:



Solution:

i) Generate GEN(B) and KILL(B) for all blocks:

Block B	GEN[B]	Bit vector	KILL[B]	Bit vector
B1	$\{d_1,d_2\}$	11000	$\{d_3,d_4,d_5\}$	00111
B2	$\{d_3\}$	00100	$\{d_1\}$	10000
B3	$\{d_4\}$	00010	$\{d_2,d_5\}$	01001
B4	$\{d_5\}$	00001	$\{d_2,d_4\}$	01010
B5	Ø	00000	Ø	00000

Computation For block B1 :

NEWIN=OUT[B2] , since B2 is the only predecessor of B1

NEWIN=GEN[B2]=00100 = IN[B1]

OUT[B1] = IN[B1] – KILL[B1] + GEN[B1]

= 00100 – 00111 + 11000

= 11000

For block B2 :

NEWIN=OUT[B1] + OUT[B5] , since B1 and B5 are the predecessor of B2

IN[B2] = 11000 + 00000 = 11000

OUT[B2] = IN[B2] – KILL[B2] + GEN[B2]

= 11000 – 10000 + 00100

= 01100

Initial pass

Block B	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass-1

Block B	IN[B]	OUT[B]
B1	00100	11000
B2	11000	01100
B3	01100	00110
B4	00110	00101
B5	00111	00111

Pass-2

Block B	IN[B]	OUT[B]
B1	01100	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

Pass-3

Block B	IN[B]	OUT[B]
B1	01111	11000
B2	11111	01111
B3	01111	00110
B4	00110	00101
B5	00111	00111

As iteration 4 and 5 produces same values, the process ends. Hence the value of definition anywhere at any point of time is deduced.

Computing ud-chains:

- From reaching definitions information we can compute ud-chains.
- If a use of variable A is preceded in its block by a definition of A, then only the last definition of A in the block prior to this use reaches the use.
  - i.e. ud-chain for this use contains only one definition.
  - If a use of A is preceded in its block B by no definition of A, then the ud-chain for this use consists of all definitions of A in IN[B].

- Since ud-chain take up much space, it is important for an optimizing compiler to format them compactly.

Applications:

- Knowing the use-def and def-use chains are helpful in compiler optimizations including constant propagation and common sub-expression elimination.
- Helpful in dead-code elimination.
- Instruction reordering.
- (Implementation of) scoping/shadowing.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V - Compiler Design – SCSA1604

V. CODE GENERATION

Issues involved in code generation – Register allocation – Conversion of three address code to assembly code using code generation algorithm – examples – Procedure for converting assembly code to machine code – Case study

Code Generation:

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

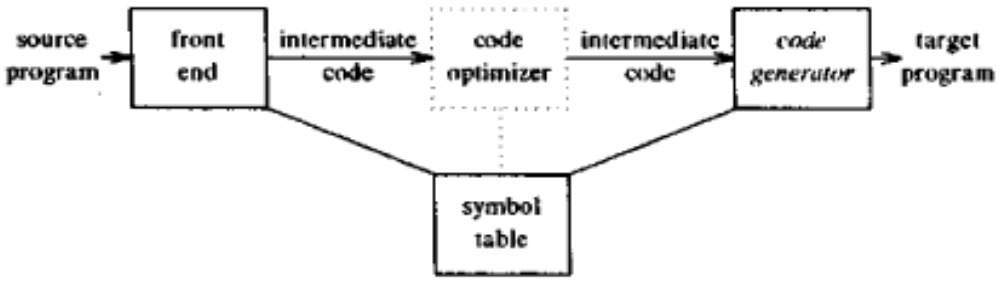


Figure 5.1 Position of code generator

- The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.
- 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.

ISSUES IN THE DESIGN OF A CODE GENERATOR:

The following issues arise during the code generation phase :

1)Input to code generator

2)Target program

3)Memory management 4)Instruction selection 5)Register allocation 6)Evaluation order

1.Input to code generator:

The input to the code generation consists of the intermediate representation of the source

program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

1)Linear representation such as postfix notation

2)Three address representation such as Quadruples

3)Virtual machine representation such as stack machine code

4)Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2.Target program:

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

a. Absolute machine language: It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language: It allows subprograms to be compiled separately.

c. Assembly language: Code generation is made easier.

3.Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions.

For example,

j : goto *i* generates jump instruction as follows :

if $i < j$, a backward jump instruction with target address equal to location of

code for quadruple *i* is generated.

if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

Instruction selection:

1. The instructions of target machine should be complete and uniform.

2. Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
3. The quality of the generated code is determined by its speed and size.

For example:

Every three-address statement of the form

x=y+z

where x, y and z are statically allocated.

Code sequence generated is shown as:

```
MOV y,R0   /* load y into register R0 */
ADD z,R0    /* add z to R0  */
MOV R0,x   /* store R0 into x */
```

Unfortunately, this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements,

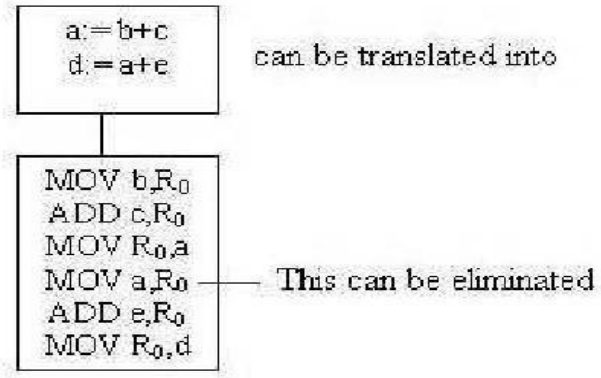


Figure 5.2 Code Translation

The quality of the generated code is determined by its speed and size.A target machine with a rich instruction set may provide several ways of implementing a given operation.

For example:

If the target machine has an "increment" instruction(INC), then the three address statement

a=a+1

may be implemented more efficiently by the single instruction,

INC a

instead of ,

MOV a,R0

ADD #1,R0

MOV R0,a

Register allocation:

Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two sub problems :

Register allocation – the set of variables that will reside in registers in the program is selected.

Register assignment - the specific register that a variable will reside is selected.

Certain machine requires even-odd register pairs for some operands and results. For example consider the division instruction of the form :

Div x, y

where, x – dividend in even register in even/odd register pair, y – divisor in even register holds the remainder odd register holds the quotient

Evaluation order:

At last, the code generator decides the order in which the instruction will be executed. The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

- Picking the best order is a difficult task.
- Initially avoid this problem by generating code for the three address statements in the order in which they have been produced by the intermediate code generator.
- It creates schedules for instructions to execute them.

When instructions are independent, their evaluation order can be changed to utilize registers and save on instruction cost. Consider the following instruction:

a+b-(c+d)*e

The three-address code, the corresponding code and its reordered instruction are given below:

Three-address code	Code	Reordered three-address code	Code	Inference

t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
	MOV a,R0
	ADD b,R0
	MOV R0,t1
	MOV c,R1
	ADD d,R1
	MOV e,R0
	MUL R1,R0
	MOV t1,R1
	SUB R0,R1
	MOV R1,t4
		t2:=c+d
		t3:=e*t2
		t1:=a+b
		t4:=t1-t3
			MOV c,R0
			ADD d,R0
			MOV e,R1
			MUL R0,R1
			MOV a,R0
			ADD b,R0
			SUB R1,R0
			MOV R0,t4
				The reordered instructions reduced the number of final code by 2 and thus saved in cost. The three-address code is reordered so that t1 is computed after computing t2 and t3. This reordering has saved in the instruction cost.

TARGET MACHINE:

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. The target computer is a byte-addressable machine with 4 bytes to a word. It has *n* general-purpose registers, R0, R1, . . . , Rn-1.It has two-address instructions of the form:

op source, destination

where, *op* is an op-code, and *source* and *destination* are data fields.

It has the following op-codes :

MOV (move *source* to *destination*)

ADD (add *source* to *destination*)

SUB (subtract *source* from *destination*)

The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Table 5.1 Mode and address allocation table

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+contents(R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c+ contents(R))	1

For example : *contents(a)* denotes the contents of the register or memory address represented by a. A memory location M or a register R represents itself when used as a source or destination.

e.g. MOV R0,M → stores the content of register R0 into memory location M.

Instruction costs :

Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.

Address modes involving registers have cost zero. Address modes involving memory location or literal have cost one. Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.

For example :

1. The instruction **MOV R0, R1** copies the contents of register R0 into R1. It has cost

one, since it occupies only one word of memory.

2. The (store) instruction **MOV R5,M** copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
3. The instruction **ADD #1,R3** adds the constant 1 to the contents of register 3,and has cost two, since the constant 1 must appear in the next word following the instruction.
4. The instruction **SUB 4(R0),*12(R1)** stores the value contents(contents(12+contents(R1)))-contents(contents(4+R0)) into the destination *12(R1). Cost of this instruction is three, since the constant 4 and 12 are stored in the next two words following the instruction.

For example :

MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.

The three-address statement a : = b + c can be implemented by many different instruction sequences :

```
MOV b, R0
ADD  c, R0
MOV R0, a                    cost = 6

MOV b, a
ADD c, a                     cost = 6
```

Assuming R0, R1 and R2 contain the addresses of a, b, and c :

```
MOV *R1, *R0
ADD  *R2, *R0                cost = 2
```

In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

Run-Time Storage Management:

Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.

The two standard storage allocation strategies are:

- Static allocation
- Stack allocation

In static allocation, the position of an activation record in memory is fixed at compile time.

In stack allocation, a new activation record is pushed onto the stack for each execution of a

procedure. The record is popped when the activation ends.

The following three-address statements are associated with the run-time allocation and deallocation of activation records:

- Call
- Return
- Halt

We assume that the run-time memory is divided into areas for:

- Code
- Static data
- Stack

Static allocation:

- In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.
- As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* +20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,

callee.static_area – Address of the activation record

callee.code_area– Address of the first instruction for called procedure

#here +20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :

GOTO * *callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation:

- Using the relative address, static allocation can become stack allocation for storage in activation records.
- The position of the record for an activation of a procedure is not known until run time.
- In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.
- The indexed address mode of target machine is used for this purpose.
- Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP     /* initializes stack */
Code for the first procedure
HALT /* terminate execution */
```

Implementation of Call statement:

```
ADD #caller.recordsize, SP          /* increment stack pointer */
MOV #here +16, *SP                  /*Save return address */
GOTO callee.code_area
```

where,

caller.recordsize – size of the activation record

#here +16 – address of the instruction following the GOTO

Implementation of Return statement:

```
GOTO * (SP ) /*return to the caller */
SUB #caller.recordsize, SP    /* decrement SP and restore to previous value */
```

A SIMPLE CODE GENERATOR:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement a := b+c can have the following sequence of codes:

ADD Rj, Ri Cost = 1 // if Ri contains b and Rj contains c

(or)

ADD c, Ri Cost = 2 // if c is in a memory location (or)

MOV c, Rj Cost = 3 // move c from memory to Rj

ADD Rj, Ri

Register and Address Descriptors:

A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three -address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z those registers will no longer contain y or z.

Generating Code for Assignment Statements:

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

t : = a – b

u : = a – c

v: = t + u

d := v + u

Code sequence for the example is:

Table 5.2 Shows the code sequence and the register allocation

Statements	Code Generated	Register descriptor	Address descriptor
t : = a – b	MOV a, R0 SUB b, R0	Register empty	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 R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Generating code for Indexed Assignments

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

Generating code for Pointer Assignments

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p: = a	MOV a, *Rp	2

Generating code for Conditional Statements

Conditional Statements are part of any programming construct to take an appropriate branch. Conditional jumps are implemented by finding out the value of the register. If the value of a register is negative, zero, positive, non-negative, non-zero, non-positive are the various possibilities to check to branch to a particular situation. The compiler typically uses a set of condition codes to indicate whether the computed quantity of a register is zero, positive or negative.

- First case of conditional statement: if x < y goto z - The code that is generated should involve subtracting ‘y’ from ‘x’ which is in register R and then jump to location ‘z’ if R is negative
- Second case of conditional statement: CMP x, y - Sets the condition code to positive

if x> y and so on.

- CJ < z - Jump to z if value is negative

Consider the following example:

x := y + z

if x < 0 goto z

The following would be the target code

MOV y, R0

ADD z, R0

MOV R0, x //x is the condition code

CJ < z

Register Allocation and Assignment:

Register allocation is only within a basic block. It follows top-down approach.

Local register allocation

- Register allocation is only within a basic block. It follows top-down approach.
- Assign registers to the most heavily used variables
- Traverse the block
- Count uses
- Use count as a priority function
- Assign registers to higher priority variables first

Need of global register allocation:

- Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
- To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Usage Counts:

A simple method of determining the savings to be realized by keeping variable x in a register for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a register.

An approximate formula for the benefit to be realized from allocating a register to x within a loop L is:

$$\sum_{blocks\ B\ in\ L} (use(x,B) + 2 * live(x,B))$$

where,

-use(x, B) is the number of times x is used in B prior to any definition of x;

-live(x,B) is 1 if x is live on exit from B and is assigned a value in B and 0 otherwise.

Example: Consider the basic block in the inner loop in Fig 5.3 where jump and conditional jumps have been omitted. Assume R0, R1 and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in the figure.

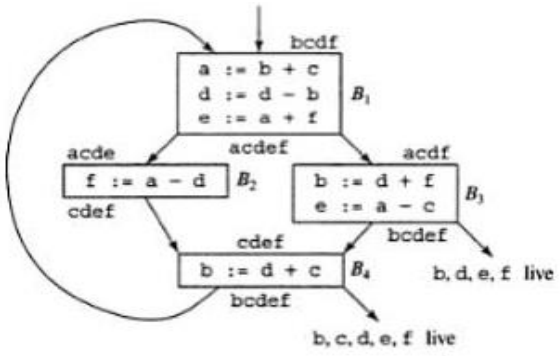


Fig 5.3 Flow graph of an inner loop

Fig 5.4 shows the assembly code generated from Fig 5.3.

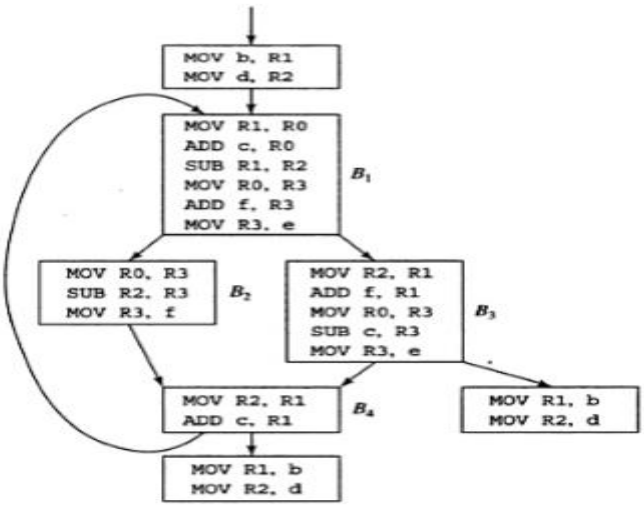


Fig 5.4 Code sequence using global register assignment