1. *Discuss on the relation between DFA and minimal DFA.*

2.18 Language for Specifying Lexical Analyzers-LEX

- For efficient design of compiler, various tools have been built for constructing lexical analyzers using the special purpose notations called **regular expressions.**
- Basically LEX is a unix utility which generates the lexical analyzer.
- A LEX lexer is very much faster in finding the tokens as compared to the handwritten LEX program in C.

- LEX scans the source program in order to get the stream of tokens and these tokens are related together so that various programming constructs such as expressions, block statements, procedures, control structures can be realised.
- The LEX specification file can be created using the extension .l(often pronounced as **dot L**). For example, the specification file can be **x.l** .
- This **x.l** file is then given to LEX compiler to produce **lex.yy.c**.
- This **lex.yy.c.** is a C program which is actually a **lexical analyzer program.** The LEX specification file stores the regular expressions for the tokens and the **lex.yy.c.** file consists of the **tabular representation** of the transition diagrams constructed for the regular expression. The lexemes can be recognized with the help of this tabular representation of transition diagram.
- Finally the compiler compiles this generated **lex.yy.c** and produces an object program **a.out.** When some input stream is given to a.out then sequence of tokens get generated. The above described scenario can be modelled below.

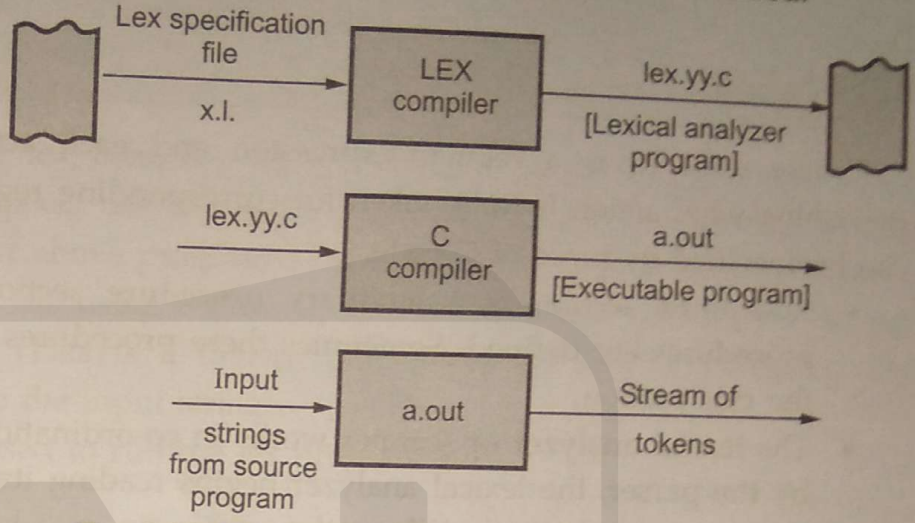


Fig. 2.18.1 Generation of lexical analyzer using LEX

2.18.1 Structure of LEX

Now the question arises how do we write the specification file? Well, the LEX program consists of three parts -

1. Declaration section **2. Rule section and** **3. Procedure section.**

```
%{
Declaration section
%}
%%
Rule section
%%
Auxiliary procedure section
```

- In the **declaration section** declaration of variable constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.
- The **rule section** consists of regular expressions with associated actions. These translation rules can be given in the form as -

```
R1    {action1}
R2    {action2}
.
.
.
Rn    {actionn}
```

Where each $\mathbf{R_i}$ is a regular expression and each **action**$_i$ is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of C code.

- And third section is a **auxiliary procedure** section in which all the required procedures are defined. Sometimes these procedures are required by the actions in the rule section.
- The lexical analyzer or scanner works in co-ordination with parser. When activated by the parser, the lexical analyzer begins reading its remaining input, character by character at a time. When the string is matched with one of the regular expressions R_i then corresponding action$_i$ will get executed and this action$_i$ returns the control to the parser.
- The repeated search for the lexeme can be made in order to return all the tokens in the source string. The lexical analyzer ignores white spaces and comments in this process. Let us learn some LEX programming with the help of some examples -

```
%{
%}
%%
      "Rama" |
      "Seeta" |
      "Geeta" |
      "Neeta" |          printf("\n Noun");
      "Sings" |
      "dances" |
      "eats"             printf ("\n Verb");
%%
      main ()
      {
            yylex();
      }
      int yywrap()
      {
            return 1;
      }
```

This is a simple program that recognizes noun and verb from the string clearly. There are 3 sections in above program.

- The section starting and ending with %, { and %} respectively is a **definition section**.

Q.1 Why white space characters are generally not allowed in identifier names?

Ans. : The lexical analyser scans the entire source program line by line and generates tokens. The finite amount of input is stored in an input buffer. The beginning pointer points to first character of the string and there is a forward pointer which go on reading each next character. And when forward pointer reads white space it declares all previous read characters as an identifier. Thus white spaces are important for lexical analysers because they serve as separators between two tokens. And if white spaces are allowed as identifiers then it will create difficulty for a lexical analyser to separate out token from source program. Hence white spaces are not allowed in identifier names.

Q.2 Discuss the significance of lexemes with longest prefix with suitable examples.

Ans. : The lexical analysis is a process of recognizing tokens from source program. And compiler does this job by constructing recognizer that looks for the lexemes stored

in the input buffer. This working is based on the rule - " If more than one pattern matches then recognizer has to choose the longest lexeme matched.

For example : If we have two input lexemes 'for' and 'formula' then because of the rule of matching longest pattern we can recognize 'formula' as an identifier and not a keyword. Because 'for' is a keyword and 'formula' is an identifier. If there would not be such rule then compiler would have returned keyword for the string 'formula', and not the identifier.

Q.3 With suitable example, discuss the significance of ordering of lexical rules.

Ans. : In the lexical analysis process, a specification file LEX is written in order to recognize the token. Suppose there is a lexeme "printf" which matches with rule [a-zA-Z]*. But this rule is generally given for identifying the identifiers. So if we write -

```
"printf"        {return KEYWORD ;}
[a-zA-Z]*       {return IDENTIFIER;}
```

"Then matching the first rule" will help to identify "printf" as keyword and not as identifier. Hence ordering of lexical rules in the LEX specification file is of great importance.

Q.4 What are the functions of lexical analyzer ?

Ans. : 1. It produces stream of tokens.

2. It eliminates blank and comments.

3. It generates symbol table which stores the information about identifiers, constants encountered in the input.

4. It keeps track of line numbers.

5. It reports the error encountered while generating the tokens.

Q.5 What are the issues of the lexical analyzer ? **AU : Dec.-09, May-13, 14, Marks 4**

Ans. : Various issues in lexical analysis are -

1) The lexical analysis and syntax analysis are separated out for simplifying the task of one or other phases. This separation reduces the bourdon on parsing phase.

2) Compiler efficiency gets increased if the lexical analyzer is separated. This is because a large amount of time is spent on reading the source file. And it would be inefficient if each phase has to read the source file. Lexical analyzer uses buffering techniques for efficient scan of source file. The tokens obtained can be stored in input buffer to increase performance of compiler.

3) Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer. The representation of non standard symbol can be isolated in lexical analyzer. For example in Pascal, ↑ can be isolated in lexical analyzer.

Q.6 What are sentinels ? What is its usage ? AU : May-04, Dec.-10

Ans. : Sentinels are special characters that are used to indicate an end of buffer. The sentinel is not a part of source language. One can avoid a test on end of buffer using sentinels at the end of buffer. For example : 'eof'.

Q.7 What is recognizer ? AU : Dec.-06

Ans. : Recognizers are machines. These are the machines which accepts the strings belonging to certain language. If the valid strings of such languages are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected. For example -

1. The finite state machines are recognizers for regular expressions or regular languages.
2. The push down automata are recognizers for context free languages.

Q.8 Define the term DFA.

Ans. : A deterministic finite automata (DFA) is a collection of following things -

1) A finite set of states denoted by Q.
2) The set of input symbols denoted by Σ
3) The start state $q_0 \in Q$.
4) The set of final states F such that $F \in Q$.
5) The mapping function δ This function takes two parameters - one is current state and other is input symbol. This function returns the next state. The DFA is denoted by $A = (Q, \Sigma, \delta, q_0, F)$.

Q.9 What is the difference between DFA and NFA ?

Ans. : The DFA is a deterministic finite automata whereas NFA is the non deterministic finite automata. In DFA, for the given state on given input we can reach to unique next state, but in case of NFA we can have more than one next states for the same input.

Q.10 What is the role of lexical analyzer ? AU : Dec.-11

Ans. : The lexical analyzer scans the source program and separates out the tokens from it.

Q.11 Give the transition diagram for an identifier. AU : Dec.-11

Ans. : The regular expression for denoting identifier will be -

r.e. = letter (letter + digit)*

The transition diagram will be

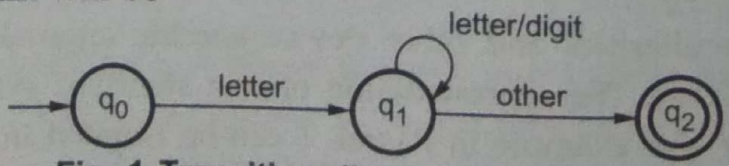


Fig. 1 Transition diagram for identifier

Q.12 What are the possible error recovery actions in lexical analyzer ? AU : May-12

Ans. : Various possible error recovery actions in lexical analyzer are -

i) Deleting extra characters that might appear in the source statement.

ii) Inserting missing character

iii) Interchanging two adjacent characters.

iv) Replacing an incorrect character by the correct character.

Q.13 What is the input to lexical analyzer generator ? What is its output ? AU : May-11

Ans. : The lexical analyzer generator takes the source program as input and generates the stream of tokens as output.

Q.14 Write the Regular expressions for identifier and number. AU : Dec.-12

Ans. : 1. Regular expression for identifier is

R.E.=letter(letter+digit)*

2. Regular expression for integer number is

R.E.=digit.digit*

Q.15 Define tokens, patterns and lexemes. AU : May-13

Ans. : Token : It describes the class or category of input string. For example, identifiers, keywords, constants are called tokens.

Pattern : The set of rules that describe the token.

Lexemes : It represents the sequence of characters in the source program that are matched with pattern of token.

Q.16 Write the regular expression for the identifier and whitespace. AU : Dec.-13

Ans. : Refer answer of Q.11 for regular expression

Regular expression for white space is

r.e. = (|\t\n)*

Q.17 Why is buffering used in lexical analysis ? What are the commonly used buffering methods ? AU : May-14

Ans. : The lexical analyzer reads the input source program and identifies the tokens from it. The input is read from the secondary storage. But reading the input in this way is costly. Hence the input source program are sentences are stored in the buffer. That is why buffering is required in lexical analysis.

The one buffer and two buffer schemes are the commonly used buffering methods.

Q.18 Define lexeme. (Refer answer of Q.15) AU : May-14

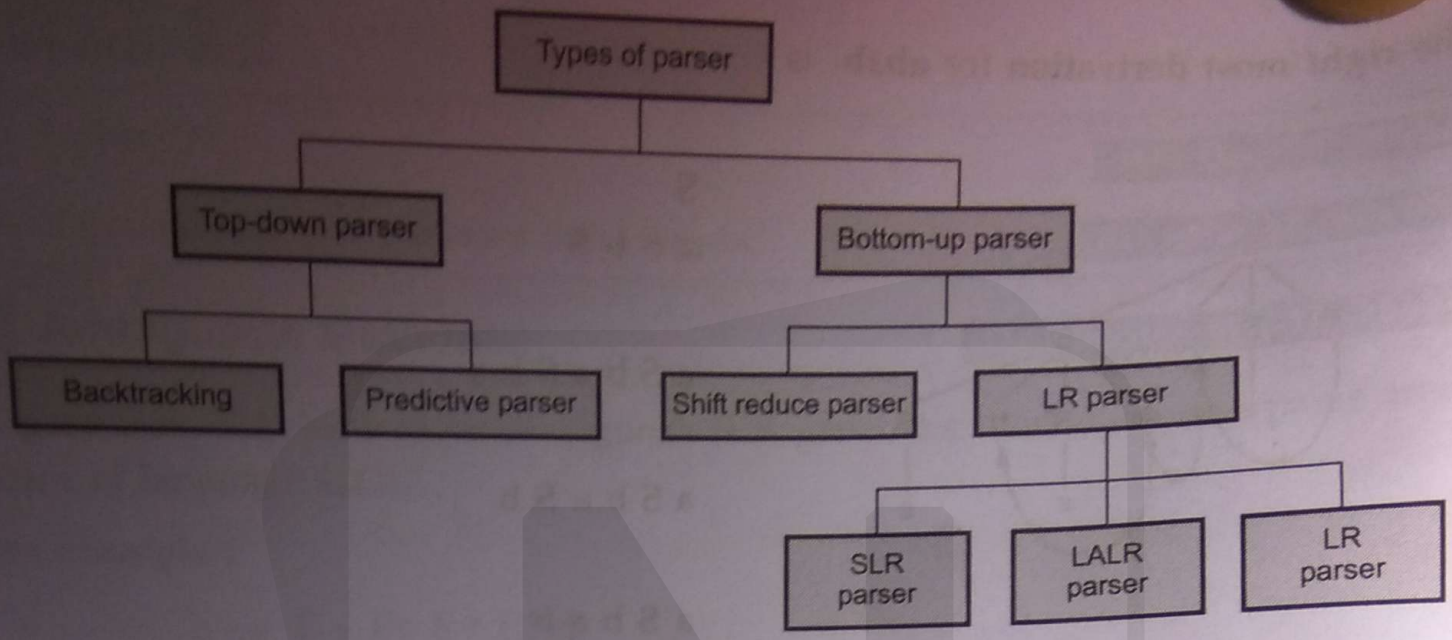


Fig. 3.6.1 Parsing techniques

Let us discuss the parsing techniques in detail.

Comparison between top down and bottom up parser

Sr. No.	Top down parser	Bottom up parser
1.	Parse tree can be built from root to leaves.	Parse tree is built from leaves to root.
2.	This is simple to implement.	This is complex to implement.
3.	This is less efficient parsing techniques. Various problems that occur during top down technique are ambiguity left recursion	When the bottom up parser handles ambiguous grammar conflicts occur in parse table.
4.	It is applicable to small class of languages.	It is applicable to a broad class of languages.
5.	Various parsing techniques are 1) Recursive descent parser 2) Predictive parser	Various parsing techniques are 1) Shift reduce 2) Operator precedence 3) LR parser.

3.15 Comparison of LR Parsers

AU : Dec.-10, Marks 2

It's a time to compare SLR, LALR and LR parser for the common factors such as size, class of CFG, efficiency and cost in terms of time and space.

Sr. No.	SLR parser	LALR parser	Canonical LR parser (CLR)
1.	SLR parser is smallest in size.	The LALR and SLR have the same size.	LR parser or canonical LR parser is largest in size.
2.	It is an easiest method based on FOLLOW function.	This method is applicable to wider class than SLR.	This method is most powerful than SLR and LALR.
3.	This method exposes less syntactic features than that of LR parsers.	Most of the syntactic features of a language are expressed in LALR.	This method exposes less syntactic features than that of LR parsers.
4.	Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Immediate error detection is done by LR parser.
5.	It requires less time and space complexity.	The time and space complexity is more in LALR but efficient methods exist for constructing LALR parsers directly.	The time and space complexity is more for canonical LR parser.

Graphical representation for the class of LR family is as given below :

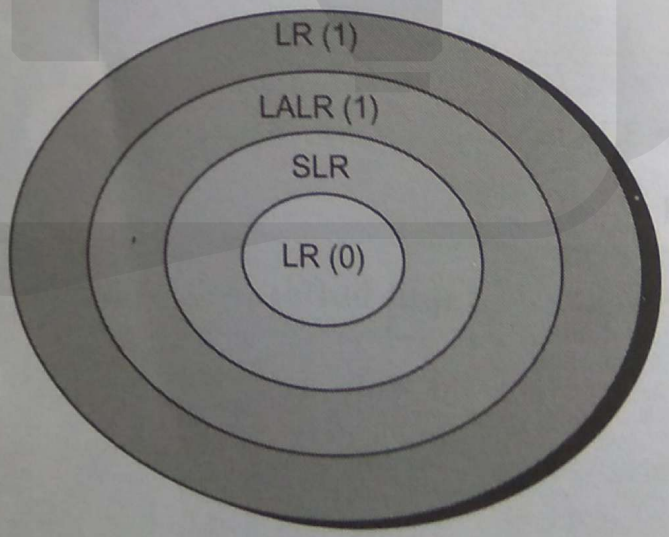


Fig. 3.15.1 Classification of grammars

3.18 Strategies to Recover from Syntactical Errors

AU : May-06, 07, 11, Marks 8

Parser employs various strategies to recover from syntactic errors. These strategies are

i) Panic mode ii) Phrase level iii) Error productions iv) Global correction

No one strategy is universally acceptable but can be applied to a broad domain. These strategies are given in detail as below -

i) Panic mode

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

ii) Phrase level recovery

- In this method, on discovering error parser perform local correction on remaining input.
- It can replace a prefix of remaining input by some string. This actually helps the parser to continue its job.

- The local correction can be replacing comma by semicolon, deletion of extra semicolons or inserting missing semicolon; The type of local correction is decided by compiler designer.
- While doing the replacement a care should be taken for not going in an infinite loop.
- This method is used in many error-repairing compilers.
- The drawback of this method is it finds difficult to handle the situations where the actual error has occured before the point of detection.

iii) Error production

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

iv) Global production

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

Global production is thus simply a theoretical concept.

Review Question

1. *Explain in detail about the error recovery strategies in parsing.*

AU : May-06, Marks 8 ; May-07, 11, Marks 6

3.19 YACC

We have discussed the manual method of construction of LR parser. This involves lot of work for parsing the input string. Hence there is a need for automation of this process in order to achieve the efficiency in parsing the input. Certain automation tools for parser generation are available. YACC is one such automatic tool for generating the parser program. YACC stands for **Y**et **A**nother **C**ompiler **C**ompiler which is basically the utility available from UNIX. Basically YACC is LALR parser generator. The YACC can report conflicts or ambiguities (if at all) in the form of error messages. In earlier chapter

we have seen one such tool LEX for lexical analyzer. LEX and YACC work together to analyse the program syntactically.

The typical YACC translator can be represented as shown in Fig. 3.19.1.

First we write a YACC specification file; let us name it as x.y. This file is given to the YACC compiler by UNIX command

```
yacc x.y
```

Then it will generate a parser program using your YACC specification file. This parser program has a standard name as **y.tab.c**. This is basically parser program in C generated automatically. You can also give the command with – d option

```
yacc – d x.y
```

By – d option two files will get generated one is **y.tab.c** and other is **y.tab.h**. The header file y.tab.h will store all the tokens and so you need not have to create y.tab.h explicitly. The generated y.tab.c program will then be compiled by C compiler and generates the executable a.out file. Then you can test your YACC program with the help of some valid and invalid strings.

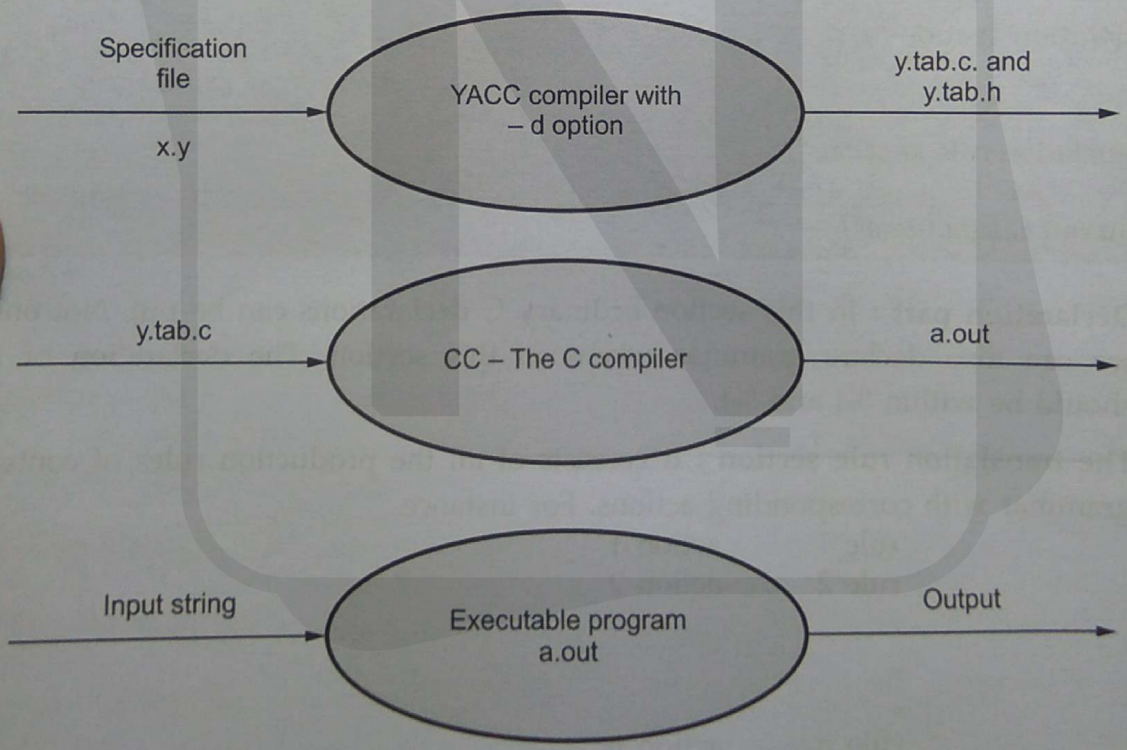


Fig. 3.19.1 YACC : Parser generator model

Writing YACC specification program is the most logical activity. This specification file contains the context free grammar and using the production rules of context free grammar the parsing of the input string can be done by y.tab.c.

Let us learn how to write YACC program.

Q.1 Explain various issues in parsing.

Ans : The first important issue in parsing is specification of syntax in programming language. It should be precise and unambiguous. The second important issue in parsing is representation of the input after parsing. The third crucial issue is use of parsing algorithm. The parsing algorithm can be using either the top down approach or using the bottom up approach.

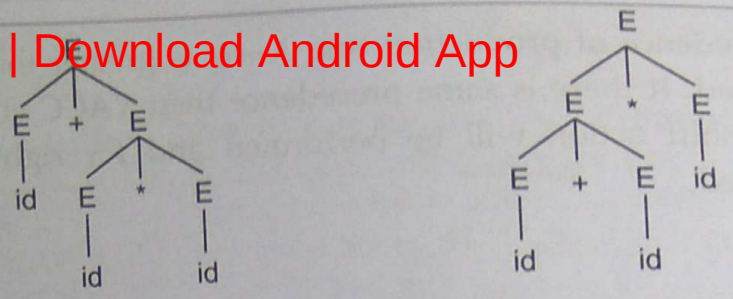
Q.2 Why lexical and syntax analyser are separated out ? AU : Dec.-05

Ans : The lexical analyzer scans the input program and collects the tokens from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages - Firstly it **accelerates** the process of **compilation** and secondly the **errors** in the source input can be **identified precisely.**

Q.3 What is ambiguous grammar ? AU : May-12

OR Define Ambiguous grammer. AU : May-16

Ans : The ambiguous grammar is a grammar in which more than one parse trees can be generated for the same input. For example - The string id+id*id can be represented as -



(a) Parse tree 1 (b) Parse tree 2

Fig. 3.1

Q.4 What is recursive descent parsing ?

Ans : Recursive descent parsing is a kind of parsing in which the recursive procedures can be used for parsing the given input string. It is a top down parsing technique.

Q.5 What is the function of parser ?

Ans : There are two important functions that can be performed by the parser and those are - parsing of input string and generating error messages if there exists errors in the input.

Q.6 What is handle ?

Ans : "Handle of right sentential form γ is a production A->β and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ ".

Q.7 What is handle pruning ?

AU : May-11, Dec.-11

Ans : In bottom up parsing the process of detecting handle and using them in reduction is called handle pruning. For example -

Consider the grammar,

$E \rightarrow E+E$

$E \rightarrow id$

Now consider the string id + id + id and the rightmost derivation is

$$E \underset{rm}{\Rightarrow} E + E$$
$$E \underset{rm}{\Rightarrow} E + E + E$$
$$E \underset{rm}{\Rightarrow} E + E + id$$
$$E \underset{rm}{\Rightarrow} E + id + id$$
$$E \underset{rm}{\Rightarrow} id + id + id$$

The bold strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	$E \rightarrow id$
E + id + id	id	$E \rightarrow id$

E + E + id	E + id	E → id
E + E + E	E + E	E → E + E
E + E	E + E	E → E + E
E		

Q.8 What are the two important rules used in shift reduce parsing ?

Ans: Following are the two rules that can be used in shift reduce parsing -

- If the incoming operator has more priority than in stack operator then perform **shift**.
- If in stack operator has same or less priority than the priority of incoming operator then perform **reduce**.

Q.9 What are the disadvantages of operator precedence parsing ?

AU : May-06, Dec.-06

Ans : Following are the disadvantages of operator precedence parsing -

1. The operator like minus has two different precedence (unary and binary). Hence it is hard to handle tokens like minus sign.
2. This kind of parsing is applicable to only small class of grammars.

Q.10 Explain the kernel and non kernel items

Ans : **Kernel items** - It is the collection of items S'-> • S and all the items whose dots are at the left end of the R.H.S of the rule.

Non Kernel items - It is the collection of all the items in which • are the left end of the R.H.S. of the rule.

Q.11 What is viable prefix ?

Ans : It is the set of prefixes in the right sentential form of a production A->α. This set can appear on the stack during shift/reduce action.

Q.12 What is the significance of lookahead operator?

Ans : The significance of lookahead **symbols is that the parser can decide the reductions based on the k lookahead symbols. Hence this parsing procedure becomes an efficient one.**

Q.13 Eliminate left recursion from the following grammar A→Ac|Aad|bd|c.
[Refer example 3.7.2] AU : May-13

Q.14 What should the error handler in a parser do ? AU : May-11

Ans. : The error handler in-parser reports the syntactical error if any.

Q.15 Which of the parser-bottom-up or top-down parser-is called LR parser ? Why is it called LR ? AU : May-11

Ans. : The LR parsers are called bottom up parsers because in this type of parsing the input is parsed from bottom to top.

Q.16 Compare syntax tree and Parse tree AU : Dec.-12

Ans: Parse trees are much more detailed representation of the source language than that of syntax trees. Syntax trees are smaller than parse trees and these are much space and time efficient.

Q.17 Write the rule to eliminate left recursion in a grammar AU : Dec.-12

Ans: If the left recursive grammar is

$$A \to A\alpha \mid \beta \quad \xrightarrow{\text{Eliminated as}} \quad A \to \beta A' \;\; A' \to \alpha A' \mid \varepsilon$$

Q.18 Eliminate the left recursion for the grammar
S->Aa | b
A->Ac | Sd | ε AU : Dec.-13

Ans. : The rule to eliminate left recursion is

if $A \to A\alpha \mid \beta$ then $A \to \beta A', A' \to \alpha A' \mid \varepsilon$.

$A \to Ac \mid Sd \mid \varepsilon$

We can replace S by Aa | b.

The grammar then becomes -

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

By eliminating immediate left recursion from A productions we get -

$S \to Aa \mid b$

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

Q.19 Write a CF grammar to represent palindrome. AU : Dec.-14

Ans : The context Free(CF) grammar to represent the palindrome is

CFG G=(V,T,P,S) where

V is set of nonterminal symbols = (S),

T is a set of terminal symbols =(a,b),

S is a start symbol

The set of production rules P = { S->aSa|bSb|a|b|∈}

Q.20 What is the role of parser ? AU : May-15

Ans : Parser collects the tokens from the lexical analyzer in order to check the syntax of the source code. If any error is detected during the analysis of syntax, the syntax error messages are also displayed by the parser.

Q.21 Write the algorithm for FIRST and FOLLOW in parser. AU : May-16

Ans : FIRST function

FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST (α).

Following are the rules used to compute the FIRST functions.

1. If the terminal symbol a then FIRST(a) = {a}.
2. If there is a rule $X \rightarrow \varepsilon$ then FIRST(X)={ε}.
3. For the rule $A \rightarrow X_1 X_2 X_3 \ldots X_k$ FIRST(A) = (FIRST(X_1) ∪ FIRST(X_2) ∪ FIRST(X_3)... ∪ FIRST(X_k))

   where k $X_j \leq n$ such that $1 \leq j \leq k-1$.

FOLLOW function

FOLLOW (A) is defined as the set of terminal symbols that appear immediately to the right of A. In other words

FOLLOW(A) = { a | $S \overset{*}{\Rightarrow} \alpha$ Aa β where α and β are some grammar symbols may be terminal or non-terminal}.

The rules for computing FOLLOW function are as given below -

1. For the start symbol S place $ in FOLLOW(S).
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without ε is to be placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and FIRST(β)= {ε} then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A). That means everything in FOLLOW(A) is in FOLLOW(B).

Q.22 Write a grammar for branching statements. AU : May-16

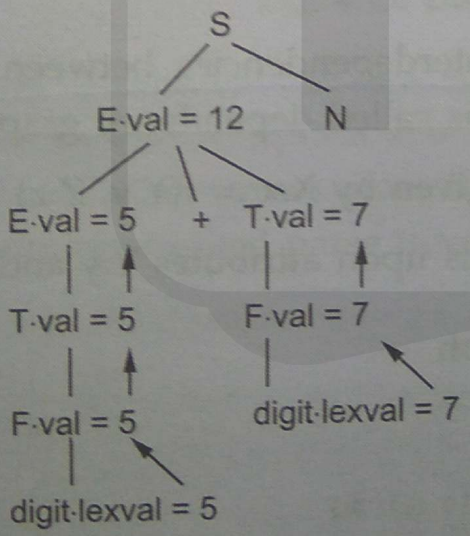
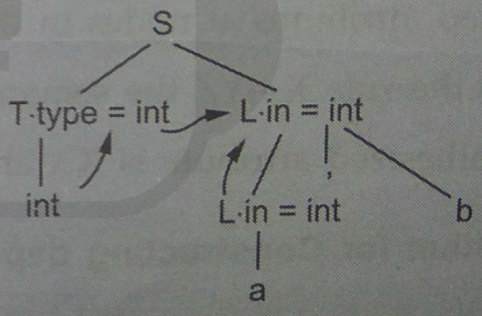
Ans : The context free grammar will be for if-else statement as follows

$S \rightarrow iEtS$

$S \rightarrow iEtSeS$

$S \rightarrow a$

$E \rightarrow b$

No.	Synthesized translation	Inherited translation
1.	The attribute 'a' is called synthesized attribute of X and $b_1, b_2, ..., b_k$ are attributes belonging to the production symbols. When $x \rightarrow \alpha$ is a rule associated with $a := f^*(b_1, b_2, ... b_k)$	The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production and $b_1, b_2, ..., b_k$ are belonging to either X or production α. When $x \rightarrow \alpha$ is a rule associated with $a := f^*(b_1, b_2, ... b_k)$
2.	In synthesized translation the synthesized attributes are computed.	In inherited translation, the inherited attributes are computed.
3.	The synthesized attributes are computed using **values of children.**	The inherited attributes are computed using values of **parent** and **sibling nodes.**
4.	Information is passed up in syntax tree.	Information is passed down in syntax tree.
5.	Computing synthesized attributes is **simple.**	Computing inherited attributes is **complex.**
6.	**Example :** 	**Example :** 

Ans. : i) Each identifier must be declared before the use

ii) The use of identifier must be within the scope.

iii) An identifier must not have multiple declarations at a time within the same scope.

Q.4 Mention the role of semantic analysis. AU : Dec.-12

Ans: The semantic analysis is carried out in order to get the precise meaning of programming constructs. The data type of the identifier is obtained by type checking system of semantic analysis phase.

Q.5 Give examples for static check. AU : May-13

Ans: The compiler enforces to check the static semantics of the programming constructs at compile time. This is known as static check. The typical examples of static check are -

1. Type checks
2. Flow of control checks
3. Uniqueness checks
4. Name related checks

Q.6 What is meant by Coercion ? AU : Dec.-13

Ans. : If the type conversion is done automatically by the compiler then it is called implicit conversion. The implicit conversions are also called as coercion.

Comparison between Quadruple, Triple and Indirect Triple

Quadruple	Triple	Indirect triple
The format of quadruple is (operator, operand1, operand2, result)	The format of triple is (operator, operand1, operand2)	The format of triple is (operator, operand1, operand2) but it makes use of **two tables.**

In this representation, the entries in the symbol table against the temporaries can be obtained. One can quickly access the value of temporary variables using symbol table.	In triple representation the **pointers** are used. By using pointers one can access directly the symbol table entry.	In indirect triple list of all references to computations is made separately and stored. Thus indirect triple and quadruple representations are similar as far as their utility is concerned. But indirect triple saves some amount of space as compared with quadruple representation.
The **quadruple** representation is beneficial for **code optimization.**	The **use of pointer** allows to **access the symbol table** entries quickly.	During the code optimization the **task of moving the instructions around and deleting the instructions** is involved. These are the easy tasks for quadruples but difficult for triples.

5.4 Types of Three Address Statements

AU : May-12, Marks 8

The form of three address code is very much similar to assembly language. Here are some commonly used three address codes for typical language constructs.

Language construct	Intermediate code form	Meaning
Assignment statement	x := y op z	Here binary operation is performed using operator 'op'.
Assignment statement	x := op y	Here the unary operation is performed. The operator 'op' is an unary operator.
Copy statement	x := y	Here the value of y is assigned to x.
Unconditional jump	goto L	The control flow goes to the statement labeled by L.
Conditional jump	if x relop y goto L	The relop indicates the relational operators such as <,=,>= .If x relop y is true then it executes goto L statement.
Procedure calls	param x_1 param x_2 ... param x_n call p,n	Here the parameters $x_1, x_2 \ldots x_n$ are used as parameters to the procedure p.
	return y	The return statement indicates the return value y.
Array statements	x:=y[i]	The value at i^{th} index of array y is assigned to x.
	x[i]:=y	The value of identifier y is assigned at the index i of the array x.
Address and pointer assignments	x:=&y	The value of x will be the address or location of y
	x:=*y	The y is a pointer whose value is assigned to x
	*x :=y	The r-value of object pointed by x is set by the *l*-value of y.

Q.4 What are the different ways of implementing the three address code ?

Ans. : The implementation of three address code can be done using :

1. Quadruple representation
2. Triple
3. Indirect Triple.

Q.5 Write down the equation for arrays.

Ans. : The equation for two dimensional arrays is :

$$A[i,j] = ((i * n2) + j) * w + (base - ((low1 * n2) + low2) * w)$$

Q.6 What is backpatching ? AU : May-07, 13; Dec.-09, 12

Ans : Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process.

Q.7 What is marker non terminal ?

Ans: The marker non terminal is a non terminal that is inserted in the production rule present in RHS. The purpose of marker non terminal is to mark the exact point where the semantic action is picked. The attribute **state** is associated with M and is used to record the address of the statement.

Q.8 Why are quadruples preferred over triples in an optimizing compiler ? AU : May-12

Ans. : In quadruple representation, the values of temporaries can be accessed using the symbol table. Hence the quadruples are preferred over triples in optimizing compiler.

Q.9 List out the motivations for back patching.

Ans. : i) In a Boolean expression suppose there are two expression E1 && E2. If the expression E1 is false then there is no need to compute rest of the expression. But if we generate the code for the expression E1 we must know where to jump on encountering the false E1 expression. But this can be accomplished by the backpatching technique.

ii) For the flow of control statements(such as if, else, while), the jumping location can be identified using the backpatching.

Q.10 What is the ideo behind generating three address codes during compilation ? AU : May-11

Ans. : The intermediate language is an easy form of source language which can be generated efficiently by the compiler. It leads to efficient code the generation and it acts as an effective mediator between front end and back end. Hence it is necessary to generate three address codes during compilation.

Q.11 What is DAG AU : Dec.-11, May-16

Ans. : Directed Acyclic Graph is similar to syntax tree but it is in compact form. It is a representation of an intermediate code.

3. Recursive procedures are not supported by this type of allocation.

6.4.2 Stack Allocation

a. Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called **control stack.**

b. As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.

c. The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.

d. The data structures can be created dynamically for stack allocation.

Limitations of stack allocation

The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static allocation.

6.4.3 Heap Allocation

a. If the values of non local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last In First Out nature. For retaining of such local variables heap allocation strategy is used.

b. The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object. This allocated memory can be deallocated when activation ends. This deallocated (free) space can be further reused by heap manager.

c. The efficient heap management can be done by

i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.

ii) Allocate the most suitable block of memory from the linked list. i.e. use best fit technique for allocation of block.

6.4.4 Comparison between Static, Stack and Heap Allocation

Sr.No.	Static allocation	Stack allocation	Heap allocation
1.	Static allocation is done for all data objects at compile time.	In stack allocation, stack is used to manage runtime storage.	In heap allocation, heap is used to manage dynamic memory allocation.

2.	Data structures can not be created dynamically because in static allocation compiler can determine the amount of storage required by each data object.	Data structures and data objects can be created dynamically.	Data structures and data objects can be created dynamically.
3.	**Memory allocation :** The names of data objects are bound to storage at compile time.	**Memory allocation :** Using Last In First Out (LIFO) activation records and data objects are pushed onto the stack. The memory addressing can be done using index and registers.	**Memory allocation :** A contiguous block of memory from heap is allocated for activation record or data object. A linked list is maintained for free blocks.
4.	**Merits and limitations :** This allocation strategy is simple to implement but supports static allocation only. Similarly recursive procedures are not supported by static allocation strategy.	**Merits and limitations :** It supports dynamic memory allocation but it is slower than static allocation strategy. Supports recursive procedures but references to non local variables after activation record can not be retained.	**Merits and limitations :** Efficient memory management is done using linked list. The deallocated space can be reused. But since memory block is allocated using best fit, holes may get introduced in the memory.

- FORTRAN compiler can create a number of **data areas** in which value of objects is stored. In FORTRAN, there is i) one data area for each procedure, ii) one data area for each named COMMON block and iii) one data area for blank COMMON block if used.
- The symbol table contain the entries for each static object. Conceptually, we bind the static objects to absolute addresses. The address of static object is denoted as a pair(DataArea, offset).
  - The **DataArea** denotes the starting address of the one of the COMMON block or it can be the start of a block of storage for the variables local to subroutine.
  - The **offset** denotes the position of the object relative to beginning of the data area.
- For COMMON blocks, record for each block must be kept during the processing of all procedures. This makes the storage allocation task somewhat **dynamic**.
- If procedures are separately compiled, ***a link editor*** must be used to select the size of the COMMON block.
- Then Equivalence operation is performed to specify the sharing of storage units by two or more entities in a program unit. This causes association of the entities that share the storage units. The equivalence operation is based on the equivalence algorithm.
- After performing equivalence operation a memeory map for each COMMON block is created by scanning the list of names present in that block
- **Example :** If a we create a record for a COMMON block named MyBlock, then its declaration is as follows -

COMMON /MyBlock/ x,y

The compiler will handle this declaration as follows -

1. The record for a block named **MyBlock** will be created in the symbol table if such entry is not created earlier.
2. In the symbol table entries for the data objects **x** and **y** a pointer to **MyBlock** is set to indicate that these data objects are in COMMON block and are actually the members of MyBlock.
3. However, if this is not the first declaration of MyBlock then simply link x and y to end of the list of names for MyBlock.
4. After procedures are processed, the equivalence algorithm will be applied to perform equivalence operation(to specify sharing of storage units).
5. Then a memory map for each COMMON block is created.

Q.1 Enlist the storage allocation strategies used in run time environment.

Ans. : The storage allocation strategies are -

1. Static allocation
2. Stack allocation
3. Heap allocation.

Q.2 What is the purpose of control stack used in run time storage organization ?

Ans. : The control stack is used to manage the active procedures. Managing the active procedures means that when a call occurs then the execution of activation is interrupted and information about the status of the control flow is saved. When the control returns from the call this suspended activation is resumed after storing the values of relevant registers.

Q.3 What are various ways to pass the parameters to the function ? AU : May-05

Ans. : Various ways to pass the parameters to the function are -

1. **Call by value :** In this method actual value of the parameter is passed.
2. **Call by Reference :** In this method the address of actual parameter is passed.
3. **Copy-restore :** This method is hybrid between call by value and call by reference.
4. **Call by name :** In this method procedure is treated like Macro.

Q.4 Suggest a suitable approach for computing the hash function. AU : Dec.-05

Ans. : Using hash function we should obtain exact locations of name in symbol table. The hash function should result in uniform distribution of names in symbol table. The hash function should be such that there will be minimum number of collisions. Collisions is such a situation where hash function results in same location for storing the names.

Q.5 What are the limitations of static allocation ? AU : Dec.-07, May-11

Ans. :

1. The static allocation can be done only if the size of data object is known at compile time.
2. The data structures can not be created dynamically. In other words, the static allocation can not manage the allocation of memory at run time.
3. Recursive procedures are not supported by this type of allocation.

7.10 Data Flow Analysis of Flow Graphs

AU : May-07, 10, 13, 14, Dec-09, 11, Marks 16

The data flow property represents the certain information regarding usefulness of the data items for the purpose of optimization. These data flow properties are -

1. Available expressions,
2. Reaching definitions,
3. Live variables,
4. Busy expressions.

The data flow analysis is a process of computing values of data flow properties.

In this section we will discuss some of the data flow properties in detail.

Review Questions

1. *Write about data flow analysis of structural programs.* **AU : May-07, 13, Marks 8**
2. *Explain the dataflow analysis concept with suitable example.* **AU : Dec-09, Marks 16**
3. *Write about data flow analysis of structural programs.* **AU : Dec.-11, Marks 8**
4. *How to trace data-flow analysis of structured program ? Discuss.* **AU : May-12, Marks 6**
5. *What is data flow analysis ? Explain data flow abstraction with examples.* **AU : May-14, Marks 8**

7.10.1 Data Flow Properties

Before discussing the data flow properties consider some basic terminologies that will be used while giving the data flow property.

- A program point containing the definition is called **definition point**.
- A program point at which a reference to a data item is made is called **reference point**.
- A program point at which some evaluating expression is given is called **evaluation point**.

For example :

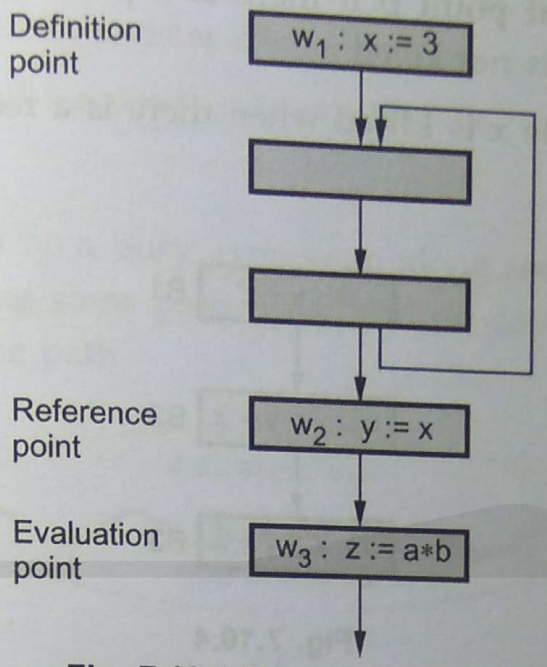


Fig. 7.10.1 Program points

1. Available expression : An expression x+y is available at a program point w if and only if along all paths are reaching to w.

1. The expression x+y is said to be available at its evaluation point.

2. The expression x+y is said to be available if no definition of any operand of expression (here either x or y) follows its last evaluation along the path. In other word, if neither of the two operands get modified before their use.

For example :

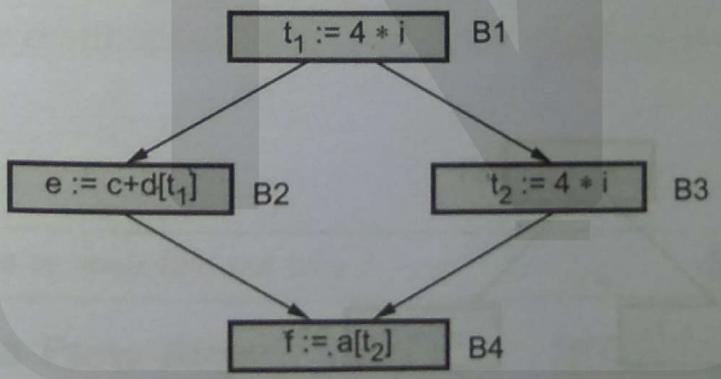


Fig. 7.10.2

The expression 4 * i is the available expression for B2, B3 and B4 because this expression is not been changed by any of the block before appearing in B4.

Advantage of available expression

The use of available expression is to eliminate common sub expressions.

2. Reaching definitions

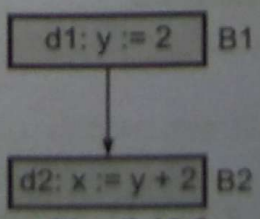


Fig. 7.10.3

A definition D reaches at point p if there is a path from D to p if there is a path from D to p along which D is not killed.

A definition D of variable x is killed when there is a redefinition of x.

For example :

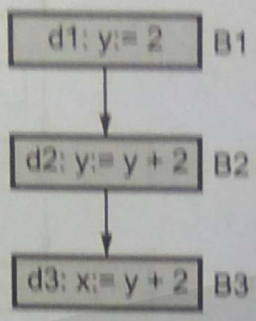


Fig. 7.10.4

The definition d1 is said to a **reaching definition** for block B2. But the definition d1 is not a reaching definition in block B3, because it is **killed** by definition d2 in block B2.

Advantage of reaching definition

- Reaching definitions are used in constant and variable propagation.

3. Live variable

A variable x is live at some point *p* if there is a path from *p* to the exit, along which the value of x is used before it is redefined. Otherwise the variable is said to be **dead** at that point.

For example :

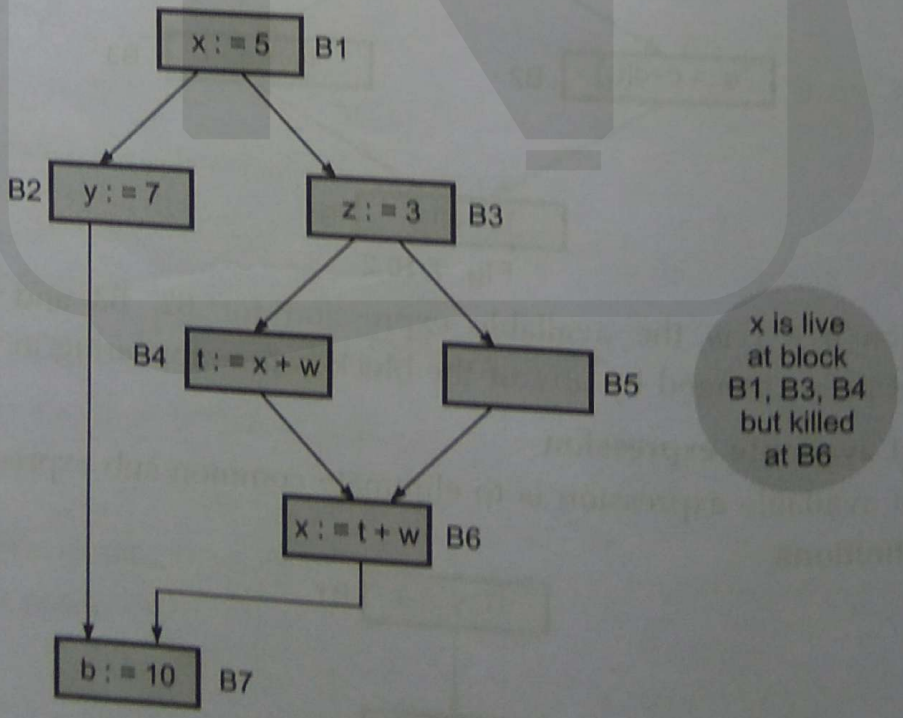


Fig. 7.10.5 Live variables

Advantages of live variables

- Live variables are useful in register allocation.
- Live variables are useful for dead code elimination.

4. Busy expression

An expression e is said to be a busy expression along some path $p_i \cdots p_j$ if and only if an evaluation of e exists along some path $p_i \cdots p_j$ and no definition of any operand exists before its evaluation along the path.

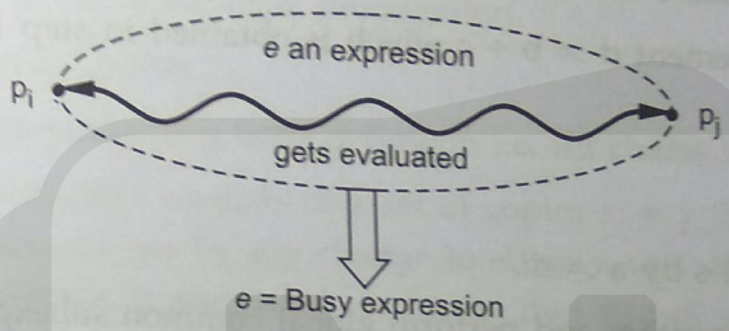


Fig. 7.10.6

Advantage of busy expression

- Busy expressions are useful in performing code movement optimization.

Key Point *By performing data flow analysis we get available expressions, reaching definitions, live variables and busy variables and these are useful for applying various global optimizing transformations such as global common sub expression elimination, copy propagations, code movement optimization, elimination of induction variables.*

Review Question

1. *Which variable is to be made live and why ?* **AU : May-10, Marks 2**

7.11 Efficient Data Flow Algorithms

AU : May-13, Dec.-09, Marks 16

7.11.1 Redundant Common Subexpression Elimination

We have already discussed "what is common subexpression ?" we have also seen how to eliminate such an expression while performing local optimization. In this section we will learn how to perform global optimization using transformations such as common subexpression elimination, copy propagation and induction variable elimination.

The *available expressions* allow us to determine if an expression at point p in a flow graph is common subexpression. Using following algorithm we can eliminate common subexpressions.

Q.1 What is the need of code optimization ?

Ans. : The code optimization is used to produce the efficient target code.

Q.2 What are the criteria that needs to be considered while applying the code optimization techniques ? AU : Dec.-11

Ans. : There are two important issues that need to be considered while applying the techniques for code optimization and those are,

1. The semantic equivalence of the program must not be changed. In simple words the meaning of program must not be changed.
2. The improvement over the program efficiency must be achieved without changing the algorithm of the program.

Q.3 Give the criteria for achieving machine dependant optimization.

Ans. : The machine dependant optimization obtained using following criteria -

i) Allocation of sufficient number of resources to improve the execution efficiency of the program.

ii) Using immediate instructions wherever necessary.

iii) The use of intermix instructions. The intermixing of instructions along with the data increases the speed of execution.

Q.4 Give the criteria for achieving machine independent optimization.

Ans. : The machine independent optimization can be obtained using following criteria.

i) The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.

ii) Use appropriate program structure in order to improve the efficiency of target code.

iii) Eliminate unreachable code from the source program.

iv) Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

Q.5 What are properties of optimizing compilers ? (Refer section 7.2.1) AU : May-06,13,16

Q.6. What are the transformation techniques that can be applied in code optimization ? AU : May-11

Ans. : Following are some commonly used transformations that can be applied during the code optimization -

1. Constant folding
2. Constant propagation
3. Common sub expression elimination
4. Variable propagation

5. Code movement
6. Strength reduction
7. Dead code elimination
8. Loop invariant computation
9. Loop optimization

Q.7 What is code motion ? AU : May-07, 08

Ans. : Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

For example -

```
while (i < Max – 1)
{
  sum = sum + a [i];
}
```

→

```
n = Max – 1
while (i < = n)
{
  sum = sum +a [i];
}
```

Q.8 Explain loop fusion or loop jamming.

Ans. : In loop fusion method several loops are merged into one loop.

For example

```
for i:=1 to n do
for j:=1 to m do
a[i, j]:=10
```

→

```
for i:=1 to nm do
a[i]:=10
```

Q.9 Define peephole optimization technique AU : Dec.-05

Ans. : Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

Q.10 What are structure preserving transformations ? AU : May-07, May-11

Ans. : The structure preserving transformation is a DAG based transformation. That means a DAG is constructed for the basic block then the above said transformations can be applied. The structure preserving transformations can be applied by applying some principle techniques such as common sub expression elimination, variable and constant propagation, code movement, dead code elimination.

Q.11 What are dominators ? (Refer section 7.8(1))

Q.12 What are natural loops ? (Refer section 7.8(2)

Q.13 What do you mean by available expression ?

Ans. : An expression x+y is available at a program point w if and only if along a paths are reaching to w.

1. The expression x+y is said to be available at its evaluation point.
2. The expression x+y is said to be available if no definition of any operand o expression (here either x or y) follows its last evaluation along the path. I other word, if neither of the two operands get modified before their use.

Q.14 Define the term evaluation point.

Ans. : A program point at which some evaluating expression is given is calle evaluation point.

Q.15 What are the advantages of using live variables ?

Ans. : Following are some advantages of using live variables -

1. Live variables are useful in register allocation
2. Live variables are used in dead code elimination.

Q.16 Write short note on global data flow analysis. AU : May-08

Ans. : The local optimization has a very restricted scope on the other hand the global data flow optimization is applied over a broader scope such as procedure or function body.

The data flow analysis is the analysis made on the data flow. The data flow analysis determines the information regarding the definition and use of the data in the program. The global data flow analysis is basically a process in which the values are computed using data flow properties. These data flow properties are -

- Available expressions
- Reaching definitions
- Live variables
- Busy variables

Q.17 What is dead code elimination ? AU : May-11

Ans. : Dead code elimination is an optimization technique in which the code containing the variable whose value is never used is removed.

Q.18 What is done in data flow analysis ? Why it is done ? AU : May-11

OR

What is data flow ananlysis ? AU : Dec.-12

Ans. : During the data flow analysis the information about the set of values is calculated at various points in a computer program. This is done in order to obtain optimization in the flow of data in the program execution.

Q.19 What is the use of algebraic identities in optimization of basic blocks ? AU : May-12

Ans. : The algebraic identities are used in peephole optimization techniques. Sometimes simple transformations can be applied on the code in order to optimize it. For example - instead of using 2*a we can use a+a similarly instead of using a/2 one can use a+0.5.

Q.20 List out two properties of reducible flow graph. AU : May-12

Ans. : Following are the two properties of reducible flow graphs -

i) The forward edges in reducible flow graph form an acyclic graph

ii) The reducible flow graphs contain the back edges whose head dominate their tail

Q.21 When does dangling reference occur ? AU : Dec.-11, May-12,16

Ans. : A reference in which the pointer is pointing to a variable which is deleted and pointer is still pointing to that memory location then it is called dangling reference.

Q.22 Define live variable AU : Dec-12

Ans. : A variable x is live at some point p, if there is path from p to the exit along which the value of x is used before it is redefined.

Q.23 What is constant folding ? AU : May-13

Ans. : Constant folding is an optimization technique in which the computation of constant is done at compile time instead of execution time.

Q.24 Define loop unrolling with example AU : Dec.-13

Ans. : The loop unrolling is the loop optimization technique in which the number of jumps and tests can be reduced by writing the code two times.

For example -

```
int i = 1;
while(i<=100)
{
a[i]=b[i];
i++;
}
```

```
int i = 1;
while(i<=50)
{
a[i]=b[i];
i++;
a[i]=b[i];
i++;
}
```

Q.25 What is an optimizing compiler ? AU : Dec.-13

Ans. : The optimizing compiler is a compiler that take minimum amount of time to execute the program and also takes less amount of memory for executing the program.

Q.26 Name the techniques in loop optimization AU : May-14

Ans. : Various techniques in loop optimization are -

1. Code motion

2. Induction variable and strength reduction
3. Loop invariant method
4. Loop unrolling
5. Loop fusion

Q.27 How would you represent the dummy blocks with no statements indicated in global data flow analysis ? AU : May-14

Ans. : The dummy blocks are represented with open circles with no statements within it, through which the control just flows before it enters and just before it leaves the region.

For example -

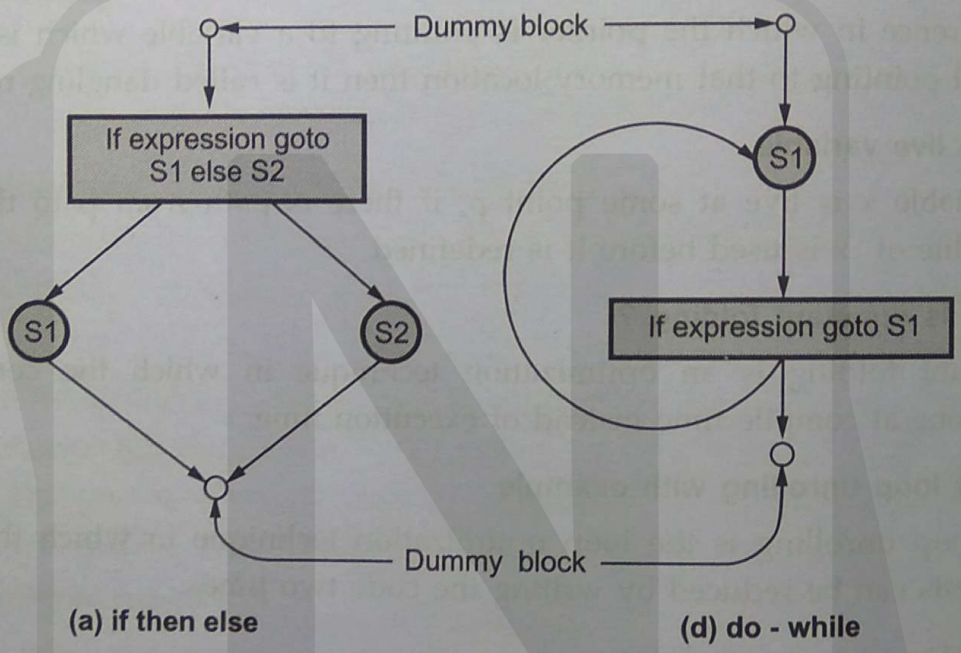


Fig. 7.1 Control construct representation

Q.28 Define basic blocks. AU : May-10, Dec.-11

Ans. : The basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

An example of the basic block is as shown below.

```
t1 := a5
t2 := t1+7
t3 := t2-5
t4 := t1+t3
t5 := t2+b
```

Q.29 Give an algorithm to determine the leaders into the basic block AU : May-08

Ans. : The leaders can be determined by using following rules -

1) The first statement is a **leader**.
2) Any target statement of conditional or unconditional goto is a leader.
3) Any statement that immediately follow a goto or unconditional goto is a leader.

Q.30 What are the applications of DAG ? AU : Dec.-06, May-13

Ans. : The DAG is used in

1. Determining the common sub-expressions (expressions computed more than once).
2. Determining which names are used inside the block and computed outside the block.
3. Determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

Q.31 List the advantage of DAG. [Refer Q.30] AU : Dec-12

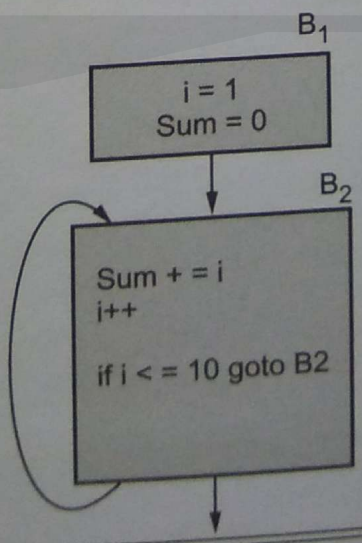
Q.32 Differentiate between basic block and flow graph. AU : Dec.-14

Ans. : Basic Block : The basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Flow Graph : Flow graph is directed graph in which the flow of control information is added to basic block. The nodes of flow graph are represented by basic blocks.

Q.33 Represent the following in flow graph i = 1, sum = 0; while (i < = 0) {Sum + = i; i ++ }. AU : Dec.-14

Ans. :



Q.34 How is Liveness of a variable is calculated? AU : May-15

Ans : Liveness of variable can be calculated as follows -

1. If variable 'a' is in use[n] then 'a' is live-in at node n.
2. If variable 'a' is live out at node n, then 'a' is live out at all nodes in pred[n].
3. If variable 'a' is live out at node n and not in def[n], then 'a' is also live-in at node n

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Review Questions

1. *Discuss about the code generation algorithm in detail.*

   AU : Dec.-06, Marks 10, May-12, Marks 8

2. *Describe in detail about a simple code generator with appropriate algorithm.*

   AU : Dec.-07, 11, May-13, Marks 8; May-11, Marks 10

3. *Explain - code generation phase with simple code generation algorithm.*

   AU : Dec.-14, Marks 8

4. *Write note on simple code generator.*

   AU : May-16, Marks 8

Two Marks Questions with Answers

Q.1 What are the properties of object code generation phase ?

Ans. : Following are the properties of object code generation -

- **Correctness** - It should produce a correct code and do not alter the purpose of source code.
- **High quality** - It should produce a high quality object code.
- **Efficient use of resources of the target machine** - While generating the code it is necessary to know the target machine on which it is going to get generated. By this the code generation phase can make an efficient use of resources of the target machine. For instance memory utilization while allocating the registers or utilization of arithmetic logic unit while performing the arithmetic operations.

- **Quick code generation** - This is a most desired feature of code generation phase. It is necessary that the code generation phase should produce the code quickly after compiling the source program.

Q.2 What are various forms of object code ?

Ans. : Various forms of object code are

1. Absolute code 2. Relocatable code 3. Assembler code

1. **Absolute code** - Absolute code is a machine code that contains reference to actual addresses within program's address space. The generated code can be placed directly in the memory and execution starts immediately.
2. **Relocatable code** - Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution with the help of a linking loader.
3. **Assembler code** - Producing an assembly language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help in generation of code.

Q.3 Explain various addressing modes used in code generation.

Ans. :

Addressing 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
literal	#c	c	1

Q.4 How do you calculate the cost of an instruction ? AU : Dec.-04, May-05

Ans. : The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "added cost".

Instruction	Cost	Interpretation
MOV R0,R1	1	Cost of register mode +1=0+1=1
MOV R1,M	2	Use of memory variable +1=1+1=2
SUB 5(R0),10(R1)	3	Use of first constant +use of second constant +1=3

Q.5 What is the use of temporary names in code generation process ?

Ans. : The temporaries are used to store the distinct values of the operands during the code generation. It optimizes the compilation process. Sometimes these temporaries can be reused during evaluation of expression.

Q.6 Write any four issues that should be clear before writing a code generator.

AU : May-11

Ans. : Four issues before writing a code generator are -

1. The code generation phase requires the complete error free intermediate code as input.
2. The output of the code generator phase is target code which comes in three forms - absolute machine language, relocatable machine language and assembly language.
3. The uniformity and completeness of instruction set is an important factor for the code generator.
4. The evaluation order is an important factor in generating an efficient target code.

Q.7 What are the uses of register and address descriptors in code generation ?

AU : Dec-12

Ans. : The register descriptor is used to keep track of the contents within the registers. The address descriptor stores the location where the current value of the name can be found at run time.

❑❑❑