

# PREVIOUS YEARS QUESTIONS

## PART-A

Q.1 Construct NFA to accept  $a(a/b)^*b$ . [R.T.U. 2016]

Ans. NFA =  $a(a/b)^*b$   
Starting state =  $q_0$   
Final state =  $q_2$   
Total No. of state = 3

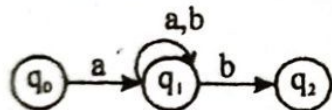


Fig.

Q.2 What do you mean by preprocessor?

Ans. **Preprocessor** : A source program may be divided into modules stored in separate files the task of collecting the source program is some times entrusted to a distinct program, called "Preprocessor". The preprocessor may also expand short hands called macros, into source language statements.

Q.3 Define static checkers. ✓

Ans. **Static Checkers** : A static checker reads a program, analyzes it and attempts to discover potential bugs without running the program.

Q.4 What is the role of code optimization?

Ans. **Code Optimization** : The code optimization phase attempts to improve the intermediate code, so that faster running-machine code will result.

Q.5 Explain the function of syntax analysis.

Ans. The "syntax analysis" imposes a hierarchical structure on the token stream, more specifically in tree structure.

## PART-B

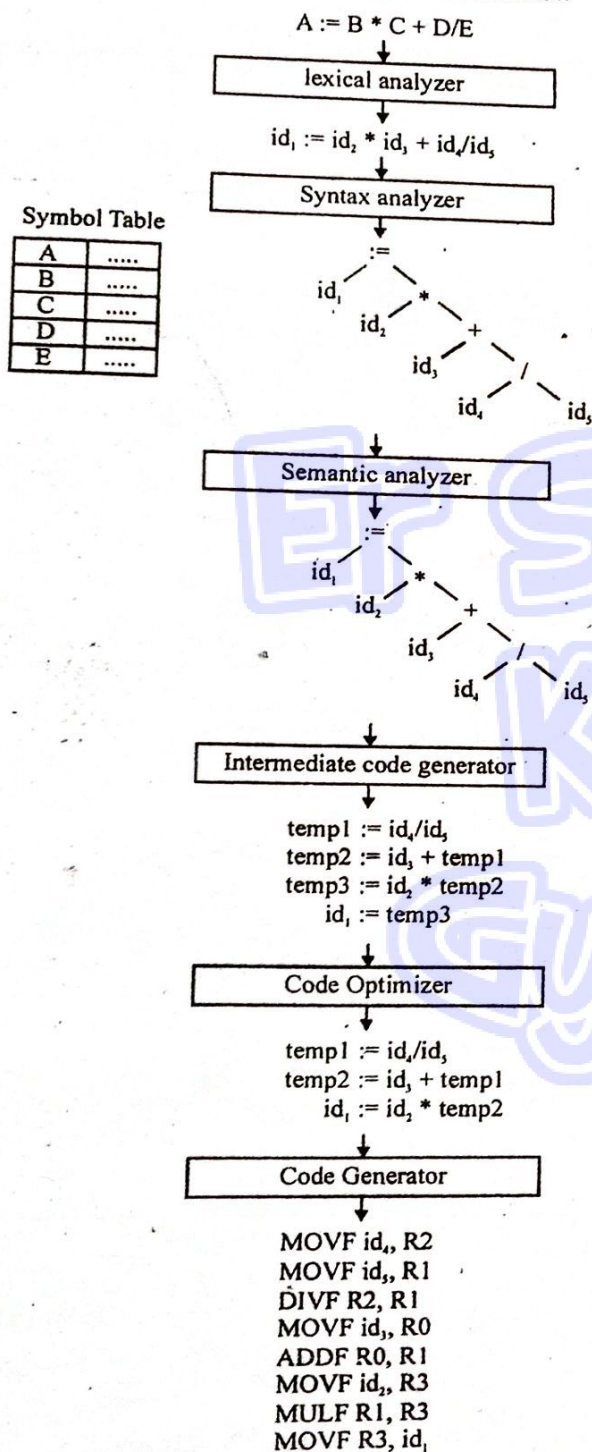
Q.6 Illustrate the translation of the following statement on all phase of compiler

$A := B * C + D / E$

[R.T.U. 2018, 2012]



Ans. Translation can take place as described below



Q.7 Explain the following terms :

(i) Translators, Compilers and Interpreters  
[R.T.U. 2018, 2012]

OR

Differentiate between compiler and interpreter.  
[R.T.U. 2013, 2010, 2008]

(ii) Bootstrapping

[R.T.U. 2018, 2012, 2010, 2008]

Ans.(i) Difference Between Translator, Compiler and Interpreter

**Translator:** A translator is a program that takes input as a program written in one programming language and converts it in output program of other programming language.

**Compiler :** Compilers are the translators which translate a program written in high level language like C++, FORTRAN, COBOL, PASCAL into machine code for some computer architecture. The generated code can be later executed many times against different data each time.



so in this sense compiler, compile the whole program at a time.

**Interpreter :** An interpreter reads an source program written in high level programming language as well as data for this program and it runs the program against the data to produce some results.

So we can say that interpreter compiles the source program line to line and an interpreter is generally slower than a compiler because it processes and interprets each statement in a program as many times as the number of the evaluation of this statement.

Table : Difference between Compiler and Interpreter

S. No.	Compiler	Interpreter
1.	Translates the whole program at once.	Translates one-by-one each statement of the program.
2.	Its output is in form of machine code.	Its output is an intermediate code.
3.	Its execution time is fast.	Execution time is slow.
4.	Needs more amount of memory.	Needs lesser memory.

(ii) **Bootstrapping :** A full assembler is itself a major piece of software, rather simple when compared with a



compiler for a really high level language, as we shall see. It is, however, quite common to define one language as a subset of another, so that subset 1 is contained in subset 2 which in turn is contained in subset 3 and so on, that is :

Subset 1  $\subseteq$  Subset 2  $\subseteq$  Subset 3 of ASSEMBLER

One might first write an assembler for subset 1 of ASSEMBLER in machine code, perhaps on a load-and-go basis (more likely one writes in ASSEMBLER and then hand translates it into machine code). This subset assembler program might, perhaps, do very little other than convert mnemonic opcodes into binary form. One might then write an assembler for subset 2 of ASSEMBLER in subset 1 of ASSEMBLER and so on. This process, by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known as "bootstrapping", by analogy with the idea that it might be possible to lift oneself off the ground by tugging at one's bootstraps.

In computing, bootstrapping (from an old expression "to pull oneself up by one's bootstraps") is a technique by which a simple computer program activates a more complicated system of programs. In the start up process of a computer system, a small program (such as BIOS) initializes and tests that a basic requirement of hardware, peripherals and external memory devices are connected. It then loads a program from one of them and passes control to it, thus allowing the loading of larger programs (such as an operating system). A different use of the term bootstrapping is to use a compiler to compile itself, by first writing a small part of a compiler of a new programming language in an existing language to compile more programs of the new compiler written in the new language. This solves the "chicken and egg" causality dilemma.

**Q.8** How can we represent tree as terms? Illustrate your explanation with an example.

[R.T.U. 2018, 2014]

**Ans.** A tree is the representation of a source code written in a particular programming language. Each node of a tree denotes a construct occurring in the source code. For instance, in the given example grouping parenthesis are implicit in the tree structure, and a syntactic construct like an if-else condition then expression may be denoted by means of a single node with three branches.

The tree is often build by a parser during the source code translation and compiling process. The following diagram illustrates the tree representation of if-else statement:

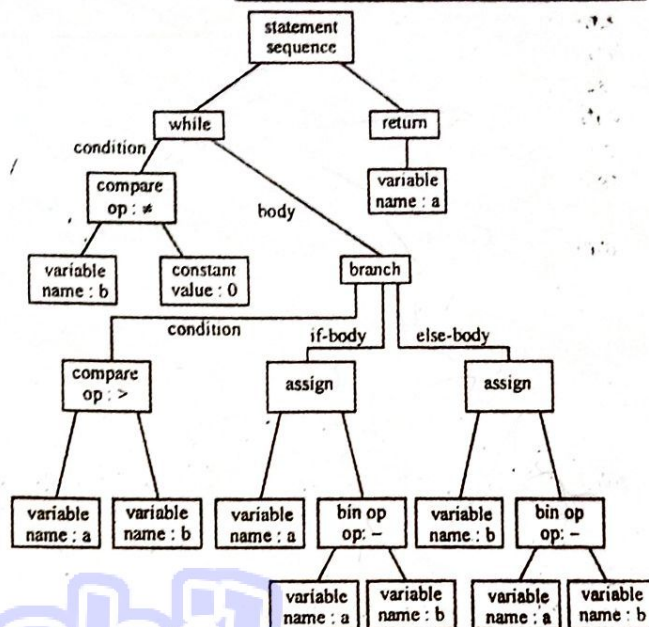


Fig.

A tree is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

The following things must be included while designing a tree for a source code :

- Variables types must be preserved as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

Some operations will always require two elements, such as the two terms for addition. However, some language constructs require an arbitrarily large number of children, such as argument lists passed to programs from the command shell. As a result tree has to be flexible enough to allow for quick addition for an unknown quantity of children.

Another major design requirement for a tree is that it should be possible to unparse a tree into source code form. The source code produced should be sufficiently similar to the original in appearance and identical in execution on recompilation.

- Q.9** (a) Discuss data structures used by a compiler.  
(b) Discuss various error recovery techniques for a lexical analysis.

[R.T.U. 2017]



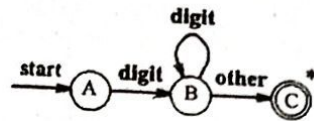


Fig. 4

**Q.11** Why do we need syntax trees when constructing compilers?

[R.T.U. 2014]

**Ans.** Syntax tree formation is one of the essential part to be carried out while constructing compilers to know that whether the grammar being inputted have a valid sequence of tokens.

Tokens are valid sequence of symbols, keywords, identifiers etc. The parser takes the token produced during the lexical analysis stage, and attempts to build a certain memory structure to represent that input. Frequently, that structure is an 'abstract syntax tree' (AST).

The parse tree is used to construct the AST which is precise representation of the program that is used by later phases in the compiler, in particular the type checker (for statically typed languages) and the code generator.

The advantage of AST over the program representations such as strings is that ASTs make access to the immediate sub-programs of a program easy, eg. The program has

If C then P else Q

Three immediate sub-programs namely C, P and Q. The AST for the program has three points to the sub-programs.

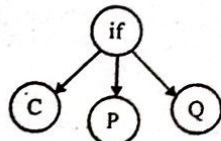


Fig. 1

Which provides exact information needed for efficient type checking and code-generation.

Let us consider an arithmetic expression  $4 * (3 + 17)$  in the obvious grammar of arithmetic expression:

$E \rightarrow E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid 2 \mid \dots$

Let us ignore the ambiguity and left recursion in that grammar. Parse tree for  $4 * (3 + 17)$  is

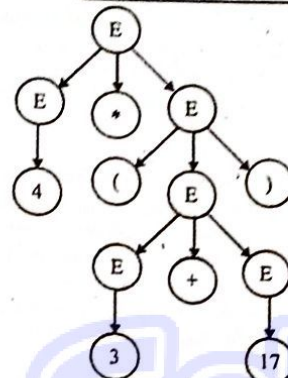


Fig. 2

The corresponding AST for above tree is :

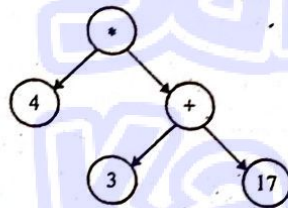


Fig. 3

Above example illustrates how syntax tree models the specific order in which operators are applied.

**Q.12** What are the fundamental differences between parse trees and abstract syntax tree?

[R.T.U. 2014]

**Ans.** Parse tree which is also known as concrete-syntax trees are a representation of grammars in a tree-like form.

- A parse tree pictorially shows how the start symbol of grammar derives a string in the language.
- CST is a one to one mapping from the grammar to a tree form.
- In parse tree, the '+' expression can look in different ways as :  
(a)  $2+3$       (b)  $(+2\ 3)$   
(c)  $(2\ 3\ +)$     (d) the sum of 2 and 3.
- Parse tree illustrates how tokens are grouped together, Basically the tree is initially constructed by the parser.
- The structure of the tree:
  - Nodes : Non-terminals
  - Leaves : Terminals

### Compiler Design

- Parse tree are defined by content free grammars  
 $\text{Exp} ::= \text{Exp} '+' \text{Exp};$   
 $\text{Exp} ::= '2';$   
 $\text{Exp} ::= '3';$
- Parse tree for the above expression will be drawn as :

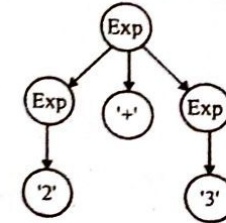


Fig. 1

### Abstract Syntax Tree (AST)

- AST are simplified syntactic representation of the source code, and they are most often expressed by the data structures of the language used for implementation.
- AST discard all the information that may be important for parsing the string, but isn't needed for analyzing it.
- AST are usually the last product of the front-end of a compiler.
- The expression in AST only shows the significant parts. For eg. A sum expression has its two operand expression as its significant part.

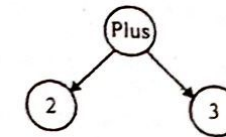


Fig. 2

- Abstract tree shows the semantically significant structure i.e. it gives the tree returned by the parser and manipulated by type checker.
- The structure of the tree:
  - Nodes : Constructor functions
  - Leaves : Atoms
- Abstract trees are defined by constructor type signatures.  
 $\text{Plus} : (\text{Exp}, \text{Exp}) \rightarrow \text{Exp}$   
 $2 : \text{Exp}$   
 $3 : \text{Exp}$



AST for the above expression will be drawn as fig. 3:

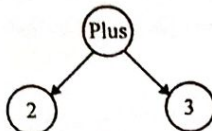


Fig. 3

**Q.13** What is finite automata? Explain NFA and DFA with an example. Construct NFA, that recognizes  $(a/b)^*abb$ . Also show that whether the string  $aabb$  is accepted by this NFA or not. [R.T.U. 2013]

**Ans. Finite Automata :** The finite automata are used as the mathematical model that can be used to recognize the regular expressions. The regular expressions are built to match the pattern of the lexeme to identify the tokens. In order to recognize tokens the regular expression can be created. These regular expressions then can be converted into the equivalent NFA. The NFA then can be converted to DFA. Input string is read character by character and lexical analysis is done using the finite automata so that valid tokens can be generated.

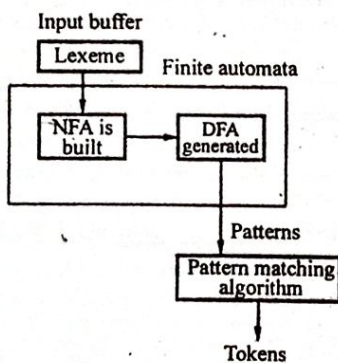


Fig.

#### Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA, for short) is a mathematical model that consists of:

1. A set of states  $S$ .
2. A set of input symbols  $\Sigma$  (the input symbol alphabet).
3. A transition function move that maps state-symbol pairs to sets of states.

4. A state  $s_0$  that is distinguished as the start (or initial) state.
5. A set of states  $F$  distinguished as accepting (or final) states.

An NFA can be represented diagrammatically by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function. This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol  $\epsilon$  as well as by input symbols.

#### Construction of NFA

The transition graph for an NFA that recognizes the language  $(a/b)^*abb$  is shown in fig. 1. The set of states of the NFA is  $\{0, 1, 2, 3\}$  and the input symbol alphabet is  $\{a, b\}$ . State 0 in Fig. 2 is distinguished as the start state and the accepting state 3 is indicated by a double circle.

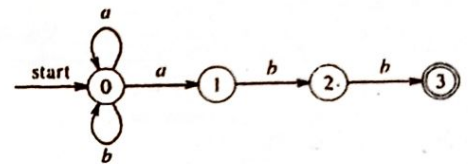


Fig. 1 : A nondeterministic finite automaton

In a computer, the transition function of an NFA can be implemented in several different ways, as we shall see. The easiest implementation is a transition table in which there is a row for each state and a column for each input symbol and  $\epsilon$ , if necessary. The entry for row  $i$  and symbol  $a$  in the table is the set of states (or more likely in practice, a pointer to the set of states) that can be reached by a transition from state  $i$  on input  $a$ . The transition table for the NFA of fig. 1 is shown in fig. 2.

The transition table representation has the advantage that it provides fast access to the transitions of a given state on a given character; its disadvantage is that it can take up a lot of space when the input alphabet is large and most transitions are to the empty set.

State	Input Symbol		
	A	B	C
0	{0, 1}	{0}	-
1	-	{2}	-
2	-	{3}	-

Fig. 2 : Transition table for the finite automaton of fig. 1

Adjacency list representations of the transition function provide more compact implementations, but access to a given transition is slower. It should be clear that we can easily convert any one of these implementations of a finite automaton into another.



An NFA accepts an input string  $x$  if and only if there is some path in the transition graph from the start state to some accepting state, such that the edge labels along this path spell out  $x$ . The NFA of fig. 3 accepts the input strings  $abb, aabb, babb, aaabb, \dots$ . For example,  $aabb$  is accepted by the path from 0, following the edge labeled  $a$  to state 0 again, then to states 1, 2 and 3 via edges labeled  $a, b$  and  $b$ , respectively.

A path can be represented by a sequence of state transitions called moves. The following diagram shows the moves made in accepting the input string  $aabb$ :

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

In general, more than one sequence of moves can lead to an accepting state. Notice that several other sequences of moves may be made on the input string  $aabb$ , but none of the others happens to end in an accepting state. For example, another sequence of moves on input  $aabb$  keeps re-entering the non-accepting state 0:

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

The language defined by an NFA is the set of input strings it accepts. It is not hard to show that the NFA of fig. 1 accepts  $(a/b)^* abb$ .

### PART-C

Q.14 Explain the various compiler phases in brief with suitable example. [R.T.U. 2018, 2014]

OR

Draw and discuss phases of a compiler. [R.T.U. 2017]

OR

What are the phases of a compiler? Explain the function of each phase in brief. [R.T.U. 2016]

OR

Explain the difference phases of compiler design with the help of suitable diagram? [R.T.U. 2015].

OR

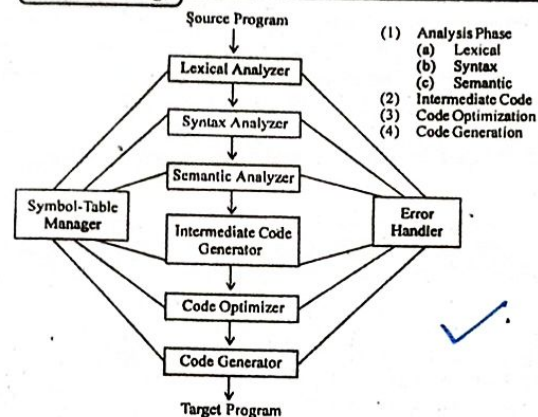
What are the different phases of compiler? Explain them with the help of suitable example. [R.T.U. Dec. 2013, 2012, 2011]

OR

Explain all phases of compiler with suitable example. [R.T.U. 2013, 2009, 2008]

**Ans. Phases of Compiler :** A compiler operates in phases each of which transforms the source program to the target program. A typical decomposition of a compiler are defined in fig. 1.

### Compiler Design



The first three phases, forming the bulk of the analysis portion of a compiler. Two other activities, symbol-table management and error handling, are shown interacting with the six phases of Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code generation, Code optimization and Code generation.

**Symbol-Table Management :** An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope and in case of procedure name, such things as the number and types of its arguments, the method of passing each argument and the type returned.

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or receive data from that record quickly.

When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.

**Example :** In a pascal declaration like :

var position, initial, rate: real;

The type real is not known when position, initial and rate are seen by lexical analyzer.

The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. The code generator typically enters and uses detailed information about the storage assigned to identifiers.

**Error Detection and Reporting :** Each phase can encounter errors. After detecting an error, a phase must

somehow deal with that error, so that compilation can proceed, allowing further errors in a source program to be detected.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g. if we try to add two identifiers, one of which is the name of an array and the other the name of a procedure.

**The Analysis Phases :** As the translation progress, the compiler's internal representation of the source program changes.

**Example :** Translation of statement

position := initial + rate \* 60

The "lexical analysis" phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters or a multi-character operator like  $=$ , the character sequence forming a token is called "lexical lexeme" for the token.

The lexical value associated with this occurrence of  $=$  points to the symbol table entry for  $=$ .

Here we shall use  $id1, id2$  and  $id3$  for position, initial and rate to emphasize that the internal representation of an identifier is different from the character sequence forming the identifier.

Thus we can write the statement as :

$id1 := id2 + id3 * 60$

We consider an intermediate form called "Three-address Code" which is like the assembly language for a machine in which every memory location can act like a register.

Three-address code consists of a sequence of instructions, each of which has at most three operands.

**Example :**

Fig. 2 shows the representation of statement for each phase :

position := initial + rate \* 60

temp1 := intto real (60)

temp2 :=  $id3 * temp1$

temp3 :=  $id2 * temp2$

$id1 := temp3$



The "syntax analysis" imposes a hierarchical structure on the token stream, more specifically in tree structure.

The "semantic analysis" phase basically checks the source program for the semantic errors gather type information for sub sequent code generation phase.

position := initial + rate \* 60

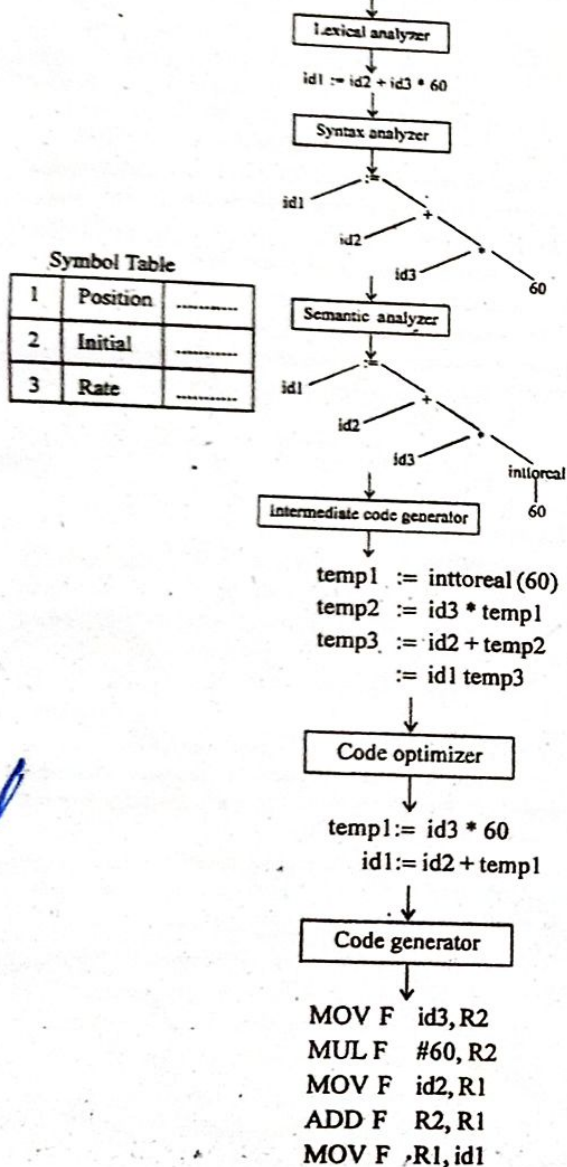


Fig. 2 : Compilation of "position := initial + rate \* 60"

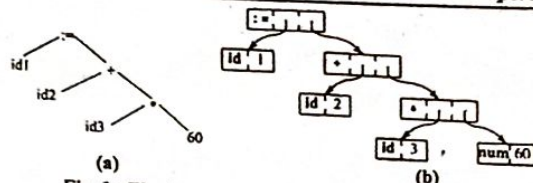


Fig. 3 : The data structure in (b) is for the tree in (a)

**Intermediate Code Generation :** After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. We use this intermediate representation for an abstract machine. This intermediate representation should have two important properties :

- It should be easy to produce.
- Easy to translate into the target program.

We consider an intermediate form called "Three-address code," which is like the assembly language for a machine in which every memory location can act like a register. Three-address code consists of sequence of instructions, each of which has at most three operands.

Three-address code

```
temp1 = intoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

...(1)

This intermediate form has several properties. First, each three-address instruction has at most one operator in addition to the assignment. Thus when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program. Second, the compiler must generate a temporary name to hold the value computed by each instruction. Third, some "three-address" instructions have fewer than three operands.

**Code Optimization :** The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial. For example, a natural algorithm generates the intermediate code (1), using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

```
temp1 := id3 * 60
id1 := id2 + temp1
```

temp3 := id1

...(2)

There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase. That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the

intoreal operation can be eliminated. Besides, temp3 is used only once, to transmit its value to id1. It then becomes safe to substitute id1 for temp3, whereupon the last statement of (1) is not needed and the code of (2) results.

There is great variation in the amount of code optimization different compilers perform. In those that do the most, called "optimizing compilers," a significant fraction of the time of the compiler is spent on this phase. **Code Generation :** The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly coded. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

For example, using register 1 and 2, the translation of the code of (2) might become

```
MOV F id3, R2
MUL F #60, R2
MOV F id2, R1
ADD F R2, R1
MOV F R1, id1
```

...(3)

The first and second operands of each instruction specify a source and destination, respectively. The F in each instruction tell us that instruction deal with floating-point numbers. This code moves the contents of the address id3 into register 2, then multiplies it with the real constant 60. The # signifies that 60 is to be treated as a constant. The third instruction moves id 2 into register 1 and adds to it the value previously computed in register 2. Finally, the value in register 1 is moved into the address of id1, so the code implements the assignment in fig.3(b).

- Q.15 (a) What do you understand by optimization of compiler.  
(b) Define ambiguity of grammar with suitable example.  
(c) What do you understand by loop optimization, and how it is done?

[R.T.U. 2017]

**Ans.(a) Optimization of Compiler :** An optimization is the process of transforming a piece of code into another functionally equivalent piece of code for the purpose of improving one or more of its characteristics. The two most important characteristics are the speed and size of the code. Other characteristics include the amount of energy required to execute the code, the time it takes to compile the code and, in case the resulting code requires Just-in-Time (JIT) compilation, the time it takes to JIT compile the code.



Q.19 What do you understand by "input buffering"?  
Explain "buffer pairs" and sentinels also.

[R.T.U. 2013]

*Reading*

OR

Explain the input buffering in brief.

[R.T.U. Dec. 2013]

### Ans. Input Buffering

We first mention a two-buffer input scheme that is useful when look-ahead on the input is necessary to identify tokens. Then we introduce some useful techniques for speeding up the lexical analyzer, such as the use of "sentinels" to mark the buffer end.

There are three general approaches to the implementation of a lexical analyzer.

1. Use of lexical analyzer generator, such as the Lex compiler to produce the lexical analyzer from a regular-expression based specification. In this case, the generator provides routines for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of input.

The three choices are listed in order to increasing difficulty for the implementor. Unfortunately, the harder to implement approaches often yield faster lexical analyzers. Since the lexical analyzer is the only phase of the compiler that reads the source program character by character, it is possible to spend a considerable amount of time in the lexical analysis phase, even though the later phases are conceptually more complex. Thus, the speed of lexical analysis is a concern in compiler design.

### Buffer Pairs

For many source languages, there are times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced. The lexical analyzers used a function `ungetc` to push lookahead characters back into the input stream. Because a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. Many buffering schemes can be used, but since the techniques are somewhat dependent on system parameters, we shall only outline the principles behind one class of schemes here.

We use a buffer divided into two N-character halves, as shown in Fig.1 Typically, N is the number of characters on one disk block, e.g., 1024 or 4096.



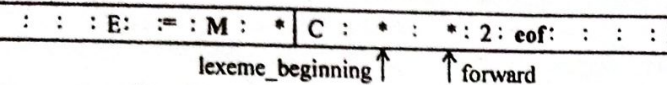


Fig. 1 : An input buffer in two halves

We read  $N$  input characters into each half of the buffer with one system read command, rather than 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 the buffer after the input characters, as in Fig. 1. That is, eof marks the end of the source file and is different from any input character. Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme. Initially, both pointers point to the first character of the next lexeme to be found. One called the forward pointer, scans ahead until a match for a pattern is found. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately past the lexeme. With this scheme, comments and white space can be treated as patterns that yield no token.

If the forward pointer is about to move past the halfway mark, the right half is filled with  $N$  new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with  $N$  new characters and the forward pointer wraps around to the beginning of the buffer.

This buffering scheme works quite well most of the time, but with it the amount of lookahead is limited and this limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer. For example, if we see

DECLARE (ARG 1, ARG2, ..., ARGn)  
in a PL/I program, we cannot determine whether DECLARE is keyword or an array name until we see the character that follows the parenthesis. In either case the lexeme ends at the second E, but the amount of look ahead needed is proportional to the number of arguments, which in principle is unbounded.

**Sentinels:** If we use the scheme of fig. 1 exactly as shown, we must check each time we move the forward pointer that we have not moved off one half of the buffer. If we do, then we must reload the other half. That is, our code for advancing the forward pointer performs tests like those shown in fig. 2.

```
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. 2 : Code to advance forward pointer

Except at the ends of the buffer halves, the code in fig. 2 requires two tests for each advance of the forward pointer. We can reduce the two tests to one if we extend each buffer half to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program. A natural choice is eof; fig. 3 shows the same buffer arrangement as fig. 1, with the sentinels added.

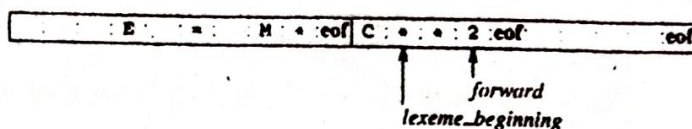


Fig. 3 : Sentinels at end of each buffer half

Most of the time the code performs only one test to see whether forward points to an eof. Only when we reach the end of a buffer half or the end of the file do we perform more tests. Since  $N$  input characters are encountered between eof's, the average number of tests per input character is very close to 1.

```
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. 4 : Lookahead code with sentinels

We also need to decide how to process the character scanned by the forward pointer; does it make the end of a token, does it represent progress in finding a particular keyword or what? One way to structure these tests is to use a case statement, if the implementation language has one. The test if forward ↑ = eof can then be implemented as one of the different cases.

□□□



# PREVIOUS YEARS QUESTIONS

2021	PART-A
2020	
2019	
2018	

Q.1 Consider the augmented expression grammar given below :

$E' \rightarrow E$

$E \rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow (E)/id$

If  $I$  is the set of two items  $[E' \rightarrow E]$  and  $[E \rightarrow E + T]$ , then calculate - goto ( $I, +$ ) [R.T.U. 2013]

Ans. If  $I$  is the set of two items  $\{[E' \rightarrow E], [E \rightarrow E + T]\}$ , then GOTO ( $I, +$ ) contains the items

$E \rightarrow E + .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

We computed GOTO ( $I, +$ ) by examining  $I$  for items with + immediately to the right of the dot.  $E' \rightarrow E$  is not such an item, but  $E \rightarrow E + T$  is. We moved the dot over the + to get  $E \rightarrow E + .T$  and then took the closure of this singleton set.

Q.2 What do you mean by parsing?

Ans. Parsing : Parsing is the "process of determining" if a string of token can be, generated by grammar.

There are different parsing method that can be applied to construct Syntax - directed translation.

Basically  $\rightarrow$  "top down method" and "bottom-up method," for parsing.

Q.3 What is the role of parser?

Ans. The Role of the Parser : The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

Q.4 Explain FIRST and FOLLOW approach.

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

Q.5 What is predictive parsing?

Ans. Predictive Parsing : Recursive - descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A special form of Recursive - descent - parsing is called predictive parsing, in which the look-ahead symbol unambiguously determines the procedure selected for each non-terminal.

	PART-B
--	--------

Q.6 What do you mean by context free grammar? Give distinction between regular and context free grammar and limitations of context free grammar. [R.T.U. 2018, 2012, Raj. Univ. 2006]



Write down a short note on the Context free grammar.  
[R.T.U. 2018, Dec. 2013]

**Ans. Context-free Grammars :** Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars. For example, we might have a conditional statement defined by a rule such as :

If  $S_1$  and  $S_2$  are statements and  $E$  is an expression, then

"if  $E$  then  $S_1$  else  $S_2$ " is a statement. ... (1)

This form of conditional statement cannot be specified using the notation of regular expressions; we saw that regular expressions can specify the lexical structure of tokens, using the syntactic variable  $stmt$  to denote the class of statements and  $expr$  the class of expressions, we can readily express (1) using the grammar-production

$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \quad \dots (2)$

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, each of the keywords *if*, *then* and *else* is a terminal.
2. Nonterminals are syntactic variables that denote sets of strings.  $stmt$  and  $expr$  are non-terminals. The non-terminals define sets of string that help define the language generated by the grammar. 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 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 (sometimes the symbol  $::=$  is used in place of the arrow), followed by a string of non-terminals.

#### Regular Expressions vs. Context-Free Grammars :

Every construct that can be described by a regular expression can also be described by a grammar. For example, the regular expression  $(a|b)^*abb$  and the grammar

$$A_0 \rightarrow aA_0 | bA_0 | aA_1$$

$$A_1 \rightarrow bA_2$$

$$A_2 \rightarrow bA_3$$

$$A_3 \rightarrow \epsilon$$

describe the same language, the set of strings of  $a$ 's and  $b$ 's ending in  $abb$ . We can mechanically convert a nondeterministic finite automaton (NFA) into a grammar that generates the same language as recognized by the NFA. 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  $A_i \rightarrow aA_j$ . If state  $i$  goes to state  $j$  on input  $\epsilon$ , The production  $A_i \rightarrow A_j$ . If  $i$  is an accepting state, introduce  $A_i \rightarrow \epsilon$ . If  $i$  is the start state, make  $A_i$  be the start symbol of the grammar.

Since every regular set is a context-free language, we may reasonably ask, "Why use regular expressions to the lexical syntax of a language"? There are several reasons:

1. The lexical rules of a language are frequently quite simple and to describe them we do not need a notation as powerful as grammars.
2. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.
3. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.
4. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

**Q.7 Show whether the following grammar is LL(1) or not**

$$E \rightarrow TE/ + TE/ \epsilon$$

$$T \rightarrow FT/ * FT/ \epsilon$$

$$F \rightarrow (E)/id$$

And explain the model of a predictive parser.

[R.T.U. 2018, 2012]

**Ans.** With the help of the grammar above, we can generate predictive parsing table

	Non-Terminal			Input Symbol		
	Id	+	*	(	)	\$
E	$E \rightarrow TE$	$E \rightarrow + TE$		$E \rightarrow ( TE$	$E \rightarrow )$	$E \rightarrow \$$
T	$T \rightarrow FT$	$T \rightarrow \epsilon$	$T \rightarrow * FT$		$T \rightarrow )$	$T \rightarrow \$$
F	$F \rightarrow id$			$F \rightarrow ( E$		

#### Parsing Table (M) for Grammar

A grammar whose parsing table has no multiply-defined entries is said to be LL(1). This grammar for arithmetic expression is LL(1).



"A predictive parser is an efficient way of implementing recursive descent parsing by handling the stack of activation records explicitly". Model of predictive parser is given below :

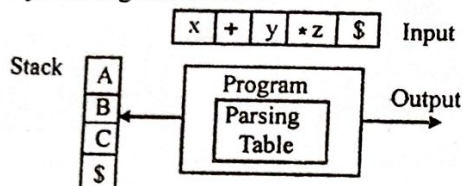


Fig.

The table-driven parser can be implemented using an input buffer, a stack and a parsing table.

- The input buffer is used to hold the string to be parsed. This string is followed by '\$' which is used as a right end marker to indicate the end of the input string.
- The stack is used to hold the sequence of grammar symbol. '\$' indicates bottom of the stack. Initially, the stack has the starting symbol of the grammar above '\$'.
- The parsing table is a two-dimensional array  $T[A, a]$  where  $A$  is the non-terminal and  $a$  is either a terminal or '\$'.

The parser is controlled by a program that behaves as follows :

The program determines  $X$  (the symbol on top of the stack) and  $a$  (the current input symbol). These two symbols decide the action of the parser. There are three possibilities :

- If  $X = a = \$$ , the parser announces successful completion of parsing and it halts.
- If  $X = a \neq \$$ , the parser pops  $X$  off to the stack and advances the input pointer to the next input symbol.
- If  $X$  is a non-terminal, the program consults the entry  $T[X, a]$  of the parsing table  $T$ . This entry will be either a  $X$ -production or an error entry. If  $T[X, a] = \{X \rightarrow UVW\}$  then the parser replaces  $X$  on top of the stack by  $WVU$  (with  $U$  on top.) If  $T[X, a] = \text{error}$ , the parser calls error recovery routine.

Q.8 Write down a short note on operator precedence parser for regular expression.

[R.T.U. 2018, Dec. 2013]

Ans. Operator Precedence Parser : It is bottom up parser which handles only a small class of grammars as defined below :

- There should not be any production with right side containing symbol  $\epsilon$ .
- No production can contain two adjacent on the right side.

Operator precedence parsing use there distinct precedence relation,  $<, \equiv, >$  as defined below :

Relation	Means
$a < b$	$a$ gives precedence to $b$
$a = b$	$a$ has some precedence as $b$
$a > b$	$a$ has more precedence than $b$

Example :  $E \rightarrow E+E|E|E|E^*E|E|E$   
 $E \rightarrow E|E$   
 $E \rightarrow (E)$   
 $E \rightarrow -E$   
 $E \rightarrow id$

have to parse the string  $id + id * id$ , then

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

- Step 1 :  $\$ < id > + < id > * < id > \$$   
 Step 2 : Scan from left to right till first  $>$  is encountered.  
 Step 3 : Scan backward till  $<$  is encountered.  
 Step 4 : By Step 2, and Step 3, we have  $id$  as a terminal and it can be reduced using

$E \rightarrow id$   
 $E + < id > * < id > \$$

- Step 5 : Repeat step 2 to we have  
 $E + E * < id > \$$

- Step 6 : Similarly  
 $E + E * E$

Now non terminal can be removed as follows :

- Step 7 :  $\$ < + < * > \$$   
 Step 8 : Repeating the above process,  
 $\$ < + > \$$  By ( $E \rightarrow E^*E$ )  
 Step 9 : and  $\$ \$$  By ( $E \rightarrow E+E$ )

Q.9 Write down a short note on YACC error handling in LR parser. [R.T.U. 2018, Dec. 2013]

Ans. YACC Error Handling : Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A



general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, YACC provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat : error ';' ;
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error '\n' { printf( "Reenter last line: " ); }
input { $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the

error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yerror;
```

in an action resets the parser to its normal mode. The last example is better written

```
input : error '\n'
      { yerror;
        printf( "Reenter last line: " ); }
input
      { $$ = $4; } ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error { resynch();
             yerror;
             yyclearin; } ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

**Q.10 Construct LALR parsing table for the following grammar.**

$S \rightarrow AA$

$A \rightarrow aA \mid b$

[R.T.U. 2017]

**Ans. LALR Parser :** LALR parser are same as CLR parser with one difference. In CLR parser if two states differ only in lookahead then we combine those states in LALR parser. After minimisation if the parsing table has no conflict that the grammar is LALR also.

Eg:

consider the grammar  $S \rightarrow AA$

$A \rightarrow aA \mid b$



**Q.12 Write a short note on operator precedence parsing and function.** [R.T.U. 2016]

**Ans. Operator Precedence Parsing :**

**Operator Grammar :** The grammar is handle by operator precedence parsing have the following property:

1. There should not be any production with right side containing symbol Null ( $\epsilon$ ).
2. No production can contain two adjacent non-terminal on the right side.
3. It use three disjoint precedence relation  $<, = \& .>$ .
4. If the grammar is ambiguous then only operator precedence can parse the grammar.

Relation	Meaning
$a < b$	'a' yields precedence to 'b'
$a = b$	'a' has equal precedence as 'b'
$a .> b$	'a' takes precedence over 'b'

**Example :** Construct operator precedence grammar.

$E \rightarrow E + E / E * E / id$  and Input  $\rightarrow id + id * id \$$



Operator Relation Table :

	id	+	*	\$
Higher Precedence Table → id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
Lower Precedence Table → \$	<	<	<	-

Size of Table ( $n^2$ )

E	E	E			
\$	id	+	id	*	id

Stack

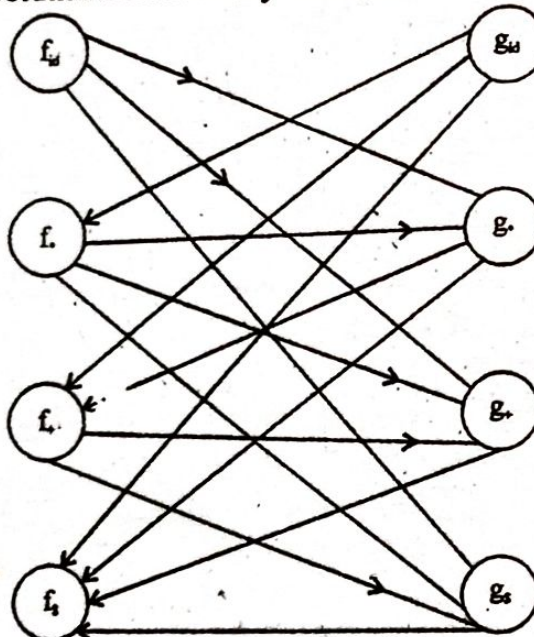
 $E + E * E$ 

- add \$ in stack.
- check operator table for entry [\$, id] if less precedence push otherwise pop.
- Repeat this step untill the end of string.

**Operator Function :**

Row is denoted by function = g

Column is denoted by function = f



No cycle in this graph

Longest path

 $f_{id} \rightarrow g_{+} \rightarrow f_{+} \rightarrow g_{+} \rightarrow f_{\$}$  $g_{id} \rightarrow f_{+} \rightarrow g_{+} \rightarrow fg \rightarrow gf \rightarrow f_{\$}$ 

Function Table

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

**Advantages :**

- Size of function table is less than  $2n$  relation type.
- Blank entry is present in relation which is known as error, which are less in function table.



Ans. (i) FIRST and FOLLOW

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

FIRST(E') = {+, ε}

FIRST(T') = {\*, ε}

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

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

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

(ii) Predictive parsing table for Grammar G

Non-Terminal	Input-Symbol					
	id	+	*	(	)	\$
E	E → TE'			E → (		
E'		E' → + TE'			E' → ε	E' → ε
T	T → FT'			T → (		
T'		T' → ε	T' → * FT'		T' → ε	T' → ε
F	F → id			F → (		

## PART-C

Q.15 Write down a short note on the difference between bottom up and top down parsing with suitable example.

OR

Explain top down and bottom up parsing techniques in detail.

Ans. **Top-down Parsing** : For a given input string, top-down parsing attempt to derive a string identical to it by successive application of grammar rules to the grammar's distinguished symbol.

When such a string is obtained a tree representing its derivation would be the syntax tree for the input string. Thus if w is the input string, a top-down parser determines a derivation sequence.

$$s \Rightarrow \dots \Rightarrow \dots \Rightarrow w$$

Where s is starting non-terminal in given grammar.

Basically top-down parsing attempts to find the left most derivation for the input string w, since string w is scanned by the parser left to right, one Symbol/Token at a time and the left-most derivations generate the leaves of the parse tree in left-to-right order, which matches the input scan order.

For example

$V_N = \{expr, term, rest\}$

$V_T = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \epsilon\}$

$expr \rightarrow term \text{ rest}$

$rest \rightarrow + term \text{ rest} \mid - term \text{ rest} \mid \epsilon$   
 $term \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

If we want to show the construction of the parse tree for the input string 9 - 5 + 2, then parse tree is as follows :

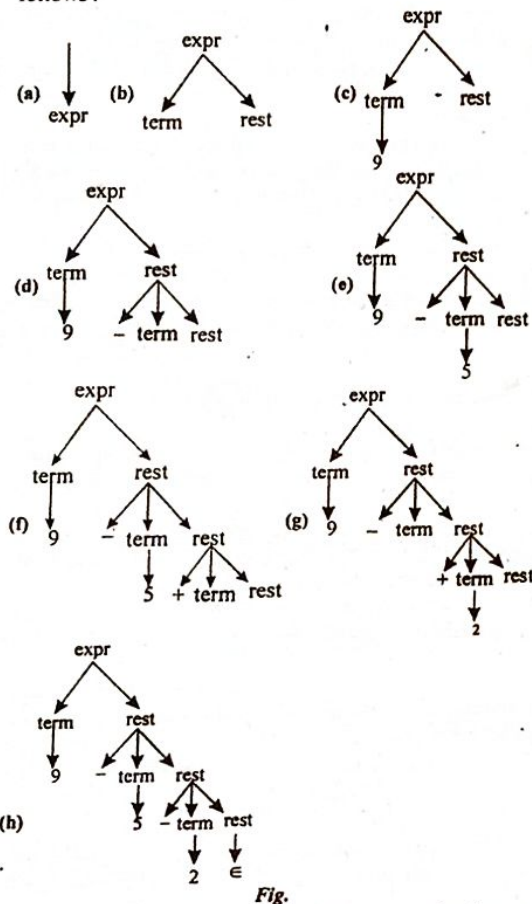


Fig.

So in this manner we show the input string by top-down parsing.

**Bottom-up Parsing** : Bottom-up parsing is an attempt to reduce the input string w to the start symbol of a grammar by tracing out the right-most derivations of w in reverse. This is equivalent to constructing a parse tree for the input string w in the reverse order.

Bottom-up parsing involves the selection of a substring that matches the right side of the production, whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation. That is, it leads to the generation of the previous right most derivation.

For Example :

Consider the grammar

$S \rightarrow aT U e$

$T \rightarrow Tbc/b$

$U \rightarrow d$

and let us find out the right most derivation of the sentence for the string abbcd e.

$S \Rightarrow aT U e$   
 $\Rightarrow aT d e$   
 $\Rightarrow aTbc d e$   
 $\Rightarrow abbcd e$

Q.16 Give the model for LR parser and explain its actions.

OR

What do you mean by LR parser? What is the model of an LR parser? Explain.

OR

Give the model for LR parser and explain its action.

Ans. **The LR Parsing Algorithm**

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 parsers; only the parsing table changes from one parser to another. The parsing program reads characters from input buffer one at a time. The program uses a stack to store a string of the form  $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$ , where  $s_m$  is on top. Each  $X_i$  is a grammar symbol and each  $s_i$  is a symbol called a state.

Each state symbol summarizes the information contained in the stack below it and the combination of the state symbol on, top of the stack and the current input symbol are used to index the parsing table and determine the shift- reduce parsing decision. In an implementation, the grammar symbols need not appear on the stack; however, we shall always include them in our discussions to help explain the behavior of an LR parser.

The parsing table consists of two parts, a parsing function action and a goto function. The program driving the LR parser behaves as follows. It determines  $s_m$ , the state currently on top of the stack and  $a_i$ , the current input symbol. It then consults action  $[s_m, a_i]$ , the parsing action table entry for state  $s_m$  and input  $a_i$ , which can have one of four values :

1. Shifts, where s is a state,
2. Reduce by a grammar production  $A \rightarrow \beta$ ,
3. Accept and
4. Error

The function goto takes a state and grammar symbol as arguments and produces a state, the goto function of a



most derivation of the

parser and explain its  
[R.T.U. 2016]

parser? What is the  
plain. [R.T.U. 2013]

ser and explain its  
[Raj. Univ. 2005, 2004]

ut, a stack, a driver  
as two parts (action  
he same for all LR  
ges from one parser  
ads characters from  
gram uses a stack to  
..  $X_m s_m$ , where  $s_m$  is  
ol and each  $s_i$  is a

s the information  
combination of the  
d the current input  
table and determine  
an implementation,  
pear on the stack;  
n in our discussions  
parser.

vo parts, a parsing  
he program driving  
determines  $s_m$ , the  
and  $a_i$ , the current  
[ $s_m, a_i$ ], the parsing  
input  $a_i$ , which can

on  $A \rightarrow \beta$ ,

l grammar symbol  
goto function of a

parsing table constructed from a grammar G using the SLR, canonical LR or LALR method is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G which are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right most handle. The initial state of this DFA is the state initially put on top of the LR parser stack.

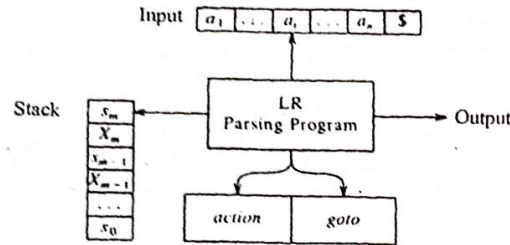


Fig. 1 : Model of an LR parser

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i a_{i+1} \dots a_n \$)$$

This configuration represents the right-sentential form  $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$

in essentially the same way as a shift-reduce parser would, only the presence of states on the stack is new.

The next move of the parser is determined by reading  $a_i$ , the current input symbol and  $s_m$ , the state on top of the stack and then consulting the parsing action table entry  $action[s_m, a_i]$ . The configurations resulting after each of the four types of move are as follows:

1. If action  $[s_m, a_i] = \text{shift } s$ , the parser executes a shift move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s a_{i+1} \dots a_n \$)$$

Here the parser has shifted both the current input symbol  $a_i$  and the next state  $s$ , which is given in action  $[s_m, a_i]$ , onto the stack;  $a_{i+1}$  becomes the current input symbol.

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

$$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s a_{i+1} \dots a_n \$)$$

where  $s = \text{goto}[s_{m-r}, A]$  and  $r$  is the length of  $\beta$ , the right side of the production. Here the parser first popped  $2r$  symbols off the stack ( $r$  state symbols and  $r$  grammar symbols), exposing state  $s_{m-r}$ . The parser then pushed both  $A$ , the left side of the production and  $s$ , the entry for  $\text{goto}[s_{m-r}, A]$ , onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct,  $X_{m-r+1} \dots X_m$ , the sequence of grammar symbols popped off the stack, will always match  $\beta$ , the right side of the reducing production.

### Compiler Design

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

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

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

The LR parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

#### LR Parsing Algorithm

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

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

**Method :** Initially, the parser has  $s_0$  on its stack, where  $s_0$  is the initial state and  $w\$$  in the input buffer. The parser then executes the program in Fig. until an accept or error action is encountered.

set ip to point to the first symbol of  $w\$$ ;

repeat forever begin

let  $s$  be the state on top of the stack and

$a$  the symbol pointed to by ip;

if action  $[s, a] = \text{shift } s'$  then begin

push  $s'$  then  $s'$  on top of the stack;

advance ip to the next input symbol

end

else if action  $[s, a] = \text{reduce } A \rightarrow \beta$ , then begin

pop  $2 * |\beta|$  symbols off the stack;

let  $s'$  be the state now on top of the stack;

push  $A$  then goto  $[s', A]$  on top of the stack;

output the production  $A \rightarrow \beta$ ,

end

else if action  $[s, a] = \text{accept}$  then

return

else error ( )

end

Fig. 2 : LR parsing program

Q.17 Explain why Bottom-up parsing is more generally applicable than top-down parsing?

[R.T.U. 2015]

Ans. Differ  
up parsers:

To
Start at the derivation
Picks a pr tries to ma
May requi backtrack

Some gra backtrack
Top down can't han recursion not termi

In a start symb to reach th to use con guessworl Mo which sca right) whe Typ input to gu can ultim tokens. Fo

we the next guaranter input ch productio So that we h

TI copies of characte If we ch choose t wrong. producti

T nonterm match i recursic



CD.40

Syntax trees for assignment statements are produced by the syntax directed definition. Non-terminals generates an assignment statement.

**Three Address Code :** Three address code is a sequence of statements of the general form.

$x := y \text{ op } z$

where  $x$ ,  $y$  and  $z$  are names, constants or compiler-generated temporaries, 'op' stands for any operator.

Thus a source language expression like  $x + y * z$  might be translated into sequence

$t_1 := y * z$

$t_2 := x + t_1$

where  $t_1$  and  $t_2$  are compiler generated temporary names.

**Syntax Directed Translation Into Three Address Code :** When three address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal,  $E$  on the left side of  $E \rightarrow E_1 + E_2$  will be computed into a new temporary  $t$ .

**Declarations :** As the sequence of declarations in a procedure or block is examined, we can layout storage for names local to the procedure,

For each local name, we create a symbol table entry with information like the type and the relative address of the storage for the name,

The relative address consists of an offset from the base of the static-data area or the field for local, data in an activation record.

B.Tech. (V Sem.) C.S. Solved Papers

**Keeping Track of Scope Information :** In a language with nested procedures, names local to each procedure can be assigned relative addresses using the approach of procedure. When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. A new symbol table is created when a procedure declaration is seen and entries for the declaration are created in the, new table. The new table points back to the symbol table of enclosing procedure, the name represented by id itself is local to the enclosing procedure.

**Assignment Statements :** Expressions can be of type integer, real, array and record. As part of the translation of assignment into three address code, we show how names can be looked up in symbol table and how elements of array and records can be accessed.

1. **Names in Symbol Table :** We formed three address statements using names themselves, with the understanding that the names stands for pointers to their symbol table entries. The translation scheme shows how such symbol table entries can be found.

2. **Reusing Temporary Names :** We have been going along assuming that "newtemp" generates a new temporary name each time a temporary is needed. It is useful especially in optimizing compilers, is actually create a distinct name each time "newtemp" is called.

However, the temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table and space has to be allocated to hold their values.

Compiler Design

Q.3 What are the benefits of using a machine independent intermediates form?

Ans.

1. Retracing is facilitated, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to intermediate representation.

Q.4 Define dependency graphs.

Ans. **Dependency Graphs :** If an attribute  $b$  at a node in a parse tree depends on an attribute  $C$ , then the semantic rule for  $b$  at that node must be evaluated after the semantic rule that defines  $C$ .

The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a "dependency graph".

Q.5 Define L-attributed definition.

Ans. **L-Attributed Definitions**

A syntax-directed definitions is L-attributed if each inherited attributed of  $X_j, 1 \leq j \leq n$ , on the right side of  $A \rightarrow X_1 X_2 \dots X_n$ , depends only on :

1. The attributes of the symbols  $X_1, X_2, \dots, X_{j-1}$  to the left of  $X_j$  in the production and
2. The inherited attributes of  $A$ .

Every S-attributed definition is L-attributed, because the restrictions (1) and (2) apply only to inherited attributes.

## PREVIOUS YEARS QUESTIONS

### PART-A

Q.1 Write syntax directed definition for a given assignment statement.

$S \rightarrow id = E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow id$

[R.T.U. 2016]

Ans.

Grammar	Semantic Rule
$S \rightarrow id = E$	$S.val = id.lexval = E.val$
$E \rightarrow E + E$	$E.val = E_1.val + E_2.val$
$E \rightarrow E * E$	$E.val \rightarrow E_1.val * E_2.val$
$E \rightarrow -E$	$E.val = -E_1.val$
$E \rightarrow (E)$	$E.val = E_1.val$
$E \rightarrow id$	$E.val = id.lexval$

Q.2 What is type system?

Ans. **Type System :** A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system.

### PART-B

Q.6 Let  $G$  be a formal grammar with nonterminal symbol  $S$  and  $D$ , terminal symbol 'b', '0' and '1', start symbol  $S$  and the following production rule.

$S \rightarrow bD$

$D \rightarrow 0D$

$D \rightarrow 1D$

$D \rightarrow 0$

$D \rightarrow 1$

(a) Is  $G$  regular? Why (not)?

(b) Turn  $G$  systematically into a finite automation.

[R.T.U. 2018, 2014]



```

a = _t0;
_L0: _t1 = 10;
      _t2 = a < _t1;
If Z _t2 Goto _L1;
      _t3 = 2;
      _t4 = a % _t3;
      _t5 = 0;
      _t6 = _t4 == _t5;
PushParam _t6;
LCall_PrintBool;
PopParams 4;
      _t7 = 1;
      _t8 = a + _t7;
      a = _t8;
Goto _L0;
_L1:
EndFunc;

```

**Q.9** What do you mean by DAG? Write an algorithm for constructing a DAG. [R.T.U. 2013]

OR  
Write short notes on DAG. [R.T.U. 2013]

### Ans. Directed Acyclic Graphs

Directed acyclic graphs (DAGs) are usual data structures for implementing transformations on basic blocks. A DAG given a picture of how the value computed by each statement in a basic block is used in subsequent of the block. Constructing a DAG from three-address statement is a good way of determining common sub-expression (expression computed more than once) within a block, determining which names are used into the block but evaluated outside the block and determining which statements of the block could have their computed value used outside the block.

A DAG for a basic block (or just DAG) is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constant. From the operator applied to a name we determine whether the l-value or r-value of a name is needed; most leaves represent r-values. The leaves represent initial value of name and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent compute values and the identifiers labeling a node are deemed to have that value.

### Compiler Design

It is important not to confuse DAGs with flow graphs. Each node of a flow graph can be represented by a DAG, since each node of the flow graph stands for a basic block.

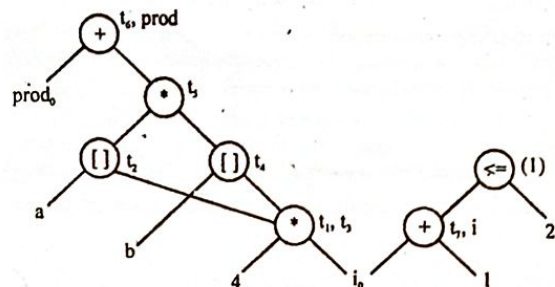


Fig. : An example DAG

### DAG construction

To construct a DAG for a basic block, we process each statement of the block in turn. When we see a statement of the form  $x := y + z$ , we look for the nodes that represent the "current" values of  $y$  and  $z$ . These could be leaves or they could be interior nodes of the DAG if  $y$  and/or  $z$  had been evaluated by previous statements of the block. We then create a node labeled '+' and give it two children; the left child is the node for  $y$ , the right the node for  $z$ . Then we label this node  $x$ . However, if there is already a node denoting the same value as  $y + z$ , we do not add for new node to the DAG, but rather give the existing node the additional label  $x$ .

Two details should be mentioned. First if  $x$  (not  $x_0$ ) has previously labeled some other node, we remove that label, since the "current" value of  $x$  is the node just created. Second, for assignment such as  $x := y$  we do not create a new node. Rather, we append label  $x$  to the list of names on the node for the "current" value of  $y$ .

### Algorithm Construction a DAG.

**Input.** A basic block

**Output.** A DAG for the basic block containing the following information:

1. A label for each node. For leaves the label is an identifier (constants permitted) and for interior nodes, an operator symbol.
2. For each node a (possibly empty) list of attached identifiers (constants not permitted here).

**Q.10** Explain the syntax directed translation schemes in details. [R.T.U. 2014]

**Ans. Syntax Directed Translation (SDT) :** It makes the user life easy by hiding many implementation details and free the user having to specify explicitly the order in which translation takes place.

The translation of token string takes place by evaluating semantic rules, it is not necessary to follow syntax directed definition, it can be implemented in a single pass by evaluating semantic rules during parsing itself.

Translation scheme indicates the order of evaluation of semantic action associated with a production rule. In other words, translation scheme gives a little bit important about implementation detail.

We can point out syntax directed translation as:

- (i) Intermediate code represented as a list of instructions. Instruction sequences are concatenated using the operator  $\parallel$ .
- (ii) Attributes for expression  $E$ :
  - $E.place$ : denotes the location that holds the value of  $E$ .
  - $E.code$ : denotes the instruction sequence that evaluates  $E$ .
- (iii) Attributes for statement  $S$ :
  - $S.begin$ : denotes the first instruction in the code for  $S$ .
  - $S.after$ : denotes the first instruction after the code for  $S$ .
  - $S.code$ : denotes the instruction sequence that represents  $S$ .
- (iv) Auxiliary Functions:
  - $newtemp()$ : returned a new temporary each time it is called. Returns a pointer to the ST entry of a temp. May take a parameter specifying the type of the temp.
  - $newlabel()$ : returns a new label name each time it is called.
- (v) Notation: we write  $gen(x := y + z)$

to represent the instruction  $x := y + z$ .

**Example :** For the boolean expression  $a < b$  or  $c < d$  and  $e < f$  perform SDT scheme.

**Solution :** The SDT scheme based on parse tree becomes :

```

00 : if a < b goto 03
01 : t1 = 0
02 : goto 04
03 : t1 = 1
04 : if c < d goto 07
05 : t2 = 0
06 : goto 08
07 : t2 = 1
08 : if e < f goto 11
09 : t3 = 0
10 : goto 12
11 : t3 = 1
12 : t4 = t2 and t3
13 : t5 = t1 or t4

```



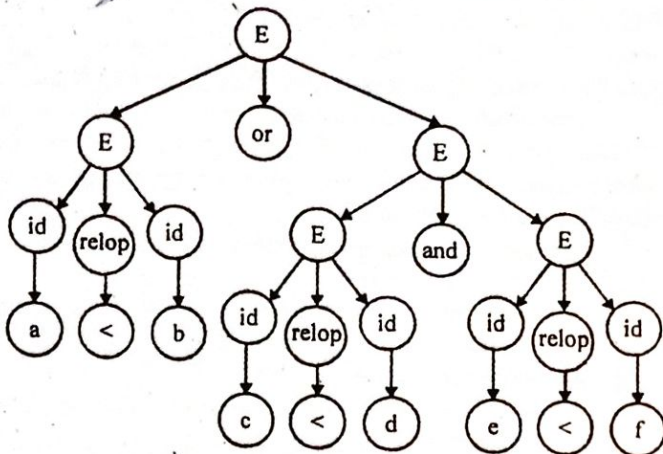


Fig.

**Q.11** What is the process and importance of intermediate code generation. [R.T.U. 2014]

**Ans.** Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form that can be readily executed by a machine.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code.

Tasks which are part of code generation phase include:

- (i) **Instructions Selection** : which instruction to use.
- (ii) **Instructions Scheduling** : In which order to put those instructions.
- (iii) **Register Allocation** : The allocation of variables to processor registers.
- (iv) **Debug Data** : Debug data generation if required so the code can be debugged.

Instruction selection is typically carried out by doing a recursive post order traversal on the abstract syntax tree, matching particular tree configuration against templates.

For eg: the tree  $W := \text{ADD}(x, \text{MUL}(y, z))$  might be transformed into linear sequence of instructions by recursively generating the sequences for  $t1 := x$  and  $t2 := \text{MUL}(y, z)$ , and then emitting the instruction  $\text{ADD } w, t1, t2$ .

In a compiler that uses an intermediate language, there may be two instruction selection stages—one to convert the parse tree into intermediate code, and a

second phase much later to convert the intermediate code into instructions from the instruction set of the target machine.

**Q.12** Give a syntax-directed definition to translate infix expression into infix expression without redundant parenthesis. For example, since + and \*. Associate to the left,  $((a*(b+c))*(d))$  can be rewritten as  $a*(b+c)*d$ . [R.T.U. Dec. 2013]

**Ans.**  $S \rightarrow E$

$E.iop = \text{nil}$

$S.equation = E.equation$

$E \rightarrow E \text{ subone } + T$

$E \text{ subone}.iop = E.iop$

$T.iop = E.iop$

$E.equation = E \text{ subone}.equation \parallel '+' \parallel T.equation$

$E \rightarrow T$

$E.stop = '+'$

$E \rightarrow T$

$T.iop = E.iop$

$E.equation = T.equation$

$E.sop = T.sop$

$T \rightarrow T \text{ subone } * F$

$T \text{ subone}.iop = '*'$

$F.iop = '*'$

$T.equation = T \text{ subone}.equation \parallel '*' \parallel F.equation$

$T.sop = '*'$

$T \rightarrow F$

$F.iop = T.iop$

$T.equation = F.equation$

$T.sop = F.sop$

$F \rightarrow \text{Char}$

$F.equation = \text{Char.lexval}$

$F.sop = \text{nil}$

$F \rightarrow (E)$

if  $(F.iop == '*' \ \&\& \ E.sop == '+')$  {  $F.equation = '(' \parallel E.equation \parallel ')' \}$

else {  $F.equation = E.equation$  }

$F.sop = \text{nil}$

**Q.13** Write short note on S-attributed definitions and L-attributed definitions. [R.T.U. 2011]

OR

Write short note on Synthesized attribute.

[R.T.U. 2013, 2010, 2008]

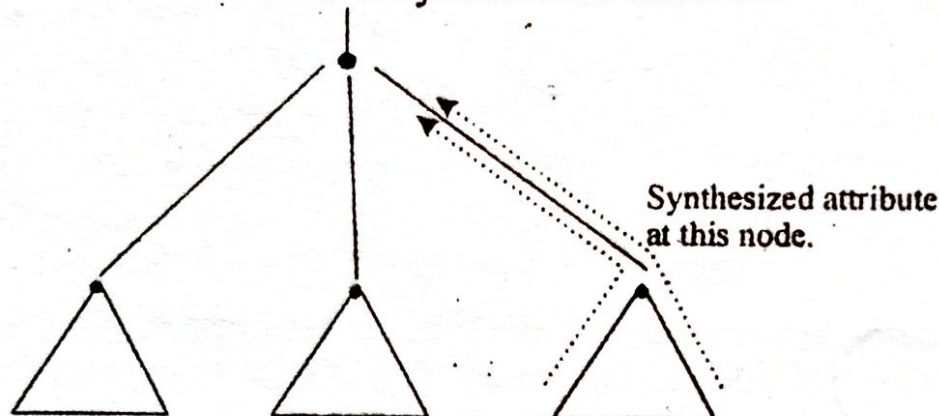
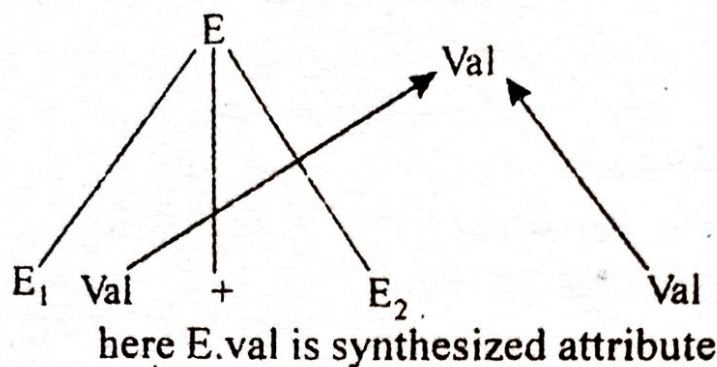


**Compiler Design**

**Ans. Synthesized Attributes :** An attribute at a node is synthesized if its value is computed from the attributed values of the children of that node in the parse tree.

**Example :**  $E \rightarrow E_1 + E_2$

$E.val = E_1.val + E_2.val$  — semantic rule



Synthesized attribute can be evaluated by a single bottom-up traversal of the parse tree.

Synthesized attributes also refer as s-attributes. Synthesized attributes also define in the L-attributes.

"Grammar containing only synthesized attributes is called **S-attributed**".

Synthesized attributes can be conveniently handled during bottom up parsing as it builds the parse tree bottom-up.

"On the other hand grammar for which the attributes can always be evaluated by a depth-first, L-to-R traversal of the parse tree, is called **L-attributed grammar**.

**L-Attributed Definitions :** Refer to Q.5.



Q.16 Translate the arithmetic expression :

$(a + b) * (c + d) + (a + b + c)$  into

- (i) Syntax tree
- (ii) Three address code
- (iii) Quadruples.
- (iv) Triples.
- (v) Indirect triples.

[R.T.U. 2016, 2015]

Ans. Expression:  $(a+b)*(c+d)+(a+b+c)$

(i) Syntax tree:

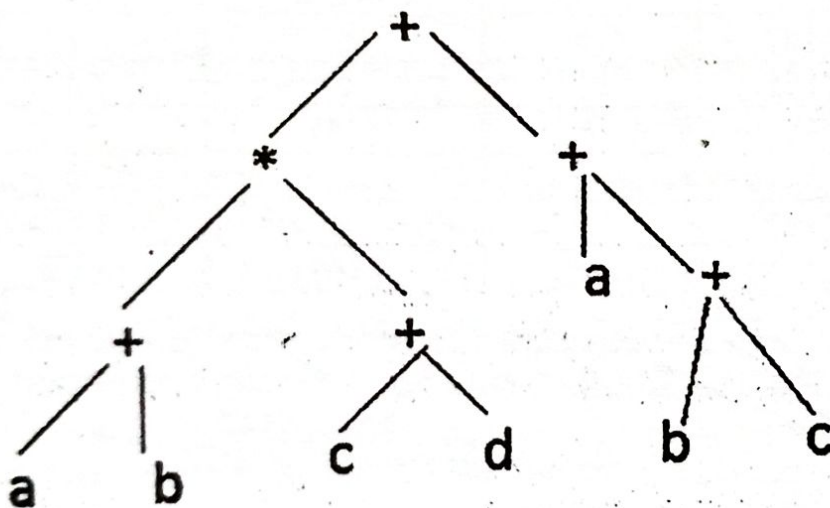


Fig.

(ii) Three Address Code : Three-address code(TAC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.

Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example,  $t1 = t2 + t3$ .

Let  $t1 = a + b$

Let  $t2 = c + d$

Let  $t3 = t1 + c$

Let  $t4 = t1 * t2$

Let  $t5 = t4 + t3$  (the three address code desired!)



## (iii) Quadruples representation

	OP	arg 1	arg 2	Result
(0)	+	a	b	$t_1$
(1)	+	c	d	$t_2$
(2)	*	$t_1$	$t_2$	$t_3$
(3)	+	a	b	$t_4$
(4)	+	$t_4$	c	$t_5$
(5)	+	$t_3$	$t_5$	$t_6$

## (iv) Triples representation

	OP	arg 1	arg 2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	a	b
(4)	+	c	(3)
(5)	+	(2)	(4)

## (v) Indirect triples representation

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

	OP	arg 1	arg 2
(14)	+	a	b
(15)	+	c	d
(16)	*	(14)	(15)
(17)	+	a	b
(18)	+	c	(17)
(19)	+	(16)	(18)

**Q.17** Define syntax directed definition. Explain the various forms of syntax directed definition.

[R.T.U. 2015, 2012]

**Ans. Syntax Directed Definition :** A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol. If we think a node for the grammar symbol in a parse tree as a record with fields for holding information, then an attribute corresponds to the name of a field.

An attribute can represent anything we choose: a string, a number, a type, a memory location or whatever. The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production rule at the node, the value of a synthesized attribute at a node is

computed from the values of attribute at the children of that node in the parse tree; the value of an inherited attribute is computed from the values of attributes at the sibling and parent of that node.

Semantic rules set up dependency between attributes that will be represented by a graph. From the dependencies graph, we drive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the node in the parse tree for the input string. A semantic rule may also have side effect e.g., printing a value or updating a global variable. Of course, an implementation need not explicitly construct a parse tree or a dependency graph; it just has to produce the same output for each input string.

A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing the attribute values at the node is called annotating or decorating the parse tree.

**Form of a syntax directed definition**

In a syntax-directed definition each grammar production  $A \rightarrow \alpha$  has associated with it a set of semantic rules of the form  $b = f(c_1, c_2, \dots, c_k)$  where  $f$  is a function and either

1.  $b$  is a synthesized attribute of  $A$  and  $c_1, c_2, \dots, c_k$  are attributes belonging to the grammar symbols of the production.
2.  $b$  is an inherited attribute of one of the grammar symbols on the right side of the production and  $c_1, c_2, \dots, c_k$  are attributes belonging to the grammar symbols of the production.

In either case, we say that attribute  $b$  depends on attributes  $c_1, c_2, \dots, c_k$ . An attribute grammar is a syntax-directed definition in which the functions in semantic rules cannot have side effects.

Function in semantic rules will often be written as expressions. Occasionally, the only purpose of a semantic rule in a syntax-directed definition is to create a side effect. Such semantic rules are written as procedure calls or program fragments. They can be thought of as rules defining the values of dummy synthesized attributes of the non-terminal on the left side of the associated production; the dummy attribute and the  $=$  sign in the semantic rules are not shown.

**Example:** The syntax directed definition in fig. is for a desk-calculator program. This definition associates an integer-valued synthesized attribute called  $val$  with each of the non-terminal  $E, T$  and  $F$ . For each  $E, T$  and  $F$ -production, the semantic rule computes the value of attributes  $val$  for non terminal on the left side from the value of  $val$  for the nonterminals on the right side.

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

Fig. : Syntax-directed definition of a simple desk-calculator

The token digit has a synthesized attribute  $lexval$  whose value is assumed to be supplied by the lexical analyzer. The rule associated with the production  $L \rightarrow En$  for the starting nonterminal  $L$  is just a procedure that prints output of the value of the arithmetic expression generated by  $E$ ; we can think of this rule as defining a dummy attribute for the nonterminal  $L$ .

In a syntax directed definition, terminals are assumed to have synthesized attributes only, as the definition does not provide any semantic rules for terminals. Values for attributes of terminals are usually supplied by the lexical analyzer.

**Q.18 (a)** Define Syntax Directed Definitions? Define the expressions used by type checker. [R.T.U. 2013]

OR

Define 'type expressions' used by types checker. [R.T.U. 2009, Raj. Univ. 2004]

**(b)** For the assignment statement  $X = (a + b) * (c + d)$ . Construct the translation scheme and an annotated parse tree. [R.T.U. 2013]

OR

Obtain the translation scheme for obtaining the three address code for the grammar :

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

**Ans.(a) Syntax Directed Definition :** Refer to Q.17 Type Expressions

The type of a language construct will be denoted by a "type expression." Informally, a type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following definition of type expressions are used :



## PART-A

Q.1 What is the objectives of sub-division of Run-time memory?

Ans. The run-time storage might be subdivided to hold :

1. The generated target code.
2. Data objects.
3. A counterpart of the control stack.

Q.2 Write two limitations of static allocation.

Ans. Limitations of Static Allocation :

1. The size of a data object and constraints on its position in memory must be known at compile time.
2. Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.

Q.3 Define activation records.

Ans. Activation Records : Information needed by a single, execution of a procedure is managed using a continuous block of storage called an "activation record" or "frame".

Q.4 Why storage allocation strategies is used?

Ans. A different storage allocation strategy is used in each of the three data areas in the organization :

1. Static allocation lays out storage for all data objects at compile time.
2. Stack allocation manages the run time storage as a stack.

Q.5 What do you mean by dangling reference.

Ans. Dangling Reference : A dangling reference occurs when there is a reference to storage that has been deallocated.

## PART-B

Q.6 Write short notes on :

(a) Nesting depth and Access links

(b) Static versus Dynamic storage allocations

[R.T.U. 2018, 2014]

Ans.(a) **Nesting Depth** : The nesting depth is the number of statement blocks that are nested due to the use of control structures (branches, loops). The maximum nesting depth is restricted to 256 by the ABAP compiler. In control structure, the restriction on the maximum nesting depth within a procedure is to five level.

This nesting depth is defined at the level of a procedure (method). Implementations must not occur at other points.

Access links can be defined in two ways as static links and dynamic links. When we want to pop out the current frame of the caller and restore the caller frame, this can be done with the help of a pointer in the current frame, called the dynamic link that points to the previous frame (caller's frame). Thus all the frames are linked together in the stack using dynamic link.

The second thing, is if we allow nested functions, we need to be able to access the variables stored in previous activation records in stack. This is done with the help of static link.

Ans.(b) Static memory allocation is used when you know the memory requirement in advance. Dynamic memory allocation is assigned by the compiler at runtime.

The space is allocated once, when your program is started and is never freed. Static allocation is what happens when we declare a static or global variable. Each static or global variable defines one block of space, of fixed size.

Dynamic memory allocation is used when the memory we need, or long we continue to need it, depends on the factor that are not known before the program runs. The only way to get dynamically allocated memory is via a system call and the only way to refer to dynamically allocated space is through a pointer.



The actual process of dynamic allocation requires more computation times and hence its slower than static memory allocation.

**Q.7** Discuss symbol table with following subcategories.

- (a) Basic operations on symbol table.
- (b) Implementation of symbol table.

[R.T.U. 2017]

#### Ans.(a) Basic Operations on Symbol Table

A symbol table, either linear or hash, should provide the following operations.

**insert()**

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The **insert()** function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

`int a;`

should be processed by the compiler as:

`insert(a, int);`

`lookup()`

`lookup()` operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of `lookup()` function varies according to the programming language. The basic format should match the following:

`lookup(symbol)`

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

**Ans.(b) Implementation of symbol table :** If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

**Q.8** Explain the symbol table management system.

[R.T.U. 2016]

**OR**  
Write short note on Symbol table. [R.T.U. 2015]

**Ans. Symbol table :** A symbol table mechanism allows us to add new entries and to find existing entries efficiently. The two symbol table mechanisms described here are linear lists and hash tables. We evaluate each scheme on the basis of the time required to add  $n$  entries and make  $e$  inquiries. A linear list is the simplest to implement, but its performance is poor when  $e$  and  $n$  get large. Hashing schemes provide better performance for somewhat greater programming effort and space overhead. Both mechanisms can be adapted readily to handle the most closely nested scope rule.

Each entry in the symbol table is for the declaration of a name. The format of entries does not have to be uniform, because the information saved about a name depends on the usages of the name. Each entry can be implemented as a record consisting of a sequence of consecutive words of memory to keep symbol table records uniform, it may be convenient for some of the information about a name to be kept outside the table entry, with only a pointer to this information stored in the record.

The symbol table entry itself can be set up when the role of a name becomes clear, with the attribute values being filled in as the information becomes available. In some cases, the entry can be initiated from the lexical analyzer as soon as a name is seen in the input. More often, one name may denote several different objects, perhaps even in the same block or procedure.

For example, the C declarations

`int x;`

`struct x { float y, z; };`

use `x` both as an integer and as the tag of a structure with two fields. In such cases, the lexical analyzer can only



return to the parser the name itself (or a pointer to the lexeme forming that name), rather than a pointer to the symbol table entry. The record in the symbol table is created when the syntactic role played by this name is discovered. For the above declarations two symbol table entries for  $x$  would be created; one with  $x$  as an integer and one as a structure.

**Q.9** Write a short note on Symbol table and Dangling References. [R.T.U. Dec. 2013]

**Ans. Symbol Table :** Refer to Q.8.

**Dangling References:** It is link or pointer to something (instruction, table element, index item, etc) that no longer contains the same content. If the reference is not a currently valid address or if it is a valid address but there is no content in that location, it may cause the computer to crash. If the content has changed, it can also cause the system to crash or, at the very least, produce erroneous output.

**Creation of Dangling Pointer :** Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. As the system may reallocate the previously freed memory to another process, if the original program then dereferences the (now) dangling pointer, unpredictable behavior may result, as the memory may now contain completely different data.

**Problems Caused by Dangling Pointer :** If the program writes data to memory pointed by a dangling pointer, a silent corruption of unrelated data may result, leading to suitable bugs that can be extremely difficult to find or cause segmentation faults (\*NIX) or general protection faults (Windows). If the overwritten data is book keeping data used by the system's memory allocator, the corruption can cause system instabilities. Hence dangling references are known as problems.

## PART-C

**Q.10** Explain the organization of symbol table in detail. Also explain the various data structures used in symbol tables. [R.T.U. 2013]

OR

Explain the various strategies of symbol table creation and organization. [R.T.U. 2018, 2014]

OR

**Q.** Discuss symbol table with Data structure used in symbol table subcategories. [R.T.U. 2017]

**Ans. Symbol Table Management System :** It contains

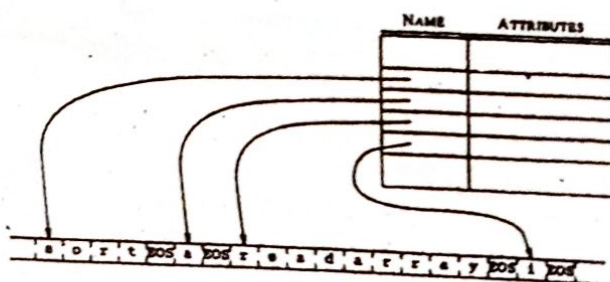
- (1) Characters in a Name.
- (2) Storage Allocation Information.
- (3) Hash Table.

(1) Characters in a Name

There is a distinction between the token id for an identifier or name, the lexeme consisting of the character string forming the name and the attributes of the name. Strings of characters may be unwieldy to work with, so compilers often use some fixed-length representation of the name rather than the lexeme. The lexeme is needed when a symbol table entry is set up for the first time and when we look up a lexeme found in the input to determine whether it is a name that has already appeared. A common representation of a name is a pointer to a symbol table entry for it.

NAME	ATTRIBUTES
s	
o	
r	
t	
a	
r	
e	
a	
d	
a	
r	
r	
a	
y	

(a) In fixed-size space within a record



(b) In a separate array

Fig. 1 : Storing the characters of a name

(2) Storage Allocation Information

Information about the storage allocations that will be bound to names at run time is kept in the symbol table. Consider the names with static storage first. If the target code is assembly language, we can let the assembler take care of storage locations for the various names. In the case of names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all the compiler plans out the activation record for each procedure.



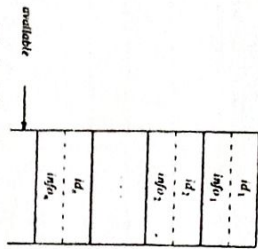


Fig. 2: A linear list of records

If the symbol table contains  $n$  names, the work necessary to insert a new name is constant if we do the insertion without checking to see if the name is already in the table. If multiple entries for names are not allowed, then we need to look through the entire table before discovering that a name is not in the table, during work proportional to  $n$  in the process. To find the data about a name, on the average, we search  $n/2$  names, so the cost of an inquiry is also proportional to  $n$ .

Thus, since insertions and inquiries take time proportional to  $n$ , the total work for inserting  $n$  names and making  $e$  inquiries is at most  $c(n^2 + e)$ , where  $c$  is a constant representing the time necessary for a few machine operations.

Variations of the searching technique known as hashing have been implemented in many compilers. Here we consider a rather simple variant known as open hashing, where "open" refers to the property that there need be no limit on the number of entries that can be made in the table. Even this scheme gives us the capability of performing  $e$  inquiries on  $n$  names in time proportional to  $n(n + e)/m$ , for any constant  $m$  of our choosing.

Since  $m$  can be made as large as we like, up to  $n$ , this method is generally more efficient than linear lists and is the method of choice for symbol tables in most situations.

The space taken by the data structure grows with  $m$ , so a time-space trade-off is involved.

The basic hashing scheme is illustrated in fig. 3. There are two parts of the data structure:

1. A hash table consisting of a fixed array of  $m$  pointers to table entries.

2. Table entries organized into  $m$  separate linked lists, called buckets (some buckets may be empty). Each record in the symbol table appears on exactly one of these lists. Storage for the records may be drawn from an array of records. Alternatively, the dynamic storage allocation facilities of the implementation language can be used to obtain space for the records, often at some loss of efficiency.

To determine whether there is an entry for string  $s$  in the symbol table, we apply a hash function  $h$  to  $s$ , such that  $h(s)$  returns an integer between 0 and  $m - 1$ . If  $s$  is in the symbol table, then it is on the list numbered  $h(s)$ . If  $s$  is not yet in the symbol table, it is entered by creating a record for  $s$  that is linked at the front of the list numbered  $h(s)$ .

As a rule of thumb, the average list is  $n/m$  records long if there are  $n$  names in a table of size  $m$ . By choosing  $m$  so that  $n/m$  is bounded by a small constant, say 2, the time to access a table entry is essentially constant.

Array of list headers, indexed by hash value

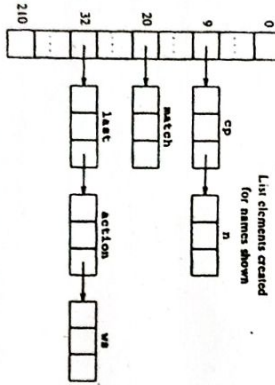


Fig. 3: A hash table of size 211

Following are commonly used data structures for symbol table construction.

#### (i) List Data Structure for Symbol Table

- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive.
- The pointer 'available' is maintained at the end of all stored records. The figure 4 for list data structure using arrays is as given below.

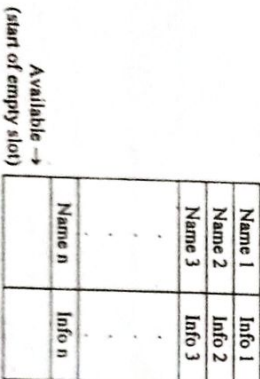


Fig. 4: List data structure

To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name we get an error "use of undeclared name".

While inserting a new name we should ensure that it should not be already there. If it is there another error occur i.e., "Multiple defined name". The advantages of list organization is that it takes minimum amount of space.

#### (ii) Self Organizing List

This symbol table implementation is using linked list. A link field is added to each record. We search the records in the order pointed by the link of link field. A pointer "First" is maintained to point to first record of the symbol table.

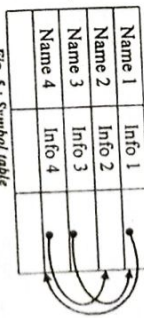


Fig. 5: Symbol table

The reference to these names can be Name2, Name1, Name4, Name2.

When the name is referenced or created it is moved to the front of the list. The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

#### (3) Hash Tables

- Hashing is an important techniques used to search the records of symbol table. This method is superior to list organization.
- In hashing schemes two tables are maintained a hash table and symbol table.
- The hash table consists of  $k$  entries from 0, 1 to  $k-1$ . These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such the  $h(\text{name})$  will result may integer between 0 to  $k-1$ . We can search any name by position =  $h(\text{name})$
- Using the position we can obtain the exact locations of name is symbol table.
- The hash table and the symbol table can be shown as below.
- 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 collision. Collision is such a situation where hash function results in same location for storing the names. Various collision resolution techniques are open addressing, chaining, rehashing. The advantage is hashing is that quick search is possible and the disadvantage is that hashing is complicated to implement. Some extra space is required. Obtaining scope of variables is very difficult.

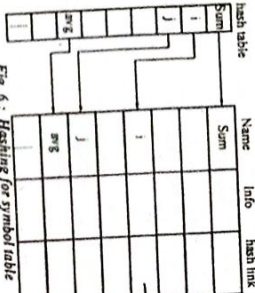


Fig. 6: Hashing for symbol table

Q.11 What are Activation Trees and Activation Records? Explain the Data Access Process without Nested Procedures. [R.TU.2018,2014]

Ans. Activation Tree : It is a graphical representation of flow of activations when a program is running. It describes the flow of control over the procedure call chains. In this:

- (1) Each call is represented by a node.
  - (2) The root represents the activation of the main procedure.
  - (3) Node  $a$  is parent of  $b$ , if the flow of control goes from  $a$  to  $b$  ( $a$  is calling  $b$ ).
  - (4) Node  $a$  is to the left of  $b$  if  $a$  terminates before  $b$ . In this, actual flow of control corresponds to depth-first traversal of tree. We make the following assumptions about the flow of control among procedures during the execution of program.
1. Control flows sequentially; that is, the execution of program consists of a sequence of steps, with control being at some specific point in the program at each step.
  2. Each execution of a procedure starts at the beginning to the point immediately following the place where the procedure was called.
- Activation Record : The information needed by a simultaneous execution of a procedure is managed using a continuous



## Compiler Design

than a  $[I_0]$  - to  $I_0$ , where  $I_0$  is the initial value of  $i$ . This phenomenon occurs because the location of  $x$  in the assignment  $x = \text{temp of swap}$  is not evaluated until needed, by which time the value of  $i$  has already changed. A correctly working version of swap apparently cannot be written if call-by-name is used.

**Example :** Suppose that the function  $f$  in the assignment  $x = f(A) + f(B)$  is called by value. Here the actual parameters  $A$  and  $B$  are expressions. Substituting expression  $A$  and  $B$  for each occurrence of the formal parameter in the body of  $f$  leads to call-by-name; recall  $a[i]$  in the last example. Fresh temporary variables can be used to force the evaluation of the actual parameters before execution of the procedure body :

$t_1 = A;$   
 $t_2 = B;$   
 $t_3 = f(t_1);$   
 $t_4 = f(t_2);$   
 $x = t_3 + t_4;$

Now in-line expression will replace all occurrences of the formal by  $t_1$  and  $t_2$  when the first and second calls, respectively are expanded.

**Q.14 Differentiate between stack allocation and heap allocation?** [R.T.U. 2016, 2015]

OR

**Explain the various storage allocation strategies?** [R.T.U. 2013, 2009, 2008]

OR

**Write short note on storage allocation strategies.** [R.T.U. 2015, Dec. 2013]

OR

**Explain the differences between stack allocation and heap allocation strategies.**

[R.T.U. Dec. 2013, R.T.U. 2012, 2009, Raj. Univ. 2007]

**Ans. Storage Allocation Strategies :** A different storage allocation strategy is used in each of the three data areas in the organization.

1. Static allocation lays out storage for all data objects at compile time.
2. Stack allocation manages the run-time storage as a stack.
3. Heap allocation allocates and deallocates storage as needed at run time from a data area known as a heap.

**Static Allocation :** In static allocation, names are bound to storage as the program is compiled, so there is no need for a run time support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage locations. This property allows the values of local names to be retained

across activations of a procedure. That is, when control returns to a procedure, the values of the locals are the same as they were when control left the last time.

From the type of a name, the compiler determines the amount of storage to set aside for that name. The address of this storage consists of an offset from an end of the activation record for the procedure. The compiler must eventually decide where the activation records go, relative to the target code and to one another. Once this decision is made, the position of each activation record and hence of the storage for each name in the record is fixed. At compile time we can therefore fill in the addresses at which the target code can find the data it operates on.

Some limitations go along with using static allocation alone.

1. The size of a data object and constraints on its position in memory must be known at compile time.
2. Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
3. Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

**Stack Allocation :** Stack allocation is based on the idea of a control stack; storage is organized as a stack and activation records are pushed and popped as activation begin and end respectively. Storage for the locals in each call of a procedure is contained in the activation record for that call. The values of locals are deleted when the activation ends; that is, the values are lost because the storage for locals disappears when the activation record is popped.

We first describe a form of stack allocation in which the sizes of all activation records are known at compile time. Situations in which incomplete information about sizes is available at compile time are considered below.

Suppose that register  $\text{top}$  marks the top of the stack. At run time, an activation record can be allocated and deallocated by incrementing and decrementing  $\text{top}$ , respectively, by the size of the record.

Figure 1 shows the activation records that are pushed onto and popped from the run-time stack as control flows through the activation tree. Dashed lines in the tree go to activations that have ended. Execution begins with an activation of procedure  $s$ . When control reaches the first call in the body of  $s$ , procedure  $r$  is activated and its activation record is pushed onto the stack. When control returns from this activation, the record is popped leaving just the record for  $s$  in the stack. Whenever control is in activation, its activation record is at the top of the stack.



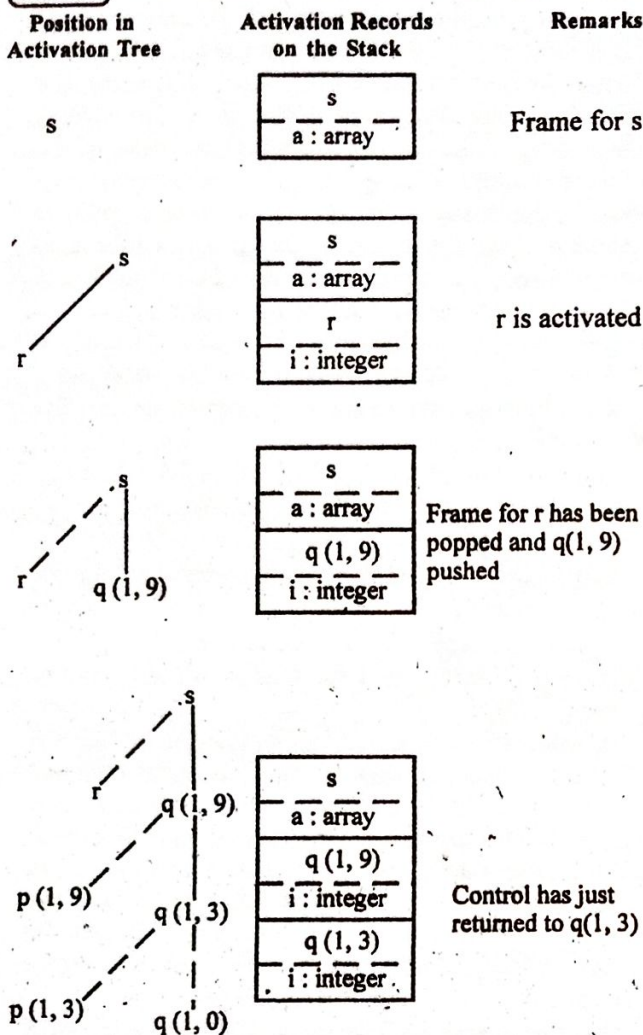


Fig. 1

Several activations occur between the last two diagrams in fig.1. In the last diagram activations p(1,3) and q(1,0) have begun and ended during the lifetime of q(1,3), so their activation records have come and gone from the stack, leaving the activation record for q(1,3) on top.

**Heap Allocation :** Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so overtime the heap will consist of alternate areas that are full and in use.

The stack allocation strategy discussed above cannot be used if either of the following is possible :

1. The value of local names must be retained when activation ends.
2. A called activation outlines the caller. This possibility cannot occur for those languages where activation

trees correctly depict the flow of control between procedures.

In each of the above cases, the deallocation records need not occur in a last-in first-out fashion, so storage cannot be organized as a stack.

The difference between heap and stack allocation of activation records can be seen from the fig.1 and the fig.2.

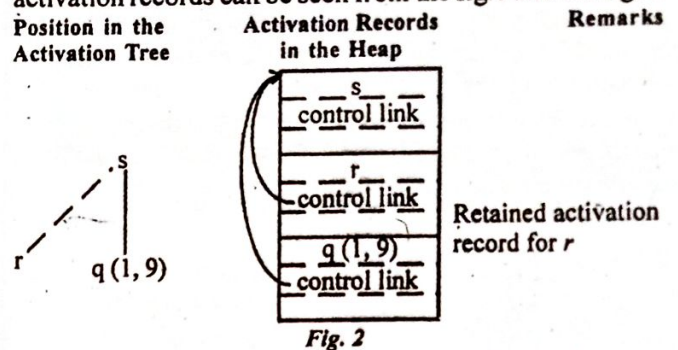


Fig. 2

In fig.2, the record for an activation of procedure r is retained when the activation ends. The record for the new activation q(1,9) therefore cannot follow that for s physically, as it did in fig.1. Now if the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q(1,9). It is left to the heap manager to make use of this space.

Differences between stack allocation and heap allocation:

Stack Allocation	Heap Allocation
Used for static memory allocation.	Used for dynamic memory allocation.
Variables allocated are stored on memory.	Variables on heap are allocated at runtime.
Access to this memory is very fast.	Slow memory access.
Allocation is dealt with when the program is compiled.	No dependency between data, any element can be accessed randomly.
Execution follows LIFO Order, that is, the last reserved block is always the first one to be freed.	Memory blocks can be allocated at any time and freed at any time.
Keeping track is very simple. Only need to adjust the stack pointer.	Keeping track of which blocks are free and which are not is much more complex.
Stacks are used when data to be allocated is known at compile time and is not too big.	Heaps are used if details about data are not known at compile time or if data is big.
Stacks are thread specific.	Heaps are application specific.



## PREVIOUS YEARS QUESTIONS

### PART-A

Q.1 Consider the following basic block and then construct the DAG for it.

$$t_1 = a + b$$

$$t_2 = c + d$$

$$t_3 = e - t_2$$

$$t_4 = t_1 - t_3$$

[R.T.U. 2013]

Ans.

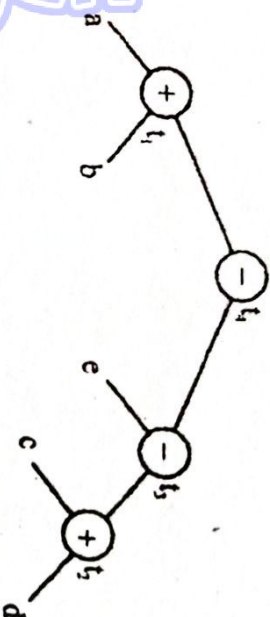


Fig. : DAG for basic block

Now that the order is the one we would naturally obtain from a syntax-directed translation of the expression  $(a + b) - (e - (c + d))$ .

Q.2 What do you understand by basic blocks.



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

### Q.3 Define flow graph.

**Ans. Flow Graph :** We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of the flow graph are basic blocks.

### Q.4 What do you mean by machine independent optimizations?

**Ans. Machine Independent Optimizations :** A "machine independent optimizations" are program transformations that improve the target code without taking into consideration, any properties of the target machine.

### Q.5 Define optimizing compilers.

**Ans. Optimizing Compilers :** Compilers that apply code-improving transformations are called "optimizing compilers"

## PART-B

### Q.6 Construct a DAG for the basic block whose code is given below:

$D := B * C$   
 $E := A + B$   
 $B := B * C$   
 $A := E - D$

[R.T.U. 2018, Dec. 2013]

**Ans. Given**

$D := B * C$   
 $E := A + B$   
 $B := B * C$   
 $A := E - D$

DAG for the given code is

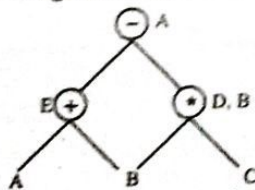
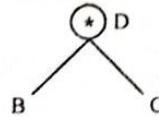


Fig. : DAG of given code

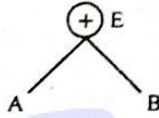
To generate the DAG we follows the following

step :

Step 1 :  $D := B * C$

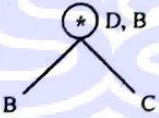


Step 2 :  $E := A + B$



Step 3 :  $B := B * C$

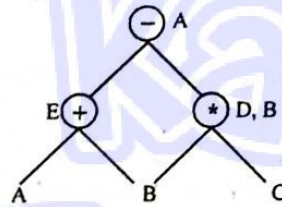
It is same as step 1. so



Step 4 :  $A := E - D$

i.e. Step 2 (Step 1, Step 3)

i.e.



### Q.7 Explain the basic block and control flow graph.

[R.T.U. 2018, Dec. 2013]

OR

Write short notes on :

(a) Flow graph

(b) Basic block

[R.T.U. 2016]

OR

Define basic blocks and flow graphs? Explain structure preserving transformation on basic blocks in detail.

[R.T.U. 2013]

**Ans. Flow Graphs :** A graph representation of three address statements, called a flow graph, is useful for understanding code generation algorithms, even if the graph is not explicitly constructed by a code generation algorithm.

**Basic Blocks :** A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three address statements forms a basic block :

$t_1 := a * a$

$t_1 := a * b$

$t_1 := 2 * t_1$

$t_1 := t_1 + t_1$

$t_1 := b * b$

$t_1 := t_1 + t_1$

A three address statement  $x := y + z$  is said to define  $x$  and to use (or reference)  $y$  and  $z$ . A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three address statements into basic blocks.

**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 the following :

- The first statement is a leader.
  - Any statement that is the target of a conditional or unconditional goto is a leader.
  - 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.

### Structure Preserving Transformation

The various structure preserving transformations on basic blocks are:

- Common sub-expression elimination.
  - Dead-code elimination.
  - Renaming of temporary variables.
  - Interchange of two independent adjacent statement.
- Let us now examine some transformation in more detail:

#### 1. Renaming Temporary Variables

Suppose we have a statement  $t := b + c$ , where  $t$  is a temporary. If we change this statement to  $u := b + c$  where  $u$  is a new temporary variable and change all uses of this instance of  $t$  to  $u$ , then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a new temporary block. We call such a basic block a normal-form block.

#### 2. Interchange of Statements

Suppose we have a block with the two adjacent statements :

$t_1 := b + c$

$t_2 := x + y$



OR  
Write short notes on Advantages of DAG. [R.T.U. 2017]

OR  
What is peephole optimization? Explain it. [R.T.U. Dec. 2013]

OR  
What are the advantages of DAG? Explain the peephole optimization. [R.T.U. 2014]

#### Ans. Advantages of DAG

- When a DAG is created, common sub-expressions are detected.
- Creating a DAG makes it easy to see variables and expression which are used or defined within a block.
- We can generate triples and quadruples representing DAG. They are easier for humans to read than an abstract syntax tree.
- It provides us more opportunity for optimization at code generation time.
- It allows us to determine the loops and thus the loops can be resolved.

**Peephole Optimization :** In compiler theory, peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognising sets of instructions that don't actually do anything or that can be replaced by a leaner set of instructions.

**Replacement Rules :** Common techniques applied in peephole optimization :

- Constant Folding :** Evaluate constant subexpressions in advance.
  - Strength Reduction :** Replace slow operations with faster equivalents.
  - Null Sequences :** Delete useless operations.
  - Combine Operations :** Replace several operations with one equivalent.
  - Algebraic Laws :** Use algebraic laws to simplify or reorder instructions.
  - Special Case Instructions :** Use instructions designed for special operand cases.
  - Address Mode Operations :** Use address modes to simplify code.
- There can, of course, be other types of peephole optimization involving simplifying the target machine instructions, assuming that the target machine is known in advance. Advantages of a given architecture and instruction sets can be exploited in this case.

**Examples :** Replacing slow instructions with faster ones

```

...
aload l
aload l
mul

...
can be replaced by

...
aload l
dup
mul

```

This kind of optimization, like most peephole optimizations, makes certain assumptions about the efficiency of instructions. For instance, in this case, it is assumed that the dup operation (which duplicates and pushes the top of the stack) is more efficient than the aload X operation (which loads a local variable identified as X and pushes it on the stack).

**Removing Redundant Code :** Another example is to eliminate redundant load stores :

$a = b + c;$

is straightforwardly implemented as :

```

MOV b, R0 # Copy b to the register
ADD c, R0 # Add c to the register, the register
            is now b + c

```

```

MOV R0, a # Copy the register to a
MOV a, R0 # Copy a to the register
ADD c, R0 # Add c to the register, the register
            is now a+(b+c)+c]

```

```

MOV R0, d # Copy the register to d but can be
            optimised to

```

```

MOV b, R0 # Copy b to the register
ADD c, R0 # Add c to the register, the register
            is now b + c (a)

```

```

MOV R0, a # Copy the register to a
ADD c, R0 # Add c to the register, which is
            now b+c+(a)+c]

```

```

MOV R0, d # Copy the register to d
            Furthermore, if the compiler knew that the variable a
            was not used again, the middle operation could be omitted.

```

**Removing Redundant Stack Instructions :** If the compiler saves registers on the stack before calling a subroutine and restores them when returning, consecutive calls to subroutines may have redundant stack instructions.

Suppose the compiler generates the following Z80 instructions for each procedure call :

```

PUSH AF
PUSH BC

```

```

PUSH DE
PUSH HL
CALL_ADD R
POP HL
POP DE
POP BC
POP AF

```

If there were two consecutive subroutine calls, they would look like this :

```

PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL_ADD R1
POP HL
POP DE
POP BC
POP AF
PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL_ADD R2
POP HL
POP DE
POP BC
POP AF

```

The sequence POP regs followed by PUSH for the same registers is generally redundant. In cases where it is redundant, a peephole optimization would remove these instructions. In the example, this would cause another redundant POP/PUSH pair to appear in the peephole and these would be removed in turn. Removing all of the redundant code in the example above would eventually leave the following code :

```

PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL_ADD R1
CALL_ADD R2
POP HL
POP DE
POP BC
POP AF

```

**Implementation :** Modern architectures typically allow for many hundreds of different kinds of peephole optimizations and it is therefore often appropriate for compiler programmers to implement them using a pattern matching algorithm.

**Characteristics of Peephole Optimization**

- Redundant instruction elimination.
- Flow of control optimization
- Algebraic simplifications
- Use of machine idioms

Q14 Explain in brief the various issues of design of code generator. [R.T.U. 2018, Dec. 2013]

OR  
What are the various issues in design of code generator loop optimization? [R.T.U. 2014]

**Ans. Issues in the Design of a Code Generator ,**

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation and evaluation order are inherent in almost all code generation problems.

#### 1. Input to the Code Generator

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation. There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.).

We also assume that the necessary type checking has taken place, so type conversion operators have been inserted whenever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

#### 2. Target Programs

The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language or assembly language. Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed



quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute 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 by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

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 generate code. The price paid is the assembly step after code generation. Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must use several passes.

### 3. Memory Management

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the "back patching". Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as  $j$ : goto  $i$  is encountered and  $i$  is less than  $j$ , the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple  $i$ . If, however, the jump is forward, so  $i$  exceeds  $j$ , we must store on a list generated for quadruple  $j$ . Then we process quadruple  $i$ , we fill in the proper machine location for all instructions that are forward jumps to  $i$ .

### 4. Instruction Selection

The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement we can design a code skeleton that outlines the target code to be generated for that construct.

For example, every three address statement of the form  $x := y + z$ , where  $x$ ,  $y$  and  $z$  are statically allocated, can be translated into the code sequence

```
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

```
a := b + c
d := a + e
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

would be translated into

Here the fourth statement is redundant and so is the third if 'a' is not subsequently used.

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. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. 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, rather than by a more obvious sequence that loads a into a register, add one to the register and then stores the result back into a.

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

### Compiler Design

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

### 5. Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

- During register allocation, we select the set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed. Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

$M, x, y$

where  $x$  is the multiplicand, is the even register of an even/odd register pair. The multiplicand value is taken from the odd register pair. The multiplier  $y$  is a single register. The product occupies the entire even/odd register pair. The division instruction is of the form

$D, x, y$

where the 64-bit dividend occupies an even/odd register pair whose even register is  $x$ ;  $y$  represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences in fig. 1(a) and 1(b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in fig. 2. R stands for register i, L, ST and A stands for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e:

```
(a)      t := a + b
          t := t * c
          t := t / d
(b)      t := a + b
          t := t + c
          t := t / d
```

Fig. 1: Two three address code sequence

L	R1, a	L	R0, a
A	R1, b	A	R0, a
M	R0, c	A	R0, c
D	R0, d	SRDA	R0, 32
ST	R1, t	D	R0, d

Fig. 2: Optimal machine code sequence

### 6. Choice of Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

### 7. Approaches to Code Generation

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested and maintained is an important design goal.

### Various Issues in Design of Loop Optimization

Loop optimization can be viewed as the application of a sequence of specific loop transformation to the source code or intermediate representation, with each transformation having an associated test for legality.

A transformation or optimization generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program.

Evaluating the benefit of a transformation or sequence of transformation can be quite difficult with this approach, as the application of one beneficial transformation may require the prior use of one or more other transformation that by themselves, would result in reduced performance.

### Q.15 Write short notes on Global Data Flow analysis.

[R.T.U. 2017]

**Ans. Global Data Flow Analysis:** In order to do code optimization and a good job of code generation, compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph. A compiler could take advantage of "reaching definitions", such as knowing where a variable like  $deb$  was last defined before reaching a given block, in order to perform transformations as just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.