COMPILER DESIGN

Sessional : 50 Marks
Theory : 100 Marks
Total : 150 Marks
Duration of Exam. : 3 Hrs.

ators need of translators, structure of
ools.
f lexical analyzer, regular expressions ,
g, A language specifying lexical analyzer.
on to finite automata, and vice versa,
of lexical analyzer.

ars, definition of parsing.
precedence parsing, top down par sing,

er.
ition, construction of syntax trees, syntax
directed translation, three address code.

mbol tables, its contents and data struc-
bles. Errors, lexical phase error, syntactic

eration, forms of objects code, machine
mporary and user defined variables.

total, with two questions from each
h will be Q.1. This Q.1 is compulsary
equal mark (20marks). Students have
on from each section.

COMPILER DESIGN

Model Test Paper-I
Paper Code:PCC-CSE-302-G

Note : *Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.*

Q.1.(a) Write short note on compiler construction tools.

Ans. Compiler Construction Tools : Writing a computer is tedious and time consuming task. There are some specialized tools for helping in implementation of various phases of compilers. These tools are called complier construction tools. These tools are also called as compiler-compiler, compiler-generators, or translator writing system. Various complier construction tools are given as below :

(i) Scanner generator : These generators lexical analyzers. The specification given to these generators are in the from of regular expressions.

The UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

(ii) Parser generators : These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammer. Typically UNIX has a tool called YACC which is a parser generator.

(iii) Syntax-directed translation engines : In this tool the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.

(iv) Automatic code generator : These geneators take in intermediate code as input and converts each rule of intermediate language into equivalent mechine language. The template matching technique is used. The intermediate code statements are replaced templates that represent the corrponding sequence of machine instructions.

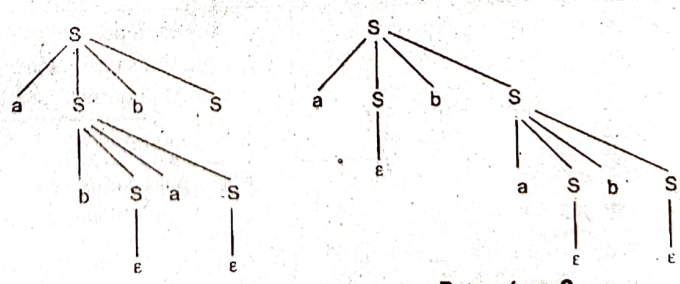
(v) Data flow engines : The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

Q.1.(b) Consider the grammar :

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

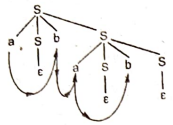
Show that this grammar is ambiguous by constructing two different rightmost derivations for the sentence *abab*.

Ans. Consider a string 'abab'. We can construct parse trees for deriving 'abab'



Parse tree 1 **Parse tree 2**

The rightmost derivation for abab is -



S
a S b S
a S b a S b s
a S b a S b
a S b a b
a b a b

This is a language containing all the strings with equal number of a's and b's.

Q.1.(c) Register and address descriptor in code generation.

Ans. Register and Address Descriptors : The code generation algorithm uses descriptors to keep track of register contents and address for names.

(i) A *register descriptor* keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor that all registors are empty. (If registers are assigned across blocks, this would not be the case). As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.

(ii) An *address descriptor* keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the acccessing method for a name.

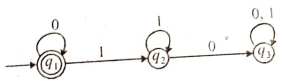
Q.1.(d) Compare single pass and multipass compiler.

Ans. Difference between Single pass compiler and Multi-pass compiler :

Sr.No	*Single pass compiler*	*Multi-pass compiler*
1.	This type compiler passes through the source code of each compilation unit only once.	This type of compiler processes the source code of a program several times.
2.	It does not "look back" at code it previously processed.	In this each pass takes the result of the previous pass as the input and creates an intermediate output.
3.	It is also called narrow compiler.	It is sometimes called wide compiler.
4.	One pass compilers may be faster than multi-pass compiler.	Multi-pass complier are slower as compare with one pass compiler.
5.	Pascal's compiler is an example of single pass compiler.	C++ compiler is multi-pass complier.

Section-A

Q.2.(a) Design the regular expression for given transition diagram.



Ans.

$q_1 = q_1 0 + \wedge$ ...(i)
$q_2 = q_1 1 + q_2 1$ ...(ii)
$q_3 = q_2 0 + q_3 (0 + 1)$ ...(iii)

from equation (i)

$q_1 = q_1 0 + \wedge$

By Arden's rule

$q_1 = \wedge 0^*$
$q_1 = b^*$

Put the value of q_1 in equation (ii)

$q_2 = 0^* 1 + q_2 1$

By Arden's rule

$q_2 = 0^* 1 1^*$

Put the value of q_2 in equation (iii)

$q_3 = 0^* 1 1^* 0 + q_3 (0 + 1)$

By Arden's rule

$q_3 = 0^* 1 1^* 0 (0 + 1)^*$

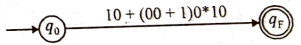
Since q3 is our final state therefore regular expression for this diagram is

$r = 0^* 1 1^* 0 (0 + 1)^*$

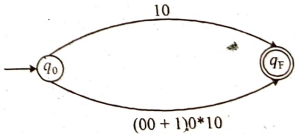
Q.2.(b) Construct a DFA with reduced states equivalent to the regular expression 10 + (00 + 1)0* 10.

Ans. The given regular expression is 10 + (00 + 1)0 * 10

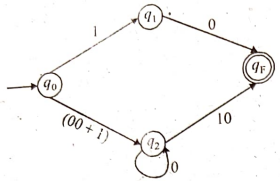
Step 1.



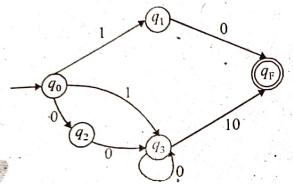
Step 2.



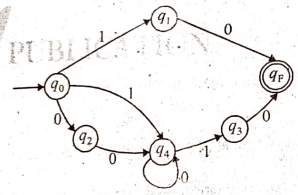
Step 3.



Step 4.



Step 5.



Step 6. We now design transition table of above drawn transition graph.

Input	0	1
q_0	q_2	$\{q_1\ q_4\}$
q_1	q_F	ϕ
q_2	q_4	ϕ
q_3	q_5	ϕ
q_4	q_4	q_3
q_5	ϕ	ϕ

The equivalent DFA will be

States	0	1
q_0	q_2	$[q_1\ q_4]$
$[q_0\ q_1]$	$[q_2, q_F]$	$[q_1\ q_4]$
$[q_0\ q_2]$	$[q_2, q_4]$	$[q_1\ q_4]$
$[q_0\ q_1\ q_2]$	$[q_2, q_F, q_4]$	$[q_1\ q_4]$
$[q_0\ q_2\ q_4]$	$[q_2, q_4, q_F]$	$[q_1\ q_4]$
$[q_0\ q_3\ q_4]$	$[q_2, q_F, q_4]$	$[q_1\ q_4, q_3]$
$[q_0\ q_4\ q_5]$	$[q_2, q_4]$	$[q_1\ q_4, q_3]$

Q.3. Explain the various phases of compiler.

Ans. Phases of Compiler : Following are the phases of compiler :

(i) Lexical Analysis : The lexical analysis is also called scanning. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group for strings called token. A token is a sequence of characters having a collective meaning. For example if some assignement statement in your source code is as follows.

total = count + rate * 10

Then in lexical analysis phase this statement is broken up into series of tokens as follows.

(a) The identifier total
(b) The assignment symbol
(c) The identifier count
(d) The plus sign
(e) The identifier rate
(f) The multiplication sign.
(g) The constant number 10

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

(ii) Syntax Analysis : The syntax analysis is also called parsing. In this phase the tokens generated by the lexical analyzer are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens togther. For the expression total = count + rate * 10 the phase tree and can be generated as follows.

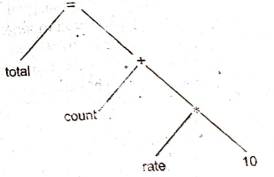


Fig.(1) : Parse tree for total = count + rate * 10

In the statement 'total = count + rate *10' first of all rate*10 will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of syntax tree the

production rules are to be designed. The rules are usually expressed by context free grammer. For the above statement the production rules are :

(1) $E \leftarrow$ identifier (2) $E \leftarrow$ number
(3) $E \leftarrow E1 + E2$ (4) $E \leftarrow E1 * E2$
(5) $E \leftarrow (E)$

where E stands for an expression.

- By rule (1) count and rate are expression and
- By rule (2) 10 is also an expression.
- By rule (4) we get rate*10 as expression.
- And finally count + rate*10 is an expression.

(iii) Semantic Analysis : Once the syntax is checked in the syntax analyzer phase i.e. the semantic anaylsis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if ... else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

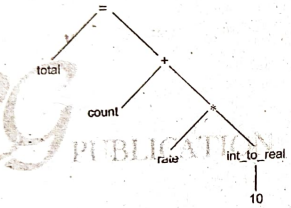


Fig.(2) : Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

(iv) Intermediate Code Generation : The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple posix*. Here we will consider an intermediate code in three address code form. This is like an assembly language.The three address code consists of instructions each of which has at the most three operands. For example,

t1 : = int_to_real (10)
t2 : = rate × t1
t3 : = count + t2
total : = t3

(v) Code Optimization : The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. Thus by optimizing the code the overall running time of the target program can be improved.

(vi) Code Generation : In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine insturctions.

MOV rate, $R1$
MUL # 10.0, $R1$
MOV count, $R2$
ADD $R2$, $R1$
MOV $R1$, total

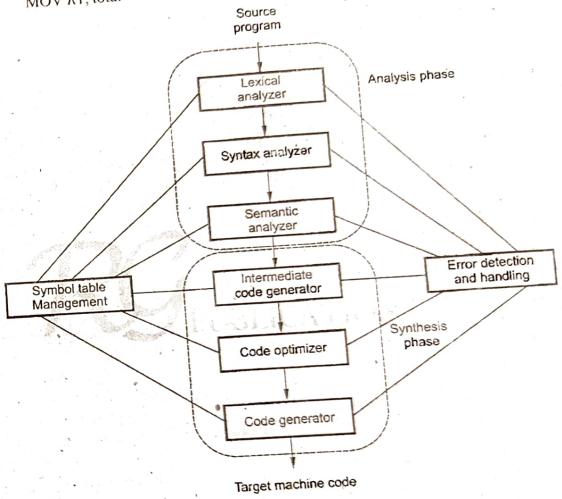


Fig.(3) : Phases of Compiler

(vii) Symbol table management : To support these phases of complier a symbol table is maintained. The task of symbol table is to store identifier (variables) used in the program.

The symbol table also stores information about attributes of each identifier. The attributes of identifiers are usually its type, its type, its scope, information about the storage allocated for it.

The symbol table also stores information about the subroutines used in the programs. In case of subroutine, the symbol table stores the name of the subroutine number of arguments passed to it, type of these arguments, the method of passing these arguments (may be call by value or call by reference) and return type if any.

Basically symbol table is a data structure used to store the information about identifiers.

The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record efficiently.

During compliation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attritbutes of an identifier and therefore the attributes are entered by remaining phases of complier.

Various phases can use the symbol table in various ways. For example while doing the semantic analysis and intermediate code generation we need to know what type of identifiers are. Then during code generation typically information about how much storage is allocated to identifier is seen.

(viii) Error detection and handling : To err is human. As programs are written by human beings therefore they can not be free from errors. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form the of message. When the input characters from the input do not from the token, the lexical analyzer detects it as error. Large number of errors can be detected in syntax phase. Such errors are popularly called as syntax errors. During sementic analysis: type mismatch kind of error is usually detected.

Section-B

Q.4.(a) What is predictive parsing ? Explain with example.

Ans. Predictive Parser : A special form a of recursive descent parser that needs no backtracking called predicitive parser.

This parser uses grammer from which left recursion is eliminated and also left factoring is applied as required.

A backtracking parser is a non-deterministic recognizer of the language generated by the grammer. The backtracking problems in the top-down parser can be solved; that is a top-down parser can function as a deterministic recognizer if it is capable of predicting or detecting which alternatives are right choices for the expansion of nonterminals during the parsing of input string e.g. predictive parser.

If $A \rightarrow a_1|\alpha_2|.... \alpha_n$ are the productions in the grammer, then a top-down parser can decide if a nonterminal A is to be expanded or not.

If A is to be expanded, the parser decides which A production should be used. It looks at the next input symbol and finds out which of the α_i derives to a string starting with a terminal symbol, it carries out the derivation of A using a production $A \rightarrow \alpha_i$, the input otherwise the parser reports the failure.

Q.4.(b) Check whether the given grammar is LL(1) ? Remove left recursion and then again verity whether it is LL(1) ?

$S \rightarrow Aa|bG$

$S \rightarrow Ac|Sd|\varepsilon$

Ans. The rule $A \rightarrow Ac \mid Sd \mid \varepsilon$ is a left recursive. Hence given grammar is not LL(1).

The left recursion can be removed as follows -

If $A \rightarrow A\alpha \mid \beta$ then $A \rightarrow \beta A'$, $A' \rightarrow \alpha A'$, $A' \rightarrow \varepsilon$.

$\therefore$

$$A \longrightarrow A\,c \mid S\,d \mid \varepsilon$$
$$\downarrow \quad \downarrow\downarrow \quad \downarrow \quad \downarrow$$
$$A \quad A\alpha \quad \beta \quad \beta$$

$\therefore$ $A \rightarrow S\,d\,A' \mid \varepsilon A'$

$A' \rightarrow c\,A' \mid \varepsilon$

$S \rightarrow Aa \mid b$

$A \rightarrow Sd\,A' \mid A'$ $\quad\therefore\quad$ $A \rightarrow \varepsilon\,A'$ becomes A'.

$A' \rightarrow CA' \mid \varepsilon$

The FIRST and FOLLOW computations are,

FIRST(S) = The first terminal apearing on R.H.S.
= {a, b, c}

FIRST(A) = {a, b, c, ε}

FIRST(A′) = {c, ε}

FOLLOW(S) can be computed as follows -

(i) As S is a start symbol add $ to FOLLOW(S)

(ii) $A \rightarrow Sd\,A'$

↑

i.e. *d* follows S

$\therefore$ *d* is added in FOLLOWS(S)

$\therefore$ FOLLOWS(S) = {d, $}

FOLLOW(A) can be computed as follows -

(i) $S \rightarrow Aa$

↑

i.e. *a* follows A

$\therefore$ *a* is added in FOLLOW(A)

$\therefore$ FOLLOW(A) = {*a*}

FOLLOW(A′) can be coputed as follows -

$$A \longrightarrow S\,d\,A'$$
$$\downarrow \quad \downarrow\downarrow$$
$$A \quad \alpha \quad B$$

By the rule if $A \rightarrow \alpha B$ then everything FOLLOWS(A) is in FOLLOW(B).

Hence FOLLOWS(A′) = FOLLOW(A) = {a}

Now we will design predictive parsing table as follows -

FIRST(S) = {a, b, c}	FOLLOW(S) = {*d*, $}
FIRST(A) = {a, b, c, ε}	FOLLOW(A) = {a}
FIRST(A′) = {c,ε}	FOLLOW(A′) = {a}

	a	b	c	d	$
S	S → Aa	S → b			
A	A → SdA′ A′ → ε	A → SdA′			
A′	A′ → cA′ A′ → ε				

As the table contains multiple entries in the M[A, a] and M[A′, a]. Hence given grammar is not LL(1).

Q.5.(a) Define LR parser. Write down its advantages and Limitations. Explain the algorithm of LR parser.

Ans. LR Parsers : In computer science, an LR parser is a parser for context-free grammars that reads input from Left to right and produces a Right most derivation.

LR parsers have several advantages :

(i) LR parsers can be implemented very efficiently.

(ii) Many programming languages can be parsed using some variation of an LR parser. One notable exception is C++.

(iii) LR parsers can detect syntatic errors as soon as possible in left to right scan of input.

(iv) LR parsers are more general than operator precedence or shift-reduce parsers.

(v) LR parsing dominates the common forms of top-down parsing without backtrack.

The main drawback of LR parsing is that too much work is needed to construct an LR parser by hand for a typical programming language. Therefore. LR parsers are usually constructed by a parser generator or a compiler-compiler.

LR Parsing Algorithm : As LR parser reads the input from left to right but needs to produce a right most derivation. Therefore, it uses reductions, instead of derivations to process the input. That is, the algorithm works by creating a "Left most reduction" of the input. The end result, when reversed, will be rightmost derivation.

Before describing the working of the LR parsing algorithm, we define the term configuration. A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the remaining input. IF the stack contains $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$ with s_m on the top of the stack and the input is $a_i a_{i+1} \ldots a_n \$$ then the configuration of an LR parser is represented by

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, \quad a_i X_{i+1} \ldots a_n \$)$$

Symbol on the top of stack Current input symbol

Depending upon the state(s) 0 the top of stack and the current input symbol (a_i) the LR parser consults an action table entry ACTION $\{s_m, a_i\}$ until the string is accepted or a syntax error is reported.

The entry ACTION $[s_m, a_i]$ can have one of four values :

(i) a shift s_n, in which

- the current terminal is removed from the input stream.
- the state n is pushed onto the stack and becomes the current state.

(ii) a reduce *rm*, in which

- the number m is written to the output stream.
- for every symbol in the right-hand side of rule m a state is removed from the stack.
- given the state that is then on the top of the stack and the left hand side of rule m a new state is looked up in the GOTO table and made the new current state by pushing it onto the stack.

(iii) an accept, meaning that string is accepted.

(iv) An error indicating by blank entries meaning that a syntax error is reported.

Q.5.(b) Explain canonical LR parser.

Ans. CLR : A canonical *LR* (*CLR*) or *LR* (1) parser is an *LR* parser in which parsing tables are constructed with similar ways as LR (0) parsers except that the items in the item sets also contain a lookahead, i.e., a terminal that is expected by the parser after the right hand side of the rule. For example, such an item for a rule $X \rightarrow Y Z$ might be $X \rightarrow Y \bullet Y, a$; which means that the parser has to read a string corresponding to Y and expects next a string corresponding to Z followed by the terminal 'a'.

Construction of the canonical LR (CLR) :

Input : An augmented grammar G'.

Output : The canonical LR parsing table functions action and goto for G'

Method :

(i) Construct $C = \{I_0, I_1, \ldots\ldots, I_n\}$, the collection of sets of LR(1) items for G'.

(ii) State I of the parser is constructed from I_i. The parsing actions for state I are determined as follows :

– If $[A \rightarrow \alpha. a, b]$ is in I_i and goto(Ii, a) = Ij, then set action[i,a] to "shift j." Here, a is required to be a terminal.

– If $[A \rightarrow \alpha., a]$ is in I_i, $A \neq S'$, then set action[i,a] to "reduce $A \rightarrow \alpha$."

– If $[S' \rightarrow S., \$]$ is in I_i then set action[i ,$] to "accept."

If a conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to be fail.

(iii) The goto transition for state i are determined as follows: If goto(I , A)= I ,then goto[i,A]=j.

(iv) All entries not defined by rules(ii) and (iii) are made "error."

(v) The initial state of the parser is the one constructed from the set containing item $[S' \rightarrow .S, \$]$.

Consider the following augmented grammar :

S' → S

S → CC

C → Cc | d

The initial set of items is :

I_0 : S' → .S , $

S → .CC, $

C → .Cc, c | d

C → .d, c | d

We have next set of items as :

I_1 : $S' \rightarrow S.,\$$
I_2 : $S \rightarrow .Cc,\$$
$C \rightarrow .Cc,\$$
$C \rightarrow .d,\$$
I_3 : $C \rightarrow c.C,\$$
$C \rightarrow .c\,C\,.\,c \mid d$
$C \rightarrow .d,\$$
I_4 : $C \rightarrow d.\,.\,c \mid d$
I_5 : $S \rightarrow CC.\,.\,\$$
I_6 : $C \rightarrow c.C,\$$
$C \rightarrow .c\,C\,.\$$
$C \rightarrow .d\,.\,\$$
I_7 : $C \rightarrow d.\,.\,\$$
I_8 : $C \rightarrow c\,C.\,.\,c \mid d.$
I_9 : $C \rightarrow c\,C.\,.\,\$$

Section-C

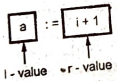
Q.6. What is symbol table ? Explain the data structures used to construct symbol tables.

Ans. Symbol Tables : Symbol table is a data structure used by compiler to keep track of semantics of variables. That means symbol table stores the information about scope and bindng information about names.

- Symbol table is built in lexical and syntax analysis phases.
- The symbol table is used by various phases as follows - Semantic analysis phase refers symbol table for type conflict issue. Code generation refers symbol table knowing how run time space is allocated ? What type of run time space is allocated ?
- I-value and r-value

The l and r prefixes come from left and right side assignement.

For example :



Data Structures for Symbol Tables : Requirements for symbol table management.

(i) For quick inserstion of identifier and related information.
(ii) For quick searching of identifier.

Following are commonly used dataa structures for symbol table construction.

(1) 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 record. The fig.(1) for list data structure using arrays is as given below,

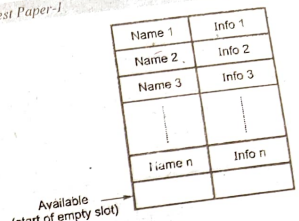


Fig. : (1)

- 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 occurs i.e: "Multiple defined Name".
- The advantages of list organization is that it takes minimum amount of space.

(2) 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 the link field.
- A pointer "First" is maintained to point to first record of the symbol table.

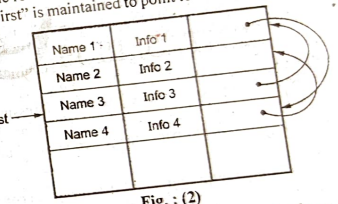


Fig. : (2)

The reference to these names can be Name 3, Name 1, Name 4, Name 2.

- 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 freqeunctly referred names will be the least.

(3) Hash tables : Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.

- In hashing scheme two tables are maintained - a hash table and symbol table.
- The hash table consists of k entries from 0.1 to $k-1$. These entriss are basically pointers to symbol table pointing to the names of symbol table.
- The determine whether the 'Name' is in symbol table, we use a hash functions 'h' such that h (name) will result any integer between 0 to $k-1$. We can search any name by position = h (name)

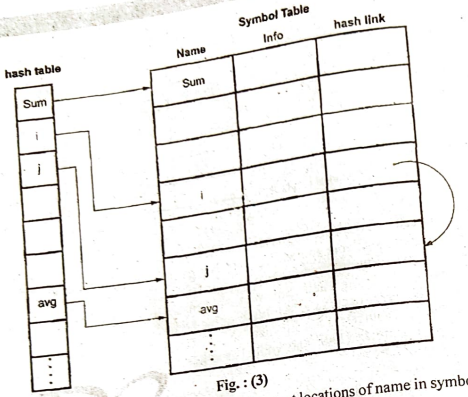


Fig. : (3)

- Using this position we can obtain the exact locations of name in symbol table.
- The hash table and symbol table can be as shown in fig.(3).
- The hash function should result in unifrom 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 of hashing is 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.

Q.7.(a) Discuss about syntactic and semantic phase errors.

Ans. Syntactic Phase Errors : These are the errors which are popularly known as syntax errors. They appear during syntax analysis phase of compiler.

Typical phase errors are :

- Errors in structure
- Missing operators
- Misspelled keywords.
- Unbalanced parenthesis.

For example :

```
Switch (Choice) :
{
    ....
    ....
}
```

The keyword switch is incorrectly written as switch. Hence "unidentified keyword/ identifier". This error occurs.

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

Strategies to Recover from Syntactical Errrors : Parser employs various strategies to recover from syntatic errors. These strategies are

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

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

(i) *Panic mode* : This strategy is used by most parsing methods. This is simply to implement. In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon or end. These tokens indicate an end of input statement. Thus in panic mode recovery a considerable amount of input in skipped without checking it for addtional errors.

This method guarantees not to go in infinite loop. If there are less number of error in the same statement then this strategy is a best choice.

(ii) *Phrase level recovery* : In this method, on discovering error parser local correction on remaining input. It can replace a prefix of remaining input by some string. This actually helps the parser to continue its job.

The local correction can be replacing comma by semicolon, deletion of extra semicolons or inserting missing semicolon. The type of local correction is decided by compiler designer.

While doing the replacement a care should be taken for not going in an infinite loop.

This method is used in many error reparing compilers.

The drawback of this method is it finds difficult to handle the situations where the actual errors has occured before the point of defection.

(iii) *Error production* : If we have a knowledge of common errors that can be encountered then we can incorporate these errors by augmenting the grammer of the corresponding language with error productions that generate the erroneous constructs.

If error production is used then during parsing, we can generate appropriate error message and parsing can be continued.

This method is extremely difficult to maintain. Because if we change the grammer then it becomes necessary to change the corresponding error productions.

(iv) *Global production* : We often want such a compiler that makes very few changes in processing an incorrect input string. We expect less number of insertions, deletions, and changes of tokens to recover from erroneous input.

Such methods increase time and space rquirements at parsing time. Global production is thus simply a theoretical concept.

Semantic Errors : Semantic errors are those errrors which get detected during semantic analysis phase.

Typical errors in this phase are :

- Incompatible types of operands
- Undeclarced variables.
- Not matching of actual arguments with formal arguments.

For example :
int a[10], b ;
....
....
a = b ;
It generates a semantic error.

Error Recovery : If the error "Undeclared identifier" is caught then to recover from this error the symbol table entry for corresponding identifier is made.

If the types of two operands in an expressions are not compatible with each other then automatic type conversion is done by compiler. This type of conversion is called implicit conversion.

Q.7.(b) What is three address code ? Explain the type of three address statements.

Ans. Three Address Code : Three address code is the typical example of low level representation. The term "three address code" contains three addresses two for the operands and one for the result.

Three address code is a sequence of statement of the general form

p: q op r

where p, q and r are names constants or compiler generated temporaries.

- op stands for any operator, such as fixed or floating point arithmetic operator or logical operator of Boolean value.

- There is only one operator on the right hand side of statement ensuring that operations and expression are simple. So no built in operators are allowed.

- A source language expression like x : x * y + z – x translated into three address sequence as,

$t_1 := x\ y$
$t_2 := t_1 + z$
$t_3 := t_2 - x$
$x := t_3$

where t_1, t_3 are compiler generated temporary names.

- Three address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

- Following Table represents the three address sequence for the syntax tree and dag in Table. Variables names can appear directly in tree address statements so Table (a) has no statements corresponding to the Table (a).

$t_1 := -c$ $t_2 := b * t_1$ $t_3 := -c$ $t_4 := b * t_3$ $t_5 := t_2 + t_4$ $a := t_5$	$t_1 := -c$ $t_2 := b * t_1$ $t_5 := t_2 + t_1$ $a := t_5$
(a) Code for the syntax tree	(b) Code for the dag.

Table : Three address code corresponding to the tree and dag.

- Three address code gives better flexibility in terms of target code generation and code optimization as its is simpler.

- The semantic rules for generating three address code from common programming language constructs are similar to those for constructing syntax trees for generating postfix notation.

Types of Three Address Statements : The form of three-address code is very much similar to assembly language. The intermediate language have following types of three address statements.

(i) Assignment statement : It is the form $x := y\ op\ z$, where *op* is a binary arithmetic or logical operand which is applied to the operand *y* and *z*.

It is also in the form $x : op\ y$, where *op* is a unary which is applied to the operand *y*. Unary operator can be unary minus, logical negation, shift operations and conversion operations.

(ii) Copy statement : It is in the form $x := y$ where the value of *y* is assigned to *x*.

(iii) Jump : Both conditional and unconditional jumps are used. The unconditional jump is of the form goto L where L is Label. i.e. control is transferred to the three address statement with label L.

The conditional jump is of the form if *x relop y goto L* where *L* is the label. Here the relop represents the relational operations such as <, >, <=, >=, ==. If *x* relop *y* is true then it transfer control to label *L*.

(iv) Procedure call/return : A call to the proceduce $p\ (x_1, x_2, x_3, ..., x_n)$ Where x_1, x_2, x_3,..... x_n are the paraments for the procedure and suppose y is the return value which is again optional. Then this procedure call is translated into following three address statements.

param x_1
param x_2
.....
param x_n
call *p*, *n*

The integer *n* indicates the number of actual parameter in "call *p*, *n*".

(v) Indexed assignment : It is the from $x := y\ [i]$ and $x\ [i] := y$. The first of these form assign the value at the i^{th} index of array *y* to *x* while second form assign value of *y* to the *x* at the i^{th} index. In both these form *x*, *y* and *i* refer to data objects.

(vi) Address and pointer assignments : It is in the from x : & *y*, and x : = *y, and *x : = *y*. In the first form the value of *x* will be the address of *y*. In the second form *x* is assigned the contents of while in next the *r*-value of object pointed by *x* is set by the l value of *y*.

(vii) Miscellaneous : Three may be some other statements needed depending upon the source language, One such statements to define the jump target is Label L.

The choice of the allowed operatior is very important issue in the design of an intermediate represnetation form. The operator set used by source language should be rich to implement all the operations in the source language. But is very easy to small operator set on a new target machine. And the rich instruction set force the front end to generate long sequences for some source language operations.

Section-D

Q.8.(a) Explain basic blocks and flow graphs in code generation.

Ans. Basic block : Basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at end statement without any halt and any possibility of branching except at the end statement. Thus, halting or branching statement may appear as last or ending statement of Basic Block.

A name in basic block is said to be live at given point in the program if its vlaue is used after that point in program (may or may not be in same block). A name in basic is said to be dead at given point in the program if its value is never used after that point in program (may or may not be in same block).

The three address statement $a : = b + c$ is said to define a and to define and to use b and c.

Each Basic block may contain many 3 address statements but each 3 address statement is in only one Basic block.

Simple example of Basic Block can given as :

$t_1 : = a * 4$
$t_2 : = b * b$
$t_3 : = 2 * t_2$
$t_4 : = t_1 * t_3$
$t_5 : = b - b$
$t_6 : = t_4 * t_5$

Flow Graph : A program is a set of instructions performing a task. The instruction of the programs are grouped to from basic blocks.

Flow graph is a directed graph which is graphical representation of flow of control among all basic blocks constituting program. Flow graph is used to add flow of control information to the set of basic block making up a program. Flow graph has one distinguished node as initial node whose leader statements is first statement of the program.

Let program have have two blocks B_1 and B_2.

B_1 is said to be predecessor of B_2 or B_2 is said to be successor of B_1 if there is direct edge from B_1 to B_2 i.e. B_2 immediately follows B_1 in any execution sequence.

There is directed edge from B_1 to B_2 if

(a) There is conditonal or unconditional jump from last statement of B_1 to first statement of B_2 or

(b) B_2 immendiately follows B_1 in order of the program and last statement of B_1 is unconditional statement.

So B_1 is called as predecessor of B_2 and B_2 is called as successor of B_1.

Q.8.(b) Explain the transformation on basic blocks.

Ans. Transformation on Basic Blocks : Each basic block computes some set of expressions. Two basic blocks are said to be equivalent if they compute same set of expressions. Many transformations can be applied to Basic blocks without changing expression computed by that Basic Block.

Transformations are applied to generate code of improved quality. These transformation leading to improved quality code rearranges computations such that required time and space is reduced.

There are two types of transformations on basic blocks.

(a) Structure presuming Transformation

(b) Algebraic Transformation

(a) *Structure presuming Transformation* : Primary structure presuming tranformation on basic blocks may include transformation which improves of code by.

(i)Eliminating redundant operands or expressions.

(ii)Eliminating dead codes.

(iii)Renaming temporary variables.

(iv) Interchanging of statements.

Code of improved quality mean improved code that performs same task as that of input code but needs lesser resources

Following are the different of primary structured transformation on basic blocks.

(i) Common subexpression elimination

(ii) Dead code elimination

(iii) Renaming temporary variable

(iv) Interchanging of two independent adjacent statements.

(b) *Algebraic Transformation* : Countless algebraic transformation are algebraic terantransformation that do not change value of varible on left hand side (*LHS*) of the statement.

Example : $p : = p + 0$ or

$p : = p + 1$

These two types of statements can be eliminated from basic block without changing value of Block.

Consider another statement $a : = b$ ** 2 usually requires a function call for its implementation. But same operation can be executed by cheaper statement.

$a : b * b$

Thus algebraic transformation one also used to simplify basic block.

Q.9. Write short notes on :

(a) Machine dependent code optimization.

(b) Code generation.

(c) Register allocation and assignment.

Ans.(a)Machine dependent code optimization : Machines dependent code optimization techniques improve the target code by applying transformation and for that it considers the following characteristics or properties of the target machine.

(a)Complete knowledge of the target machine.

(b)Register allocation efficient allocation and utilization of sufficient number of resources.

(c)Machine instruction use of immediate instructions whereover necessary.

(d)Different addressing modes.

(e)Programmng structure of the target machine.

(f)Arithmetic properties.

Machien dependent optimization can be achieved by.

(i)Efficient allocation and used of machine registers.

(ii)Use of machine idoms or inductions directly.

Ex. for $i = i + 1$ we can use INC i

Machine dependent optimzation can be efficiently applied by code generation phase of complifer.

Ans.(b)Code generation : The code generation is the last phase in the compilation process. In computer science, the code generation is the process which converts some internal representation of source code into a form (e.g., machine code) that cna be readily executed by a machine. The code generation is done by a code generator component of a compiler. The code generator translates the intermediate representation of the source program into a sequence of machine instructions.

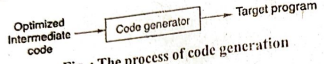


Fig. : The process of code generation

The input to a code generator is an intermediate language program that can be :

(i) A sequence of quadruples.

(ii) A sequence of triples.

(iii) A tree, or

(iv) A postfix polish string.

The output of a code generator is called the target program (or object program) that can be :

(i) An absolute machine language program

(ii) A relocatable machine language program

(iii) An assembly language program

(iv) A program in some other programming language.

Ans.(c)Register allocation : It is the process of just selecting set of variables that will reside in registers at any point in program.

Regiser assignment : After Register Allocation explicitly specific register are picked in which any variable can reside.

Optimal register assigement to varibles is difficult. Mathematicall it is NP-Complete problem.

Certain machines require register pairs such as even odd numbered register for some operand and results.For example in IBM system integer multiplication and integer divison requires pairs.

Consider following three address code :

$t = a / b$

$t = t + c$

$t = t * d$

The efficient target machine code sequence will be

```
MOV   a, R0
DIV   b, R0
ADD   c, R0
MUL   d, R0
MOV   R0, t
```

COMPILER DESIGN

Model Test Paper-II

Paper Code: PCC-CSE-302-G

Note: *Attempt five questions in all, selecting one question from each section. Q. No. 1 is compulsory.*

Q.1.(a) What is the different between DFA and NDFA ?

Ans. Differences between DFA and NDFA are as follows :

(1) "DFA" stands for "Deterministic Finite Automata" while "NDFA" stands for "Nondeterministic Finite Automata."

(2) Both are transition functions of automata. In DFA the next possible state is distinctly set while in NDFA each pair of state and input symbol can have many possible next states.

(3) NDFA can use empty string transition while DFA cannot use empty string transition.

(4) NDFA is easier to construct while it is more difficult to construct DFA.

(5) Backtracking is allowed in DFA while in NDFA it may or may not be allowed.

(6) DFA requires more space while NDFA requires less space.

(7) While DFA can be understood as one machine and a DFA machine can be constructed for every input and output, NDFA can be understood as several little machines that compute together, and there is no possibility of constructing an NDFA machine for every input and output.

Q.1.(b) What are the different phases of complier ?

Ans. Figure shows the structure of a compiler or various phase of compiler.

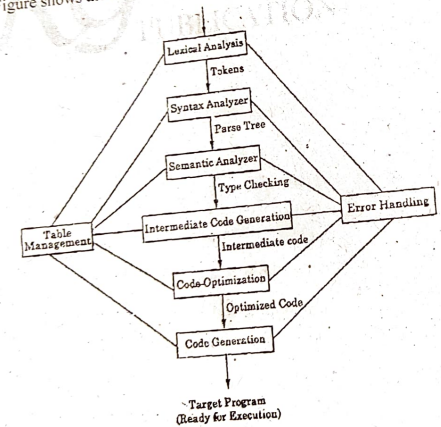


Fig. : Phases of Compiler

Q.1.(c) What is the role of parser?

Ans. The Role of the Parser (Syntax Analyzer) : Parser or Syntax analyzer is the program which performs syntax analysis or parsing.

Parsing is an important phase of the compiler design, which obtains string of tokens produced as output of Lexical Analysis.

Parser then groups tokens appearing in it's input as a statement to identify larger structures in the program. If any syntax error is found, it reports and implements recovery technique.

Q.1.(d) What is shift reduce parsing?

Ans. Shift reduce parseing attempts to construct parse tree leaves to root. Thus it works on the same principle of bottom up parser. A shift reduce parser requires following data structures:

(i) The input buffer storing the input string.

(ii) A stack for storing and accessing the L.H.S and R.H.S of rules.

Q.1.(e) Define SLR.

Ans. SLR (1) are simple LR easiest to implement but some what weak in terms of the member of grammars for which it succeeds. Parsing table constructed by this method is called SLR table. A grammer for which an SLR parser can be constructed is said to be an SLR grammar.

Q.1.(f) What is intermediate code ?

Ans. Intermediate Code : The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple posix*. Here we will consider an intermediate code in three address code form. This is like an assembly language.The three address code consists of instructions each of which has at the most three operands. For example,

t1 : = int_to_real (10)
t2 : = rate × t1
t3 : = count + t2
total : = t3

Q.1.(g) What do you mean by DAG ?

Ans. The DAG (Directed Acyclic Graph) is a useful data structure for automatically analyzing the basic blocks. A DAG is a directed graph with no cycles which gives a picture of how the value computed by each statement in a basic block is used in the subsequent statements in the block.

Q.1.(h) What is semantic error ?

Ans. Semantic errors are those errors which get detected during semantic analysis phase. Typical errors in this phase are :

(i) Incompatiable types of operands.

(ii) Undeclarted varaibles.

(iii) Not matching of actual arguments with formal arguments.

Section -A

Q.2.(a) What is translator? Why need translator? Explain different type of translator.

Ans. Translator : A translator is defined as a software program that takes as input a program written in source language and produces as output a program in another language called the object or target language. The concept of translating a program from source to object or target language is shown in fig.

Fig. : The Concept of program translation.

Need : Translators are needed to convert one form of the program into the other. Writing application program in the object code language is not possible. Hence the application programs are normally written in low level or high level languages, because these languages are English like languages and easiler to use. The example of low level language is assembly language and examples of high level languages are – C, C + +, FORTRAN and so on.

But the computer can not understand these type of languages, it only understands the machine code or object code. Further more conversion of this object code can be done easily in the form of binary language (i.e.the language of 0 and 1). Hence translator is used as a mediator to convert high/low level languages into machine level languages.

Assembler is a translor which converts the assembly language into machine language.

Compiler is a translator which converts the high level languages like C or C + + into machine language.

Types of Translator are :

(i) Compiler : A compiler is a translator (a software program) that translates a high-level language program into functionally equivalent low-level machine language program which can be exected direct. Individual source language statements usually map into many mchine-level instructions. A compiler also generates diagnostic message wherever the specifications of the source language are violated by the programmer.

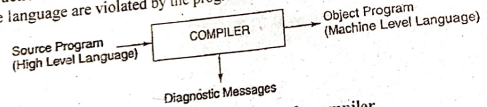


Fig. : The functional diagram of a compiler.

(ii) Interpreter : An interpreter like compiler is a translator that translates a high-level language program into functionally equivalent low-level machine code, but it does it at the moment the program is run. An interpreter reads a source program written in a high-level programming language as well as data for this program, and it runs the program against the data to produce some results.

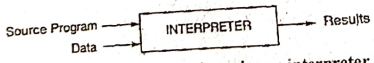


Fig. : A language translation using an interpreter.

(iii) Assembler : An assembler is a translator that translates an assembly level language program into low-level machine language program which can then be executed directly. Individual source language statements usually map one-for-one to machine-level instructions. The translation using an assembler is shown in Fig.

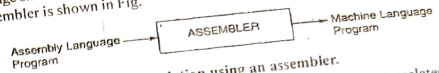


Fig. : A translation using an assembler.

(iv) Macro Assembler : A macro-assembler is a translator that translates the assembly level language instructions into machine code, and is a variation on the assembler. Most source language statements map one-for-one into their target language equivalents, but some macro statements map into a sequence of machine-level instructions-effectively providing a text replacement facility, and thereby extending the assembly language to suit the user. The concept of macro is shown in Fig.

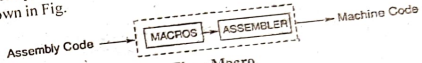


Fig. : Macro.

(v) Preprocessor : A preprocessors is a translator that translates a superset of a high-level language into the original high-level language, or that perform simple text substitutions before the translation takes place. This enables the statements of the source program to be precessed first by means of certain directives and its meaning is substituted in the source program before it goes to final compilation. The Fig. shows the preprocessor as a translator.

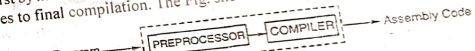


Fig. : Preprocessor

Loaders and Linkers : Loader is system programs. Loader loads the binary code in the memory ready for execution. Transfer the control to 1st instruction. Loaders are responsible for locating program in the main memory every time it is being executed.

Linking is the process of combining various pieces of code and data together to form a signle executable that can be loaded in memory.

Q.2.(b) Explain various compiler construction tools.

Ans. The principal tools useful for compiler-construction are described below :

(i) Scanner generators (Lexical analyzer generator) are used to automatically generate lexical analyzers from the specifications based on regular expressions. For example, LEX is a lexical analyzer generator available on UNIX. The basic organization of the resulting lexical analyzer is a finite automation.

(ii) Parser generators are used to generate the syntax analyzers from the input that is based on context-free grammer (CEG). For example, YACC is a LALR parser generator available as a command on the unix system. YACC stans for "Yet Another Compiler-Compiler'. One

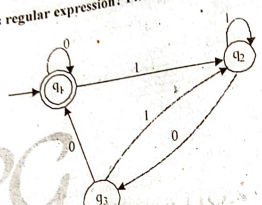
significant advantage of using a parser generator is increased reliability. A mechanically-generate parser is more likely to be correct than one produced by hand.

(iii) Syntax-directed translation engines, are used to produce the collections of routine that traverse the parse tree which generate intermediate code.

(iv) Automatic code generators are used to generate the machine language for t target machine by translating each operation of the intermediate language using a collection rules that define those translation.

(v) Date-flow engines, are used to perform data-flow analysis needed to perfor good code otpimization. The data flow-analysis involves the gathering of information about ho values are transmitted from one part of program to each other part.

Q.3.(a) What is regular expression? Find the regular expression correspondenc to



Ans. Regular expression : Regular expressions are tiny units, which are useful fo representing the set of strings belonging to some specific language.

There is only one initial state. Also there are no Λ-moves. The equations are

$$q_1 = q_1 0 + q_3 0 + \Lambda$$
$$q_2 = q_1 1 + q_2 1 + q_3 1$$
$$q_3 = q_2 0$$
$$q_2 = q_1 1 + q_1 1 + (q_2 0)1 = q_1 1 + q_2(1 + 01)$$

So,

By applying Arden's Theorem, we get

$$q_2 = q_1(1 + 01)^*$$

Also,

$$q_1 = q_1 0 + q_3 0 + \Lambda = q_1 0 + q_2 00 + \Lambda$$
$$= q_1 0 + (q_1 1 (1 + 01)^*)00 + \Lambda$$
$$= q1(0 + 1(1 + 01)^* 00) + \Lambda$$

Once again applying Arden's Theorem, we get

$$q_1 = \Lambda(0 + 1)(1 + 01)^* 00)^* = (0 + 1(1 + 01)^* 00)^*$$

As q_1 is the only final state, the regular expression corresponding to the given diagram is $(0 + 1(1 + 01)^* 00)^*$.

Q.3.(b) What is the role of a lexical analyzer? Explain how to design lexical analyzer.

Ans. Role of a lexical analyzer : Lexical analyer is the first phase of compiler. The lexical analyzer reads the input sources program from left left to right one character at a time and generates the sequare of tokens. Each token is a single logical cohesive unit such as identifier, keywords operators and punctuation marks. Then the parser to determine the syntax of the source program can use these tokens. The role of lexical analyzer in the process of compilation is as shown below :

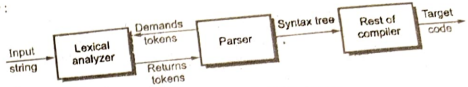


Fig. : Role of Lexical Analyzer.

As the lexical analyser scans the source program to recognize the tokens it is also called as scanner.

Functions of lexical analyzer :

(i) It produces stream of tokens.
(ii) It eliminates blank and comments
(iii) It generates sysmbol table which stores the information about identifiers, constants encountered in the input.
(iv) It keeps track of line numbers
(v) It reports the error encountered while generating the tokens.

The lexical analyzer works in two phases. In first phase it performs scan and in the second phase it does lexical analysis; means it generates the series of tokens.

Construction of Lexical Analyzer :

- Lexical analyers can be constructed in two ways.
- First method involves writting a progam to do the lexical analysis.
- Another method uses automatic generation of lexical program which is faster.
- But with coding lexical analyzer is more efficient .
- For coding, lexical analyzer needs tokens and grammer as shown in example below.

Example :

stmt	→	if expr then stmt | if expr then stmt else stmt | ε
expr	→	term relop term | term
term	→	id |num
if	→	if
then	→	then
else	→	else
relop	→	< | < = | = | < > | > | > =
id	→	letter (letter/digit)*
num	→	[0 – 9] +

- Here the terminals are if, then else id, relop, num.
- Set of strings are defined by regular definition in Example.

- In programming tokens (which) are going to the declared matched by several different regular expressions e.g. if, else, while either which may lead to ambiguity. To resolve this, we must give preference to reserve word, if string is matched by reserse word by an identifier.
- In addition to this lexemes are separated by delimiters like white space, tab, newlines. We have to define white space and if it match lexical analyzer will ignore that token and not return is to parse which strip out the while spaces. We can define white space as follows :

delim → blank | tab| newline
ws → delim + (this is rule for white spaces)

Section -B

Q.4. What do you mean by grammer? Explain context Free Grammer? Write a CFG which generate palindronte for binary number.

Ans. Defination of Grammer : A phase structure grammer (or simply grammer) is (V_n, Σ, P, S)

(i) where V_n is finite nonempty set whose elements are called variables.
(ii) Σ is finite nonempty set whose elements are called terminals.
(iii) $V_n \cap \Sigma = \phi$
(iv) S is specific variable (i.e., an element of V_n) called the start symbol.
(v) P is finite nonempty set whose elements are $\alpha \rightarrow \beta$.

Context Free Grammer : Context Free Grammer (CFG) is a type of grammer used to define the way the statements are written for a programming language like Pascal or C. Thus we can say that CFG gives the syntactic specification of progamming languages.

Basically the grammer can be defined as $G = (N, T, P, S)$

T is Set of terminals represented as small letters. E.g. = {a, b, c, ...}. These represent words of the language which are converted to tokens by the lexical analyzer so here a, b, c represent tokens. These tokens are arranged in the combination with these meta symbols called non terminals.

N is Set of Non terminals represented as captical letters E.g = {A, B, C,.....}. These are the meta symbols used to define the production rules.

P is a set of production rules. These rulers are used to produce synatactically correct statement. These rules are the grammer rules which decide how the statements are going to be written. Few production rules are defined below.

$S \rightarrow aA \mid bB$
$A \rightarrow abB \mid aB$
$B \rightarrow a \mid b \mid \varepsilon$

S is a special symbol called start symbol. Here in the above production rules, S is the start symbol.

So we can see one example

where
$G = (N, T, P, S)$
$N = (A, B)$
$T = (a, b)$
$P = \{A \rightarrow abB \mid aB, B\ a \mid b \mid \varepsilon\}$
$S = A$

The sentences produced by this grammer can be as follows :
{*a, aa, ab, aba, abb*}. All grammer can be grouped to form classes. These groups are formed according to the way the production rules are specified. The conditions imposed on the grammer rules will determine how the rules can be mentioned. The main component that defines the grammer are the production rules. And the way they are written.

$G = N = \{S,E,O\}$, $T = \{$if, then, else, when, while, do, (,), =, +, *$\}$, P, S)
$S \rightarrow$ if E then S else S| while E do S| ε
P is as follows
$E \rightarrow E\ O\ E$
$E \rightarrow (E)$
$E \rightarrow = E$
$E \rightarrow id$
$O \rightarrow + \mid - \mid *$

CFG which generate palindronte for binary number : The grammer will generate palindrome for binary numbers, that is 00, 010, 001100,....

Let G be the *CFG* $G = (V_n, V_t, P, S)$
$V_n = \{S\}$
$V_t = \{0, 1\}$
And production rule P is defined as
S - > 0S0/ 1S1
S - > 0/1/ε

Q.5. What is parsing? Explain top-down parsing and predictive parsing techniques. Check the following grammer is LL(1) or not
S → AaAb/BbBa A → ε B→ ε

Ans. Parsing : Parsing also called as Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning.

Top down Parser : Top down parsing is a strategy to find the leftmost derivation of an input string. In top down parsing, the parse tree is constructed starting from the root and proceeding toward the leaves (Similar to a derivation), generating the nodes of the tree in preorder.

For example, consider the following grammer.

$E \rightarrow cDe$
$D \rightarrow ab \mid a$

For the input string *cae*, the leftmost derivation is specified as :

$E \Rightarrow cDe \Rightarrow cae$

The derivation tree for the input string cae is shown in fig.(a)

Predictive parsing (Non backtracking parsing) : Predicative parsing does not require backtracking in order to derive the input string. Predicative parsing is possible only for the class of *LL* (*k*) grammer (context free grammar). The grammer should be free from left recursion and left factoring. Each non-terminal is combined with the next input signal to guide to parser to select the correct production rule that will lead the parser to match the complete input string.

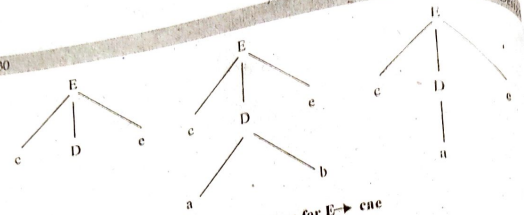


Fig.(a) : Phaser Tree for E⟶ cne

The given grammer is

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \in$

$B \rightarrow \in$

$FIRST(A) = \{\in\}$ $FOLLOW(S) = \{\$\}$

$FIRST(B) = \{\in\}$ $FOLLOW(A) = \{a,b\}$

$FIRST(S) = \{a,b\}$ $FOLLOW(B) = \{a,b\}$

The corresponding parsing table is given in Fig.(1).

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBA$	
A	$A \rightarrow \in$	$A \rightarrow \in$	
S	$A \rightarrow \in$	$A \rightarrow \in$	

Fig.(1) : Parsing table

As the parsing table of Fig.(1) contains no multiply defined entries. Therefore, the given grammar is LL(1) grammer.

The augmented grammer corresponding to given grammar is

$S' \rightarrow S$

1. $S \rightarrow AaAb$
2. $S \rightarrow BbBa$
3. $A \rightarrow \in$
4. $B \rightarrow \in$

The canonical collection LR(0) itms is constructed as given below

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

$GOTO(I_0,S) = I_1$: $S' \rightarrow S.$

$GOTO(I_0,A) = I_2$: $S \rightarrow A.aAb$

$GOTO(I_0,B) = I_3$: $S \rightarrow B.bBa$

$GOTO(I_2,a) = I_4$: $S \rightarrow Aa.Ab$

$A \rightarrow .$

$GOTO(I_3,b) = I_5$: $S \rightarrow Bb.Ba$

$B \rightarrow .$

$GOTO(I_4,A) = I_6$: $S \rightarrow AaA.b$

$GOTO(I_5,B) = I_7$: $S \rightarrow BbB.a$

$GOTO(I_6,b) = I_8$: $S \rightarrow AaAb.$

$GOTO(I_7,a) = I_9$: $A \rightarrow BbBa$

The corresponding GOTO graph is shown in Fig.(2).

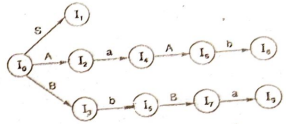


Fig.(2) : The GOTO graph

State	ACTION			GOTO		
	a	b	$	S	A	B
1	r3/r4	r3/r4	acc	1	2	3
2	s4					
3		s5			6	
4	r3	r3				7
5	r4	r4			r1	
6		s8				
7	s9					
8			r1			
9			r2			

Fig.(3) : The SLR parsing table

The table of Fig.(3) contains two multiply defined entries (reduce-reduce conflicts). Therefore, the given grammer is not SLR(1).

Section -C

Q.6.(a) What is syntax directed translation scheme ? How they are used to make syntax tree ?

Ans. Syntax Directed Translation Schemes : A syntax directed translation scheme is a formalism for describing the process of translation with in a syntax directed compiler. A

syntax directed translation scheme is a context-free grammar in which the program fragments called semantic actions (or output action or semantic rule) are embedded with in the right side of productions. For example, suppose an output action α is associated with production $A \rightarrow XYZ$. The action α is executed whenever the syntax analyzer recognizes in its input a substring w which has a derivation of the form $A \Rightarrow XYZ \Rightarrow w$.

A translation scheme is like a syntax directed definition except that the order of the evaluation of the semantic rules is explicitly shown. The position at which an action is to be executed is shown by enclosing it between braces and writing it with in the right side of a production.

Syntax-Directed Translation Scheme for Syntax Trees : In the syntax-directed translation scheme for syntax trees, the translation E. VAL is associated with a non-terminal E. The corresponding syntax-directed translation scheme is given in following table.

Sr. No.	Production rule	Semantic action
1.	$E \rightarrow E^{(1)}$ op $E^{(2)}$	{E.VAL : = NODE(*op*, $E^{(1)}$.VAL, $E^{(2)}$.VAL)}
2.	$E \rightarrow (E^{(1)})$	{E.VAL : = $E^{(1)}$.VAL}
3.	$E \rightarrow -E^{(1)}$	{E.VAL : = UNARY (–, $E^{(1)}$.VAL)}
4.	$E \rightarrow$ *id*	{E.VAL : = LEAF (id)}

In table, we make use of following functions :

(i) NODE (OP, LEFT, RIGHT) : This function creates a new node labeled by the first argument and makes the second and third arguments the left and right children of the new node, returning a pointer to the created node.

(ii) UNARY(OP, CHILD) : This function creates a new node labeled OP and makes CHILD its child. Again, a pointer to the created node is returned.

(iii) LEAF (*id*) : This function creates a new node labeled by *id* and returns a pointer to that node. This node receives no children. The label of the leaf would be a pointer to the symbol table.

Q.6.(b) Explain three address code, quadruplus and triples.

Ans. Three Address code : Three address code is the typical example of low level representation. The term "three address code" contains three addresses two for the operands and one for the result.

Three address code is a sequence of statement of the general form

p : q op r

where p, q and r are names constant or compiler generated temporaries.

op stands for any operator, such as fixed-or-floating point arithmetic operator or logical operator on Boolean value.

There is only one operator on the right hand side of a statement ensuring that operations and expressions are simple. So no built in operations are allowed.

A source language expression like $x : x * y + z - x$ translated into three address sequence as,

$t_1 : = x * y$
$t_2 : = t_1 + z$

$t_3 : = t_2 - x$
$x : = t_3$

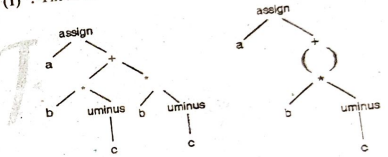
where t_1, t_2 are compiler generated temporary names.

Three address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

Following Table(1) represents the three address sequence for the syntax tree and dag in Table(1). Variables names can appear directly in three address statements so Table1.(a) has no statements corresponding to the leaves in Table1(a).

$t_1 : = - c$ $t_2 : = b * t_1$ $t_3 : = - c$ $t_4 : = b * t_3$ $t_5 : = t_2 + t_4$ $a : = t_5$	$t_1 : = - c$ $t_2 : = b * t_1$ $t_5 : = t_2 + t_1$ $a : = t_5$
(*a*) *Code for the syntax free*	(*b*) *Code for the dag*

Table (1) : Three address code corresponding to the tree and dag in the Fig.(1)



(a) Syntax tree (b) DAG

Fig.(1) : Graphical representation of a : = b * – c + b * –

Three address code gives better flexibility in terms of target code generation and code optimination as it is simpler.

The semantic rules for generating three address code from common programming language constructs are similar to those for contructing syntax trees for generating postfix notation.

Quadruple : A quadruple is a structure with at the most four field as, op arg 1, arg2 and result. The op field is used to represent internal node for the operator. The arg 1 and arg 2 represents the argument or operand of the expression and result field represent the result of the expression i.e. it stores the result of the expression.

The three-address statements with unary operators like $a : = - y$ or $a : = y$ do not use arg2. Operator like param does not use arg2 and result, it only use arg1. Conditional and unconditional jumps put the targe lable in result.

The values of the field arg1, arg2 and result are considered as pointers to symbol table entries for the names represented by these fields. Similarly whenever the temporary names are created they must be entered into the symbol table.

Example : The quadruples for the assignment x : = y * – z + y* – z are shown in following Table.

The three address statement and table for the above assignment statement is

t_1 : = uminus z
t_2 : = y * t_1
t_3 : = uminus z
t_4 : = y * t_3
t_5 : = t_2 + t_4
x : = t_5

	op	arg1	arg2	result
(0)	Uminus	z		t_1
(1)	*	y	t_1	t_2
(2)	uminus	z		t_3
(3)	*	y	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triple : Triple representation is streuture with at the most three field arg1, arg2 and op. The field arg1 adn arg2 are the arguments of the operator op. In triple, temporaries are not used instead of that pointers in the symbol table are used directly. Triple corresponds to the respresentation of a syntax tree or DAG by an array of nodes.

Example : Following Table shows the triples for the assignment statement

x : = y* – z + y* – z.

	op	Arg1	arg2
(0)	uminus	z	
(1)	*	y	(0)
(2)	uminus	z	
(3)	*	y	(2)
(4)	+	(1)	(3)
(5)	assign	x	(4)

Numbers in the round brackers are used to represent pointers into the triple structure. While symbol table pointers are represented by the names themselves

Section -D

Q.7.(a) Define LALR ? Write a algorithm for construction of an LALR parsing table.

Ans. In computer science, a lookahead LR parser of LALR parser is a specialized form of LR parser that can deal with more context-free grammars than SLR parsers. The number of states in SLR and LALR parsing tables are some but considerably less than the number of states in canonical LR parsing tables. LALR parser gives a good trade-off between the number of

grammars it can deal with and the size of the parsing table it requires. These types of parsers are most often generated by compiler-compilers such as YACC and GNU Bison.

The LALR parser uses the collection of LR(1) itemsets for the construction of LALR parsing table. After constructing LR(1) itemsets, we look for the sets of LR (1) items having the same core. Such itemsets merged together to form a single set of items and the resulting lookahead is the union of two individual lookaheads. The LALR parsing the table can be generated by applying the algorithm as follows.

Algorithm for constructing LALR parsing table :

Input : The collection of LR (1) items $I_0, I_1, \ldots I_n$ for an augmented grammar G'.

Output : An LR parsing table consisting of ACTION and GOTO functions, if possible.

Method : The method consists of following steps :

(i) Find all the item sets having the same core, and replace these sets by their union. The resulting lookhead is the union of two individual lookheads.

(ii) Construct the state *i* corresponding to new item set I_i e.g. the state corresponding to an itemset I_{47} is 47.

(iii) The ACTION table entry for state *i* are constructed. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

(iv) The GOTO transitions for the state *i* are constructed using the rule of that if GOTO $(I_i, A) = I_j$ then GOTO $(i, A) = j$ where A is a non-terminal.

(v) All the entries not define by rules (iii) and (iv) above are made "error" entries.

(vi) The inital state of the parser is the one constructed from the set of items containing $[S' \rightarrow S, \$]$

Q.7.(b) What is LR parsing? Explain with example.

Ans. LR parsing : An efficient bottom up syntax analysis technique called LR parsing that can be used to parse a large class of context free grammers. This techniques is called LR (k) parsing, the "*L*" stands for Left to Right scanning of the input, R stands for constructing right most derivation in reverse and k for the number of input symbols of lookahead which are used to make parsing decisions.

LR parsing is attractive for variety of reasons :

(a) LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammers can be written.

(b) The LR parsing method is the most general non backtraking shift reduce parsing method known, yet it can be implemented as efficiently as other shift reduce methods.

(c) An LR parser can detect a syntactic error as soon as it is possible to do so during left to right scan of input.

There are three techniques for constructung an LR parsing table of a grammer. The first method called *simple LR* (*SLR*) easier implement but least powerful second method called

canonical LR is most powerful and most expensive. The third method called *lookahead LR* (*LALR*) is intermediate in power and cost between other two.

LR Parser Structure and Algorithm :

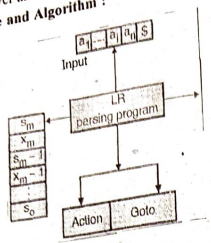


Fig.(1) : LR Parser.

The schematic form of an LR parser is shown in fig.(1). The components of LR parser are a stack, an input, an output, a derive program and a parsing table having two parts action and goto. The parsing (driver) program is same for all LR parser only the parsing table changes from one parser to another.

The program uses a stack to store a string of the from so $X_1 S_1 X_2 S_2 \ldots\ldots X_m S_m$ where X_m is grammer symbol, S_m is state symbol. This extra symbol state summarizes the information contained in the stack below it.

Both grammer and state symbol can appear on the stack but top of stack always contains a state which is used to index the parsing table and to determine the shift reduce parsing decision is combination with the current input symbol.

The parsing table contains two parts, action and goto. The program driving the LR parser behaves like, if it determines S_m is state currently on top of the stack and x_i is the current input symbol then it consults action $[S_m, x_i]$ and it makes an entry in parsing action table for S_m and x_i, which can have one of four values.

(a) Shift S where S is state.

(b) reduce by grammer production $A \rightarrow \beta$.

(c) accept.

(d) error

A configuration of an LR parser is pair whose first component is the stack content and whose second content is the unexpended input.

The initial configuration of a LR parser is written as

$(S_0 X_1 S_1 X_2 S_2 \ldots\ldots X_m S_m, a_i a_i \ldots\ldots a_n \$)$

After each move resulting configuration are as follows :

(a)If action $[S_m, a_i]$ = shift S, the parser exeutes a shift move, entering the configuration

$(S_0 X_1 S_1 X_2 S_2 \ldots\ldots X_m S_m a_i S, a_{i+1} \ldots\ldots a_n \$)$

parser shifts both input symbol a_i and state S.

(b)If action $(S_m, a_i]$ = reduce $A \rightarrow B$ then parser executes a reduce move, entering the configuration

$(S_0 X_1 S_1 X_2 S_2 \ldots\ldots X_{m-r} S_{m-r} As, a_i, a_{i+1} \ldots\ldots a_n \$)$

Here r is length of β.

(c)If action $[S_m, a_i]$ = accept and parsing is completed successfully.

(d)If action $[S_m, a_i]$ = error, the parser has discovered in error and calls an errors recovery routine.

(e)Table shows working of LR parser in each action.

Action $[S_m, a_i]$	Parser action
S_i (Shift)	Push a_i such S_i, advance next token and, continue.
r_i=(Reduce)	$A \rightarrow B$. If length $(B) = r$ pop $2r$ symbols goto $[S_k, A] = S_i$ where S_k is new top of stack push A, push S_i continue.
Accept	Successful parse, halt.
Error	call an error recovery routine.

Q.8. What do you mean by symbol table ? What is the use of symbol table ? Explain the various data structure associated with symbol.

Ans. Symbol Table : In computer science, a symbol table is a data structure used by a language translator such as a compiler where each identifier in a program is recorded along with various attributes. Each entry in the symbol table is a pair of the form (name, information). The information field about a name may include following attributes :

(i) Its type

(ii) Storage location for the name.

(iii) Its scope

(iv) Its from (e.g., a simple variable or a structure)

(v) Number and types of its arguments (if it is a procedure name).

Data Structure For Symbols Tables : In designing a symbol-table mechanism, we would like a scheme that allows us to add new entries and find the existing entries in a table efficiently. There are following three principal data structures that can be used to implement symbol tables :

(i) Linear lists
(ii) Trees
(iii) Hash tables

Each data structure has its leaf of own difficulty of implementation and efficiency. The data structures are discussed in detail one by one.

Linear lists : The linear list is a simples and easiest to implement data structure for implementing a symbol table. The names are entered in the symbol table in their order of arriv and not sorted. A linear list organization is shown in Fig.(1).

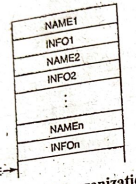


Fig.(1) : A linear list organization

When a name is encountered in a program, the linear list is searched to check whethe or not the name is already present in the symbol table. The linear list is searched upto the position marked by pointer AVAILABLE, which indicates the beginning of the empty portion of the array. If the name is not present already the record for the new name is created and added to the list at a position specified by AVAILABLE pointer. In order to insert n names and m inquiries the total work is $cn(n+m)$ where c is a constant representing the time necessary for a few machine operations. The time taken to search and insert a record can be written as $O(n^2)$.

Search Trees : The binary tree is a mathematical abstraction that plays a central role in the efficient organization of information. Like arrays (linear lists) and linked lists (Self-organizing lists), a binary tree is a data type that stores a collection of data. Binary trees play an important role in computer programming because they strike an efficient balance between flexibility and ease of implementation.

For symbol-table implementations, we use special type of binary tree to organize the data and to provide a basis for efficient implementation of the symbol-table. A binary search tree (BST) associates comparable keys with values, in a structure defined recursively as follows: A Binary Search Tree (BST) is either.

(i) Empty (null) or
(ii) A node having a key-value pair (name-information pair) and two references to BSTs a left BST with smaller keys and a right BST with larger keys.

The Binary Search Tree organization is shown in Fig.(2).

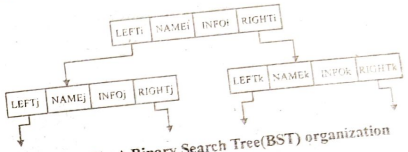


Fig.(2) : A Binary Search Tree(BST) organization

The tree shown in Fig.(2) has the property that all the names NAME j accessible from NAME i by the link left i and then following any sequence of links will precede NAME i in alphabetical order. Similarly, all names NAME k accessible starting with RIGHT; will have the property that NAMEi <NAMEk. The binary tree search routing given in Fig.(3).

```
while P = NULL do
    if NAME = NAME(P) then
        return true
    else if NAME<NAME(P) then
        P:=LEFT(P)
    else P=RIGHT(P)
```

Fig.(3) : Binary Tree Search Routine

In a Binary Search Tree (BST) organization, the time needed to enter n names and make m inquiries is proportional to $(n+m)\log n$. Therefore, the maximum time to search and insert a name is written as $O(\log n)$.

Hash Tables : The hashing scheme is superior amoung all the symbol-table organization techniques. The basic hashing scheme as shown in Fig.(4) includes two tables:

(i) *A hash table*, consisting of k words, numbered 0, 1, ... $k-1$,

(ii) *A storage table*, such that k words of has table are pointers into the storage tabl
the heads of k separate link lists.

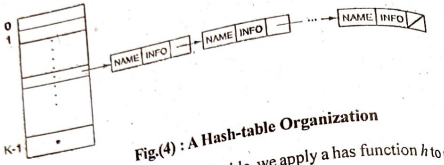


Fig.(4) : A Hash-table Organization

To determine whether NAME is in the symbol table, we apply a has function h to nam
such that h(NAME) is an integer value between 0 and $k-1$. The record for the NAME can
found on the link list numbered h(NAME). If the name is not present in that list, we create
record for name and insert it at the head of the list. In the hashing scheme organization, the fi
needed to perform m accesses on n names in time is proportional to $n(n+m)/k$, for any consta
k. Therefore, in hashing the search, insert or delete operation takes $\theta(n)$ time if the data s
searched linearly and ther are n items stored in records. Hashing in a technique which direct
refers to the memory location where the data is stored and data can be found $O(1)$ time, this
tremendous improvement over other data structures of symbol table.

**Q.9. What is code optimization? Describe local optimization and loo
optimization.**

Ans. Code Optimization : Code optimization is an optional phase designed to improv
the intermediate code so that the resulting target program

(a) Runs faster and/or

(b) Takes less space.

The code optimization is done by a program called code optimizer. The design of goo
optimizer is very important part of compiler design, as it decides the speed of execution of sourc
program. A good optimizing compiler can improve the target program by a considerable amoun
in overall speed. The principal sources of optimization are :

(i) Efficient utilization of the register and instruction set of a machine.

(ii) Inner loops.

(iii) Language implementation details inaccessible to the user.

(iv) Identification of common subexpressions.

(v) Replacement of runtime computations by compile time computations.

There are following types of optimization :

(a) Local optimization : *e.g.*, elimination of common subexpressions. The sequence
of statements

$$P := Q + R + S$$
$$T := Q + R + X$$

after eliminating might the common subexpression the code be rewritten :

$$T_1 := Q + R$$
$$P := T_1 + S$$
$$T := T_1 + X$$

(b) Loop Optimization : *e.g.*, speedup of loops. This involves moving the loop invariant
computation, outside the loop.

Moving back to our example statement

$$\text{sum} := \text{bonus} + \text{basic} \times 50$$

$T_1 := \text{inttoreal}(50)$
$T_2 := \text{id}_3 * T_1$
$T_3 := \text{id}_2 + T_2$
$\text{id}_1 := T_3$

Fig.(1) : Intermediate code statements

The intermediate code statements of fig.(1) may be optimized as :

(i) The conversion of 50 from integer to real representation can be done once and for
all at compile time, and thus eliminating the statement

$$T_1 := \text{inttoreal}(50).$$

(ii) As id_3 is used only once, to transmit its value to id_1, therefore we can substitute id_1
for T_3 and thus eliminating the last statement.

$$id_1 := T_3$$

Therefore the resulting optimized code is shown in fig.(2).

$$T_3 := id_3 \times 50.0$$
$$id_1 := id_3 + T_1$$

Fig.(2) : Optimized code statements

wn in fig.(2).

statements

COMPILER DESIGN

Model Test Paper-III
Paper Code:PCC-CSE-302-G

Note : ***Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.***

Q.1. Describe the following :

(i) Input buffering
(ii) Parsing
(iii) Syntax directed definitions
(iv) Role of Parser.

Ans.(i) Input Buffering : The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers begin-ptr (bp) and forward_ptr (fp) to keep track of the portion of the input scanned.

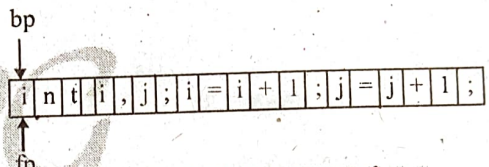


Fig.(1) : Initial configuration

Initially both the pointers point to the first character of the input string as shown below

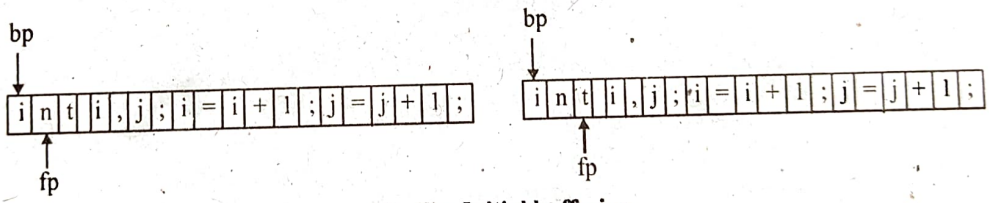


Fig.(2) : Initial buffering

The forward_ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as forward_ptr (fp) encounter a blank space the lexeme "int" is identified.

The fp will be moved ahead at white space. When fp encounters white space, it ignore and moves ahead. Then both the begin_ptr (bp) and forward_ptr (fp) are set at next token *i*.

The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. hence buffering technique is used. A block of data is first read into a buffer, and then scanned by lexical analyzer.

Fig.(3) : Initial configuration

Ans.(ii) Parsing : Parsing is the process of splitting a sentence into words. There are two types of parsing, namely the top-down parsing and the bottom-up parsing. Suppose we want to parse the sentence "Ram ate a mango." If NP, VP, N, V, ART denote noun predicate, verb, noun, verb and article, then the top-down parsing can be donw as follows :

$S \rightarrow$ NPVP
$\rightarrow$ Name VP
$\rightarrow$ Ram V NP
$\rightarrow$ Ram ate ART N
$\rightarrow$ Ram ate a N
$\rightarrow$ Ram ate a mango

The bottom-up parsing for the same sentence is
Ram ate a mango
$\rightarrow$ Name ate a mango
$\rightarrow$ Name verb a mango
$\rightarrow$ Name V ART N
$\rightarrow$ NP VN P
$\rightarrow$ NP VP
$\rightarrow$ S

In the case of formal languages, we derive a terminal string in $L(G)$ by applying the productions of G. If we know that $w \in \Sigma^*$ in $L(G)$, then $S \Rightarrow w$. The process of the reconstruction of the derivation of w is called parsing. Parsing is possible in the case of some context-free languages.

Parsing becomes important in the case of programming languages. If a statement in a programming language is given, only the derivation of the statement can give the meaning of the statement. (This is termed semantics.)

There are two types of parsing : top-down parsing and bottom-up parsing.

In top-down parsing, we attempt to construct the derivation (or the corresponding parse tree) of the input strings, starting from the root (with label S) and ending in the given input string. This is equivalent to finding a leftmost derivation. On the other hand, in bottom-up parsing we build the derivation from the given input string to the top (root with label S).

Ans.(iii) Syntax directed definitions : Syntax-directed definition is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is associated with it a set of semantic rules of the form a:= $f(b_1, b_2, ...b_k)$, where a is an attribute obtained from the function f.

Consider $X \rightarrow \alpha$ be a context free grammar and a:= $f(b_1, b_2, ... b_k)$ where a is the attribute. Then there are two types of attributes :

(1) *Synthesized attribute* : The attribute 'a' is called synthesized attribute of X and $b_1, b_2, ... b_k$ are attributes belonging to the production symbols.

The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

(2) *Inherited attribute* : The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of the production (i.e. α) and $b_1, b_2, ... b_k$ are belonging to either X or α.

The inherited attributes can be computed from the values of the attributes at the siblings and parent of that node.

Ans.(iv) The Role of the parser (Syntax Analyzer) :

- Parser or Syntax analyzer is the program which performs syntax analysis or parsing.
- Parsing is an important phase of the compiler-design, which obtains string of tokens produced as output of Lexical Analysis.
- Parser then groups tokens appearing in it's input as a statement to identify larger structures in the program. If any syntax error is found, it reports and implements recovery technique.
- Fig. shows the position of parser in compiler model.

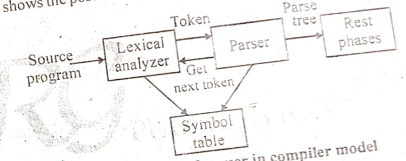


Fig. : Position of parser in compiler model

- There are different types of parsers based on the way they parse the input.
- These are :
  (1) Cocke – Younger – Kasami algorithm
  (2) Earley's algorithm
  (3) Top-down or Bottom-up parser.
- First two methods are inefficient in production of compiler hence commonly top-down and bottom-up parser used.

Section – A

Q.2.(i) What is Compiler ? Explain the different phases of Compiler in detail.

Ans. Compiler : A compiler is a translator (a software program) that translates a high-level language program into functionally equivalent low-level machine language program which can be exected direct. Individual source language statements usually map into many mchine-level instructions. A compiler also generates diagnostic message wherever the specificarjons of the source language are violated by the programmer.

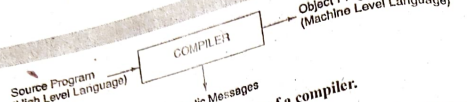


Fig. : The functional diagram of a compiler.

The process of compilation is generally broken down into subtasks called phases. Eg
phase is interdependent on other phase. Output of one phase will be input to another phase. Ty
phase of compiler takes as input the source language written by a programmer & the last ph
produces the required object language (or target program).

Figure shows the structure of a compiler or various phase of compiler.

The whole process of compilation - is carried out in different phase which are gi
below :

1. Lexical Analysis (Scanner)

→ It reads the source program one character at a time & group them into tokens.

→ A token is a sequence of characters with collective meaning e.g of Tokens. Keywor
Identifiers, operators, lables, constant.

→ Put Tokens in the symbol table.

2. Syntax Analysis (Parser)

→ Receives tokens as its input generated from previous phase (lexical Analysis)
produces a hierarchal structure called Syntax tree or parse tree.

3. Semantic Analysis

→ Performs Type checking i.e. for each operator, type compatibilities of operation
checkes.

4. Intermediate code Generation

→ Converts output of its previous phase into intermediate representation.

E.g (a) Postfix Notation

(b) Three Address code

(c) Quadruples

(d) Triples & Indirect Triples

(e) Syntax Trees

5. Code Optimization

→ It converts the intermediate representation of source program into an efficient
ie faster running code. Code optimization can be done in 3 ways

(a) Local optimization: Elimination of common sub expression

(b) Loop optimization: Removing loop invariants.

(c) Global Data Flow Analysis: Perform optimization using information flow.

6. Code Generation

→ It converts optimized intermediate code into Machine / Assembly code.

→ It allocates memory locations for variables in the program.

7. Table Management

→ Keep Recrod of each token & its attributes (ie identifier name, data type, lay
etc).

→ Symbol Tables is a data structures which contains Tokens.

8. Error Handling

→ Detects & Report errors occured at each phase of the compiler.

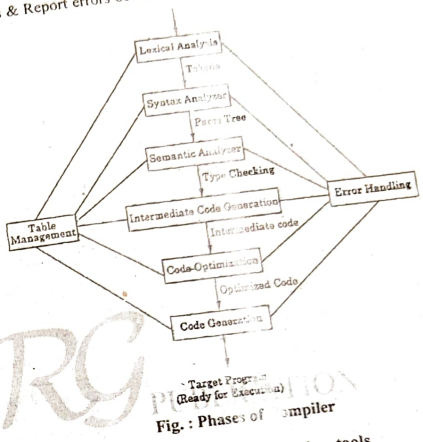


Fig. : Phases of compiler

Q.2.(ii) Explain various compiler construction tools.

Ans. A compiler is tedious and time consuming task. Therefore there are some specialized tools for implementing various phases of compiler. These tools are called compiler construction tools.

Various complier construction tools are as follows :

(i) Scanner generator - For example : LEX

(ii) Parser generator - For example : YACC

(iii) Syntax directed translation engines.

(iv) Automatic code generator.

(v) Data flow engines.

(i) *Scanner generator* : These generators generate lexical analyzers. The specification given to these generators are in the form of regular expressions.

The UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

(ii) *Parser generators* : These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammar. Typically UNIX has a tool called YACC which is a parser generator.

(iii) *Syntax-directed translation engines* : In this tool the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.

(iv) *Automatic code generator* : These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language. The template matching technique is used. The intermediate code statements are replaced templates that represent the corresponding sequence of machine instruction.

(v) *Data flow engines*: The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

Q.3. How do we implement lexical analyzer ? Explain with example.

Ans. Implementation of Lexical Analyzers : Lexical analyzer is the first phase of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens. Each token is a single logical cohesive unit such as identifier, keywords, operators and punctuation marks. Then the parser to determine the syntax of the source program can use these tokens. The role of lexical analyzer in the process of compilation is as shown below :

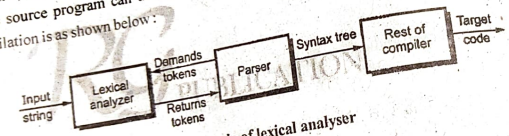


Fig. : Role of lexical analyser

As the lexical analyzer scans the source program to recognize the tokens it is also called as scanner.

Fig. shows example finite automata for part of a language.

So, as seen in above finite automata, every input lexeme is separated and token as integer shown in final state is generated or an equivalent token representing the lexeme is generated.

So for a '<' sign e.g. 'LT' token or '7' as token may be generated. For '<=', 'LE' or '8' is generated. Similarly keywords and identifiers are recognized.

While recognized a token, it is entered into a symbol table.

Following is a procedure lexical analyser.

Lexical analyser (input, look ahead, char, token, positions).

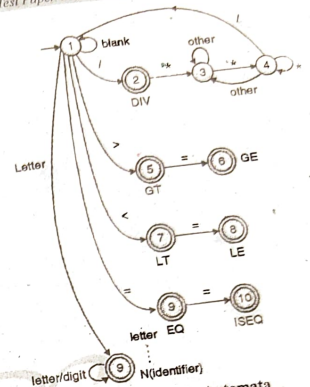


Fig. : Finite Automata

```
Position = 0;
If (lookahead = tales)
Then char = getnext (input);
Lookahead = false;
While (true){
    Switch (char){
        Case ( ) : char = getnext (input);
        Case '/' : char = getnext (input);
            If char = '*'
            Then while (true){ //loop
            Char = getnext (input)
        If char = '*'
        Then while (char = '*')
        Char = getnext (input)
        I char = '/'
        Then char = getnext (input)
        } || exit loop 1 || exit loop 1.
Else    lookahead = true.
        Token = DIV
        Return.
```

```
Case '<' :
        Char = getnext (input);
        If char = '='
        Then token = LE
Else            lookahead = true
                Token = LT

        Return.
Case '>' : char = getnext (input)
        If char = '='
        Then token = GE
        Else lookahead = true token = GT
        Return

        // similar code for other inputs.

Default : if char > = 'A' and char < = 'Z'
        Then string = char
        Char = getnext (input)
        Where (char > = 'A' && < = 'Z')
        or (char > = '0' && < = 'g')
        {concat (string, char)
        Char = getnext (input)
        }
        Lookahead = true
        If (keyword (string) > 0)
        Then token = keyword (string)
        Position = insert (string, identifier)
        Token = N

    Return.
If char = '0' && char < = 's'
Then string = char
        Char = getnext (input)
        While (char > = '0' && char < = 'g'){
        Concat (stri, char)
        Char = getnext (input);
        }
    Lookahead = true
    Position = insert (string, constant)
    Token = CONST
    Return.
}
```

In above procedure the variables used are as follows and also procedures called

Input	:	The input program lead key lexical analyser. It is read character by character.
Lookahead	:	Logical variable which tells whether next character read from 'input' can be used as look ahead character.
Char	:	variable into which a character is read.
Token	:	Stores token which is returned to syntax analyser which calls lexical analyser.
Position	:	Position of a identifier a constant in a symbol table.
Getnext	:	Reach and returns next character from input file (input).
Concat	:	Concatenates contents of 2 parameters and stores result in first parameter (variable).

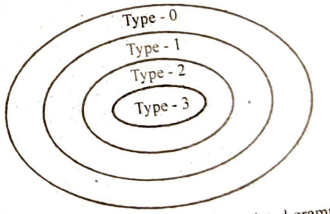
Section – B

Q.4.(i) Explain Chomsky hierarchy of grammars.

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :

Type – 0 (Unrestricted grammar)
Type – 1 (Context sensitive grammar)
Type – 2 (Context free grammar)
Type – 3 (Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.



Unrestricted (Type -0) Grammars : The unrestricted grammar is defined as

$G = (V_n, V_t, P, S)$

where
V_n = a finite set of non-terminals
V_t = a finite set of terminals
S = starting non-terminal, $S \in V_n$

and P is set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \in$. These grammars are known as type-0, phase-structure or unrestricted grammars.

Context Sensitive Grammar : Let $G = (V_n, V_t, P, S)$ be context sentive grammar where

V_n = finite set of non-terminals

V_t = finite set of terminals

S = starting non-terminals $S \in V_n$

and P is the set of fules called productions defined as

$$\alpha \rightarrow \beta$$

where β is at least as long as a that is clearly

$$|\alpha| \leq |\beta|$$

The term "Context-sentive" comes from a normal form for these grammars, where each production is of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ with $\beta \neq \in$. Replacement of variable A by string β is permitted in the "Context" of α_1 and α_2.

Context-Free Grammars : Mathematically context-free grammar is defined as follows :

"A grammar $G = (V_n, V_t, P, S)$ is said to be context-free"

where

V_n = A finite set of non-terminals, generally represented by capital letters $A, B, C, D, \ldots$.

V_t = A finite set of terminals, generally represented by small letters, like $a, b, c, d, e, f, \ldots$.

S = Starting non-terminal, called start symbol of the grammar. S belongs to V_n.

P = Set of rules or production in CFG.

G is context-free and all production in P have the form

$$\alpha \rightarrow \beta$$

where $\alpha \in V_n$ and $\beta \in (V_t \cup V_n)^*$

Regular Grammars : A regular grammar may be left linear or right linear.

"If all production of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

"If all the production of a CFG are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left linear."

A right or left linear grammar is called a regular grammar.

Q.4.(ii) What is context free grammar ? Give the left most and right most derivation for the following grammar. Also draw its derivation tree for it and check its ambiguity for id + id * id :

$$E \rightarrow E + E \mid E - E \mid E^* E \mid E / E \mid id$$

Ans. Context Free Grammer : Context Free Grammer (CFG) is a type of grammer used to define the way the statements are written for a programming language like Pascal or C. Thus we can say that CFG gives the syntactic specfication of progamming languages.

Basically the grammer can be defined as $G = (N, T, P, S)$

T is Set of terminals represented as small letters. E.g. = {a, b, c, ...}. These represent words of the language which are converted to tokens by the lexical analyzer so here a, b, c represent tokens. These tokens are arranged in the combination with these meta symbols called non terminals.

N is Set of Non terminals represented as capital letters E.g = {A, B, C,.....}. These are the meta symbols used to define the production rules.

P is a set of production rules. These rulers are used to produce synatactically correct statement. These rules are the grammer rules which decide how the statements are going to be written. Few production rules are defined below.

$$S \rightarrow aA \mid bB$$
$$A \rightarrow abB \mid aB$$
$$B \rightarrow a \mid b \mid \in$$

S is a special symbol called start symbol. Here in the above production rules, S is the start symbol.

$id + id * id$

Using LMD

LMD 1 :

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

Using LMD, we have one more derivation

LMD 2 :

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

The parse trees corresponding to these LMD's are shown in fig.[1(a) & (b)] respectively.

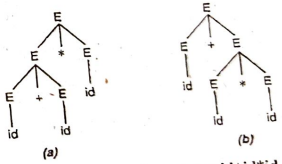


Fig.(1) : Parse tree for a string id+id*id using (a) First LMD (b) Second LMD

As there exists two distinct parse trees for the given string. Hence the grammar is ambiguous.

Q.5.(i) Test whether the grammer is LL(1) or not and construct a predictive parsing table for it.

$S \rightarrow AaAb \mid BbBa, A \rightarrow \varepsilon, B \rightarrow \varepsilon$

Ans. Consider the grammar :

$S \rightarrow AaAb$
$S \rightarrow BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

Now we will compute FIRST and FOLLOW functions.

FIRST(S) = {a, b} if we put
$S \rightarrow AaAb$
$S \rightarrow aAb$
$S \rightarrow aAb$ When $A \rightarrow \varepsilon$
Also $S \rightarrow BbBa$
$S \rightarrow bBa$ When $B \rightarrow \varepsilon$

FIRST(A) = FIRST(B) = (ε).
FOLLOW(S) = {$}
FOLLOW(A) = FOLLOW(B) = {a, b}

The LL(1) parsing table is

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

Now consider the string "ba". For parsing -

Stack	Input	Action
$S	ba$	$S \rightarrow BbBa$
$aBbB	ba$	$B \rightarrow \varepsilon$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	Accept

This shows that the given grammar is LL(1).

Q.5.(ii) Explain canonical LR parser.

Ans. CLR : A canonical *LR* (*CLR*) or *LR* (1) parser is an *LR* parser in which parsing tables are constructed with similar ways as LR (0) parsers except that the items in the item sets also contain a lookahead, i.e., a terminal that is expected by the parser after the right hand side of the rule. For example, such an item for a rule $X \rightarrow Y Z$ might be $X \rightarrow Y \bullet Y, a$; which means

that the parser has to read a string corresponding to Y and expects next a string corresponding to Z followed by the terminal 'a'.

Construction of the canonical LR (CLR) :

Input : An augmented grammar G'.

Output : The canonical LR parsing table functions action and goto for G'

Method :

(i) Construct $C = \{I_0, I_1, \ldots\ldots, I_n\}$, the collection of sets of LR(1) items for G'.

(ii) State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows :

– If $[A \rightarrow \alpha . a, b]$ is in I_i and goto(I_i, a) = I_j, then set action[i,a] to "shift j." Here, a is required to be a terminal.

– If $[A \rightarrow \alpha ., a]$ is in I_i, $A \neq S'$, then set action[i,a] to "reduce $A \rightarrow \alpha$."

– If $[S' \rightarrow S ., \$]$ is in I_i then set action[i ,$] to "accept."

If a conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to be fail.

(iii) The goto transition for state i are determined as follows: If goto(I_i , A)= I_j ,then goto[i,A]=j.

(iv) All entries not defined by rules(ii) and (iii) are made "error."

(v) The initial state of the parser is the one constructed from the set containing item $[S' \rightarrow .S, \$]$.

Consider the following augmented grammar :

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow Cc \mid d$

The initial set of items is :

I_0 : $S' \rightarrow .S, \$$
$S \rightarrow .CC, \$$
$C \rightarrow .Cc, c \mid d$
$C \rightarrow .d, c \mid d$

We have next set of items as :

I_1 : $S' \rightarrow S., \$$
I_2 : $S \rightarrow .Cc, \$$
$C \rightarrow .Cc, \$$
$C \rightarrow .d, \$$
I_3 : $C \rightarrow c.C, \$$
$C \rightarrow .c C , c \mid d$
$C \rightarrow .d, \$$
I_4 : $C \rightarrow d. , c \mid d$
I_5 : $S \rightarrow CC. , \$$
I_6 : $C \rightarrow c.C, \$$
$C \rightarrow .c C ,\$$
$C \rightarrow .d , \$$
I_7 : $C \rightarrow d. , \$$

I_8 : $C \rightarrow c\,C.\,,\ c \mid d$
I_9 : $C \rightarrow c\,C.\,,\ \$$

Section – C

Q.6. Check whether the following grammar is LR (0) or not.

$E \rightarrow E + T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Ans.

I_0: $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_1 : $E' \rightarrow E.$
$E \rightarrow E. + T$

I_2 : $E \rightarrow T.$
$T \rightarrow T. * F$

I_3 : $T \rightarrow F.$

I_4 : $F \rightarrow (.E)$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5 : $F \rightarrow id.$

I_6 : $E \rightarrow E + .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7 : $T \rightarrow T * .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_8 : $F \rightarrow (E.)$
$E \rightarrow E. + T$

I_9 : $E \rightarrow E + T.$
$T \rightarrow T. * F$

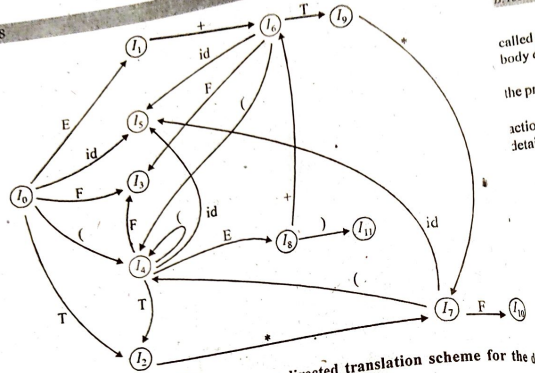
I_{10} : $T \rightarrow T * F.$

I_{11} : $F \rightarrow (E).$

Parsing Tables :

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

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



Q.7.(i) State and explain the syntax directed translation scheme for the calculator and give the parse tree and translation for the string

$(7 + 4)^* 249/3 + 26$.

Ans. Syntax-directed translation (SDT) : Syntax-directed translation (SDT) Refe to a method of complier implementation where the source language translation is complete driven by the parser, i.e., based on the syntax of the language. The parsing process and pa trees are used to direct semantic analysis and the translation of the source program. Almost modern compilers are syntax-directed.

SDT can be a separate phase of a compiler or we can augment our conventional gramm with information to control the semantic analysis and translation. Such grammars are call attribute grammars.

We augment a grammar by associating attributes with each grammar symbol t describes its properties. With each production in a grammar, we give semantic rules/actio which describe how to compute the attribute values associated with each grammar symbol i production

The general approach to Syntax-Directed Translation is to construct a parse tree syntax tree and compute the values of attributes at the nodes of the tree by visiting them in som order. In many cases, translation can be done during parsing without building an explicit tree

A class of syntax-directed translations called "L-attributed translations" (L for left- right) includes almost all translations that can be performed during parsing. Similarly, "S-attribu translations" (S for synthesized) can be performed easily in connection with a bottom-up pars

There are two ways to represent the semantic rules associated with grammar symbo

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

Syntax-Directed Translation Schemes (SDT) : SDT embeds program fragments called semantic actions with in production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Example : In the rule $E \rightarrow E1 + T$ {print '+'}, the action is positioned after the body of the production.

SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

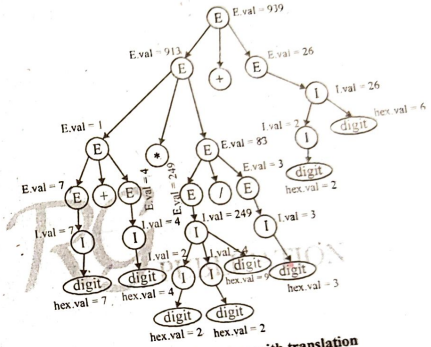


Fig. : Parse tree with translation

Q.7.(ii) What is intermediate code representation ? Explain quadruple, triples and indirect triples with the help of an example.

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

High Level IR -High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

Low Level IR -This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

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

Quadruple : A quadruple is a structure with at the most four field as, op arg 1, arg2, result. The op field is used to represent internal node for the operator. The arg 1 and arg2 represents the argument or operand of the expression and result field represent the result of expression i.e. it stores the result of the expression.

The three-address statements with unary operators like $a := -y$ or $a := y$ do not use arg2. Operator like param does not use arg2 and result, it only use arg1. Conditional and unconditional jumps put the targe lable in result.

The values of the field arg1, arg2 and result are considered as pointers to symbol table entries for the names represented by these fields. Similarly whenever the temporary names are created they must be entered into the symbol table.

Example : The quadruples for the assignment $x := y * -z + y * -z$ are shown in following Table.

The three address statement and table for the above assignment statement is

$t_1 := \text{uminus } z$
$t_2 := y * t_1$
$t_3 := \text{uminus } z$
$t_4 := y * t_3$
$t_5 := t_2 + t_4$
$x := t_5$

	op	arg1	arg2	result
(0)	Uminus	z		t_1
(1)	*	y	t_1	t_2
(2)	uminus	z		t_3
(3)	*	y	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triple : Triple representation is strcuture with at the most three field arg1, arg2 and op. The field arg1 adn arg2 are the arguments of the operator op. In triple, temporaries are not used instead of that pointers in the symbol table are used directly. Triple corresponds to the respresentation of a syntax tree or DAG by an array of nodes.

Example : Following Table shows the triples for the assignment statement

$x := y * -z + y * -z$.

	op	Arg1	arg2
(0)	uminus	z	
(1)	*	y	(0)
(2)	uminus	z	
(3)	*	y	(2)
(4)	+	(1)	(3)
(5)	assign	x	(4)

Numbers in the round brackers are used to represent pointers into the triple structure While symbol table pointers are represented by the names themselves

Indirect Triple Representation : In indirect triple representation listing pointers to triples is done instead of listing the triples themselves.

Example :

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

	OP	arg1	arg2
(14)	Uminus	z	
(15)	*	y	(14)
(16)	Uminus	z	
(17)	*	y	(16)
(18)	+	(15)	(17)
(19)	Assign	x	(18)

Section – D

Q.8.(i) Explain the various types of errors generated during the various phases of the compiler. How do we recover from these errors ?

Ans. Errors can be categorises as follows :

1. **Lexical:** Ill-formed numeric literals and identifiers. Some time a character is encountered, which is not in the language.
2. **Syntactic:** Some time programmar commits the mistake during coding process. For example unbalanced parenthesis and punctuation.
3. **Semantic:** Some time errors of declaration and scope are takes place. For example undeclared or multiple-declared identifiers. Type mismatch is another example of semantic errors.
4. **Logical:** It is next to impossible for the compilers to handle the logical errors. Infinite loops is the example of logical error. In these type of errors code is correct syntactically but it does not operate as indeed.

A good compiler have following goals for error handling:

- Detect the presence of errors and produce "meaningful" diagnostics.
- To recover quickly enough to be able to detect subsequent errors.
- Error handling components should not significantly slow down the compilation of syntactically correct programs.

Error Recovery Strategies : Error recovery strategies can be divided as follows:

1. **Phrase Level Recovery:** Replace, delete, or insert a token as a prefix to the input which will enable the parser to continue. Inserting tokens can lead to infinite loops.
2. **Error Productions:** Add rules to the grammar that describe the erroneous syntax. This strategy can resolve many, but not all potential errors.

- If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
- When an error production is used by the parser, we can generate appropriate error diagnostics.
- Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.

3. Global Correction: Replace incorrect input with input that is correct and requir
the fewest changes to create. This requires technique that are costly in terms of time and spac
- Ideally, we would like a compiler to make as few changes as possible in processin

incorrect inputs.
- We have to globally analyze the input to find the error.
- This is an expensive method, and it is not in practice.

4. Panic Mode: Discard tokens until, "synchronization" token is found, such token
are often delimiters, such as, ';', '}', ')', 'end', whose role in the source program are clear. Thi
method can fail to detect multiple errors in close proximity.

In case of panic mode error recovery strategy, an error is detected when :

(a) For a table based implementation: given non-terminal 'A' on the stack top, and 'a' a
the next input token, parsing table lookup (A,a) returns an error.

(b) For a table based implementation: The terminal on top of the stack does not match
the next input token.

(c) For a recursive descent implementation: A specific token is expected to be next in
input in the input, but is not.

Q.8.(ii) What is the use of Symbol Table ? Explain any two data structures associated with Symbol Table.

Ans. Symbol Table : In computer science, a symbol table is a data structure used by a language translator such as a compiler where each identifier in a program is recorded along with various attributes. Each entry in the symbol table is a pair of the form (name, information). The information field about a name may include following attributes :

(i) Its type
(ii) Storage location for the name.
(iii) Its scope
(iv) Its from (e.g., a simple variable or a structure)
(v) Number and types of its arguments (if it is a procedure name).

Data Structure For Symbols Tables : In designing a symbol-table mechanism, we would like a scheme that allows us to add new entries and find the existing entries in a table efficiently. There are following three principal data structures that can be used to implement symbol tables :

(i) Linear lists
(ii) Trees
(iii) Hash tables

Each data structure has its leaf of own difficulty of implementation and efficiency. These data structures are discussed in detail one by one.

Linear lists : The linear list is a simples and easiest to implement data structure for implementing a symbol table. The names are entered in the symbol table in their order of arrival and not sorted. A linear list organization is shown in Fig.(1).

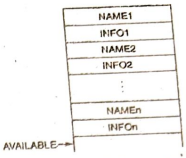


Fig.(1) : A linear list organization

When a name is encountered in a program, the linear list is searched to check whether or not the name is already present in the symbol table. The linear list is searched upto the position marked by pointer AVAILABLE, which indicates the beginning of the empty portion of the array. If the name is not present already the record for the new name is created and added to the list at a position specified by AVAILABLE pointer. In order to insert n names and m inquiries the total work is $cn\,(n+m)$ where c is a constant representing the time necessary for a few machine operations. The time taken to search and insert a record can be written as $O(n^2)$.

Search Trees : The binary tree is a mathematical abstraction that plays a central role in the efficient organization of information. Like arrays (linear lists) and linked lists (Self-organizing lists), a binary tree is a data type that stores a collection of data. Binary trees play an important role in computer programming because they strike an efficient balance between flexibility and ease of implementation.

For symbol-table implementations, we use special type of binary tree to organize the data and to provide a basis for efficient implementation of the symbol-table. A binary search tree (BST) associates comparable keys with values, in a structure defined recursively as follows : a Binary Search Tree (BST) is either.

(i) Empty (null) or
(ii) A node having a key-value pair (name-information pair) and two references to BSTs, a left BST with smaller keys and a right BST with larger keys.

The Binary Search Tree organization is shown in Fig.(2).

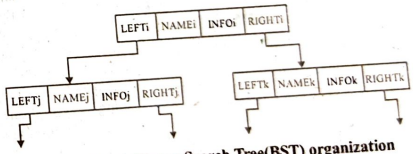


Fig.(2) : A Binary Search Tree(BST) organization

The tree shown in Fig.(2) has the property that all the names NAME j accessible from NAME i by the link left i and then following any sequence of links will precede NAME i in

alphabetical order. Similarly, all names NAME k accessible starting with RIGHT; will h
property that NAMEi <NAMEk. The binary tree search routing given in Fig.(3).

```
while P = NULL do
    if NAME = NAME(P) then
        return true
    else if NAME<NAME(P) then
        P:=LEFT(P)
    else P=RIGHT(P)
```

Fig.(3) : Binary Tree Search Routine

In a Binary Search Tree (BST) organization, the time needed to enter n name make m inquiries is proportional to $(n + m) \log n$. Therefore, the maximum time to search insert a name is written as $O(\log n)$.

Hash Tables : The hashing scheme is superior amoung all the symbol-table organiz techniques. The basic hashing scheme as shown in Fig.(4) includes two tables:

(i) *A hash table,* consisting of k words, numbered 0, 1, ... $k - 1$,

(ii) *A storage table,* such that k words of has table are pointers into the storage ta the heads of k separate link lists.

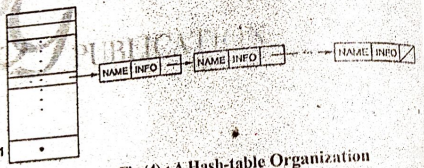


Fig.(4) : A Hash-table Organization

To determine whether NAME is in the symbol table, we apply a has function h to n such that h(NAME) is an integer value between 0 and $k - 1$. The record for the NAME co found on the link list numbered h(NAME). If the name is not present in that list, we cre record for name and insert it at the head of the list. In the hashing scheme organization, the needed to perform m accesses on n names in time is proportional to $n(n + m)/k$, for any cons k. Therefore, in hashing the search, insert or delete operation takes $\theta(n)$ time if the data searched linearly and ther are n items stored in records. Hashing in a technique which dire refers to the memory location where the data is stored and data can be found $O(1)$ time, th tremendous improvement over other data structures of symbol table.

Q.9. Write short notes on the following :

(a) Loop optimization

(b) Code generations.

Ans.(a) Loop Optimization : *e.g.*, speedup of loops. This involves moving the loop invariant computation, outside the loop.

Moving back to our example statement

sum : = bonus + basic * 50

```
T1 : = inttoreal(50)
T2 : = id3 * T1
T3 : = id2 + T2
id1 : = T3
```

Fig.(1) : Intermediate code statements

The intermediate code statements of fig.(1) may be optimized as :

(i) The conversion of 50 from integer to real representation can be done once and for all at compile time, and thus eliminating the statement

T_1 : = inttoreal (50).

(ii) As id_3 is used only once, to transmit its value to id_1, therefore we can substitute id_1 for T_3 and thus eliminating the last statement.

id_1 : = T_3

Therefore the resulting optimized code is shown in fig.(2).

```
T3 : = id3 * 50.0
id1 : = id3 + T1
```

Fig.(2) : Optimized code statements

Ans.(b) Code Generation : The code generation is the last phase in the compilation process. In computer science, the code generation is the process which converts some internal representation of source code into a form (e.g., machine code) that cna be readily executed by a machine. The code generation is done by a code generator component of a compiler. The code generator translates the intermediate representation of the source program into a sequence of machine instructions.

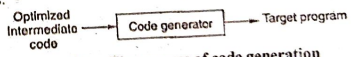


Fig. : The process of code generation

The input to a code generator is an intermediate language program that can be :

(i) A sequence of quadruples.

(ii) A sequence of triples.

(iii) A tree, or

(iv) A postfix polish string.

The output of a code generator is called the target program (or object program) that can be :

(i) An absolute machine language program

(ii) A relocatable machine language program
(iii) An assembly language program
(iv) A program in some other programming language.

Various properties desired by an object code generation phase are :

(i) *Correctness :* It should produce a correct code and do not alter the purpose of code.

(ii) *High quality* : It should produce a high quality object code.

(iii) *Efficient use of resources of the target machine* : While generating the code necessary to know the target machine on which it is going to get generated. By this the generation phase can make an efficient used of resource of the target machine. For memory utilization while allocating the registers or utilization of arithmetic logic unit performing the arithmetic operations.

(iv) *Quick code generation* : This is a most desired feature of code generation It is necessary that the code generation phase should produce the code quickly after the source program.

COMPILER DESIGN

July - 2021

Paper Code:PCC-CSE-302-G

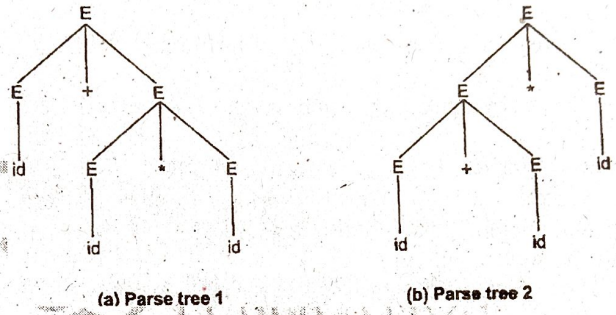
Note : ***Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.***

Q.1.(a) Write a short note on ambiguous grammar. **(2.5)**

Ans. A grammar G is said to be ambiguous if it generates more than one parse trees for sentence of language L(G).

For example : $E \rightarrow E + E \mid E * E \mid id$

Then for id + id * id



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

Fig. : Ambiguous grammer

Q.1.(b) Compiler. **(2.5)**

Ans. Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).

During this process of translation if some errors are encountered then compiler displays them as error messages. The basic model of compiler can be represented as follows.

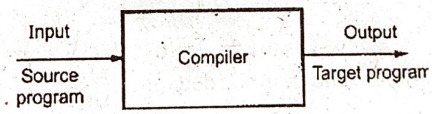


Fig. : Compiler

The compiler takes a source program as higher level languages such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

Q.1.(c) Differentiate between tokens, patterns and lexemes. **(2.5)**

Ans. Tokens : Sequence of characters that have a collective meaning.

Patterns : There is a set of strings in the input for which the same token is pro
output. This set of strings is described by a rule called a pattern associated with the to
Lexeme : A sequence of characters in the source program that is matched
pattern for a token.

Q.1.(d) Role of regular expression.

Ans. Regular Expressions : Regular expressions are mathematical symbolism
describe the set of strings of specific language. It provides convenient and useful nota
representing tokens. Here are some rules that describe definition of the regular expressio
the input set denoted by Σ.

(1) ε is a regular expression that denotes the set containing empty string.

(2) If R1 and R2 are regular expressions then R = R1 + R2 (same can also be repre
as R = R1 | R2) is also regular expressions which represents union operation.

(3) If R1 and R2 are regular expressions then R = R1.R2 is also a regular expr
which represents concatenation operation.

(4) If R1 is a regular expression then R = R1* is also a regular expression w
represents kleen closure.

A language denoted by regular expressions is said to be a regular set or a regular lang

Q.1.(e) What is phrase level error recovery ?

Ans. Phrase-level recovery : On discovering an error a parser may perform
correction on the remaining input; that is it may replace a prefix of the remaining input by
string that allows the parser to continue.

The basic local corrections to recover from syntax error are :

- Deletion of a source symbol e.g. to delete an extra semicolon.
- Replace comma by a semicolon.
- Insertion of synthetic symbol e.g. to insert a missing semicolon.
- The choice of the local correction is left to the compiler designer.
- The motive behind the local correction is to present a new string to the parser w
would lead to by passing of the error situation to continue parsing. But while performing
correction we must be careful to choose replacement that do not lead to infinite loops.
- This type of replacement can correct any input string and has been used in se
error repairing compilers.
- The method major drawback is the difficultly it has in coping with situations inv
the actual error has occurred before the point of detection.

Q.1.(f) What is register allocation in code generation ?

Ans. Register allocation : It is the process of just selecting set of variables that
reside in registers at any point in program.

Goals of Register Allocation :

1. Generate code that requires as little registers as possible
2. Avoid unnecessary memory accesses, i.e., not only temporaries, but also prog
variables are implemented by registers.

3. Allocate registers such that they can be used as much as possible, i.e., registers should not be used for variables that are only rarely accessed.
4. Obey programmer's requirements.

Unit – I

Q.2. Explain different phases of compiler. (15)

Ans. Phases of Compiler : Following are the phases of compiler :

(i) Lexical Analysis : The lexical analysis is also called scanning. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group for strings called token. A token is a sequence of characters having a collective meaning. For example if some assignement statement in your source code is as follows,

total = count + rate * 10

Then in lexical analysis phase this statement is broken up into series of tokens as follows.

(a) The identifier total
(b) The assignment symbol
(c) The identifier count
(d) The plus sign
(e) The identifier rate
(f) The multiplication sign.
(g) The constant number 10

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

(ii) Syntax Analysis : The syntax analysis is also called parsing. In this phase the tokens generated by the lexical analyzer are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens togther. For the expression total = count + rate * 10 the phase tree and can be generated as follows.

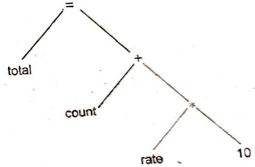


Fig.(1) : Parse tree for total = count + rate * 10

In the statement 'total = count + rate *10' first of all rate*10 will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of syntax tree the production rules are to be designed. The rules are usually expressed by context free grammer. For the above statement the production rules are :

(1) $E \leftarrow$ identifier (2) $E \leftarrow$ number

(3) $E \leftarrow E1 + E2$ (4) $E \leftarrow E1 * E2$
(5) $E \leftarrow (E)$
where E stands for an expression.

- By rule (1) count and rate are expression and
- By rule (2) 10 is also an expression.
- By rule (4) we get rate*10 as expression.
- And finally count + rate*10 is an expression.

(iii) Semantic Analysis : Once the syntax is checked in the syntax analyzer phase, is maintained. The task of symbol table is to store identifier (variables) used in the program.
The symbol table also stores information about attributes of each identifier. The attributes
of identifiers are usually its type, its type, its scope, information about the storage allocated for it.
The symbol table also stores information about the subroutines used in the programs. In
case of subroutine, the symbol table stores the name of the subroutine number of arguments
passed to it, type of these arguments, the method of passing these arguments (may be call by
value or call by reference) and return type if any.

the semantic anaylsis determines the meaning of the source string. For example meaning
source string means matching of parenthesis in the expression, or matching of if ... else statement
or performing arithmetic operations of the expressions that are type compatible, or checking
scope of operation.

For example,

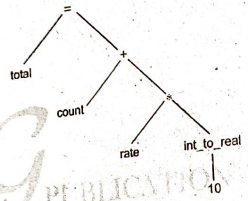


Fig.(2) : Semantic analysis

Thus these three phases are performing the task of analysis. After these phases a
intermediate code gets generated.

(iv) Intermediate Code Generation : The intermediate code is a kind of code which
is easy to generate and this code can be easily converted to target code. This code is in variet
of forms such as *three address code, quadruple, triple posix*. Here we will consider a
intermediate code in three address code form. This is like an assembly language.The three
address code consists of instructions each of which has at the most three operands. For example

```
t1 : = int_to_real (10)
t2 : = rate × t1
t3 : = count + t2
total : = t3
```

(v) Code Optimization : The code optimization phase attempts to improve the
intermediate code. This is necessary to have a faster executing code or less consumption of
memory. Thus by optimizing the code the overall running time of the target program can be
improved.

(vi) Code Generation : In code generation phase the target code gets generated. The
intermediate code instructions are translated into sequence of machine insturctions.


```
MOV rate, R1
MUL # 10.0, R1
MOV count, R2
ADD R2, R1
MOV R1, total
```

(vii) Symbol table management : To support these phases of complier a symbol table

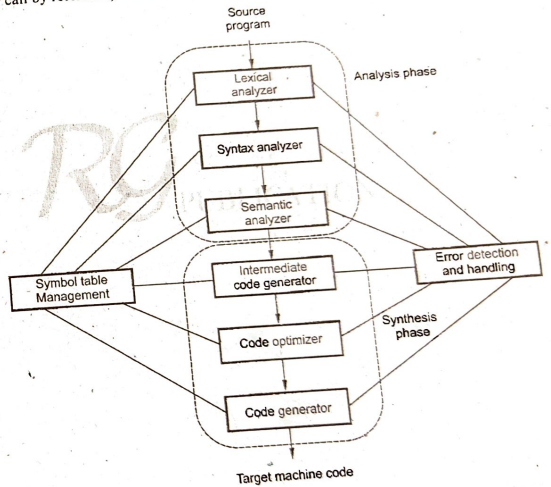


Fig.(3) : Phases of Compiler

Basically symbol table is a data structure used to store the information about identifiers.
The symbol table allows us to find the record for each identifier quickly and to store or
retrieve data from that record efficiently.

During compliation the lexical analyzer detects the identifier and makes its entry in symbol table. However, lexical analyzer can not determine all the attritbutes of an identifier therefore the attributes are entered by remaining phases of complier.

Various phases can use the symbol table in various ways. For example while doing semantic analysis and intermediate code generation, we need to know what type of identifier is. Then during code generation typically information about how much storage is allocated identifier is seen.

(viii) Error detection and handling : To err is human. As programs are written by human beings therefore they can not be free from erros. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so the compilation can proceed. Normally, the errors are reported in the form the of message. When the input characters from the input do not form the token, the lexical analyzer detects error. Large number of errors can be detected in syntax phase. Such errors are popularly known as syntax errors. During sementic analysis; type mismatch kind of error is usually detected.

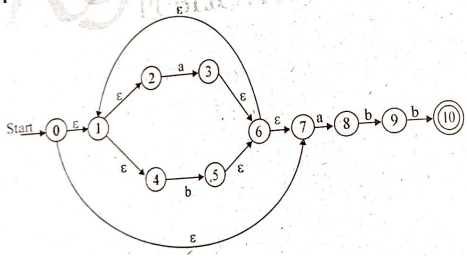
Q.3. What is Finite Automata ? Convert NFA (a|b)* abb into equivalent DFA.

Ans. Finite Automata : An automaton with a set of states, and its control moves from state to state in response to external inputs is called a finite automaton.

A finite automaton, FA, provides the simplest model of a computing device. It is central processor of finite capacity and it is based on the concept of state. It can also be given formal mathematical definition.

Conversion NFA (a|b)* abb into equivalent DFA :

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



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

ε-closure(0) = {0,1,2,4,7}

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

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

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

ε-closure(move(A,a)) = {1,2,4,5,6,7,8}

Dtran[A,a] = B

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

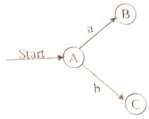
move(A,b) = {5}

ε-closure(move(A,b)) = ε-closure(5) = {3,6,7,1,2,4}

ε-closure(move(A,b)) = {1,2,4,5,6,7}

Dtran[A,b]=C

DFA and Transition table after step 2 is shown below.



DFA states	Input Symbols	
	a	b
A	B	C
B		
C		

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

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

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

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

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

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

Dtran[B,a] = B

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

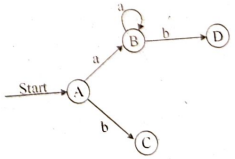
move(B,b) = {5,9}

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

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

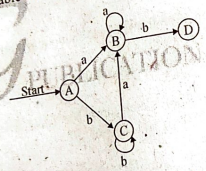
Dtran[B,b] = D

DFA and Transition table after step 3 is shown below.



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

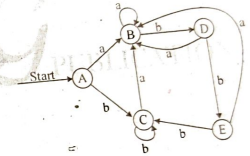
Step 4 : Compute Transition from state C on input symbol {a,b}
C = {1,2,4,5,6,7}
Step 4.1 : Compute ε-closure(move(C,a))
move(C,a) = {3,8}
ε-closure(move(C,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(C,a)) = {1,2,3,4,6,7,8}
Dtran[C,a] = B
Step 4.2 : Compute ε-closure(move(C,b))
move(C,b) = {5}
ε-closure(move(C,b)) = ε-closure(5) = {5,6,7,1,2,4}
ε-closure(move(C,b)) = {1,2,4,5,6,7}
Dtran[C,b] = C
DFA and Transition table after step 4 is shown below.



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

Step 5 : Compute Transition from state D on input symbol {a,b}
D = {1,2,4,5,6,7,9}
Step 5.1 : Compute ε-closure(move(D,a))
move(D,a) = {3,8}
ε-closure(move(D,a)) = ε-closure(3,8) = (3,8,6,7,1,2,4)
ε-closure(move(D,a)) = {1,2,3,4,6,7,8}
Dtran[D,a] = B

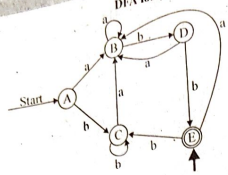
Step 5.2 : Compute ε-closure(move(D,b))
move(D,b) = {5,10}
ε-closure(move(D,b)) = ε-closure(5,10) = {5,10,6,7,1,2,4}
ε-closure(move(C,b)) = {1,2,4,5,6,7,10}
Dtran[D,b] = E
Step 6 : Compute Transition from state E on input symbol {a,b}
E = {1,2,4,5,6,7,10}
Step 6.1 : Compute ε-closure(move(E,a))
move(E,a) = {3,8}
ε-closure(move(E,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(E,a)) = {1,2,3,4,6,7,8}
Dtran[E,a] = B
Step 6.2 : Compute ε-closure(move(E,b))
move(E,b) = {5}
ε-closure(move(e,b)) = ε-closure(5) = {5,6,7,1,2,4}
ε-closure(move(E,b)) = {1,2,4,5,6,7}
Dtran[E,b] = C
DFA and Transition table after step 6 is shown below.



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

Step 7 : No more new DFA states are formed.
Stop the subset construction method.
The start state and accepting states are marked the DFA.

DFA for (a|b)*abb



DFA = Subset of NFA
A = {0,1,2,4,7} start state
B = {1,2,3,4,6,7,8}
C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}
E = {1,2,4,5,6,7,10} ← State E contains the NFA accepting state (state 10)

Unit – II

Q.4.(a) What is CFG ? (8)

Ans. A context free grammar is termed as G is a collection of the following :
(i) N is a set of non terminals
(ii) T is a set of terminals
(iii) S is a start symbol
(iv) P is a set of production rules
So finally G can be represented as G = (N, T, S, P). Production rules are given in the following form Non terminal → (N U T)*

Example for CFG :
S → aA | bB
A → abB | aB
B → a | b + ∈

Q.4.(b) Explain how regular expressions are used for token specification. (7)

Ans. Recognition of Tokens : For a programming language ther are various types of tokens such as identifier, keywords, constants and operators and so on. The token is usually represented by a pair token type and token value.

Token type	Token value

Token representation

The token type tells us the category of token and token value gives us the information regarding token. The token value is also called token attribute. During lexical analysis process

the symbol table is maintained. The token value can be a pointer to symbol table in case of identifier and constants. The lexical analyzer reads the input program and generates a symbol table for tokens.

For example :
We will consider some encoding of tokens as follows.

Token	code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

Consider, a program code as

```
if(a<10)
    i = i+2
else
    i = i – 2
```

Our lexical analyzer will generate following token stream.
1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2,(5,107), 10, (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

Location counter	Type	Value
100	identifier	a
:	:	
105	constant	10
107	identifier	i
:	:	
110	constant	2

In above example, scanner scans the input string and recognize "if" as a keyword, return token type as 1 since in given encoding code 1 indicates keyword "if" and hence 1 is a beginning of token stream. Next is a pair (8,1) where 8 indicates parenthesis and 1 indicates opening parenthesis '('. Then we scan the input 'a' it recognizes it as identifier and searches symbol table to check whether the same entry is present. If not it inserts the information about this identifier in symbol table and returns 100. If the same identifier or variable is already present in symbol table then lexical analyzer does not insert it into the table instead it returns the location where it is present.

Q.5. Perform shift-reduce parsing for string $id_1 + id_2 + id_3$ for the following grammar :

$E \rightarrow E + E \mid E * E \mid id$

Ans.

Stack	Input buffer	Parsing action
$	id1+id2+id3$	Shift
$id1	+id2+id3$	Reduce by E → id
$E	+id2+id3$	Shift
$E+	id2+id3$	Shift
$E+id2	+id3$	Reduce by E → id
$E+E	+id3$	Shift
$E+E+	id3$	Shift
$E+E+id3	$	Reduce by E → id
$E+E+E	$	Reduce by E → E+E
$E+E	$	Reduce by E → E+E
$E	$	Accept

Here we have followed two rules.

(1) If the incoming operator has more priority than in stack operator then perform shift.

(2) If in stack operator has same or less priority than the priority of incoming operator then perform reduce.

Unit – III

Q.6.(a) Explain syntax directed translation scheme.

Ans. Syntax-Directed Translation Schemes : A syntax-directed translation scheme is a context-free grammar in which attributes are associated with the grammar symbols, and semantic actions, enclosed within braces ({ }), are inserted in the right sides of the productions. These semantic actions are basically the subroutines that are called at the appropriate times by the parser, enabling the translation. The position of the semantic action on the right side of the production indicates the time when it will be called for execution by the parser. When we design a translation scheme, we must ensure that an attribute value is available when the action refers to it. This requires that :

(1) An inherited attribute of a symbol on the right side of a production must be computed in an action immediately preceding (to the left of) that symbol, because it may be referred to by an action computing the inherited attribute of the symbol to the right of (following) it.

(2) An action that computes the synthesized attribute of a nonterminal on the left side of the production should be placed at the end of the right side of the production, because it might refer to the attributes of any of the right-side grammar symbols. Therefore, unless they are computed, the synthesized attribute of a nonterminal on the left cannot be computed.

These restrictions are motivated by the L-attributed definitions. Below is an example of a syntax-directed translation scheme that satisfies these requirements, which are implemented during predictive parsing :

$D \rightarrow T$ {L.type := T.type} L;
$T \rightarrow$ int {T.type := int}
$T \rightarrow$ real {T.type := real}
$L \rightarrow$ {$L1$.type = L.type} $L1$, id{enter(id.prt, L.type);}
$L \rightarrow$ id {enter(id.prt, L.type);}

The advantage of a top-down parser is that semantic actions can be called in the middle of the productions. Thus, in the above translation scheme, while using the production $D \rightarrow TL$ to expand D, we call a routine after recognizing T (i.e., after T has been fully expanded), thereby making it easier to handle the inherited attributes. Whereas a bottom-up parser reduces the right side of the production $D \rightarrow TL$ by popping T and L from the top of the parser stack and replacing them by D, the value of the synthesized attribute T.type is already on the parser stack at a known position. It can be inherited by L. Since L.type is defined by a copy rule, L.type = T.type, the value of T.type can be used in place of L.type. Thus, if the parser stack is implemented as two parallel arrays–state and value–and state [I] holds a grammar symbol X, then value [I] holds a synthesized attribute of X. Therefore, the translation scheme implemented during bottom-up parsing is as follows, where [top] is value of stack top before the reduction and [newtop] is the value of the stack top after the redcution :

$D \rightarrow TI$;
$T \rightarrow$ int {value[newtop] = int}
$T \rightarrow$ real {value[newtop] = real}
$L \rightarrow L1$, id{enter(value[top], value[top-3]);}
$L \rightarrow$ id{enter(value[top], value[top-1]);}

Q.6.(b) Explain three-address codes, triples and quadruples. (9)

Ans. Three Address code : Three address code is the typical example of low level representation. The term "three address code" contains three addresses two for the operands and one for the result.

Three address code is a sequence of statement of the general form

p : q op r

where p, q and r are names constant or compiler generated temporaries.

op stands for any operator, such as fixed-or-floating point arithmetic operator or logical operator on Boolean value.

There is only one operator on the right hand side of a statement ensuring that operations and expressions are simple. So no built in operations are allowed.

A source language expression like $x : x * y + z - x$ translated into three address sequ

as,

$$t_1 := x * y$$
$$t_2 := t_1 + z$$
$$t_3 := t_2 - x$$
$$x := t_3$$

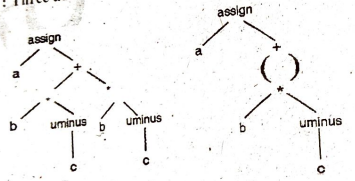
where t_1, t_2 are compiler generated temporary names.

Three address code is a linearized representation of a syntax tree or a DAG in wh
explicit names correspond to the interior nodes of the graph.

Following Table(1) represents the three address sequence for the syntax tree and da
Table(1). Variables names can appear directly in three address statements so Table1.(a) ha
statements corresponding to the leaves in Table1(a).

$t_1 := - c$ $t_2 := b * t_1$ $t_3 := - c$ $t_4 := b * t_3$ $t_5 := t_2 + t_4$ $a := t_5$	$t_1 := - c$ $t_2 := b * t_1$ $t_5 := t_2 + t_1$ $a := t_5$
(a) Code for the syntax free	*(b) Code for the dag*

Table (1) : Three address code corresponding to the tree and dag in the Fi



(a) Syntax tree (b) DAG

Fig.(1) : Graphical representation of a : = b * – c + b * –

Three address code gives better flexibility in terms of target code generation and
optimination as it is simpler.

The semantic rules for generating three address code from common programm
language constructs are similar to those for contructing syntax trees for generating postfix notat

Quadruple : A quadruple is a structure with at the most four field as, op arg 1, arg2
result. The op field is used to represent internal node for the operator. The arg 1 and a
represents the argument or operand of the expression and result field represent the result o
expression i.e. it stores the result of the expression.

The three-address statements with unary operators like $a := - y$ or $a := y$ do not use arg2. Operator like param does not use arg2 and result, it only use arg1. Conditional and unconditional jumps put the targe lable in result.

The values of the field arg1, arg2 and result are considered as pointers to symbol table entries for the names represented by these fields. Similarly whenever the temporary names are created they must be entered into the symbol table.

Example : The quadruples for the assignment $x := y * - z + y * - z$ are shown in following Table.

The three address statement and table for the above assignment statement is

$$t_1 := \text{uminus } z$$
$$t_2 := y * t_1$$
$$t_3 := \text{uminus } z$$
$$t_4 := y * t_3$$
$$t_5 := t_2 + t_4$$
$$x := t_5$$

	op	arg1	arg2	result
(0)	Uminus	z		t_1
(1)	*	y	t_1	t_2
(2)	uminus	z		t_3
(3)	*	y	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triple : Triple representation is strcuture with at the most three field arg1, arg2 and op. The field arg1 adn arg2 are the arguments of the operator op. In triple, temporaries are not used instead of that pointers in the symbol table are used directly. Triple corresponds to the respresentation of a syntax tree or DAG by an array of nodes.

Example : Following Table shows the triples for the assignment statement

$$x := y * - z + y * - z.$$

	op	Arg1	arg2
(0)	uminus	z	
(1)	*	y	(0)
(2)	uminus	z	
(3)	*	y	(2)
(4)	+	(1)	(3)
(5)	assign	x	(4)

Numbers in the round brackers are used to represent pointers into the triple structure. While symbol table pointers are represented by the names themselves

Q.7. Consider the following grammar :

E → E + T|T
T → T * F|F
F → (E)|id

and build SLR parsing table for it. **(15)**

Ans. Augmented Grammar is

E′ → . E
E → E + T | T

T → T * F | F
F → (E) | id

• LR(0) items

I_0 : E′ → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → id

I_1 : E′ → E .
E → E . + T

I_2 : E → T .
T → T . * F

I_3 : T → F .

I_4 : F → (.E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I_5 : F → id

I_6 : F → E + . T
T → . T * F
T → . F
F → . (E)
F → id

I_7 : T → T * . F
F → . (E)
F → . id

I_8 : F → (E.)
E → E . + T

I_9 : E → E + T .
T → T . * F

I_{10} : T → T * F .

I_{11} : F → (E) . action

Parsing Table for the grammar :

State	id	+	*	(	)	$	E	T	F
							goto		
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Sequence of steps for the input id * id + id $:

	STACK	INPUT	ACTIONS
1	0	id * id + id $	shift
2	0 id 5	* id + id $	reduce F → id
3	0 F 3	* id + id $	reduce T → F
4	0 T 2	* id + id $	shift
5	0 T 2 * 7	id + id $	shift
6	0 T 2 * 7 id 5	+ id $	reduce by F → id
7	0 T 2 * 7 F 5	+ id $	reduce by T → T * F
8	0 T 2	+ id $	reduce E → T
9	0 E1	+ id $	shift
10	0 E1 + 6	id $	shift
11	0 E1 + 6 id 5	$	reduce F → id
12	0 E1 + 6 F 3	$	reduce T → F
13	0 E1 + 6 T 9	$	reduce E → E + T
14	0 E1	$	accept

By using method, a parsing table can be obtained for any grammar. But the action table obtained from the method above will not necessarily be without multiple entries for every grammar. Therefore, we define a SLR(1) grammar as one for which we can obtain the action table without multiple entries.

Unit – IV

Q.8.(a) List the various error recovery strategies.

Ans. Recovery from Syntax Errors : Once a syntax error is detected and reported, the parser must be able to recover from a syntactic error. There are various methods for error recovery. Some strategies for error recovery are :

(1) Panic mode recovery :

– This is the simplest method to implement and can be used by most parsing methods. The parsing state attained is the initial state preparatory to the parsing of the next source statement.

– To attain this initial state, parser discards input symbol, one at a time until one of designated set of synchronizing token is found. The synchronizing tokens are usually delimiters such as semicolon, or end for end of statements, etc.

– The compiler designer must select the synchronizing tokens appropriate for the source amount of input without checking it for additional errors. Unlike other method panic mode recovery is simple to use and guarantees that it will not go into an infinite loop.

– The disadvantage is that it skips a considerable amount of input without checking for additional errors. And in situations where multiple errors in the same statement are rare, this method may be quite adequate.

(2) Phrase-level recovery :

– On discovering an error a parser may perform local correction on the remaining input; that is it may replace a prefix of the remaining input by some string that allows the parser to continue.

The basic local corrections to recover from syntax error are :

– Deletion of a source symbol e.g. to delete an extra semicolon.

– Replace comma by a semicolon.

– Insertion of synthetic symbol e.g. to insert a missing semicolon.

– The choice of the local correction is left to the compiler designer.

– The motive behind the local correction is to present a new string to the parser which would lead to by passing of the error situation to continue parsing. But while performing local correction we must be careful to choose replacement that do not lead to infinite loops.

– This type of replacement can correct any input string and has been used in several error repairing compilers.

– The method major drawback is the difficultly it has in coping with situations in which the actual error has occurred before the point of detection.

(3) Error production :

– While writing the grammar for a language we have a good idea of the common errors that might be encountered. We can augment the grammar for the language at hand with productions that generate the erroneous constructs.

– We then use the grammar augmented by these error productions to construct a parser. During parsing, if an error production is used, we can give an appropriate error message and the parsing can continue normally.

(4) Global Correction :

– In a given error situation multiply recovery possibility might exist. We should choose one which involves smaller number of insertions, deletions and replacements in processing of incorrect input string. This is known as minimum distance recovery.

– Also there are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction. Given an incorrect input string x and grammar G, the algorithms will find a parse tree for a related string x and grammar G, these algorithms will find a parse tree for a related string y such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.

– These methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Q.8.(b) Explain the importance of symbol tables in compiler design. (8)

Ans. Symbol table is an important data structure used in a compiler.

Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

Q.9. Explain the various strategies for code generation. (15)

Ans. Simple Code Generation Algorithm : In this method computed results can be kept in registers as long as possible.

For example

x : = a+b;

The corresponding target code is

ADD b,R_1	Here R_1 holds value of a Here cost = 2

OR

MOV b,R_1 ADD R_1,R_0	Here R_0 holds value of a Here cost = 2

– The code generator algorithm uses descriptors to keep track of register contents and addresses for names.

(1) A register descriptor is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty. As the code generation for the block progresses the registers will hold the values of computations.

(2) The address descriptor stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.

– The modes of operand addressability are as given below.
S is used to indicate value of operand in storage.
R is used to indicate value of operand in register.

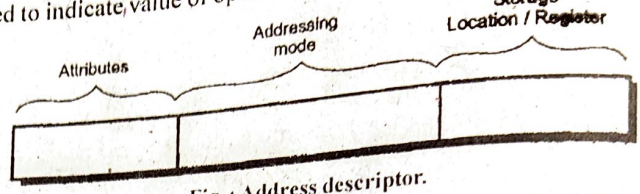


Fig. : Address descriptor.

IS indicates that the address of operand is stored in storage i.e. indirect accessing.
IR indicates that the address of operand is stored in register i.e. indirect accessing.

– The address descriptor has following fields.

The attributes mean type of the operand. It generally refers to the name of tempo variables.

The addressing mode indicates whether the addresses are of type *'S','R','IS','IR'*.

The third field is location field which indicates whether the address is in storage loca or in register.

– Similarly the register descriptor can be shown as below.

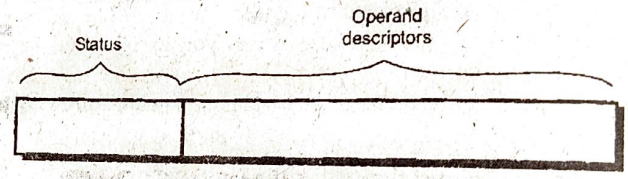
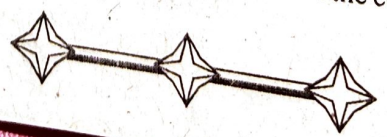


Fig. : Register descriptor.

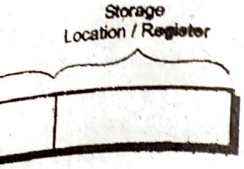
By using register descriptors we can keep track of the register which are curren occupied. The status field is of Boolean type which is used to check whether the register occupied with some data or not. When the status field holds the value 'True' then opera descriptors fields contains the pointer to the operand descriptor who is having the latest value the register.

Generic Code Generation Algorithm : The code generator generator is a tree rewriti technique in which the instruction selection can be done automatically from a high-le specification of the target machine. Thus the input to the code generator will be a sequence trees at the semantic level of target machine. The set of tree rewriting rules is called a tr translation scheme. In this tree translation scheme the templates are given and corresponding each template equivalent machine code is given. Thus

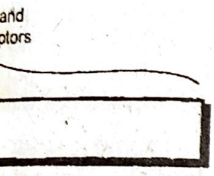
(1) Draw the tree structure for the complete expression.
(2) Traverse the tree in bottom up fashion and match the subtrees with the appropri template from the tree translation scheme.
(3) The corresponding code for the matched template is stored in a buffer.
(4) When we reach to the root node, buffer contains the complete machine code for given expression.



as given below.
ıge.
ster.



iptor.
ed in storage i.e. indirect accessing.
ed in register i.e. indirect accessing.
nerally refers to the name of temporar
ddresses are of type 'S','R','IS','IR'.
whether the address is in storage locatio
vn as below.



or.
k of the register which are currentl
used to check whether the register is
holds the value 'True' then operand
riptor who is having the latest value in
generator generator is a tree rewriting
ne automatically from a high-level
code generator will be a sequence o
tree rewriting rules is called a tree
plates are given and corresponding to
ession.
tch the subtrees with the appropriate
late is stored in a buffer.
ns the complete machine code for the

COMPLIER DESIGN

July – 2022

Paper Code:-PCC-CSE-302-G

Note : *Attempt five questions in all, selecting one question from each Section. Question No.* **1** *is compulsory. All questions carry equal marks.*

Q.1. Write short notes on :

(a) **Applications of compiler** (2.5)
(b) **Notations for regular expressions** (2.5)
(c) **Compiler construction tools** (2.5)
(d) **Benefits of Intermediate code generation** (2.5)
(e) **Ambiguous grammar** (2.5)
(f) **Code generation** (2.5)

Ans.(a) Applications of compiler :

(1) Compiler design helps full implementation Of High-Level Programming Languages.
(2) Support optimization for Computer Architecture Parallelism.
(3) Design of New Memory Hierarchies of Machines.
(4) Widely used for Translating Programs.
(5) Used with other Software Productivity Tools.

Ans.(b)Notations for regular expressions : Regular expression is a formula that describes a possible set of string. Component of regular expression..

X	the character x
.	any character, usually accept a new line
[x y z]	any of the characters x, y, z, ...
R?	a R or nothing (=optionally as R)
R*	zero or more occurrences.....
R+	one or more occurrences .....
R1R2	an R1 followed by an R2
R1|R1	either an R1 or an R2.

Ans.(c)Compiler construction tools : A compiler is tedious and time consuming task. Therefore there are some specialized tools for implementing various phases of compiler. These tools are called compiler construction tools.

Various complier construction tools are as follows :

(i) Scanner generator - For example : LEX

(ii) Parser generator - For example : YACC

(iii) Syntax directed translation engines.

(iv) Automatic code generator.

(v) Data flow engines.

(i) *Scanner generator* : These generators generate lexical analyzers. The specifica given to these generators are in the form of regular expressions.

The UNIX has utility for a scanner generator called LEX. The specification given LEX consisis of regular expressions for representing various tokens.

(ii) *Parser generators* : These produce the syntax analyzer. The specification give these generators is given in the form of context free grammar. Typically UNIX has a tool c YACC which is a parser generator.

(iii) *Syntax-directed translation engines* : In this tool the parse tree is scan completely to generate an intermediate code. The translation is done for each node of the

(iv) *Automatic code generator* : These generators take an intermediate code as in and converts each rule of intermediate language into equivalent machine language. The temp matching technique is used. The intermediate code statements are replaced templates represent the corresponding sequence of machine instruction.

(v) *Data flow engines*: The data flow analysis is required to perform good optimization. The data flow engines are basically useful in code optimization.

Ans.(d)Benefits of Intermediate code generation :

(1) It is Machine Independent. It can be executed on different platforms.

(2) It creates the function of code optimization easy. A machine-independent optimizer can be used to intermediate code to optimize code generation.

(3) It can perform efficient code generation.

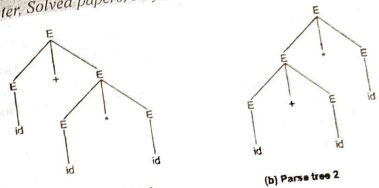
(4) From the existing front end, a new compiler for a given back end can be generat

(5) Syntax-directed translation implements the intermediate code generation, furt augmenting the parser, it can be folded into the parsing

Ans.(e)Ambiguous grammar : A grammar G is said to be ambiguous if it genera more than one parse trees for sentence of language L(G).

For example : $E \rightarrow E + E \mid E * E \mid id$

Then for id + id * id



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

Fig. : Ambiguous grammer

Ans.(f) Code generation : The code generation is the last phase in the compilation process. In computer science, the code generation is the process which converts some internal representation of source code into a form (e.g., machine code) that cna be readily executed by a machine. The code generation is done by a code generator component of a compiler. The code generator translates the intermediate representation of the source program into a sequence of machine instructions.

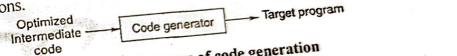


Fig. : The process of code generation

The input to a code generator is an intermediate language program that can be :

(i) A sequence of quadruples.

(ii) A sequence of triples.

(iii) A tree, or

(iv) A postfix polish string.

The output of a code generator is called the target program (or object program) that can be :

(i) An absolute machine language program

(ii) A relocatable machine language program

(iii) An assembly language program

(iv) A program in some other programming language.

Unit – I

Q.2. Describe the phases of compiler. **(15)**

Ans. Phases of Compiler : Following are the phases of compiler :

(i) Lexical Analysis : The lexical analysis is also called scanning. It is the phase of compilation in which the complete source code is scanned and your source program is broken

up into group for strings called token. A token is a sequence of characters having a collec
meaning. For example if some assignement statement in your source code is as follows,

total = count + rate * 10

Then in lexical analysis phase this statement is broken up into series of tokens as follo

(a) The identifier total
(b) The assignment symbol
(c) The identifier count
(d) The plus sign
(e) The identifier rate
(f) The multiplication sign.
(g) The constant number 10

The blank characters which are used in the programming statement are elimina
during the lexical analysis phase.

(ii) Syntax Analysis : The syntax analysis is also called parsing. In this phase th
tokens generated by the lexical analyzer are grouped together to form a hierarchical structu
The syntax analysis determines the structure of the source string by grouping the tokens toget
For the expression total = count + rate * 10 the phase tree and can be generated as follo

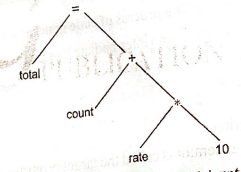


Fig.(1) : Parse tree for total = count + rate * 10

In the statement 'total = count + rate *10' first of all rate*10 will be considered becaus
in arithmetic expression the multiplication operation should be performed before the addition
And then the addition operation will be considered. For building such type of syntax tree th
production rules are to be designed. The rules are usually expressed by context free gramma
For the above statement the production rules are :

(1) $E \leftarrow$ identifier (2) $E \leftarrow$ number
(3) $E \leftarrow E1 + E2$ (4) $E \leftarrow E1 * E2$
(5) $E \leftarrow (E)$

where E stands for an expression.

- By rule (1) count and rate are expression and
- By rule (2) 10 is also an expression.

- By rule (4) we get rate*10 as expression.
- And finally count + rate*10 is an expression.

(iii) Semantic Analysis : Once the syntax is checked in the syntax analyzer phase i.e. the semantic anaylsis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if ... else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

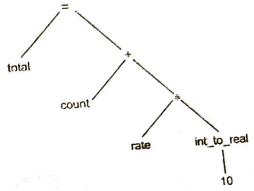


Fig.(2) : Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

(iv) Intermediate Code Generation : The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple posix*. Here we will consider an intermediate code in three address code form. This is like an assembly language.The three address code consists of instructions each of which has at the most three operands. For example,

t1 : = int_to_real (10)
t2 : = rate × t1
t3 : = count + t2
total : = t3

(v) Code Optimization : The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. Thus by optimizing the code the overall running time of the target program can be improved.

(vi) Code Generation : In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine insturctions.

MOV rate, R1

```
MUL # 10.0, R1
MOV count, R2
ADD R2, R1
MOV R1, total
```

(vii) Symbol table management : To support these phases of compiler a sy... table is maintained. The task of symbol table is to store identifier (variables) used in the prog... The symbol table also stores information about attributes of each identifier. The attri... of identifiers are usually its type, its type, its scope, information about the storage allocate... it.

The symbol table also stores information about the subroutines used in the progr... are. Then during code generation typically information about how much storage is allocated to identifier is seen.

In case of subroutine, the symbol table stores the name of the subroutine number of argum... passed to it, type of these arguments, the method of passing these arguments (may be ... value or call by reference) and return type if any.

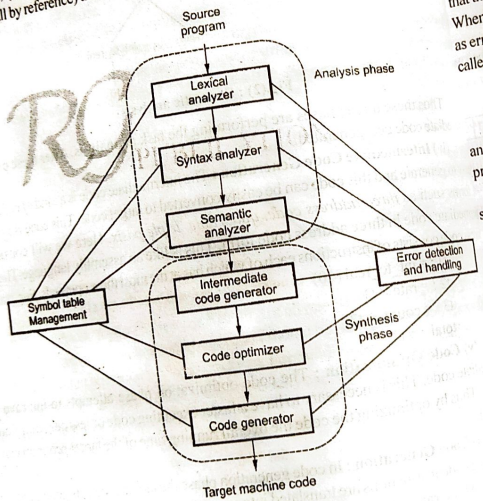


Fig.(3) : Phases of Compiler

Basically symbol table is a data structure used to store the information about identifiers. The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record efficiently.

During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of complier.

Various phases can use the symbol table in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what type of identifiers

(viii) Error detection and handling : To err is human. As programs are written by human beings therefore they can not be free from erros. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form the of message. When the input characters from the input do not from the token, the lexical analyzer detects it as error. Large number of errors can be detected in syntax phase. Such errors are popularly called as syntax errors. During semantic analysis; type mismatch kind of error is usually detected.

Q.3.(a) What is Laxical analysis ? (9)

Ans. Lexical analysis : Lexical analysis is the operation of reading the input program and breaking it into a sequence of lexemes (tokens). The syntax analyzer uses these tokens to produce parse tree.

Each token is a sequence of characters that represents a unit of information in the source program.

The interaction between lexical analyzer and parser is well defined. The parser calls a single lexical analyzer subroutine every time as it needs a new token and then subroutine (i.e. Lexical Analyzer) reads input characters until it can identify the next token and returns it to the parser. The relationship is shown in Fig.

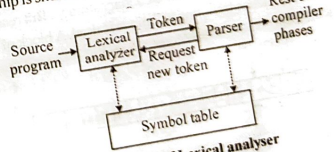


Fig. : Role of Lexical analyser

In addition the lexical analyzer also performs certain secondary tasks like remov... comments and white spaces (blank; tab and new line characters) from the source progr... may also be given the responsibility of making a copy of the source program with the... message marked in it. Each error message may also be associated with a line number.

The analyzer may keep track of the line number by tracking the number of ... characters seen while reading source program a character by character.

Q.3.(b) What is role of Input buffer in lexical analysis ?

Ans. Input Buffering : The lexical analyzer scans the input string from left to righ... **character at a time. It uses two pointers begin-ptr (bp) and forward_ptr (fp) to keep tra...** the portion of the input scanned.

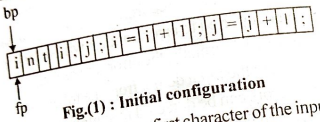


Fig.(1) : Initial configuration

Initially both the pointers point to the first character of the input string as shown in...

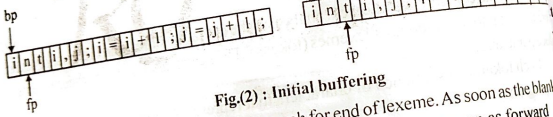


Fig.(2) : Initial buffering

The forward_ptr moves ahead to search for end of lexeme. As soon as the blank sp... is encountered, it indicates end of lexeme. In above example as soon as forward_pt... encounter a blank space the lexeme "int" is identified.

The fp will be moved ahead at white space. When fp encounters white space, it g... and moves ahead. Then both the begin_ptr (bp) and forward_ptr (fp) are set at next tok...

The input character is thus read from secondary storage. But reading in this way ... secondary storage is costly. hence buffering technique is used. A block of data is first read... a buffer, and then scanned by lexical analyzer.

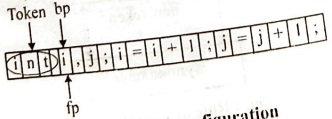


Fig.(3) : Initial configuration

There are two methods used in this context : one buffer scheme and two buffer scheme.

(1) One buffer scheme : In this one buffer scheme, only one buffer is used to store the input string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme.

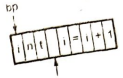


Fig. : One buffer scheme storing input string

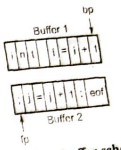


Fig. : Two buffer scheme storing input string

(2) Two buffer scheme : To overcome the problem of one buffer scheme, in this ...ethod two buffers are used to store the input string. The first buffer and second buffer are ...canned alternately. When end of current bufer is reached the other bufer is filled. The only ...problem with this method is that if length of the lexeme is longer than length of the buffer than ...canning input cannot be scanned completely.

Initally both the bp and fp are pointing to the first character of first buffer. Then the fp ...noves towards right in search of end of lexeme. As soon as blank character is recognized, the ...tring between bp and fp is identified as corresponding token. To identify the boundary of first ...uffer end of buffer character should be placed at the end of first bufer. Similarly end of second ...uffer is also recognized by the end of buffer mark present at the end of second buffer. When ...okens is identified. This *eof* character introduced at the end is called *sentinel* which is used to ...dentify the end of bufer. fp encounters first *eof*, the one can recognize end of first buffer and hence filling up of second ...uffer is started. In the same way when second *eof* is obtained then it indicates end of second ...uffer. Alternatively both the bufers can be filled up until end of the input program and stream of

Unit – II

Q.4. Calculate first and follow for the following grammar : (15)

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

Ans. $E \rightarrow (E)\ E'$

After removal of left recursion

$E \rightarrow i\ d\ E'$

$E' \rightarrow + E\ E' \mid \in$

$E' \rightarrow * E E' / \in$ Follow ( )

First ( )

E

E'

First $(E) = \{(, id\}$

First $(E') = \{+, *, \in\}$

Follow $(E) = \{), \$, +, *\}$

Follow $(E') = \{+, *, \$\}$.

Q.5. Explain top–down parsing with a suitable example.

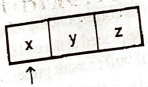
Ans. Top–down Parser : As we have seen that in top–down parsing the parse tre
generated from top to bottom (i.e. from root to leaves). The derivation terminates when
required input string terminates. The leftmost derivation matches this requirement.The m
task in topdown parsing is to find the appropriate production rule in order to produce
correct input string. We will understand the process of top–down parsing with the help of s
example.

Consider a grammar.

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

Consider the input string xyz is as shown below.



Input buffer

Now we will construct the parse tree for above grammar deriving the given input strin
And for this derivation we will make use of top down approach.

Step 1 : The first leftmost leaf of the parse tree matches with the first input symb
Hence we will advance the input pointer. The next leaf node is P. We have to expand the no
P. After expansion we get the node y which matches with the input symbol y.

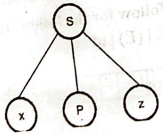


Fig. : (1)

Step 2 : Now the next nod is w which is not matching with the input symbol. Hence we go back to see whether there is another alternative of P. The another alternative for P is y which matches with current input symbol. And thus we could produce a successful parse tree for given input string.

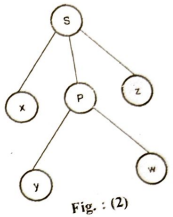


Fig. : (2)

Step 3 : We halt and declare that the parsing is completed successfully.

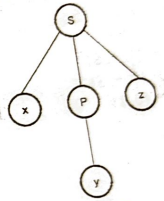


Fig. : (3)

In top–down parsing selection of proper rule is very important task. And this selection is based on trial and error technique. That means we have to select a particular rule and if it is not producing the correct input strin then we need to backtrack and then we have to try another production. This process has to be repeated until we get the correct input string. After trying all the productions if we found every production unsuitable for the string match then in that case the parse tree cannot be built.

Unit – III

Q.6. Explain the making of LR (o) parsing table for the following grammar :

$E \rightarrow E * B \mid E + B \mid B, B \rightarrow O \mid I$

Ans. $E \rightarrow E * B / E + B / B$, $B \rightarrow 0/1$.

After removal of left recussion

$E \rightarrow E * B/B$

$E \rightarrow E + B/B$

$E \rightarrow BE'$

$E' \rightarrow * BE'/ + BE'/\in$

$B \rightarrow 0$

$B \rightarrow 1$

$E \rightarrow BE'$

$E' \rightarrow * BE'$

$E' \rightarrow + BE'$

$B \rightarrow 0$

$B \rightarrow 1$

First $(E) = \{0, 1, E\}$

First $(E') = \{*, E\}$

First $(E) = \{*, +, E\}$

First $(B) = \{0\}$

First $(B) = \{0, 1\}$

Follow $(E) = \{*, +, \$\}$

Follow $(E') = \{*, +, \$\}$

Follow $(B) = \{0, 1\}$

I_0 $E' \rightarrow \cdot E$
$E' \rightarrow \cdot BE'$
$E' \rightarrow \cdot * BE'$
$E' \rightarrow \cdot + BE'$
$B \rightarrow \cdot O$
$B \rightarrow \cdot 1$

I_1 $E' \rightarrow E \cdot$

I_2 $E \rightarrow B \cdot E'$
$E' \rightarrow \cdot * \cdot BE'$
$E' \rightarrow \cdot + BE'$
$B \rightarrow \cdot O$
$B \rightarrow \cdot 1$

I_3 $E' \rightarrow * \cdot BE'$
$B \rightarrow \cdot O$
$B \rightarrow \cdot 1$

I_4 $E' \rightarrow + \cdot BE'$

$B \rightarrow \cdot O$
$B \rightarrow \cdot 1$

I_5 $B \rightarrow O \cdot$
$B \rightarrow 1 \cdot$

I_6 $E \rightarrow BE' \cdot$
$E' \rightarrow * B ; E'$

I_7 $E' \rightarrow \cdot * BE'$
$E' \rightarrow \cdot + BE'$

I_8 $E' \rightarrow + B \cdot E'$
$E' \rightarrow \cdot * BE'$
$E' \rightarrow \cdot + BE'$

I_9 $E' \rightarrow * BE' \cdot$
$E' \rightarrow + BE'$

I_{10} $E' \rightarrow BE'$

State	Action				Goto		
	O	1	+	*	$	E	E′	B
0	S_1	S_5	S_4	S_3		1	9	5
1							6	
2	r_5	r_5	S_4	S_3			9	5
3	r_5	r_5		S_7				9
4	r_5	r_5	S_8					9
5	–	–	–	–	–	–	–	–
6								
7			S_4	r_6	S_3		9	
8			S_4	S_3				
9			r_4	r_1				
10								

Q.7. How CLR parser differs from LALR parser ? Please explain. **(15)**

Ans. CLR Parser : CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR (1) items to construct the CLR (1) parsing table. CLR (1) parsing table make more number of states as compared to the SLR (1) parsing. In the CLR (1), it can locate the reduce node only in the lookahead symbols.

LALR Parser : LALR Parser is Look Ahead LR Parser. It is intermediate in power between SLR and CLR parser. It is the compaction of CLR Parser, and hence tables obtained in this will be smaller than CLR Parsing Table.

For constructing the LALR (1) parsing table, the canonical collection of LR (1) items is used. In the LALR (1) parsing, the **LR (1)** items with the equal productions but have seven look ahead are grouped to form an individual set of items. It is frequently the similar as CLR (1) parsing except for the one difference that is the parsing table.

The overall structure of all these LR Parsers is the same. There are some common factors such as size, class of context-free grammar, which they support, and cost in terms of time and space in which they differ.

Let us see the comparison between CLR, and LALR Parser.

LALR Parser	CLR Parser
It is easy and cheap to implement.	It is expensive and difficult to implement.
LALR Parser is the smallest in size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
It is intermediate in power between SLR i.e., SLR < LALR < CLR.	It is very power ful and works on a large class of grammar.
It requires more time and space complexity.	It also requires more time and space complexity.

Unit – IV

Q.8. What are the various strategies for code optimization ? (15)

Ans. Code Optimization : Code optimization is an optional phase designed to improve the intermediate code so that the resulting target program

(a) Runs faster and/or

(b) Takes less space.

The code optimization is done by a program called code optimizer. The design of good optimizer is very important part of compiler design, as it decides the speed of execution of source program. A good optimizing compiler can improve the target program by a considerable amount in overall speed. The principal sources of optimization are :

(i) Efficient utilization of the register and instruction set of a machine.

(ii) Inner loops.

(iii) Language implementation details inaccessible to the user.

(iv) Identification of common subexpressions.

(v) Replacement of runtime computations by compile time computations.

There are following types of optimization :

(a) Local optimization : *e.g.*, elimination of common subexpressions. The sequence of statements

$$P : = Q + R + S$$
$$T : = Q + R + X$$

after eliminating might the common subexpression the code be rewritten :

$$T_1 : = Q + R$$
$$P : = T_1 + S$$
$$T : = T_1 + X$$

(b) Loop Optimization : *e.g.*, speedup of loops. This involves moving the loop invariant computation, outside the loop.

Moving back to our example statement

sum : = bonus + basic × 50

T_1 : = inttoreal(50)
T_2 : = id_3 * T_1
T_3 : = id_2 + T_2
id_1 : = T_3

Fig.(1) : Intermediate code statements

The intermediate code statements of fig.(1) may be optimized as :

(i) The conversion of 50 from integer to real representation can be done once and for all at compile time, and thus eliminating the statement

T_1 : = inttoreal (50).

(ii) As id_3 is used only once, to transmit its value to id_1, therefore we can substitute id_1 for T_3 and thus eliminating the last statement.

$id_1 : = T_3$

Therefore the resulting optimized code is shown in fig.(2).

T_3 : = id_3 × 50.0
id_1 : = id_3 + T_1

Fig.(2) : Optimized code statements

Q.9. Explain code generation in detail. (15)

Ans. Code Generation : The code generation is the last phase in the compilation process. In computer science, the code generation is the process which converts some internal

representation of source code into a form (e.g., machine code) that cna be readily executed by a machine. The code generation is done by a code generator component of a compiler. The code generator translates the intermediate representation of the source program into a sequence of machine instructions.

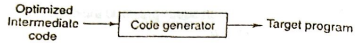


Fig. : The process of code generation

The input to a code generator is an intermediate language program that can be :

(i) A sequence of quadruples.
(ii) A sequence of triples.
(iii) A tree, or
(iv) A postfix polish string.

The output of a code generator is called the target program (or object program) that can be :

(i) An absolute machine language program
(ii) A relocatable machine language program
(iii) An assembly language program
(iv) A program in some other programming language.

Various properties desired by an object code generation phase are :

(i) *Correctness* : It should produce a correct code and do not alter the purpose of source code.

(ii) *High quality* : It should produce a high quality object code.

(iii) *Efficient use of resources of the target machine* : While generating the code it is necessary to know the target machine on which it is going to get generated. By this the code generation phase can make an efficient used of resource of the target machine. For instance memory utilization while allocating the registers or utilization of arithmetic logic unit while performing the arithmetic operations.

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

Design Issues : Let us discuss some common issues in design of code generator.

(1) Input to the code generator : The code generation phase takes intermediate code as input. The intermediate code along with the symbol table information is used to determine the runtime addresses of the data objects. These data objects are denoted by the names in the

intermediate representation. The intermediate code may be in any form such as three address code, quadruple, posix notation or it may be represented using graphical representations such as syntax trees or DAGs. The intermediate code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code. In the front end of compiler necessary type checking and type conversion needs to be done. The detection of the semantic errors should be done before submitting the input to the code generator. The code generation phase requires the complete error free intermediate code as input.

(2) Target programs : The output of code generator is target code. Typically, the target code comes in three forms such as : absolute machine kanguage, relocatable machine language and assembly language.

The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

For example : The WATFIV and PL/C are the compilers which produce the absolute code as output.

The advantage of producing the relocatable machine code as output is that the subroutines can be compiled separately. Relocatable object modules can be linked together and loaded for execution by a linking loader. This offers great flexibility of compiling the subroutines separately. If the target machine can not handle the reloacation automatically then the compiler must provide explicit relocation information to the loader in order to link the separately compiled subroutine segments.

The advantage of producing assembly code as output makes the code generation process easier. The symbolic instructions and Macro facilities of assembler can be used to generate the code. It is advantageous to have language as output for the relocatable machine code as target code.

(3) Memory management : Both the front end and code generator performs the task of mapping the names in the source program to addresses to the data objects in run time memory. The names used in the three address code refer to the entries in the symbol table. The type in a declaration statement determines the amount of storage(memory) needed to store the declared identifier. Thus using the symbol table information about memory requirements code generator determines the addresses in the target code.

Similarly if the three address code contains the labels then those labels can be converted into equivalent memory addresses. For instance if a reference to 'goto *i*' is encountered in three address code then appropriate jump instruction can be generated by computing the memory address for label *i*.

(4) Instruction selection : The uniformity and completeness of instruction set is an important factor for the code generator. The selection of instruction depends upon the instruction set of target machine. The speed of instruction and machine idioms are two important factors in selection of instructions. If we do not consider the efficiency of target code then the instruction selection becomes a straight forward task.

For each type of three address code the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

For example x:=a+b then the code sequence that can be generated as

```
MOV a,R0 /* loads the a to register R0*/
ADD b,R0 /* performs the addition of b to R0*/
MOV R0,x /* stores the contents of register R0 to x*/
```

In the above example the code is generated line by line. Such a line by line code generation process generates the poor code because the redundancies can be achieved by subsequent lines and those redundancies can not be considered in the process of line by line code generation.

For example

```
x:= y + z
a:= x + t
```

The code for the above statements can be generated as follows :

```
MOV y, R0
ADD z, R0
MOV R0, x
MOV x, R0
ADD t, R0
MOV R0, a
```

The above generated code is a poor code because MOV R0,a is not used and statement MOV a,R0 is redundant. Hence the efficient code can be

```
MOV y, R0
ADD z, R0
```



```
ADD t, R0
MOV R0, a
```

The quality of generated code is decided by its speed and size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

(5) Register allocation : If the instruction contains register operands then such a use becomes shorter and faster than that of using operands in the memory. Hence whole generating a good code efficient utilization of register is an important factor. There are two important activities done while using registers.

(1) *Register allocation* – During register allocation, select appropriate set of variables that will reside in registers.

(2) *Register assignment* – During register assignment, pick up the specific register in which corresponding variable will reside.

Obtaining the optimal (minimum) assignment of registers to variable is difficult.

Certain machines require register pairs such as even idd numbered registered for some operands and results.

For example in IBM systems integer multiplication requires register pair.

Consider the three address code

$t_1 := a + b$

$t_2 := t_1 * C$

$t_3 := t_2 / d$

The efficient machine code sequence will be

```
MOV a,R0
ADD b, R0
MUL c,R0
DIV d,R0
MOV R0,t3
```

(6) Choice of evaluation order : The evaluation order is an important factor in generating an efficient target code. Some orders require less number of registers to hold the intermediate results than the others. Picking up the best order is one of the difficulty in code

generation. Mostly, we can avoid this problem by referring the order in which the three addre code is generated by semantic actions.

(7) Approaches to code generation : The most important factor for a code generati is that it should produce the correct code. With this approach of code generation vario algorithms for generating code are designed.