B.E.

Seventh Semester Examination, 2009-2010

Compiler Design (CSE-405-E)

Note : Attempt any *five* questions. All questions carry equal marks.

Q. 1. (a) What is a translator? What is the difference between compiler and interpreter? What are various phases in compiler construction?

Ans. Translator : Once a programme has a piece of source code, he or she must convert it into machine code before the program can run on a computer. The job of converting source code into another program may be object code of translator.

"The system software that are used to translate the language from one language to another language are called the translators." The main translators are :

Assembler : An assembler is system software program used to convert low level language into the machine language.

Interpreter : This is also a system software program used to convert high level language into the machine language.

Preprocessor : A preprocessor is system software program used to convert high level language into another high level language.

Compiler : It is such a system software which converts the instruction given in high level language into machine language.

Difference between a Compiler and an Interpreter :

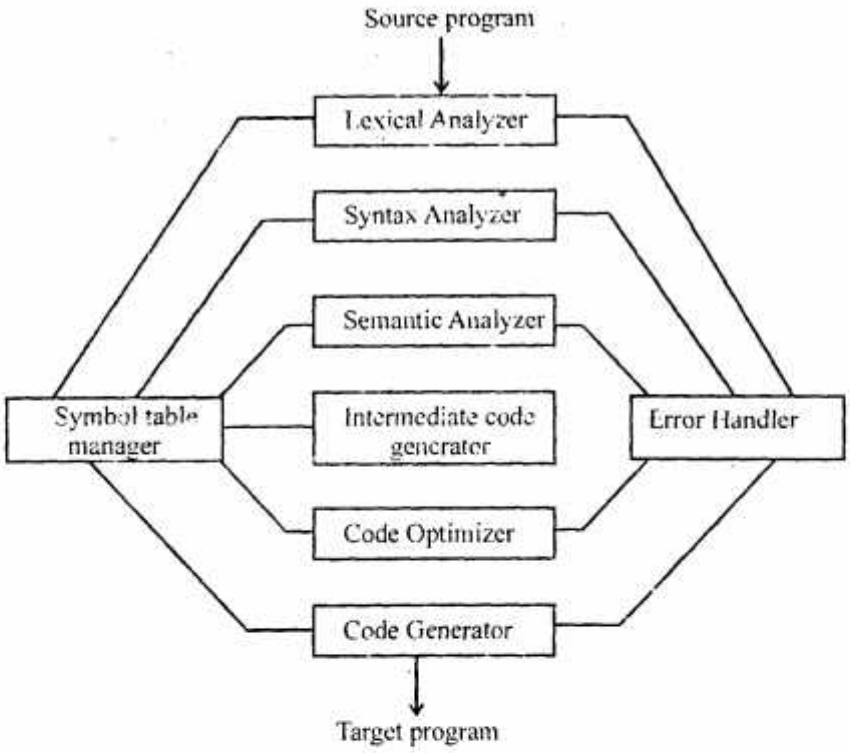
(i) In a compiler the complete source program is translated and then transferred to C.P.U. while in an interpreter each instruction is translated and then transferred to C.P.U. for execution successively. It means that each instruction is translated by the interpreter only after the previous instruction has been translated and received by C.P.U. for execution.

(ii) The object program obtained from the compiler is saved permanently for further use in future. This is not possible in case of interpreter because the complete source program is not converted into object program simultaneously.

(iii) Interpreter takes more time in the process of translation than a compiler.

(iv) It is easy to write an interpreter program which also occupies less space in memory of the computer.

The Phases of A Compiler : A compiler operates in phases, each of which transform the source program from one representation to another.



Phases of Compiler

Phase 1 : Lexical Analyzer : The lexical analyzer phase takes source program as an input and separates characters of source language into groups that are logically together. These groups are known as tokens.

Phase 2 : Syntax Analyzer : The syntax analyzer phase is also known as parsing phase. It takes tokens as input from lexical analyzer phase. The output of this phase is parse tree.

Phase 3 : Semantic Analysis : The semantic analysis phase checks the source program for semantic error and gather type information for subsequent code generation phase.

Phase 4 : Intermediate Code Generation : The next phase of compiler is intermediate code generation. It takes parse tree as an input from semantic phase & generates intermediate code. Generally a three address code is generated.

Phase 5 : Code Optimization : It is an optional phase designed to improve the intermediate code so that the ultimate object program run faster and/or takes loss space.

Phase 6 : Code Generation : It is the final phase for compiler. It generates the assembly code as target language. In this phase, the address in the binary code is translated from logical address.

Symbol Table : A symbol is a data structure containing record for each identifier, with field for the attributes of the identifier allow us to find record for each identifier quickly & to store or retrieve data from that record quickly.

Error Handler : The error handler is invoked when a flow in the source program is detected.

Q. 1. (b) Define deterministic finite state automata (DFA). Write an algorithm to simulate DFA.

Ans. Deterministic Finite Automata (DFA) : A deterministic finite automata (DFA) is a special case of a non-deterministic finite automaton in which :

(i) No state has on ε – transition, i.e., a transition on input ε and

(ii) For each state S and input symbol a, there is almost an edge labelled a leaving S.

A deterministic finite automaton has at most one transition from each state on any input. If we are using a transition table to represent the transition function of a DFA, then each entry in the transition table is a single state. As a consequence, it is very easy to determine whether a deterministic finite automaton accepts on input string, since there is at most one path from the start state labelled by that string.

Mathematically a DFA is given by

$$M = (\theta, \Sigma, \delta, q_0, F)$$

(i) θ is a finite non-empty set of state.

(ii) Σ is a finite non-empty set of input symbols.

(iii) δ is a transition system and $\delta \in \theta \times \Sigma \to \theta$.

(iv) q_0 is an initial state and $q_0 \in \theta$.

(v) F is a set of accepting states (or final states) & $F \subseteq \theta$.

Algorithm of Simulating a DFA : The following algorithm shows how to simulate the behaviour of a DFA on an input string.

Input : An input string x terminated by an end of file character eof. A DFA D with start state S_0 and set of accepting states F.

Output : The answer "yes" if D accepts x; "no" otherwise. The function move (S, C) gives the state to which there is a transition from state S on input character C. The function next char returns the next character of the input string x.

```
S:= S0 ;
C:= next char;
while C ≠ eof do
        S := move (S, C);
        C := next char;
end;
if S is in F then
        return "yes"
else return "no";
```

Q. 2. (a) What is the role of lexical analyzer in compilation process? What are lexemes and tokens? Define regular expression and tell what the use of them is.

Ans. The Role of Lexical Analyzer : The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser user for syntax analysis. This interaction is commonly implemented by making the lexical analyzer be a subroutine or a coroutine of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Since lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such tab task is stripping out from the source program comments and white space in the form of blank, tab and newline characters. Another is correlating error messages from the compiler with the source program.

Sometimes, lexical analyzers are divided into a cascade of two phases, the first called "scanning" and the second "lexical analysis." The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Tokens : There is a set of string in input for which the same token is produced as output. We treat tokens as terminal symbols in the grammar for the source language, using boldface names to represent tokens.

In most programming language, the following constructs are treated as tokens : keywords, operators, identifiers, constants, literal strings and punctuation symbols such as parenthesis, commas and semicolons.

For example in the Pascal Statement :

Const pi = 3.1416;

The substring pi is a lexeme for the token "identifier." In the example above, when the character sequence pi appears in the source program, a token representing an identifier is returned to the parser. The returning of a token is often implement by passing an integer corresponding to the token.

*** Lexeme :** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated as a lexical unit.

Certain language conventions impact the difficulty of lexical analysis. Language such as Fortran require certain constructs in fixed positions on the input line.

Thus, the alignment of a lexeme may be important in determining the correctness of a source program

Regular Expressions : Regular expressions are useful for representing certain sets of strings in an algebraic fashion. In real terms these define the language accepted by finite automaton.

We have the following definitions for regular expression :

(i) Any terminal symbol in Σ, including ε and ϕ are regular expressions.

(ii) Union of two regular expressions R_1 and R_2, written $R_1 + R_2$, is also a regular expression

(iii) The concatenation of two regular expressions R_1 and R_2 written as R_1R_2 is also a regular expression.

(iv) The closure (or iteration) of a regular expression R is written as R^* is also a regular expression.

(v) If R is a regular expression then (R) is also a regular expression.

(vi) The regular expression over Σ are precisely those obtained by recursively applying rules 1–S once or several time.

(vii) Nothing else is a regular expression.

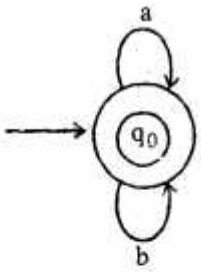
Regular expressions are used by many text editors and utilities (like in unix operating system) to search a block of text for certain patterns. e.g., to replace the found strings with some other strings. So regular expressions are useful for representing certain sets of strings in an algebraic fashion.

Q. 2. (b) Construct the transition diagram for the following regular expressions :

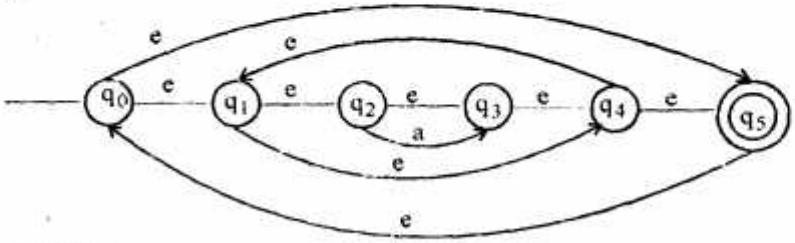
(i) $(a^*|b^*)^*$ **(ii)** $((\varepsilon|a)b^*)^*$

(iii) $(a|b)^*abb(a|b)^*$

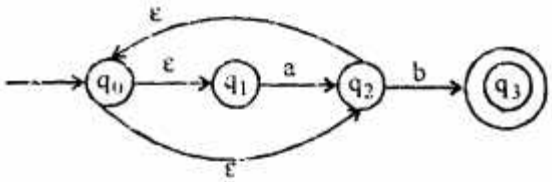
Ans. (i) $(a^*|b^*)^*$:



(ii) $((\varepsilon|a)b^*)^*$:



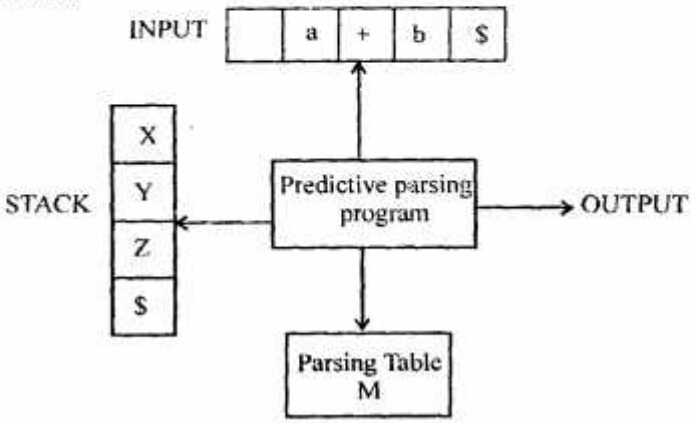
(iii) $(a|b)^*abb(a|b)^*$:



Q. 3. (a) Explain architecture and algorithm for the Non-Recursive Predictive Parser.

Ans. Non-Recursive Predictive Parser : This is a recursive descent parser which is implemented using stack instead of recursive calls.

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



Model of a Non-Recursive Predictive Parser

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

The parser is controlled by a program that behaves as follows. The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities :

(i) If X = a = $, the parser halts and announces successful completion of parsing.

(ii) If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.

(iii) If X is a non-terminal, the program consults entry $\mu[X,a]$ of the parsing table M. This entry will be either an X-production of the grammar or an error entry.

The behaviour of the parser can be described in terms of its configurations which give the stack contents and the remaining input.

Algorithm : Non-recursive predictive parsing.

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

Output : If W is in L[G], a leftmost derivation of W; otherwise, an error indication.

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

```
Set ip to point to the first symbol of W$;
repeat
let X be the top stack symbol and a the symbol pointed to by ip;
if X is a terminal or $ then
        if X = a then
                pop X from the stack and advance ip
        else error( )
else /*X is a non-terminal */
if M[X, a] = X → Y1 Y2 ..... YK . He begin pop X from the stack;
        push YK , YK-1 ..... , Y1 onto the stack, with Y1 on top;
        output the production X → Y1Y2 ..... YK
        end
else error( )
until X = $              /* stack is empty */
```

Q. 3. (b) Explain algorithm for the operator precedence parser.

Ans. Operator precedence parsing technique was first described as a manipulation on tokens without any reference to an underlying grammar. Once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using non-terminals on the stack only as placeholders for attributes associated with the non-terminals.

In operator-precedence parsing, we define three disjoint precedence relations, <, = and >, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings :

Relation	Meaning
a < b	a "yields precedence t " b
a = b	a "has the same precedence as" b
a > b	a "takes precedence over" b

Algorithm of Operator Precedence Parsing :

Input : An input string W and a table of precedence relations.

Output : If W is well formed, a skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes : otherwise, an error indication.

Method : Initially, the stack contain $ and the input buffer the string W$.

(i) set ip to point to the first symbol of W$.
(ii) repeat forever.
(iii) if $ is on top of the stack and ip points to $ then.
(iv) return
else begin
(v) let a be the top most terminal symbol on the stack and let b be the symbol pointed to by ip;
(vi) if a < b or a = b then begin
(vii) push b onto stack;
(viii) advance ip to the next input symbol;
end;
(ix) else if a > b then/* reduce */
(xi) repeat
(xi) pop the stack.
(xii) until the top stack terminal is related by < to the terminal most recently popped.
(xiii) else error ( )
end.

Q. 4. (a) Explain what are left recursion and left factoring and how to remove these problems. Why should a grammar be free from these?

Ans. Left Recursion : A grammar G (V, T, P, S) is said to be left recursive if it has a production in the form

$$A \rightarrow A\alpha / \beta$$

The above grammar is left recursive because the left of production is occurring at first position on the right side of production.

It is possible for a recursive-descent parser to loop forever. A problem arises with left recursive productions like.

$$expr \rightarrow expr + term$$

In which the leftmost symbol on the right side is same as the non-terminal on the leftside of the production.

Left Factoring : Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

A grammar G is said to be left factored if any production of it is in the form of

$$A \rightarrow \alpha\beta / \alpha\gamma$$

i.e.. on the right side of production initially α is present as first symbol in both.

The basic idea is that when it is not clear which of two alternative productions to use to expand, a non-terminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

For example, if we have the two productions

stmt → if expr then stmt else stmt

/ if expr then stmt

On seeing the input token if, we cannot immediately tell which production to choose to expand stmt. In general, if $A \rightarrow \alpha\beta_1 / \alpha\beta_2$ are two A-productions and the input begins with a non-empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However we may defer the decision by expanding A to $\alpha A'$. Then after seeing the input derived from α, we expand A' to β_1 or to β_2. That's, left-factored, the original productions become,

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 / \beta_2$$

Elimination of Left Recursion : We can eliminate left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A'/\varepsilon$$

The general form for left recursion

$$A \rightarrow A\alpha_1|A\alpha_2|.....|A\alpha_m|\beta_1|\beta_2|...../\beta_n$$

will be $A \rightarrow \beta_1 A'|\beta_2 A'|..... |\beta_n A'$ and $A' \rightarrow \alpha_1 A'|\alpha_2 A'|.....\alpha_m A'/\varepsilon$

Algorithm : Eliminating left recursion.

Input : Grammar G with no cycles or ε productions.

Output : An equivalent grammea with no left recursion.

Method :

(i) Arrange the non-terminals in same order $A_1, A_2, ..., A_n$.

(ii) for i : = 1 to n do begin

for j : = 1 to i–j do begin

replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions

$A_i \rightarrow \delta_1\gamma / \delta_2\gamma/...../\delta_R\gamma$. Where $A_j \rightarrow \delta_1|\delta_2|.....|\delta_k$

are all the current A_j-productions;

end

eliminate the immediate left recursion among the

$A_i \rightarrow$ productions

end.

Elimination of Left Factoring : The left factoring can be removed by expanding $A \rightarrow \alpha A'$. Then after seeing the input derived from α, we expand A' to β or to γ. Hence, the production will be

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta / \gamma$$

Algorithm : Elimination of left factoring.

Input : Grammar G.

Output : An equivalent grammar with no left factoring.

Method : For each non-terminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i e, there is a non-trivial common prefix, replace all the & productions $A \rightarrow \alpha\beta_1|\alpha\beta_2|.....|\alpha\beta_n|\gamma$ where γ represents all alternatives that do not begin with α by,

$$A \rightarrow \alpha A'/\gamma$$

$$A' \rightarrow \beta_1|\beta_2|.....|\beta_n$$

Here A' is a new non-terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

Top-down parsing methods cannot handle left-recursive grammars as well as left factoring grammar, so a transformation that eliminates them is required.

Q. 4. (b) Explain the working and algorithm of LR parser.

Ans. LR Parser : An efficient, bottom up syntax analysis technique that can be used to parse a large class of context free grammars. The technique is called LR(K) parsing; the "L" is for left-to-right scanning of the input, the "R" for construction a rightmost derivation in reverse and the K for the number of input symbols of lookahead that are used in making parsing decisions. When (K) is omitted, K is assumed to be 1. LR parsing is attractive for a variety of reasons :

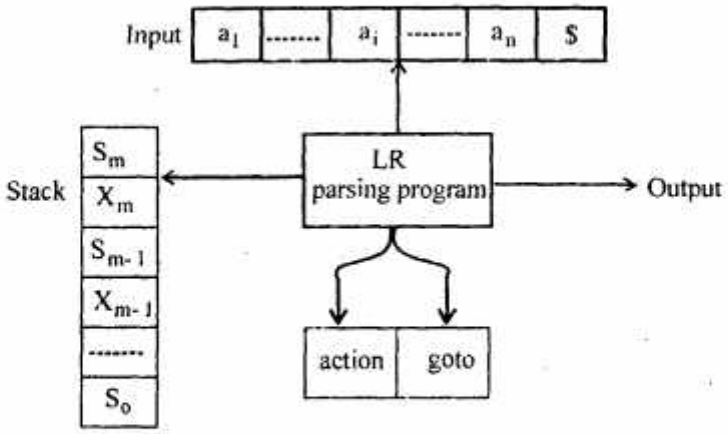
(i) LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.

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

(iii) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

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

The schematic form of an LR parser is shown in fig.



Model of an LR Parser

It consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $S_0X_1S_1X_2S_2.....X_mS_m$, where S_m is on top. Each X_i is a grammar symbol and each S_i is a symbol called a state.

The parsing table consists of two parts, a parsing action function and "action" and a goto function "goto".

All LR parsers behave in the fashion given in algorithm; the only difference between LR parsed and another is the information in the parsing action and goto fields of the parsing table.

Algorithm : LR parsing algorithm

Input : An input string W and an LR parsing table with function action & goto for a grammar G.

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

Method : Initially the parser has S_0 on its stack, where S_0 is the initial state and W$ is to input buffer. The parser than executes the program until an accept or error action is encountered.

(i) set ip to point to the first symbol of W$;
(ii) repeat forever begin.
(iii) let S be the state on top of the stack and
(iv) a the symbol pointed to by ip;
(v) if action [S_1 a] = Shift S' then begin
(vi) push a then S' on top of the stack;
(vii) advance ip to the next input symbol.
(viii) end
(ix) Else if action [S, a] = reduce $A \rightarrow \beta$ then begin

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

(xi) let S' be the state now on top of the stack;

(xii) push A then goto [S', A] on top of the stack;

(xiii) output the production $A \rightarrow \beta$

(xiv) end

(xv) else if action [S, a] = accept then

(xvi) return

(xvii) else error ( )

(xviii) end

Q. 5. Construct the LR (0) parsing table for the following grammar :

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

Is this grammar a LR(0) grammar.

Ans. The augmented grammar for above grammar is

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

The canonical collection of sets of LR(0) items for grammar as follows :

I_0: $S' = S$

$S \rightarrow \bullet R = R$

$S \rightarrow \bullet R$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet id$

$R \rightarrow \bullet L$

I_1: $S' \rightarrow S \bullet$

I_2: $S \rightarrow L \bullet = R$

$R \rightarrow L \bullet$

I_3: $S \rightarrow R \bullet$

I_4: $L \rightarrow *\bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet id$

I_5: $L \rightarrow id$

I_6: $S \rightarrow L = \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet id$

I_7: $L \rightarrow *R \bullet$

I_8: $R \rightarrow L \bullet$

I_9: $S \rightarrow L = R \bullet$

The follow are given below :

Follow (S) = {$}

Follow (L) = {=, $}

Follow (R) = {=, $}

The parsing table for above grammar is as given below :

State			Action			Goto	
	id	*	=	$	S	L	R
0	S_5	S_5			1	2	3
1				acc			
2			56 r_5	r_5			
3				r_2			
4	S_5	S_4				8	7
5			r_4	r_4			
6	S_5	S_4				8	9
7			r_3	r_3			
8			r_5	r_5			
9				r_1			

Multiple defined entry which shows a shift reduce conflict. This conflict arises from the fact that SLR parser construction method is not powerful enough to remember enough left context to decide what action parser should take on input.

Yes, this grammar is a LR(O) grammar.

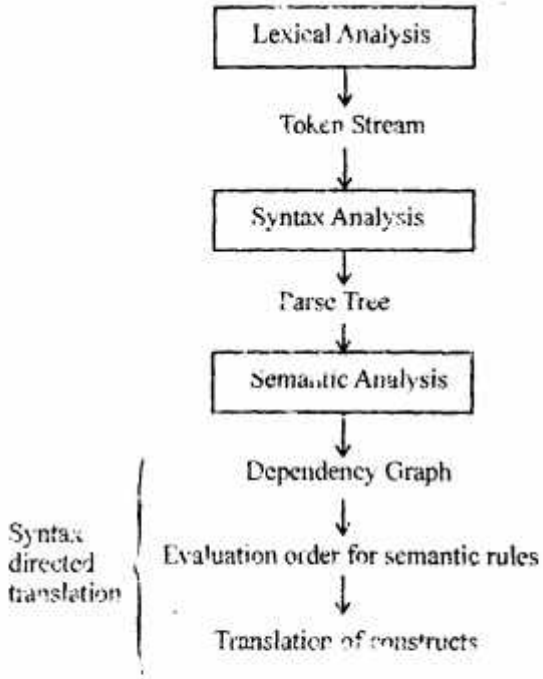
Q. 6. (a) What is syntax directed translation, why are they important?

Ans. Syntax Directed Translation : We know that every programming language contains the constructs. It is very important to know the rules of translations of the programming constructs Whenever a construct is encountered in programming language, it is evaluated/translated according to semantic rules defined in that particular programming language. The translation may be generation of intermediate code, object code or adding the information in symbol table about constructs type.

The modern compiler uses the syntax directed translation that makes user's life easy by hiding many implementation details and free the user from having to specify explicitly the order in which semantic rules are to be evaluated. The translation of token streams take place by evaluating the semantic rule.

There is a notational framework for intermediate code generation that is the extension of context free grammars. This framework is called syntax directed translation. It allows subroutines or semantic action to be attached to the productions of a context free grammar. These subroutines generates intermediate code when called at appropriate times by a parser for that grammar.

The basic structures for syntax-directed translation is given below :



The syntax directed translation is partitioned into two subset called the synthesized and inherited.

(i) Synthesised Translation : It defines the value of translation of the non-terminal on the left side of the production as a function of the translation of non-terminals on the right side i.e., left is dependent on right.

$$E \to E^{(1)} + E^{(2)} \qquad \{E.val := E^{(1)}.val + E^{(2)}.val\}$$

(ii) Inherited Translation : Translation of a non-terminal on the right side of the production is defined in terms of a translation of the non-terminal on the left i.e., right is dependent on left.

$A \rightarrow XYZ$ $[Y.val := q * A.val]$

Y which is on right side is dependent on A, which is on left.

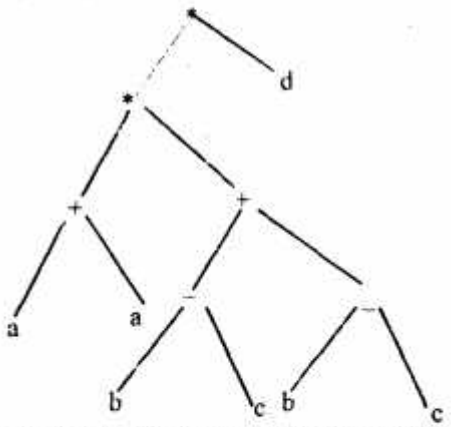
The syntax directed translation is important because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language. This allows subroutines or 'semantic actions' to be attached with context free grammar. This enable the compiler to generate intermediate code on the fly with the syntactic structure.

Q. 6. (b) Create the syntax tree and DAG for the following expression,

$$a + a * (b - c) + (b - c) * d.$$

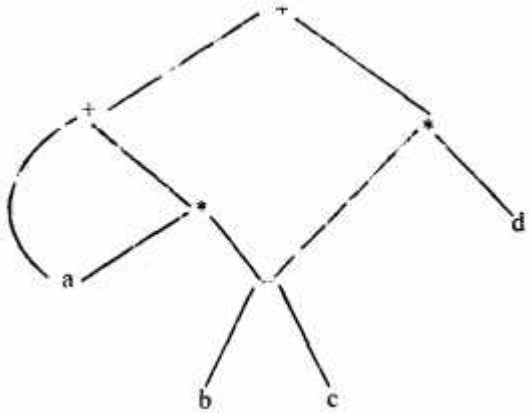
Ans. Syntax Tree :

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



DAG (Directed Acyclic Graphs) : A directed acyclic graph (DAG) for an expression identifies the common subexpressions in the expression. Like a syntax tree, a dag has a node for every subexpression of the expression; On interior node represent an operator and its children represent its operands. The difference is that a node in a dag representing a common subexpression has more than one "parent" in a syntax tree, the common subexpression would be represented as a duplicated subtree.

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



Dag for the expression $a+a*(b-c)+(b-c)*d$.

Q. 7. (a) What is a symbol table, what are various data structures used to implement the table?

Ans. Symbol Table : A symbol table is a data structure used by a compiler to keep track of scope, life and binding information about names. These names are used in the source program to identify the various program elements, like variables, constant, procedures, and the labels of statements. The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the tables as well as for adding new entries to the symbol table.

In any case, the symbol table is a useful abstraction to aid the compiler to ascertain and verify the semantics or meaning of a piece of code. It will keep track of the names, types, locations and properties of the symbols encountered in the program. The type system and the code generation system rely on the symbols encountered elsewhere in the code. It makes the compiler more efficient, since the file doesn't need to be reparsed to discover previously processed information. For efficiency, our choice of implementing data structure for the symbol table and the organization of its content should stress on minimal cost when adding new entries or accessing the information of existing entries. Also, if the symbol table can grow dynamically as needed, then it is more useful for a compiler.

For example, if defining a variable like

int X

then the name of the variable, along with its type is inserted in the symbol table.

X	int

We have following data structures available to construct symbol table :

(i) Linear lists

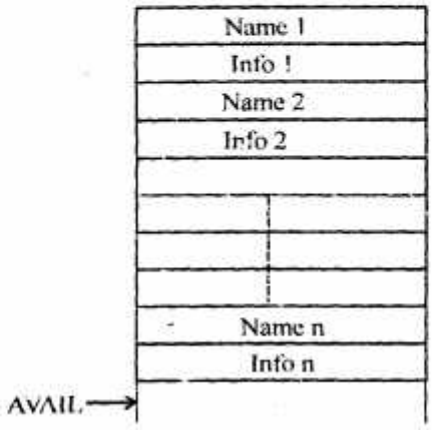
(ii) Trees

(iii) Hash tables.

Each has its own difficulty of implementation and efficiency. Linear lists are simple and easy to implement but slow due to lots of pointer adjustment needed for deletion, insertion. Trees are intermediate in power between linear lists and hash tables. Hash tables are quite fast in locating an element, but require more programming effort and space.

(i) Lists : This is simple and easy to implement data structure for symbol table. We can use single array or sequence of arrays.

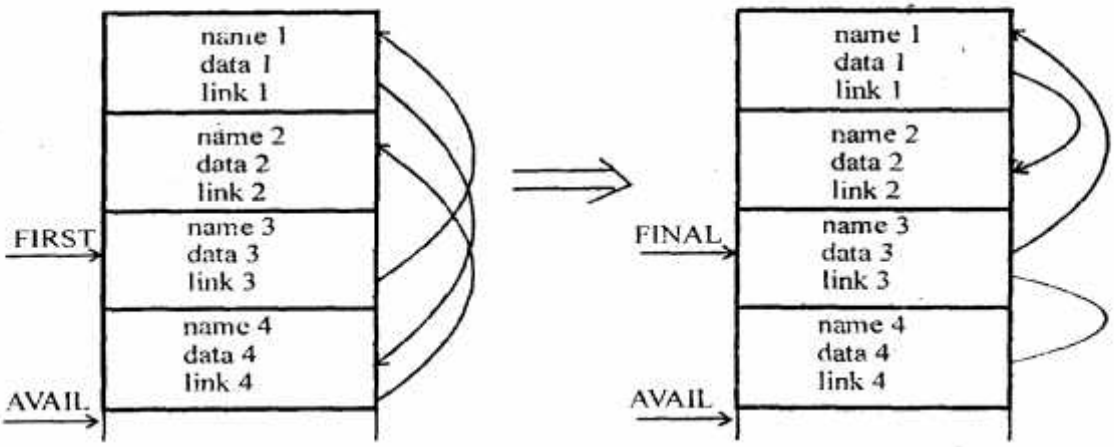
To insert a new name we scan down the list to search whether it is there or not. If it is not there new name in the words immediate following AVAIL is stored and AVAIL pointer is incremented to new space.



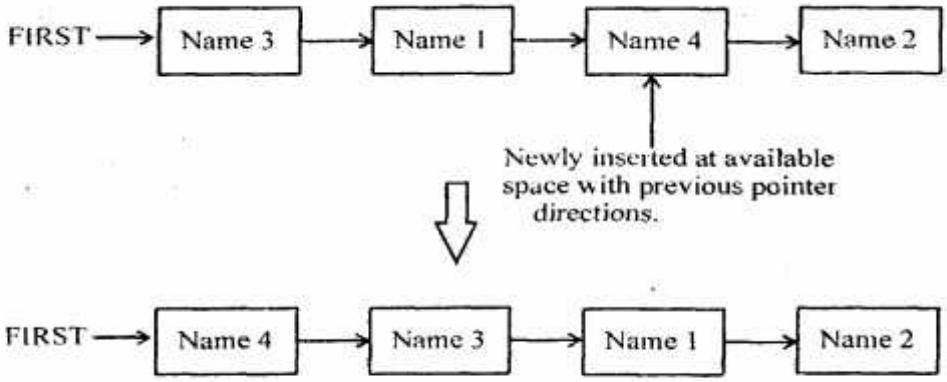
The efficiency is $O(n^2)$, it is the time taken to search and insert a record.

Self Organising Lists : At the cost of some extra space we can expedite the search process using LINK field. This field points to next record in sequence, whenever a new record is inserted, it is inserted at the beginning of list with little adjustment of pointers.

IRST represent first item of symbol table

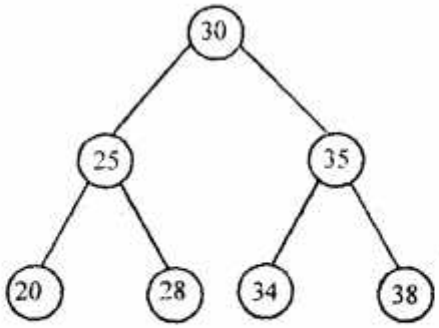


Order is : Name 3, Name1, Name 4, Name 2



Name 4 moves to first position

(ii) Search Trees : We have studied binary search trees in data structures in which point node is greater than the left child and smaller than right child. We know that binary tree has height $\log_n$, therefore, maximum time to search and insert a node is $O(\log_n)$.



BST Algorithm :

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

(iii) Hashing : This is the method of converting symbols, the indexes of entries into symbol table. e.g., 0 to n–1. The index is obtained by hashing the symbol i.e., performing some arithmetic or logical operations on symbol. In hashing the search, insert or delete operation takes $\theta(n)$ time if the data are searched linearly and there are n items stored in records. Hashing is a technique which directly refers to the memory location where the data is stored and data can be found in $0(1)$ time, this is tremendous improvement over other data structure of symbol table. The functions which calculatethe position of records is called hash function and records are stored in hash tables. There is a key associated with each record.

1	100
2	200
3	300
4	400
5	500
10	1000

Suppose we want to find the key value 400. We can directly refer to it using some hash functions that calculates its location.

$$h(400) = 400 \text{ DIV } 100 = 4$$

Thus, data is located at 4th position. This takes $\theta(1)$ time to find the key value.

Q. 7. (b) Explain various target for the code optimization.

Ans. The process of optimization is aimed at improving the execution efficiency of a program. Optimization aims mainly at rearranging the computation in a program so as to gain the advantage of execution speed without changing the meaning of a program. So finally we can say that code optimization refers to the technique used by the compiler to improve the execution efficiency of the generated object code.

The basic work of compiler is to take source code as input & produce object code/target code as output.

The code generated by intermediate code generator may not be efficient as it should; for example, it is using temporary variables. Therefore it is the responsibility of code optimization phase to generate the code which should be efficient on machine. The efficiency is defined in terms of time & space taken by a computer program to produce the desired output.

The code optimization refer to the techniques used by the compiler to improve the execution efficiency of generated object code. It involves a complex analysis of intermediate code & performs various transformations but every optimizing transformation must also preserve the semantic of the program. That is, a compiler should not attempt any optimization that would load to a change in program's semantics.

Optimization can be machine dependent or machine independent. The machine independent optimizations can be performed independently of the target machine for which the compiler is generating the code i.e., the optimization are not tied to the target machine's specific platform or language.

Examples of machine independent optimization are :

(i) Elimination of loop invariant computation

(ii) Induction variable elimination

(iii) Elimination of common sub-expression.

On the other hand, machine-dependent optimization require knowledge of target machine. An attempt to generate object code that will utilize the target machine registers more efficiently is an example of machine dependent code optimization.

Actually, code optimization is misnames (i.e., not actually as the name says), even after performing various optimizing transformations, there is no guarantee that the generated code will be optimal. Hence, we are actually performing code improvement. When attempting any optimizing transformations, the following criteria should be applied :

(i) The optimization should capture most of the potential improvements without an unreasonable amount of efforts.

(ii) The optimization should be such that the meaning of source program is preserved.

(iii) The optimization should, on average, reduce the time and space expended by the object code.

The code optimization is divided into two parts :

(i) Loop optimization

(ii) Data Flow Analysis

(i) Loop Optimization : Looping plays an important role in any programming language. Loop plays an important role in the optimization because loop is a place where compiler spend bulk of its time. The running time of a program may be improved if we decreases the number of instructions in the loop. There are three important techniques for loop optimization. They are :

(i) Code motion

(ii) Induction-variable elimination

(iii) Reduction in strength.

(if) Data Flow Analysis : To optimize the code efficiently compiler collects the information about the programs as whole and distribute this information to each block of the flow graph. This process is known as data flow analysis. To get the data flow information we define certain data flow equations and solution of those equation gives the data-flow information.

Data-flow-equations;

A typical equation is,

$$Out[B] = (In[B] - kill[B] \cup gen[B] \quad ...(i)$$

where : B : is basic block.

gen[B] . The set of all the definitions generated in block B.

Kill[B] : The set of all definitions outside block B that define the same variable as are defined in block B.

$$int[B] : \cup out[P], \text{ where P is predecessor of B} \quad ...(ii)$$

Equation (i) & (ii) are data flow equations.

Q. 8. (a) List the various types of intermediate code, explain 3 address code in detail.

Ans. Intermediate Code : Intermediate codes are machine independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language the intermediate code generator.

Intermediate language can be many different language and the designer of the compiler decides this intermediate language.

(i) Syntax trees can be used as an intermediate language.

(ii) Postfix notation can be used as an intermediate language

(iii) Three-address code can be used as an intermediate language.

(i) Syntax-Trees : A syntax tree specifies the translation of a construct in terms of attributes associated with its syntactic components. Syntax tree are used to specify many of its translation that takes place in the front and of a compiler.

(li) Postfix Notation : The postfix notation for an expression E can be defined inductively as follows :

(i) If E is a variable or constant, then the postfix notation for E is E itself.

(ii) If E is an expression of the form E_1 & E_2 ; where op is any binary operator, then the postfix notation for E is E_1 E_2 op, where E_1' and E_2' are the postfix notations for E_1 and E_2 respectively.

(iii) If E is an expression of the form (E_1), then the postfix notation for E_1 is also postfix notation for E.

Three-Address Code : Three-address code is a sequence of statement of the general form

$$X := Y \text{ op } Z$$

Where x, y & z are names, constants or compiler-generated temporaries; op stands for any operator, such as a fixed-or floating-point arithmetic operator or a logical operator on boolean-valued data. Note that no built up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus, a source language expression like x+y*z might be translated into a sequence.

$$t_1 := y * z$$

$t_2 := x + t_1$

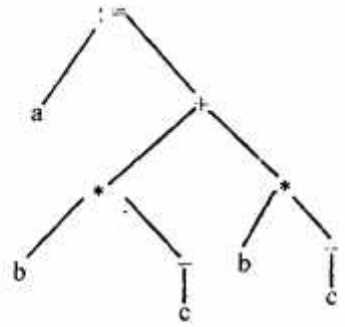
Where t_1 and t_2 are compiler-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements make three-address code desirable for target code generation and optimization.

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.

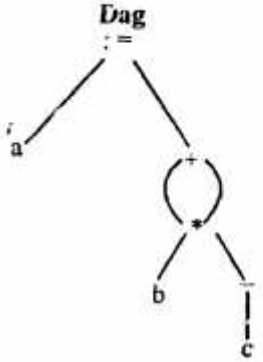
The reason for the term "three-address code" is that each statement usually contains three addresses. two for the operands and one for the result. In the implementations of three-addresses code, a programmer defined name is replaced by a pointer to a symbol-table entry for that name

$a := b * -c + b * -c$

Syntax Tree :



Dag



$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a : t_5$

Code for the syntax tree

$t_1 := -c$

$t_2 := b * t$

$t_5 := t_2 + t_2$

$a := t_5$

Code for the dag

Q. 8. (b) What are basic blocks and flow diagram, explain PEEPHOLE optimization technique.

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

Or we can say A basic blocks is a sequence of statement that enters at the start and ends with a branch at the end.

The following sequence of three address statements forms a basic block :

$t_1 := a * a$

$t_2 := a * b$

$t_3 := 2 * t_2$

$t_4 := t_1 + t_3$

$t_5 := b * b$

$t_6 := t_4 + t_5$

A basic block computes a set of expressions. There expressions are the values of the names live on exit from the block. Two basic blocks are said to be equivalent if they compute the same set of expressions.

Flow Diagram : It is used to portray the basic blocks and their successors by a directed graph. The nodes of the flow diagram or flow graph are the basic blocks. One node is distinguished as initial : it is the block, whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if.

(i) Here is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or.

(ii) B_2 immediately follows B_1 in the order of the program and B_1 does not end in an unconditional jump.

We say that B_1 is a predecessor of B_2 and B_2 is a successor of B_1.

Peephole Optimization Technique : A statement-by-statement code generation strategy often produces target code that contains redundant instructions and suboptimal constructs.

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instruction (called the peephole) and replacing these instruction by a shorter or faster sequence, whenever possible. This technique can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, moving window on the target program. The code in the peephole feed not be contiguous, although some implementations do require this.

It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit.

Following are the examples of program transformation that are characteristic of peephole optimizations :

(i) redundant-instruction elimination

(ii) flow-of-control optimizations

(iii) algebraic simplifications

(iv) use of machine idioms.

(i) Redundant Instruction Elimination : One of the opportunity for peephole optimization is the removal of unreachable instruction. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example : For debugging purpose, a large program may have within it certain segments that are executed only if a variable debug is 1.

(ii) Flow-of-Control Optimization : The intermediate code generation algorithm frequently produce jumps to jumps, jumps to conditional jumps or conditional jumps to jumps. These unnecessary jumps can be elimi-

nated in either the intermediate code or the target code by the peephole optimization.

(iii) Algebraic Simplification : There is no end to the amount of algebraic simplification that ca. be attempted through peephole optimization. However, only a few algebraic identities occur frequently enough that it is worth considering implementing them.

For example, statement such as,

$$X := X + 0$$

Or

$$X := X * 1$$

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

(iv) Use of Machine Idioms : The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permits the use of these instructions can reduce execution time significantly.