END TERM EXAMINATION (MAY 2018)

COMPILER DESIGN

ETCS-302

Faculty Name: Mrs. Garima	**Student name:** Tarun Aggarwal
	Semester: 6th, CSE
	Group: 6C13
	Roll No.: 41396402717

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

MAIT

उद्यमेन हि सिध्यन्ति
कार्याणि न मनोरथैः

Maharaja Agrasen Institute of Technology, PSP Area, Sector – 22, Rohini, New Delhi – 110085

TARUN AGGARWAL
41396402717

Q.1.Define the following terms and explain with example.
(a) Lexeme, Token and Pattern

- **Lexeme:**

A lexeme is an arrangement of alphanumeric characters in a token. Lexemes are a piece of the information stream from which tokens are identified. An invalid or an illegal token delivers a mistake. A lexeme is one of the structure squares of language. It is identified by the lexical analyser as an example of that token. These are arrangements of characters coordinated by pattern shaping the token.

- **Token:**

A token is a pair comprising of a token name and a optional attribute value. The token name is a abstract symbol representing to a sort of lexical unit, for example a specific keyword, or grouping of info characters meaning an identifier. The token names are the information symbols that the parser processes.

- **Pattern:**

The set of rules that define a token. A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

(b) Activation Record
An activation record is another name for Stack Frame. It's the data structure that makes a call stack. A general initiation record comprises of the accompanying things:

- Local variables: hold the information that is local to the execution of the system.
- Temporary values: stores the values that emerge in the assessment of an articulation.
- Machine status: holds the data about the status of machine not long before the function call.
- Access link(optional): alludes to non-neighbourhood information held 1n other initiation records.
- Control link(optional): focuses to initiation record of guest.
- Return value: utilized by the called strategy to restore an incentive to calling system
- Actual parameters

(c) Back Patching
Bootstrapping is simply the procedure for creating a gathering compiler that is accumulated and written in the source programming. language that it expects to gather. Bootstrapping a compiler has the accompanying favourable circumstances:

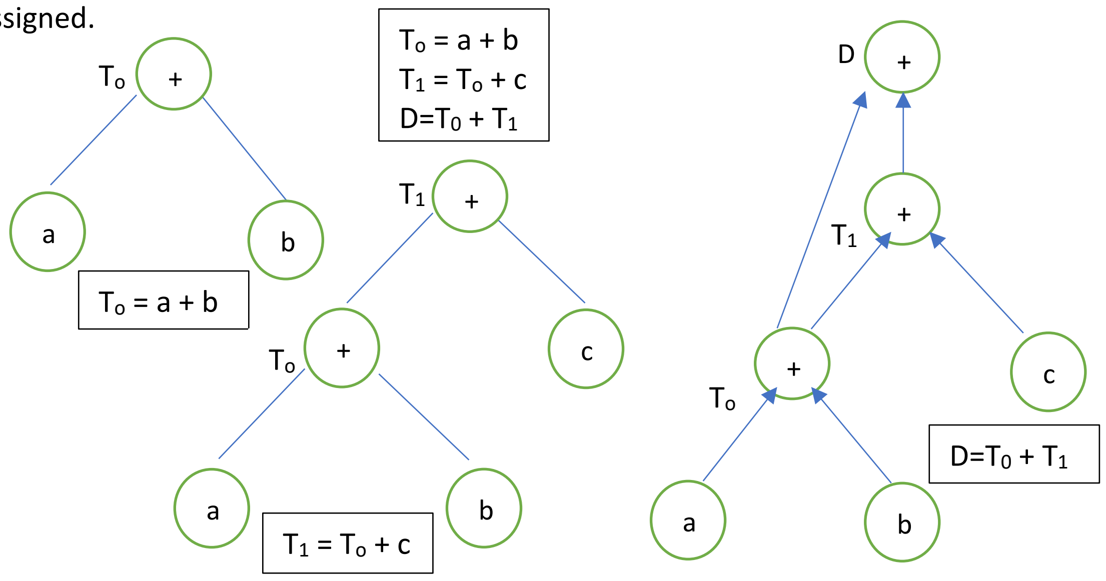
- It is a non-trivial test of the language being compiled.
- Compiler engineers and bug announcing some portion of the network just need to realize the language being incorporated.
- Compiler improvement should be possible in the more significant level language being accumulated.
- Improvements to the compiler's backend improve universally useful projects as well as the compiler itself.

- It's a far reaching consistency check as it ought to have the option to recreate its own item code.

(d) Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the basic blocks, and offers optimization too.

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators
- Interior nodes also represent the results where the values are to be stored or assigned.



(e) Left Recursion

Left recursion: Types of recursion:

- Left Recursion
- Right Recursion
- General Recursion

A production of grammar is said to have left recursion if the leftmost variable of its RHS is the same as the variable of its LHS. A grammar containing a production having left recursion is called a Left Recursive Grammar.
For example: S -> Sa/ϵ
First eliminate Left Recursion -
If we have the left-recursive pair of productions: A -> Aα/β
where B does not begin with an A.
Then, we can eliminate left recursion by replacing the. pair of productions with

A -> β/A'
A -> αA'/ϵ

Thus, left recursion is eliminated by converting the grammar into a right recursive grammar having right recursion. This right recursive grammar functions same as left recursive grammar.

(f) Shift/Reduce and Reduce/Reduce Conflicts
The Shift-Reduce Conflict is the most common type of conflict found in grammars. It is caused when the grammar allows a rule to be reduced for a particular token, but. at the same time, allowing another rule to be shifted for that same token. As a result, the grammar is ambiguous, so a program can be interpreted more than one way. This error is often caused by recursive grammar definitions where the system cannot determine when one rule is complete and another is just started.

Reduce/Reduce Conflict is a categorised as when a grammar allows two or more different rules to be reduced at the same time, for the same token. When this happens, the grammar becomes ambiguous since a program can be interpreted more than one way. This error can be caused when the same rule is reached by more than one path.

(g) Back patching
The syntax directed definition can be implemented in two or more passes when we have both synthesized attributes and inherited attributes. Build the tree first and then Walk the tree in the depth-first order.

The main difficulty with code generation in one pass is that we may not know the target of a branch when we generate code for flow of control statements.

Backpatching is the technique to get around this problem. Generate branch instructions with empty targets When the target is known, fill in the label of the branch instructions. Backpatching is the process of leaving blank entries for the go-to instruction where the target address is unknown in the forward transfer in the first pass and filling
these are unknown in the second pass.

(h) Handle pruning
Handle pruning is the general approach used in shift-and-reduce parsing variables of a handle is a substring that matches the body of a production. Handle reduction is a step in the reverse of rightmost derivation in reverse can be obtained by handle pruning. The implementation of handle pruning involves the following data-structures a stack - to hold the grammar symbols; an input buffer that contains the remaining input and a table to decide handles.
The implementation of handle pruning involves the following data structures:

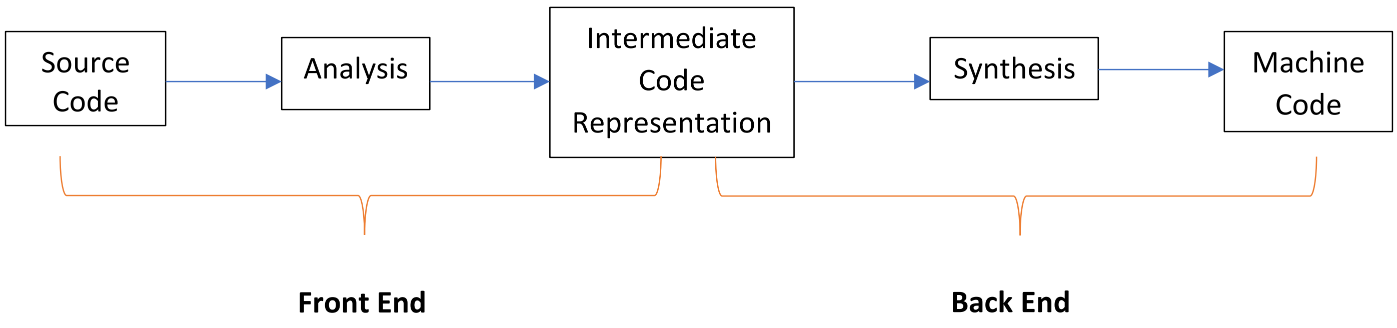
- a stack - to hold the grammar symbols,
- an input buffer that contains the remaining inputs
- a table to decide handles.

Q2. (a) What do you mean by the front end and back end of a compiler?
The phases of a compiler are front end and back. end.

- **Front End Phase:**

The front end incorporates all analysis stages and the middle of the code generator that is the reason it is otherwise called Analysis phase; The front end breaks down the source program and creates intermediate code. The analysis phase of the compiler peruses the

source program, separates it into care parts and afterward checks for lexical, sentence structure and linguistic structure errors. The analysis
stage creates an intermediate of the road portrayal of the source program and symbol table which ought to be taken care of to the Synthesis phase as input.



- **Back End Phase:**

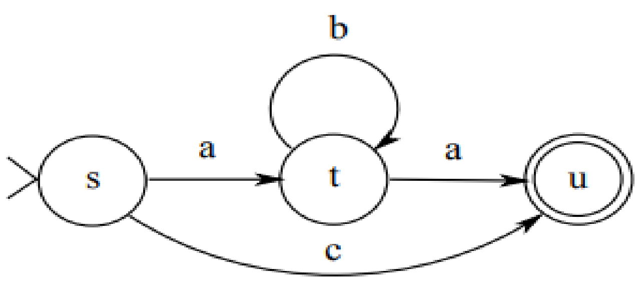
The back end incorporates the code enhancement stage and final code age stage.

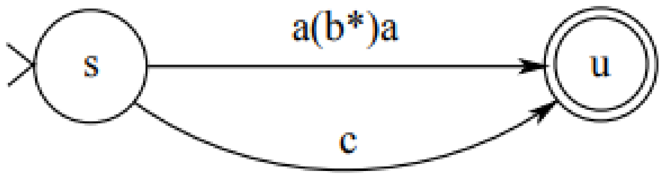
The back end incorporates the objective program from the intermediate code. This backend stage is otherwise called synthesis phase. The synthesis phase produces the objective program with the assistance of intermediate source code portrayal and symbol table. A compiler can have numerous phases and passes.

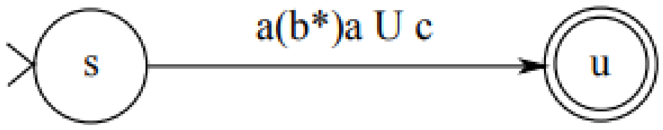
- **Pass:** A pass alludes to the traversal of a compiler through the whole Program.
- **Phase:** A period of a compiler is a recognizable stage, which takes contribution from the previous stage, processes and yields output that can be utilized as contribution for the following stage. A pass can have more than one phase.

Q 2 (b) Construct a nondeterministic finite automaton (NFA) and then deterministic finite automaton (DFA) for the regular expression ((a/b)c*)*

((a/b) c*)* => ((a + b) 0*) *
((a+b) 0*)* *10







Q 2. (c) Explain Chomsky hierarchy of grammars.

According to Chomsky hierarchy, grammars are divided of 4 types:

- **Type 0 known as unrestricted grammar:**

Type- 0 grammars include all formal grammars. Type 0 grammar languages are recognized by turing machine. These languages are also known as the recursively enumerable languages.

- **Type 1 known as context sensitive grammar:**

Type- 1 grammars generate the context-sensitive languages. The language generated by the grammar is recognized by the Linear Bounded Automata.
In Type 1 grammar:

1. First of all Type 1 grammar should be Type 0.
2. Grammar Production in the form of

$$\alpha —> \beta$$

count of symbol 1n 0: is less than or equal to B

- **Type 2 known as context free grammar:**

Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Non-Deterministic Push down Automata. Type-2 grammars generate the context-free languages.
In Type 2 grammar:

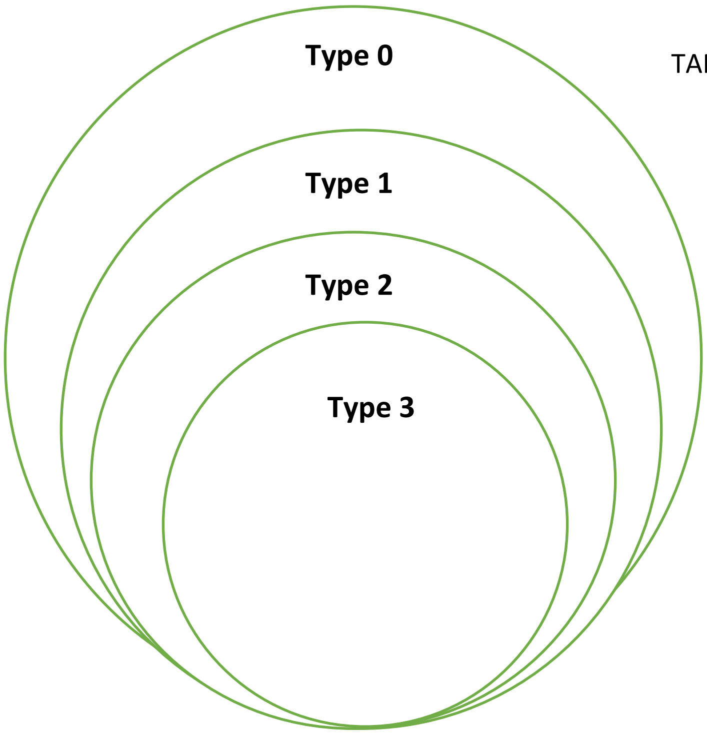
1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.

- **Type 3 known as Regular Grammar:** '

Type-3 grammars generate the regular languages. These languages are exactly all languages that can be decided by a finite state automaton. Type 3 is most restricted form of grammar Type 3 should be in the given form only:

1. V→VT*/T*
2. V→T*V/T*

TARUN AGGARWAL
41396402717



Q. 3. (a) Show that the given Grammar' is LL(1) but not SLR(1) .
S -> aAaAb I BbBa
A -> e
B -> e
To show that it Is not SLR(1) but LL(1) , consider the LR(0) items in the initial set for the augmented grammar

FIRST(S) = a, b
FIRST(A) = ε
FIRST(B) = ε
FOLLOW(A) = a, b
FOLLOW(B) = a, b

And then, by definition an LL(1) grammar has to:

- If A⇒a and A⇒b are two different rules of the grammar, then it should that FIRST (a) ∩ FIRST (b) = ∅. Hence, the two sets haven't any common element.

- If for any non-terminal symbol AA you have A⇒∗ε, then it should be that FIRST (A) ∩ FOLLOW (A) =∅. Hence, if there is a zero production for a non-terminal symbol, then the FIRST and FOLLOW sets can't have any common element.

So, for the given grammar:

- We,have FIRST(AaAb)∩FIRST(BbBa)=∅FIRST(AaAb)∩FIRST(BbBa)=∅ since FIRST(AaAb)={a} while FIRST(BbBa)={b}FIRST(BbBa)={b} and they don't have any common elements.

- FIRST(A)∩FOLLOWA)=∅ since FIRST(A)={a,b}FIRST(A)={a,b} while FOLLOW(A)=∅, and now FIRST(B)∩FOLLOW(B)=∅ since FIRST(B)={ε}while FOLLOW(B)={a,b}.

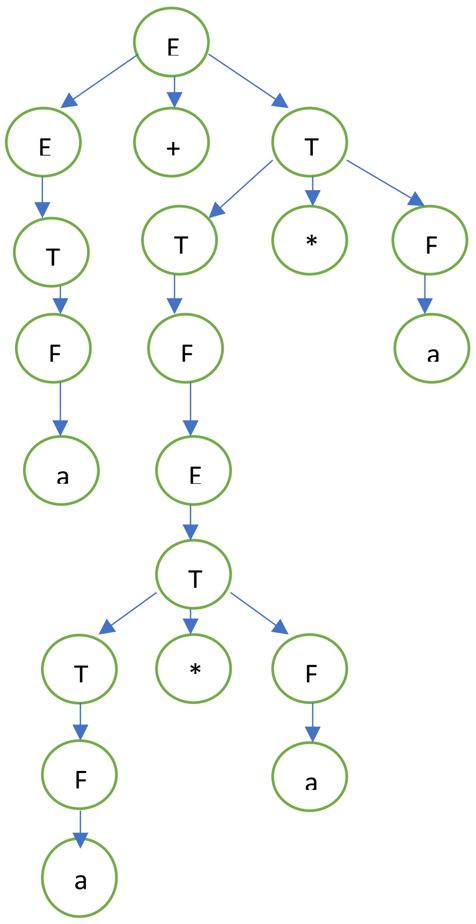
TARUN AGGARWAL
41396402717

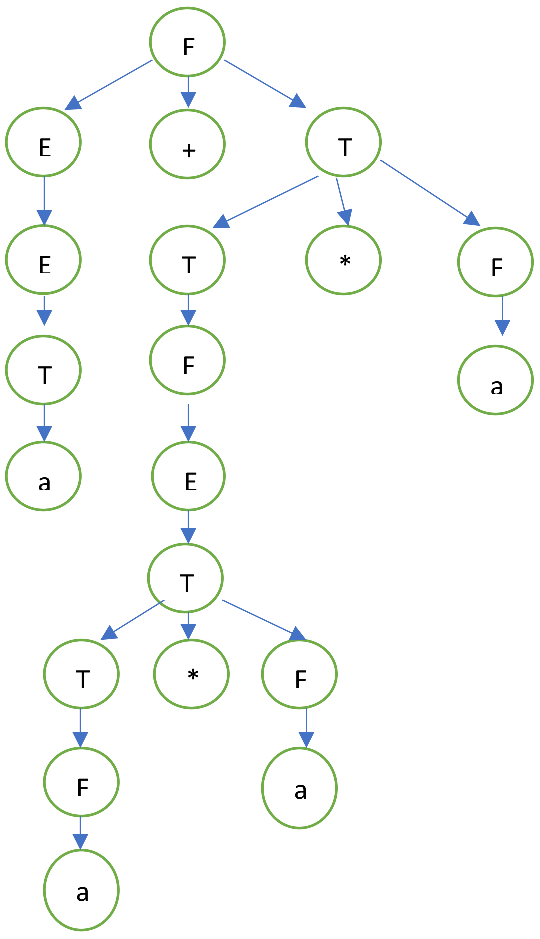
Q-3 (b) Give parse trees and derivations (leftmost and rightmost) for the grammar and input string given below.
E → E+T / T
T → T*F / F
F → (E) / a
Input string: a + (a*a) *a

E → E+T / T
T → T*F / F
F → (E) / a
Input string: a + (a*a) *a

Leftmost Derivation:

E → E+T
E → T+T
E → F+T
E → a+T
E → a+T*F
E → a+F*F
E → a+(E)*F
E → a+(T)*F
E → a+(T*F)*F
E → a+(F*F)*F
E → a+(a*F)*F
E → a+(a*a)*F
E → a+(a*a)*a

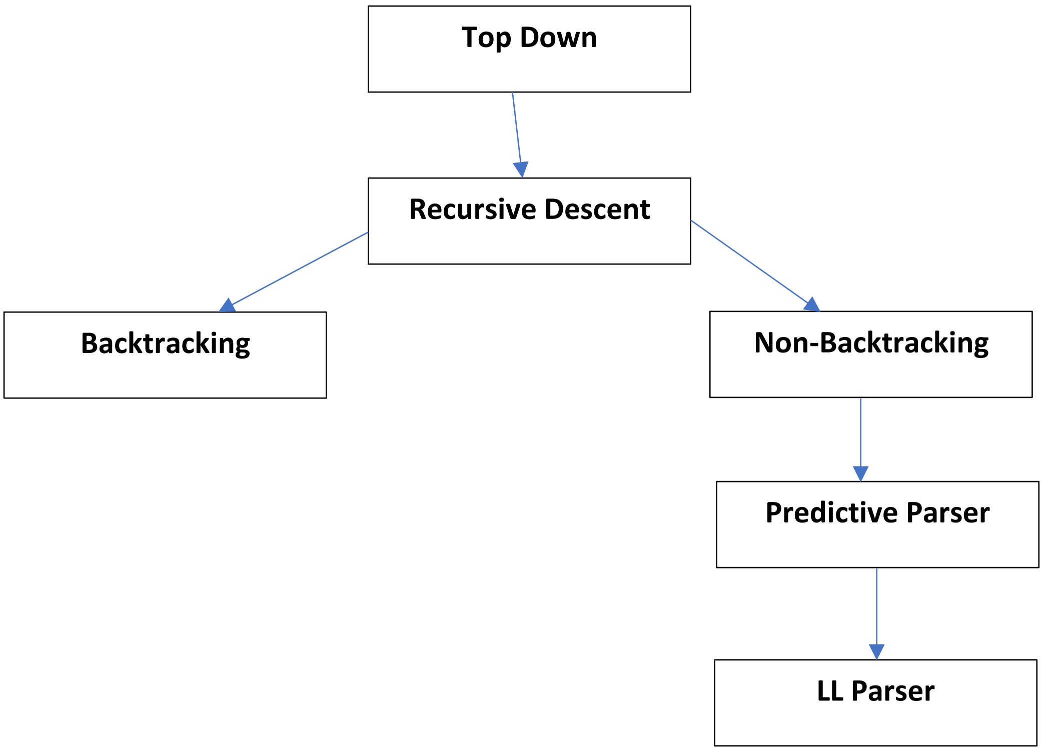




Rightmost Derivation

E → E+T
E → E+T*F
E → E+T*a
E → E+F*a
E → E+E*a
E → E+T*a
E → E+(T*F)*a
E → E+(T*a)*a
E → E+(a*a)*a
E → F+(a*a)*a
E → T+(a*a)*a
E → a+(a*a)*a

Q. 4. Explain Recursive descent Parser with example.
Recursive descent is a top-down parsing method that builds the parse tree from the top and the information is perused from left to right. It utilizes methodology for each terminal and non-terminal substance. This parsing method recursively parses the contribution to cause a to parse tree, which could possibly require backtracking. It recursively parses the info string implies based on utilizing one occurrence of an order or occasion to create another.
Top- down parsers start from the root node (start symbol) and match the input string against the production roles to replace them.



Q 4(b) Construct the canonical parsing table for the grammar given below.

S -> aA Be
A -> A bc I b
B -> d

S0: *S -> *a A B e

S1: S -> a*A B e
A-> *Abc I *b

A-> *Abc l *b

S2: A-> b*

S3: S-> aA*Be

B -> *d

A -> A*bc

S4: A-> Ab*c

S5: A-> Abc*

S6: S -> aA* Be

B -> *d

S7: B -> d*

S8: S -> aAB *e

S9: S -> aABe*

S10: S' -> S*

	(	)	a	b	$	S	A	B
0	1		9				4	
1	1		9			2	4	
2		3						
3		S→(S)			S→(S)			
4			7	8				5
5		S→AB		6	S→AB			
6		B→Bb		B→Bb	B→Bb			
7			A→Aa	A→Aa				
8		B→b		B→b	B→b			
9			A→a	A→a				

Q. 5. (a) Construct a syntax directed translation scheme that translates arithmetic expressions from' infix' to postfix notation. The solution should include context free grammar and semantic rules. Show the application of your scheme with the input given below.
1+2/3-4*5

- Parenthesize (using standard precedence) to get (1+(2/3))- (4*5) '
- Apply the above rules to calculate P{(1+(2/3))- (4*5)}, where P{X} means "convert the infix expression X to postfix".

A. P { (1 + (2 / 3)) - (4 * 5) }
B. P { 1 + (2 / 3)) } P { (4 * 5) } -
C. P { 1 + (2 / 3) } P { 4 * 5 } -
D. P { 1 } P { 2 / 3 } + P { 4 } P { 5 } * -
E. 1 P { 2 } P { 3 } / + 4 5 * -
F. 1 2 3 / + 4 5 * -

Q5(b) Translate the following expression to quadruple and triple representation A= - B*(C+D)/E

Quadruple representation

t_1 = Uminus B
t_2 = C
t_3 = t_2+D
t_4 = t_1 * t_3
t_5 = e
A = t_4/t_5

	OP	**Arg1**	**Arg2**	**Result**
0	Uminus	B		T_1
1		C		T_2
2	+	T_2	D	T_3
3	*	T_1	T_3	T_4
4		E		T_5
5	/	T_4	T_5	A

Triple representation

Number	**OP**	**Arg1**	**Arg2**
0	Uminus	B	
1		C	
2	+	1	D
3	*	0	2
4		e	
5	/	3	4

Q 6(e) What are the uses of the Symbol table in different phases of compiler Design?

Symbol Table is a significant data structure made and kept up by the compiler so as to keep semantics of variable or it stores data about extension and restricting data about names data about occurrences of different entities, for example, variables and function names and so on.

It is utilized by different periods of compiler as follows :-

- Lexical Analysis: Create new table sections in table, model like entries about token.

- SyntaxAnalysis : Adds Information with respect to attribute type, Scope, measurement, line of reference, use, and so forth in the table.

- Semantic Analysis: Uses accessible data in the to check for semantics for example to verify those expressions.
- Intermediate Code generation: Refers symbol table for knowing how much and what kind of run-time is assigned and table aides in including temporary variable data.

- Code Optimization: Uses data present in symbol table dependent improvement.

- Target Code generation: Generates code by utilizing address data for identifiers present in the table.

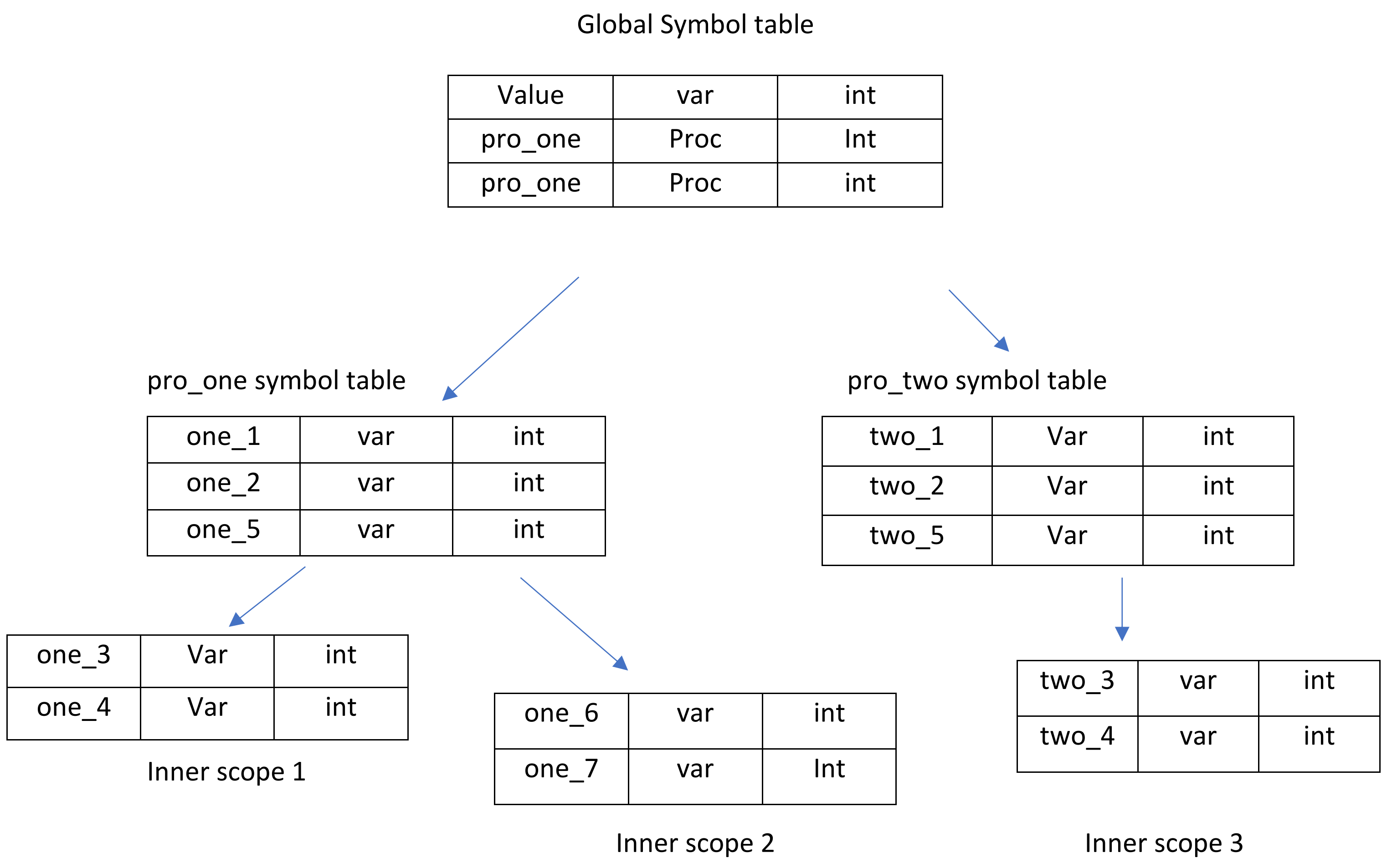
- Every section in the symbol table is related with characteristics that help the compiler in various phases.

Q6(b) Explain how scope information is represented by the symbol table.
Compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program. To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
int value=10;
void pro_one()
{
 int one_l;
 int one_2;
{
int one_3;
int one_4;
}
int one_5;
{
 int one_6;
int one_7;
}
}

void pro_two()
{
int two_l;
int two_2;
 {
 int two_3;
 int two_4;
}
}
```

The above program can be represented in a hierarchical structure as follows:



Global Symbol table

Value	var	int
pro_one	Proc	Int
pro_one	Proc	int

pro_one symbol table

one_1	var	int
one_2	var	int
one_5	var	int

pro_two symbol table

two_1	Var	int
two_2	Var	int
two_5	Var	int

one_3	Var	int
one_4	Var	int

Inner scope 1

one_6	var	int
one_7	var	Int

Inner scope 2

two_3	var	int
two_4	var	int

Inner scope 3

Q 6. (c) Explain different data structures for symbol table implementation and compare them.

Following are different data structure used for implementing symbol table: -

List

In this technique, an array is utilized to store names and related data. A pointer "accessible" is kept up toward the end of all stored records and new names are included the request as they show up.

- To scan for a name we start from start from beginning till accessible pointer and if not discovered we get an error "use of undeclared name"

- While inserting another name we should guarantee that it isn't now present in any case error occurs i. e. "Multiple defined name".

- Insertion is fast 0(1), but lookup is slow for large tables O(n).

- Advantage is that it takes a minimum amount of space.

Linked List

This execution is utilizing linked list. A link field is added to each record. Looking of names is done all together pointed by link of link field.

- A pointer "First" is kept up to point to first record of symbol table
- Insertion is fast 0(1), but lookup' is slow for large tables— O(n) on average

Hash Table
In hashing scheme two tables are kept up — a hash table and symbol table and is the most generally used strategy to implement symbol tables.

- A hash table is an array with index range: O to tablesize-1.These passages are pointers highlighting names of symbol tables.
- To look for a name we utilize a hash function that will bring about any whole number between 0 to tablesize — 1.
- Insertion and query can be made quick - 0(1).
- Bit of leeway is speedy search is conceivable and limitation is that hashing is complicated to use.

Binary Search Tree
Another way to deal with supplement symbol table is to utilize binary search tree or we include two connection fields for example left and right child.

- All names are made as children of root node that consistently follow the property of the binary search tree.
- Insertion and lockup are O(log2 n) on average.

Q. 7. (a) Why do we need code optimization? Explain Strength reduction, Subexpression elimination, dead code elimination and loop optimization techniques.

Code optimization:
The code optimization in the synthesis stage is a program change procedure, which attempts to improve the intermediate code by causing it to expend less assets so quicker—running machine code will result. Compiler advancing procedure should meet the accompanying objectives:

- The optimization must be right, it must not at all, change the meaning of the program.
- Optimization should speed up and execution of the program.
- The Compilation time must be kept sensible.
- The improvement procedure ought not postpone the general incorporating process.

- Optimization of the code is frequently performed toward the finish of the development stage since it lessens readability and adds code that is used to build the presentation.

Quality decrease: In compiler development, strength reduction is a compiler optimization where costly activities are replaced with identical however more affordable operations, the exemplary case of solidarity decrease "strong" increases inside a circle into "more fragile" augmentations '— ' something that frequently happens in' cluster ' tending to.

Instances of strength reduction include:

- replacing a duplication inside a' loop with an addition.

- replacing an exponentiation inside a loop with a duplication.

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

can be replaced with successive weaker conditions

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

.

Q7. (b) Write short notes on:
(i). Problems in code generation
Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. The code generator is expected to generate the correct code.

Issues in code generation:

Input to code generator:
The input to code generator is the intermediate code created by the front end, alongside data in the symbol table that decides the run-time locations of the data objects indicated by the names in the intermediate portrayal accepting that the they are liberated from all of syntactic and state semantic errors, the vital type checking has occurred and the type conversion operators have been inserted any place fundamental.

Target program:
Target program is the yield of the code generator. The yield might be outright machine language, relocatable machine language, low level computing language.

- Absolute machine language as a yield has focal points that it very well may be set in a fixed memory area and can be promptly executed.
- Relocatable machine language as a yield permits subprograms and subroutines to be compiled independently. Relocatable object modules can be linked together and loaded by linking loaders.
- Assembly language as a yield makes the code generation simpler. We call create generic symbolic instructions and utilize macro-facilities of assemblers in generating code.

Memory Management:
Mapping the names in the source program to addresses of information objects is finished by the front end and the code generator. A name in the location proclamation alludes to the image table section for name. At that point from the symbol table passage, a relative location can be resolved for the name.

Instruction selection:
Choosing best instruction will improve the effectiveness of the program, It incorporates the directions that ought to be finished and uniform. Instruction speeds and machine figures of speech additionally assumes a significant job when productivity is considered But on the off chance that we couldn't care less about The effectiveness of the objective program, at that point guidance determination is straight-forward.

Register allocation issues:
Use of registers make the computations quicker in contrast with that of memory, so productive use of registers is significant. The utilization of registers are partitioned into two subproblems:

- During Register assignment - we select just those set of variables that will live in the registers at each point in the program.
- During an ensuing Register task stage, the particular register is picked to get to the variable.

Evaluation order:
The code generator chooses the request where the instruction will be executed. The determination of computations influences the effectiveness of the objective code. Among numerous computations, some will require just less registers to hold the halfway outcomes. Be that as it may, picking the best request in the general case is a troublesome NP-complete issue.

Ways to deal with code generation issues: Code generator should consistently produce the right code. It is fundamental in view of the quantity of unique cases that a code generator may confront. A portion of the structure objectives of code generator are:

- Correct
- Easily viable
- Testable
- Maintainable

Q.7. (b)(ii) Uses of Basic Blocks.

Basic block is a lot of articulations which consistently executes in a grouping one change the other without getting stop in the centre or any chance of branching. Basic blocks don't contain any sort of jump statements in them. All the statements in a basic blocks gets execute in a similar request as they show up in the block without losing the low control of the program. A Basic block is a straight-line code succession with no. branches in but to the section and no branches out with the exception of at the exit. The code in a Basic block has:

- One section point, which means no code inside it is the goal of an in instruction any place in the program.
- One exit point, which means just the last instruction can make the program be executing code in an alternate basic block.
- The code might be source code, get together code or some other succession of guidance A sequence of instructions forms a basic block if the instruction in each position rules, or consistently executes previously, every one of those in later positions.
- No other instruction executes between two instructions in the sequence. For example: Consider the three address code for the expression a = b +c+d.

```
.  Tq = b+c
   T, = T,+ d
    a = T2
```

These statements will be executed in a sequence one after the other and therefore they form a basic block.
