

Q.1.a. Differentiate between Parse tree and Syntax tree.

(05)

Ans:

Parse Tree	Syntax Tree
Parse tree contain redundant information.	Syntax tree do not contains any redundant information.
It requires more memory storage.	Requires less memory storage.
Parse tree may be viewed as a graphical representation for a derivation.	A syntax tree is a compressed representation of the parse tree in which the operators appear at the interior node.
Ex.	Ex.

Q.1.b. State the reasons for the assembler to be multipass program.

(05)

Ans:

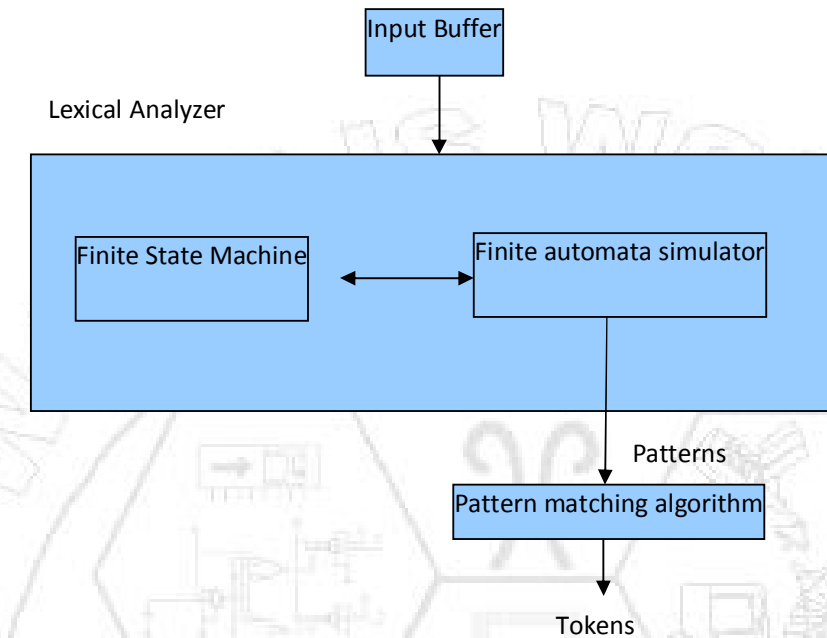
- Because symbols can appear before they are defined, it is necessary to make two passes over the input for the assembler.
- So the purpose of the PASS I is to define symbols and literals and purpose of PASS II is to generate instruction.
- Specifically the assembler must do the following;
- **Generate Instructions :**
- Evaluate the mnemonic in the operation field to produce its machine code.
- **Process pseudo ops :**
- We can group these tasks into two passes over the input associated with each task are one assembler module.

Q.1.c. Explain role of finite automata in Compiler theory.

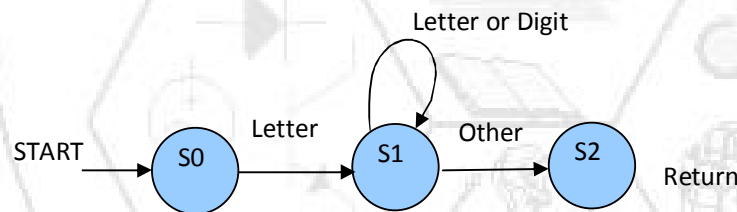
(05)

Ans:

- Lexical analysis is the process of recognizing tokens from input source program.
- Lexical analyzer stores the input in a buffer. It binds the regular expression for corresponding tokens.
- From this regular expression, finite automata are built. When LEX matches with the pattern generated by finite automata, the specific token gets generated.



- The regular expression for **identifier** will be
- r.e = letter (letter / digit )\*
- Transition diagram for **identifier** will be



Q.1.d. With example explain the process of elimination of left recursion. (05)

Ans:

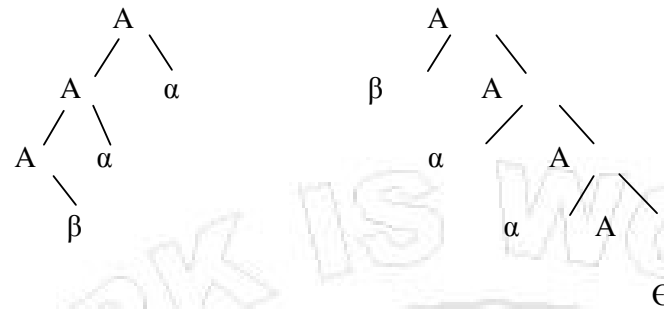
- If we have productions  $A \rightarrow A\alpha \mid \beta$ , where  $\beta$  does not begin with an A, then we can eliminate the left-recursion by replacing this pair of production with

$$A \rightarrow \beta A'$$

$$A' \rightarrow A\alpha \mid \epsilon$$

- We do not change the set of string derivable from A.

- Fig. Illustrate the nature of the transformation.



**Ex.**

Consider the grammar.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

Eliminating the immediate left recursion ( production of the form  $A \rightarrow A\alpha$ ) we obtain,

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

Q.1.e. *What is system programming? List some System Programs and write their functions.* (05)

Ans:

The process of converting the user defined code into format understood by the machine and then implementing the machine code is called **System Programming**.

**Assembler :** It converts assembly level language program into machine level language.

**Macro processor :** It convert HLL which contains macro definitions to a HLL which doesn't contain any macro call or macro definition. .

**Compiler:** It converts HLL into machine language. .

**Loader:** It loads the object program into memory for execution. .

Q.2.a. *With the help of following grammar and given string explain role of operator precedence parser.* (10)

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * V \mid V$$

$$V \rightarrow a \mid b \mid c \mid d$$

String to parse 'a + b\*c\*d'.

Ans:

## System Programming and Compiler Construction

Now,

$a < b$  ---a yields precedence to b

$a > b$  ---a has precedence as b

$a = b$  --- a takes precedence over b

Consider precedence relation for the above grammar:

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

If < then **Shift**.

If > then **Reduce**.

Let us step through the actions as operator precedence parser would take in parsing the input string “a+b\*c\*d”

Stack	Input	a	b	Action
\$	a+b*c*d\$	\$	a	Shift
\$a	+b*c*d\$	a	+	Reduce
\$V	+b*c*d\$	\$	+	Shift
\$V+	b*c*d\$	+	b	Shift
\$V+b	*c*d\$	b	*	Reduce
\$V+V	*c*d\$	+	*	Shift
\$V+V *	c*d\$	*	c	Shift
\$V+V *c	*d\$	c	*	Reduce
\$V+V *V	*d\$	*	*	Reduce
\$V+V *T	*d\$	*	*	Reduce
\$V+T	*d\$	+	*	Shift
\$V+T *	d\$	*	d	Shift
\$V+T *d	\$	d	\$	Reduce
\$V+T *V	\$	*	\$	Reduce

**System Programming and Compiler Construction**

\$T*V	\$	*	\$	Reduce
\$T*E	\$	*	\$	Reduce
\$E	\$	\$	\$	<b>Accept</b>

Q.2.b. Explain when will a macro be used in a program ? How is macro different from subroutine ?

(10)

Ans:

- A macro is a unit of specification for program generation through expansion.
- A macro consists of a name, a set of formal parameter and a body of code. The use of macro name with a set of actual parameter is replaced by some code generated from its body. This is called **Macro Definition**.
- Macro differ from subroutine in one fundamental aspect. use of macro name in the mnemonic field of an assembly statements leads to its **expansion**. Whereas use of subroutine name in a call instruction leads to its execution. Thus programs using macro and subroutine differ significantly in terms of program size and execution efficiency.
- The function of macro processor essentially involves the substitution of one group of character or lines for another. In some cases, the macro processor performs no analysis of the text it handles.
- This means that the design of macro processor is not directly related to the architecture of the computer on which it has to run.
- While subroutine gets involved in the code generation for the machine, which effects on the performance of the machine execution.
- So if there is no involvement of any logical operation in the development and we require to expand one line to number of lines then, the programmer may think over the use of macro processor.
- A macro is similar to subroutine. but there are important **difference** between them.
- A subroutine is a section of a program that is written once, and can be used many times by simply calling it from any point in the program. Similarly macro is a section of code that the programmer writes once, and then can use many times.

Q.3.a. Generate three address code for a given expressions.

While ( A < B ) do  
if ( C < D ) then X = Y + Z

(10)

Ans:

L1 : if A < B Goto L2  
Goto Last :

L2 : if C < D Goto L3  
Goto L1

L3 : t1 = X + 2

X = t1

Last:

The quadruple is a structure with at most four fields such as op,arg1,arg2,Result.

The op field is used to represent internal code for operator and arg1 and arg2 represent the two operand used and result field is used to store Result

Number	Op	Arg1	Arg2	result
1	+	X	2	t1
2	=	---	t1	X

In this representation the use of temporary variable is avoided by referring the pointer in the symbol table.

Number	Op	Arg1	Arg2
1	+	X	2
2	=	---	t1

Q.3.b. Write a note on JAVA Compiler and Environment.

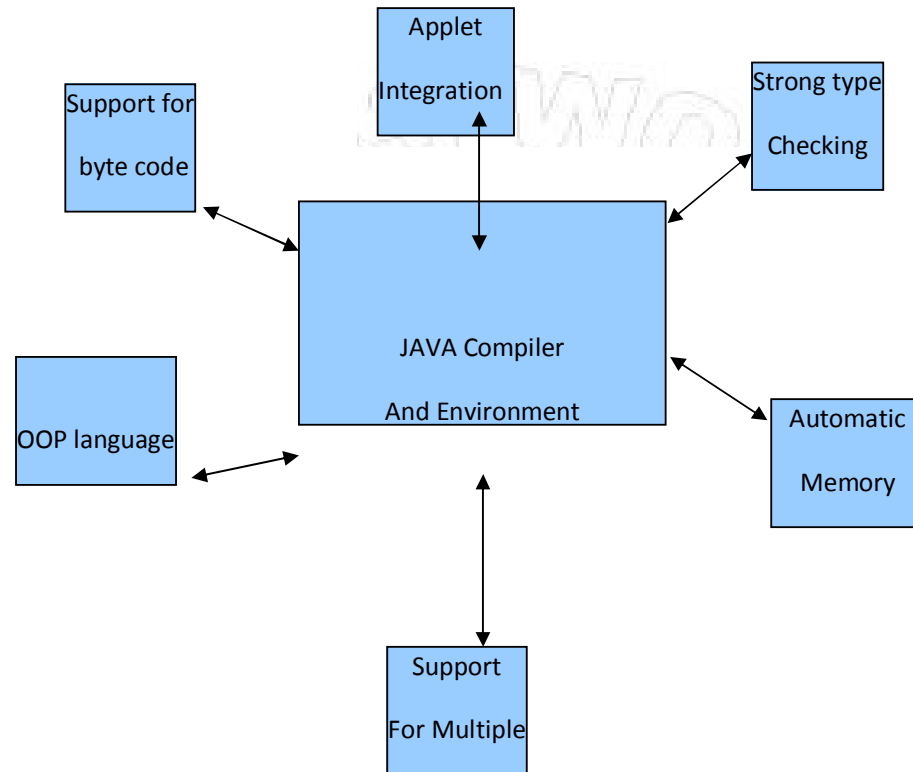
(05)

Ans:

- Java is a programming language which is developed by Sun Microsystems. This programming language is specially designed to work in internetworking environment.
- Using java we can write the high performance application which can be run on various operating systems or hardware. This future of Java is called as platform independence.
- Java is a platform independent language. That is the programs written in Java can be executed on variety of platform without any modifications.
- It is derived from C and C++ but does not have pointer or goto statements.
- Memory management is done automatically. This mechanism is called garbage collection and compaction.
- Compiler performs the strong type checking Due to which the errors can be detected at the early stage of program execution.
- Java is an object oriented programming language. Everything is Java is an object. Arrays and strings are also treated as objects. There are no

## System Programming and Compiler Construction

procedures or functions in Java, instead of that, classes and methods are used.  
 - Java supports multiple thread execution. Thread is nothing but ,a light weight process. Due to execution of Thread many processes can run concurrently. Hence Java library provides the. Method for starting a thread execution, stopping the thread, checking the status of thread and synchronizing various operations of multiple threads.



**What are assembler directives ? Explain with example.**

(05)

- These are the statements which are neither executable or nor declarative. These are the statements which direct the assembler to do specified task.

**START:** It indicates start of the program.

**END:** It indicate end of the program.

**USING:** it indicates which register is used as base register in program.

**DROP:** It indicates which register is no longer available to use in the program.

**EQU :** Whenever a symbol is defined is defined using EQU statement no memory would be allocated.

**Q.4.a.** Test whether the grammar is LL(1) or not, and construct a predictive parsing table for it.

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

(10)

Ans:

Consider the grammar:

$$S \rightarrow AaAb$$

$$S \rightarrow BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Now we will compute FIRST and FOLLOW functions.

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

Then,

$$S \rightarrow AaAb$$

$$S \rightarrow aAb \quad \text{When } A \rightarrow \epsilon$$

Also,

$$S \rightarrow BbBa$$

$$S \rightarrow bBa \quad \text{When } B \rightarrow \epsilon$$

$$\text{FIRST}(A) = \text{FIRST}(B) = \{\epsilon\}$$

$$\text{FOLLOW}(S) = \{\$\}$$

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

The LL(1) parsing table is,

	A	B	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	---
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	---
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	---

Example:

Now consider the string "ba" for parsing-

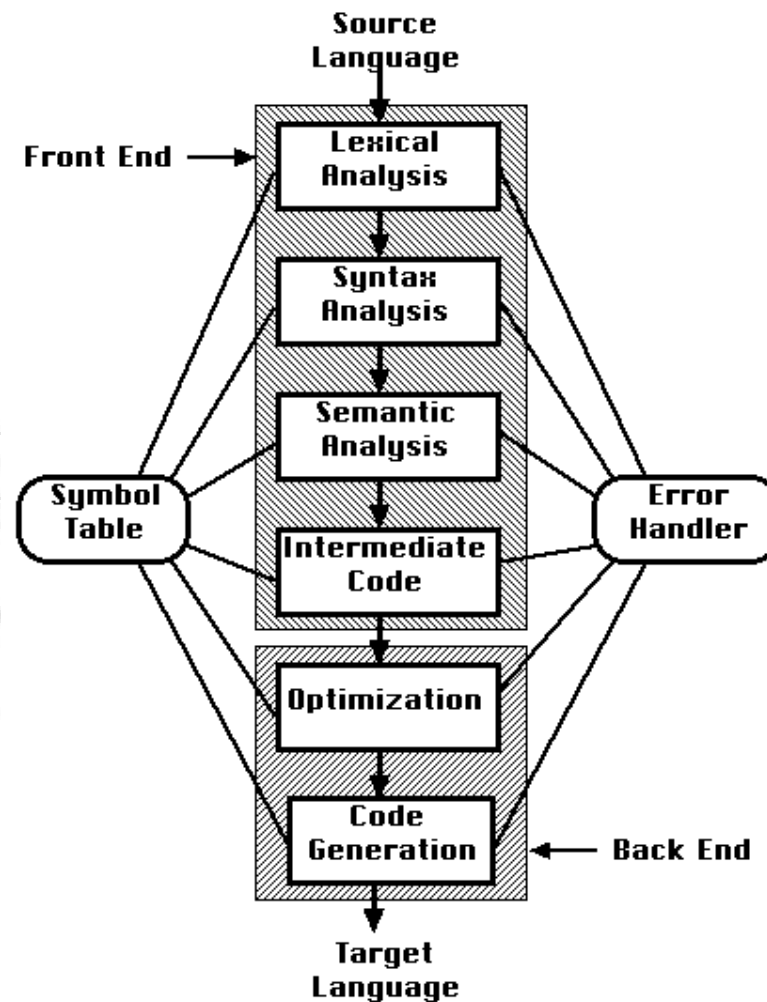
Stack	Input	Action
\$S	ba\$	$S \rightarrow BbBa$
\$aBbB	ba\$	$B \rightarrow \epsilon$
\$aBb	ba\$	---
\$aB	a\$	$B \rightarrow \epsilon$
\$a	a\$	---
\$	\$	<b>ACCEPT</b>

Q.4.b. Explain various phases of compiler with suitable example.

(10)

Ans:

The phases of compiler:



- The process of compilation is carried out in two parts analysis and synthesis.
- The analysis phase is carried out in three phases: Lexical analysis, Syntax Analysis and semantic analysis and synthesis phase is carried out in three phases: Intermediate code generation, code optimization and code generation.
- The process of **lexical analysis** is called lexical analyzer, scanner, or tokenizer. Its purpose is to break a sequence of characters into a subsequence's called tokens.
- The **syntax analysis** phase, called parser, reads tokens and validates them in accordance with a grammar. Vocabulary, i.e., a set of predefined tokens, is composed of word symbols (reserved words), names (identifiers), numerals (constants), and special symbols (operators). During compilation, a compiler will find errors such as lexical, syntax, semantic, and logical errors. If a token is found not belonging to the vocabulary, it is an lexical error.
- **Semantic analysis** deals with meaning of the statements means matching of parenthesis, matching of If...Else statements, performing arithmetic operations etc.

- **ICG** is the code which is very easy to generate and this code can easily be converted to target code. The ICG having variety of forms such as Three Address Code, Syntax tree, Postfix notations.
- This phase improves the ICG. This is necessary to have a faster execution and it takes less memory.
- In this phase the final code is generated. IC instructions are translated into sequence of machine instructions.
- Example

**Q.5.a.** With reference to assembler explain the following tables with suitable example.  
(1) POT (2) MOT (3) ST (4) LT.

(10)

Ans:

**MOT (Mnemonic opcode table)**

Mnemonic opcode	Machine opcode	Format Info	Routine

MOT is fixed length table.

In PASS I using Mnemonic opcode MOT is consulted to obtain instruction length to update location counter.

In PASS II using Mnemonic opcode MOT is consulted to obtain Binary opcode, instruction length, instruction format.

Ex:

Mnemonic Opcode	Machine Opcode	Format Info	Routine
START			
END			

**(2) POT (Pseudo Opcode table):**

Pseudo Opcode	Address Routine

POT is fixed length table.

In PASS I using Pseudo opcode, POT is consulted to process some pseudo opcodes like DS, DC, EQU etc.

Ex:

**System Programming and Compiler Construction**

Pseudo Opcode	Address Routine
USING	
DC	

**(3)ST (Symbol Table):**

Symbol	Value	Length	Relocation

- ST is used to keep a track on the symbol defined in the program.
- In PASS I whenever a symbol is defined an entry for that symbol is made in ST.
- In PASS II Symbol table is used to generate the address of symbol.

Ex:

**(4) LT (Literal Table):**

- LT is used to keep a track on the literals encountered in the program.
- In PASS I whenever a literal is encountered an entry is made in the literal table.
- In PASS II literal table is used to generate the address of the literal.

Q.5.b. What are different functions of loader? Explain difference between linkage editor and linking loader.

(10)

Ans:

There are four functions of loader

1. Allocation
2. Linking
3. Relocation
4. Loading

**Allocation:**

- it allocates the space for program in the memory, by calculating the size of the program. this activity is called allocation.

**- Linking:**

The symbol defined in one segment may be used in others. So, an establishment of the proper correspondence and (he use of the correct attribute is known as resolution of inter segment (inter-program) symbolic references or *linking*.

So, in general a program can contain -

- *internal references to externally defined symbols (i.e. external definitions)*
- Internally defined symbols which may he used externally, (i.e. public definitions).

It resolve the symbolic references between the object module by assigning all the user subroutine addresses. This activity is called as Linking.

**System Programming and Compiler Construction****Relocation:**

- Allocation means fixing the memory required by the programs. This task may be done by an assembler if the program does not contain external or library subroutines or else otherwise linker knows the size of the total linked programs together.
- There are some address dependant location in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.

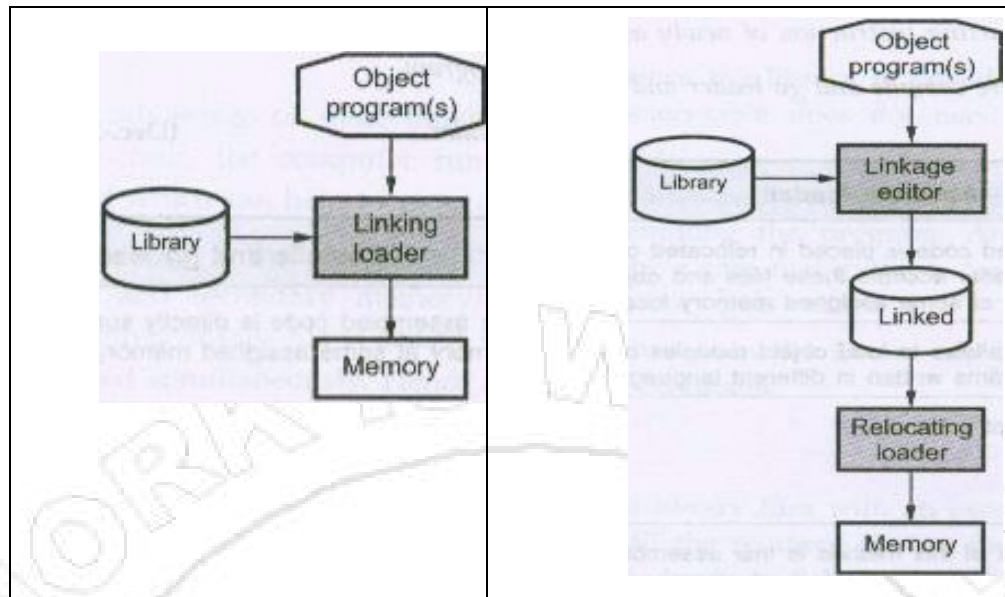
**Loading:**

- Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory Thus program now becomes ready for execution, this activity is called loading.

**Difference between linkage editor and linking loader**

<b>Linking loader</b>	<b>Linkage editor</b>
<ul style="list-style-type: none"> <li>• The linking loader performs all the linking and relocation operations including automatic library search then loads the program directly into the memory for execution</li> </ul>	<ul style="list-style-type: none"> <li>• The linkage editor produces a linked version of the program. Such a linked version is also called as load module or executable image. This load module is generally written in a file or library for later execution</li> </ul>
<ul style="list-style-type: none"> <li>• There is no need of relocating loader</li> </ul>	<ul style="list-style-type: none"> <li>• The relocating loader loads the load module into the memory.</li> </ul>
<ul style="list-style-type: none"> <li>• The linking loader searches the libraries and resolves the external references every time the program is executed.</li> </ul>	<ul style="list-style-type: none"> <li>• If program is executed many times without being reassembled then linkage editor is the best choice.</li> </ul>
<ul style="list-style-type: none"> <li>• When program is in development stage then at that time the linking loader can be used.</li> </ul>	<ul style="list-style-type: none"> <li>• When program development is finished or when the library is built then linkage editor can be used.</li> </ul>
<ul style="list-style-type: none"> <li>• The loading may require two passes</li> </ul>	<ul style="list-style-type: none"> <li>• The loading can be accomplished in one- pass.</li> </ul>

**System Programming and Compiler Construction**

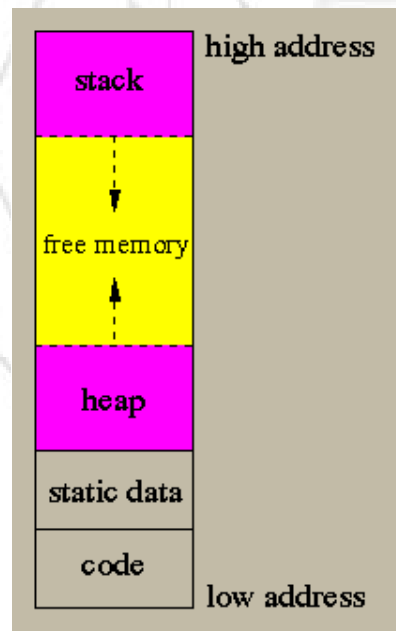


Q.6.a. *Explain run time organization in detail.*

(10)

Ans:

- During the execution of a program, the same name in the source can denote different data objects in the computer.
- The allocation and deal location of data objects is managed by the run-time support Package .



This is the layout in memory of an executable program.

The machine code of the program is typically located at the lowest part of the layout. Then, after the code, there is a section to keep all the fixed size static data

in the program.

- The dynamically allocated data (ie. the data created using malloc in C) as well as the static data without a fixed size (such as arrays of variable size) are created and kept in the heap. The heap grows from low to high addresses.

-When you call malloc in C to create a dynamically allocated structure, the program tries to find an empty place in the heap with sufficient space to insert the new data; if it can't do that, it puts the data at the end of the heap and increases the heap size.

- The focus of this section is the stack in the memory layout. It is called the *run-time stack*. The stack, in contrast to the heap, grows in the opposite direction from high to low addresses, which is a bit counterintuitive. The stack is not only used to push the return address when a function is called, but it is also used for allocating some of the local variables of a function during the function call, as well as for some bookkeeping.

### Activation Record:

<b>Returned value</b>
<b>Actual parameters</b>
<b>Optional control link</b>
<b>Optional access link</b>
<b>Saved machine status</b>
<b>Local data</b>
<b>Temporaries</b>

-The activation record is a block of memory used for mapping information needed by a single execution of a procedure.

-The **Temporaries** are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.

-The **Local data** is a data that is local to the execution of procedure is stored in this field of activation record.

-The **Saved machine status** holds the information about status of the machine just before the procedure is called.

-The **Optional access link** refers to the non local data in other activation record.

-The **Optional control link** points to the activation record of the calling procedure.

-**Actual parameters** holds the information about the actual parameters.

-**Returned value** is used to store the result of a function call.

Q.6.b. Explain role of code optimization in compiler designing with suitable Example. (10)

Ans:

- Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output.

#### -Local Optimizations

-Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block.

-Optimization is the field where most compiler research is done today. The tasks of the front-end (scanning, parsing, semantic analysis) are well understood and unoptimized code generation is relatively straightforward.

-There are a variety of tactics for attacking optimization. Some techniques are applied to the intermediate code, to streamline, rearrange, compress, etc. in an effort to reduce the size of the abstract syntax tree.

-Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to spill,

#### Constant Folding

-Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant. In its simplest form, it involves determining that all of the operands in an expression are constant-valued, performing the evaluation of the expression at compile-time, and then replacing the expression by its value.

:		:
:		:
a = 4/2;		a = 2;
b = a+3;		b = 5;
:		:
:		:

Before Optimization                      After Optimization

#### Strength Reduction

Operator strength reduction replaces an operator by a "less expensive" one.

:		:
:		:
a * 2;		a << 2;
b / 2;		b >> 5;
:		:
:		:

Before Optimization                      After Optimization

#### Dead Code Elimination

If an instruction's result is never used, the instruction is considered "dead" and can be

removed from the instruction stream.

```

:           |           :
:           |           :
debug=false; | debug=false;
if(debug);   |           :
:           |           :
:           |           :

```

Before Optimization

After Optimization

### Common Sub expression Elimination

It implies performing the evaluation of common subexpression result ones and replacing the result instead of recomputing.

```

:           |           :
:           |           :
a = b*d + 3; | a = t+3;
c = b*d + 5; | c = t+5;
e = 6 + b*d; | e = 6+t;
:           |           :

```

Before Optimization

After Optimization

### Frequency Reduction

It implies that movement of the code from a portion where it is frequently getting executed to a portion where it would not be frequently executed.

```

:           |           :
:           |           :
for(i=1;i<100;i++); | t = b*c;
{               | for(i=1;i<100;i++);
t = b*c;       | {
}               | }
:           |           :

```

Before Optimization

After Optimization

Q.7. *What is binding ? Explain static and dynamic binding.*

(10)

Ans:

### Binding:

Binding is nothing but attaching a property to an entity. For example, name is a property find it can be attached with a person. Date of birth is another such property and it is also attached with a person, but this is a type of fixed binding On the contrary, age is a property which is attached with a person keeps changing.

### System Programming and Compiler Construction

As far as programming languages goes, binding is nothing but attaching some property to program entity. For example, if we take, operator overloading as a property, then, whether operator overloading is to be allowed or not, must be decided by a language designer, where as, if we say, `int i`; then `i` is an integer This property is attached at compile time.

If we take data [data object] in a program as a program entity, then this data object can undergo various bindings during its life time.

Let us look at some examples.

- (i) The binding of a data object to one or more values. This binding may be modified by assignment statement .e.g. `a = 5;a=10`;
- (ii) The binding of a data object to one or more names. These bindings are usually set up by declarations and modified by assignments , subprogram calls and returns.  
e.g. `a=b;a=c;c=b` etc.

The time at which a binding takes place is called binding time.

Binding can take place at language design time, language implementation time, compile time, link time, load time, or run time. For example, symbol (\*) is usually bound to multiplication or comment (depending on its context) at the language design time. A data type, such as integer is bound to range of possible values at language implementation time. At compile time, a variable in language C or Pascal program is bound to a particular data type, A call to a library subprogram is bound to subprogram code at link time. A variable may be bound to a storage cell when the program is loaded in memory. That same binding does not happen until run time in some cases.

An Example of Binding :

To illustrate variety of bindings and binding times, consider single assignment statement.

`Y =X+ 10`

Now here, the meaning of symbol plus ('+') is addition . It can be overloaded i.e. it can have more than one functions depending on the context it is in . Now in the example , if X is integer , then it is an integer addition ,but if X is float , then whether X is to be converted to integer or 10 is to be converted to float , that decision must be taken at language definition time Moreover , if Y is integer then there are two ways to look at this situation In the first situation , X can be converted to integer, then added to 10 to get a value of Y which is integer . In the second situation , X can be kept as it is , 10 can be converted to float (as 10.0) , float addition is performed to get answer in float , but because Y is integer , the answer is once again converted back to integer. The method explained in the second situation is to be used if one does not want to lose finer details ( precision ). The better way is to widen ( i.e. int to real , real to double and so on ) at the right hand side of the assignment and shorten if asked to do the left hand side. Doing process of widening and shortening is called 'coercion' and this is done at language implementation time.

**Q.7.b. Explain syntax directed translation with respect to construction of syntax tree.**

(10)

Ans:

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser.

-The parsing process and parse trees are used to direct semantic analysis and the

**System Programming and Compiler Construction**

translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation.

-We augment a grammar by associating attributes with each grammar symbol that describes its properties. An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register.

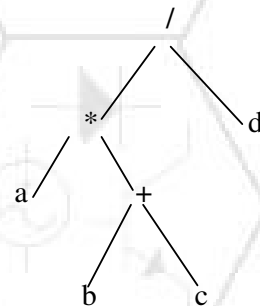
-With each production in a grammar, we give semantic rules or actions, which describe how to compute the attribute values associated with each grammar symbol in a production.

-A value associated with the grammar symbol is called a translation of that symbol. We shall usually denote the translation fields of a grammar symbol  $X$  with names such as  $X.value$ ,  $X.true$  and so on.

-For example, suppose we have the production and semantic action  $E \rightarrow E + E$  then  $\{E.VAL = E(1).VAL + E(2).VAL\}$ . the semantic action is enclosed in braces and it appears after production.

**Construction of Syntax tree:**

-It is easy to define a parse tree or syntax tree in terms of syntax directed translation. Now  $E.VAL$  is a translation whose value is a pointer to a node in the syntax tree.

**Fig : Syntax Tree**

	<b>Production</b>	<b>Semantic Action</b>
1	$E \rightarrow E(1) \text{ op } E(2)$	$\{E.VAL = \text{NODE}(\text{op}, E(1).VAL, E(2).VAL)\}$
2	$E \rightarrow ( E(1) )$	$\{E.VAL = E(1).VAL\}$
3	$E \rightarrow -E(1)$	$\{E.VAL = \text{UNARY}(-, E(1).VAL)\}$
4	$E \rightarrow \text{id}$	$\{E.VAL = \text{LEAF}(\text{id})\}$

-The function  $\text{NODE}(\text{OP}, \text{LEFT}, \text{RIGHT})$  takes three arguments. The first is the name of operator, the second and third are pointers to roots of sub trees.

**System Programming and Compiler Construction**

---

-The function creates a new node labeled by the first argument and makes the second and third arguments the left and right children of new nod, returning a pointer to the created node.

-The function UNARY(op,child) creates a new node labeled op and makes **child** its child.

-The function LEAF(id) creates a new node labelled by **id** and returns a pointer to that node. This node receives no children. In practice the label of the leaf would be a reorientation of a particular name, such as a pointer to the symbol table.

