CD SESSIONAL –II

IMPORTANT QUESTIONS

UNIT-3

1. **Compare LR parsers.**

SLR Parser	LALR Parser	CLR Parser
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.

2. **Construct LALR Parsing Table for the following grammar**

   S → C C

   C → c C | d

Consider items I_3 and I_6. Both these items set have the same core but they differ in their look-ahead and hence we combine them and call it as item I_{36} as given below.

$$I_{36}: \quad C \to c \cdot C,\ c/d/\$$$
$$C \to \cdot cC,\ c/d/\$$$
$$C \to \cdot d,\ c/d/\$$$

Similarly items I_4 and I_7 could be combined together as item I_{47} and items I_8 and I_9 as I_{89} .

I_{47}: $C \rightarrow d\cdot,\ c/d/\$$

I_{89}: $C \rightarrow cC\cdot,\ c/d/\$$

LALR Parsing Table

State	action			goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

UNIT-4

1.Explain in brief about synthesized and inherited attributes with an example.

Synthesized Attributes

An attribute is said to be Synthesized if its value at a parse tree node is determined from attribute values at the child nodes.

To illustrate, assume the following production S → ABC if S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

The production must have non-terminal as its head.

It can be evaluated during a single bottom-up traversal of the parse tree.

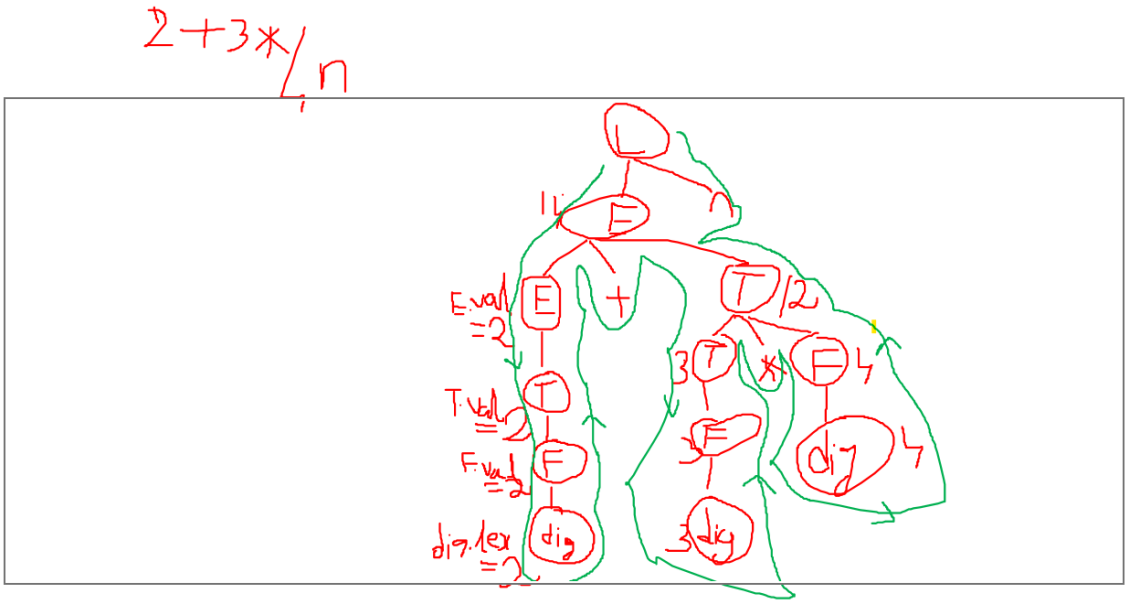
A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition.

Example

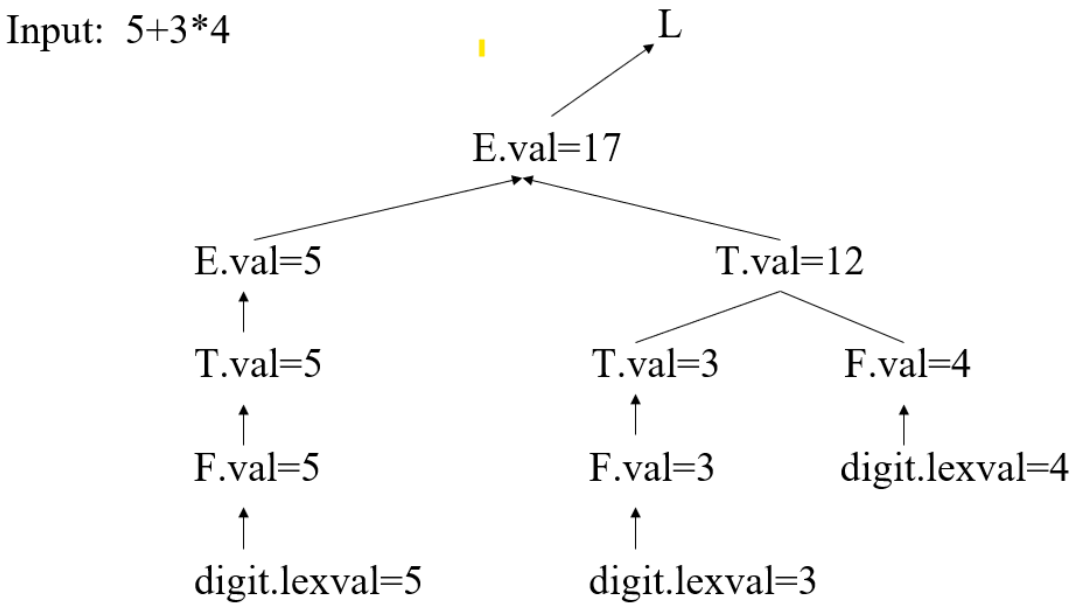
Production	**Semantic Rules**
L → E **n**	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated ParseTree for 2+3*4n



Dependency Graph



Inherited Attributes

An attribute is said to be Inherited if its value at a parse tree node is determined by the attribute value at parent and/or siblings node.

In case of S → ABC if A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B then S is said to be Inherited Attribute.

The production must have non-terminal as a symbol in its body.

It can be evaluated during a single top-down and sideways traversal of parse tree.

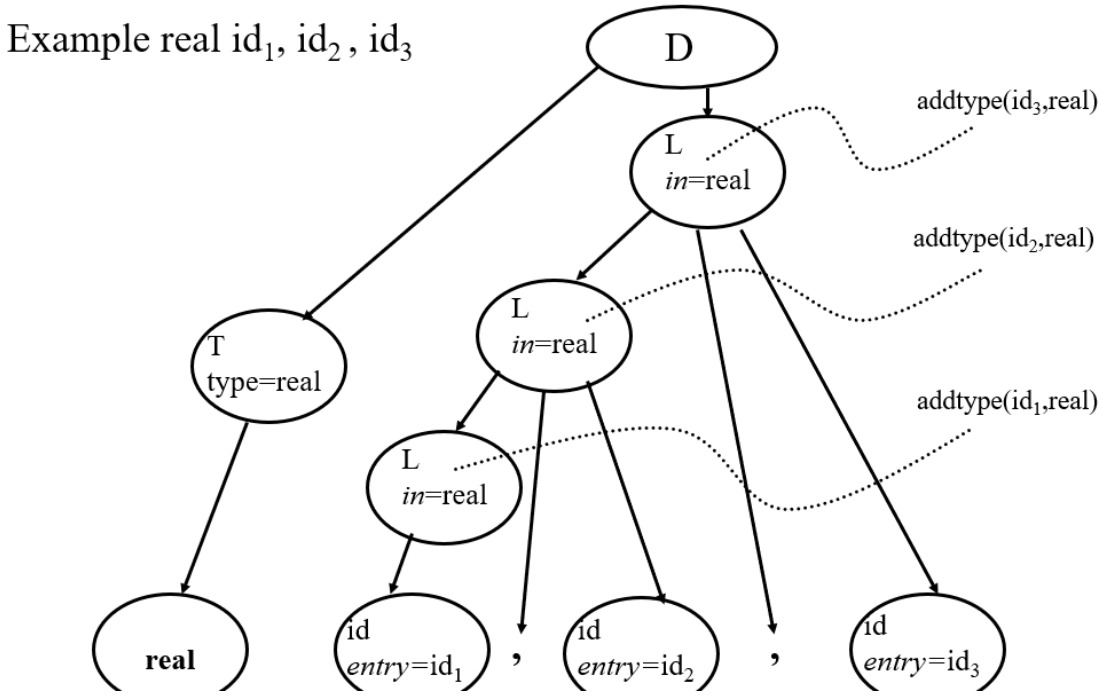
Example

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 , **id**	L_1.in = L.in, addtype(**id**.entry,L.in)

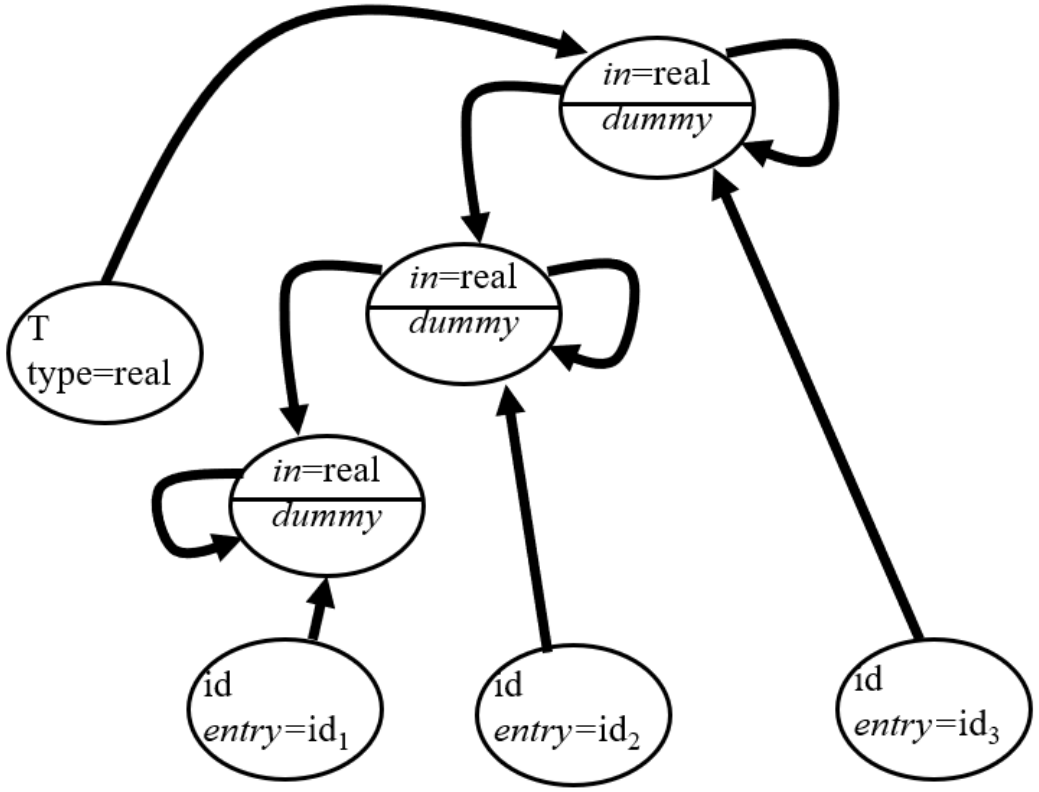
L → **id** addtype(**id**.entry,L.in)

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in.*

Annotated ParseTree



Dependency Graph



2.Differences b/w S-attributed and L-attributed grammars with examples.

Sl. No.	S-attributed	L-attributed
1.	If it uses only synthesized attributes, it is called as S-attributed.	It uses both synthesized attributes and inherited attributes
2.	Semantic actions are placed at right end of production. A->XYZ{ }	Semantic actions are placed anywhere in RHS. A->{ }BC |D{ }E |FG{ }
3.	Attributes are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.	Attributes are evaluated by traversing parse tree depth-first and left-to-right manner.

Sl. No.	S-attributed	L-attributed
4.	The attributes in S-attributed grammars can only depend on the attributes of their children or their siblings. They cannot depend on attributes from nodes higher in the parse tree.	The attributes in L-attributed grammars can depend on attributes of their siblings (preceding and following) and their parents.

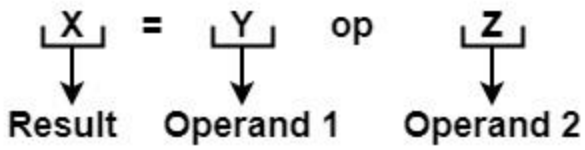
5.	Example: Let's consider a simple S-attributed grammar for arithmetic expressions:
	`E -> E + T { E.value = E1.value + T.value }`
	`E -> T { E.value = T.value }`
	`T -> T * F { T.value = T1.value * F.value }`
	`T -> F { T.value = F.value }`
	`F -> ( E ) { F.value = E.value }`
	`F -> number { F.value = number.value }`
		Example: Consider an L-attributed grammar for a simple programming language with assignments:
		`S -> id = E { id.value = E.value; }`
		`E -> E + T { E.value = E.value + T.value; }`
		`E -> T { E.value = T.value; }`
		`T -> T * F { T.value = T.value * F.value; }`
		`T -> F { T.value = F.value; }`
		`F -> ( E ) { F.value = E.value; }`
		`F -> id { F.value = id.value; }`

6.	In this grammar, the attribute "value" is synthesized and represents the value of an expression. The attribute is evaluated by propagating values from the children of a node to the parent node using appropriate semantic rules.	In this grammar, both synthesized and inherited attributes are used. The synthesized attribute "value" represents the value of an expression, while the inherited attribute "id.value" represents the value of an identifier. The attribute evaluation follows a preorder traversal, and the attribute dependencies are defined based on the grammar rules.

3.Explain about different Types(Forms) of three address code Statements with example.

The three-address code is a sequence of statements of the form A−=B op C, where A, B, C are either programmer-defined names, constants, or compiler-generated temporary names, the op represents an operator that can be constant or floatingpoint arithmetic operators or a Boolean valued data or a logical operator.

General form of Three-Address Code Representation is



TYPES:

1) Assignment statements of the form $x = y \text{ op } z$ where op is a binary arithmetic (or) logical operator.

Assignment Statement: **x:=y op z**

2) Assignment statements of the form $x = \text{op } y$ where op is a unary operator

Assignment Statement: **x:=op z**

3) copy statement of the form $x = y$.

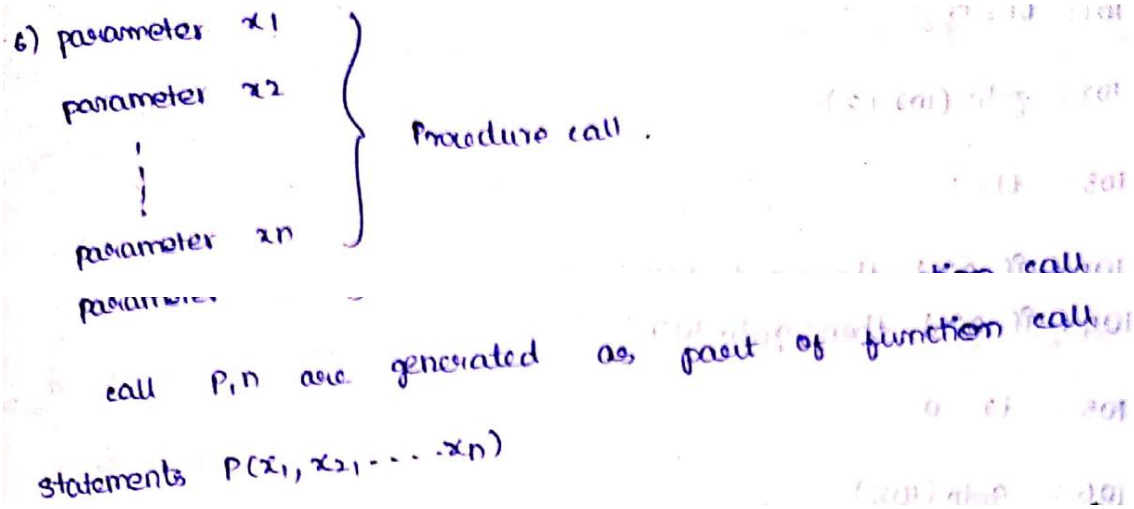
Copy Statement: **x:=z**

4) The unconditional jump statement goto L

Unconditional Jump: **goto L**

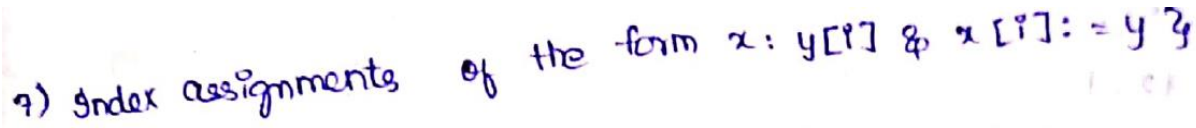
5) conditional jump statement is if x relop y goto L

Conditional Jump: **if x relop y goto L**



Procedure:

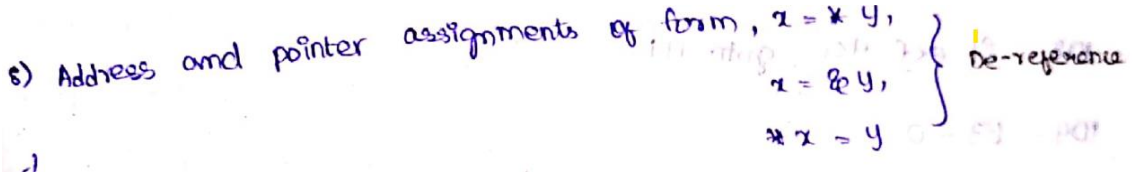
param x_1
param x_2
...
param x_n
call p,n ➔ $p(x_1, \ldots, x_n)$



Index Assignments:

x:=y[i]

x[i]:=y **array references**



Address and Pointer Assignments:

x:=&y

x:=*y **Dereferencing**

***x:=y**

EXAMPLE:

3. g=(a+b)*(c+d)*(e-f)

Three Address Code:

t1=a+b

t2=c+d

t3=e-f

t4=t1*t2

g=t4*t3

4.Quadraples,triples,indirect triples with examples.(like a=b* -c + b* -c)

a = b * – c + b * – c

Three address code:

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

Quadruples

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Triples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

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

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	=	a	(18)

Indirect triples

5.Discuss about different storage allocation techniques(static, stack and heap allocation).

The various storage allocation strategies to allocate storage in different data areas of memory are:

Static Allocation

• Storage is allocated for all data objects at compile time

In a static environment there are a number of restrictions:

- Size of data objects are known at compile time
- No recursive procedures
- No dynamic memory allocation

Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented statically.

No data structure can be created dynamically as all data is static.

Stack Allocation

• The storage is managed as a stack

In a stack-based allocation, the previous restrictions are lifted

- procedures are allowed to be called recursively
- Dynamic memory allocation is allowed
- Pointers to data locations are allowed

Storage is organized as a stack.

Activation records are pushed and popped.

Locals and parameters are contained in the activation records for the call.

This means locals are bound to fresh storage on every call.

We just need a stack_top pointer.

- To allocate a new activation record, we just increase stack_top.
- To deallocate an existing activation record, we just decrease stack_top.

Heap Allocation (It is one of Dynamic Storage Allocation)

• The storage is allocated and deallocated at runtime from a

data area known as heap

- ✓ The efficient heap management can be done by:
  - I. Creating a linked list for the free blocks and when any memory deallocated that block of memory is appended in the linked list.
  - II. Allocate the most suitable block of memory from the linked list.

Heap allocation retains the activation record even after the activation is completed.Later when it is deallocated the free space left out will be filled by the heap manager for some other objects**.**

UNIT-5

CODE OPTIMIZATION:

1.Explain in brief about different Principal sources of optimization techniques(Local Optimization techniques, Loop optimization techniques) with suitable examples(i.e., Machine Independent opt or Transformation Techniques)

CODE GENERATION:

2.Explain the generic issues in the design of code generator

The issues in the design of code generation phase :

Input to code generator

- IR + Symbol table
- IR has several choices
    - Postfix notation
    - Syntax tree
    - Three address code.

Target program

- Target program is the output of the code generator.
- The output may be
    - Absolute machine language(executable code)
    - Relocatable machine language(object files for linker)
    - Assembly language(facilitates debugging)

Memory Management

- Mapping the names in the source program to addresses of data objects is done by the front end and the code generator.

- The type in a declaration determines the width, i.e., the amount of storage, needed for the declared name.

Instruction selection

- Selecting best instructions will improve the efficiency of the program.

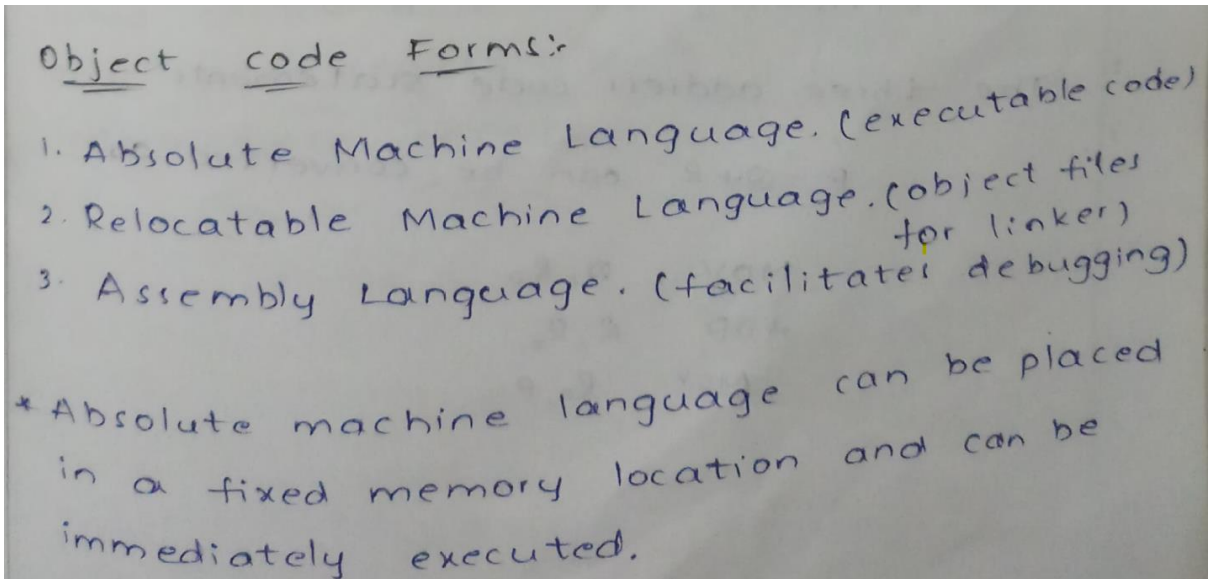
Register Allocation

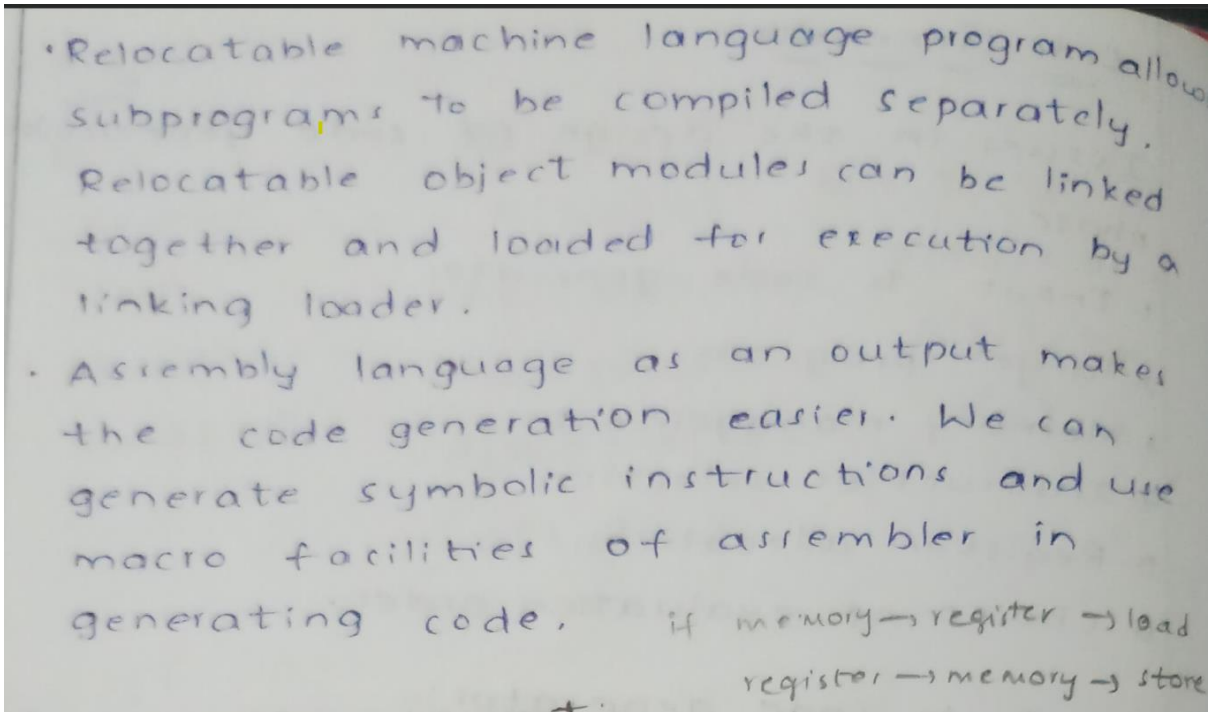
- A Key problem in code generation is deciding what values to hold in what registers.
- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important.

Evaluation order

- The order in which the computations are performed will effect the efficiency of the target code
- Some computational orders, some will require only fewer registers to hold the intermediate results.

3.Explain the different object code forms in detail.



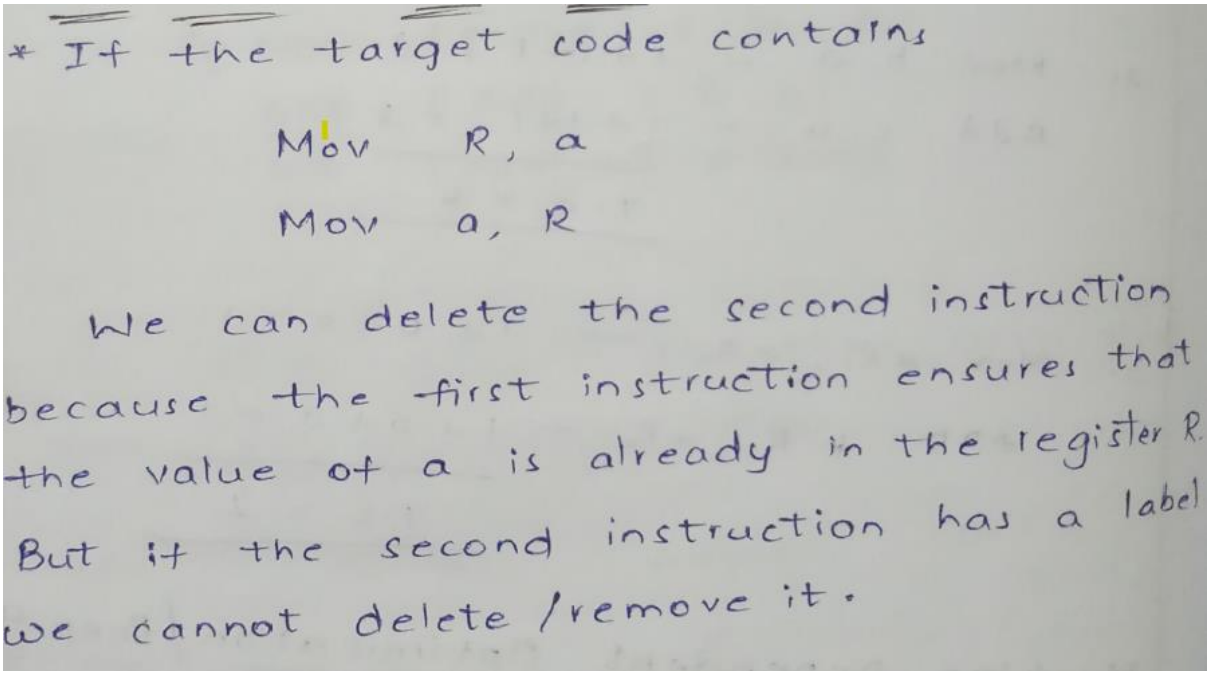


4.Explain about peephole optimization with example.(Machine Dependent opt).

- Peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window".
- Peephole: a short sequence of target instructions that may be replaced by a shorter/faster sequence.

Common techniques applied in peephole optimization are:

- Redundant instruction elimination



- Eliminating Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs.

- Flow of control optimizations

If we have jumps to jumps, then these unnecessary jumps can be removed

- Algebraic simplification

```
      goto L1                    if a < b goto L1
      ...                        ...
L1:   goto L2              L1:   goto L2
       ⇓                          ⇓
      goto L2                    if a < b goto L2
      ...                        ...
L1:   goto L2              L1:   goto L2
```

- Strength reduction

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

- Use of machine idioms

The target instructions have equivalent machine operations to perform some operations.

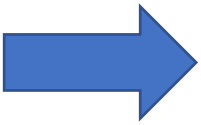
We can replace the target instructions by equivalent machine instructions in order to improve the efficiency.

Eg. a=a+1

MOV a,R

ADD #1,R

MOV R,a



INC a