Q1) Compilation process contains the sequence of various phases. Each phase takes source program in one representation & produces output in another representation.

* Various phases of compiler :-

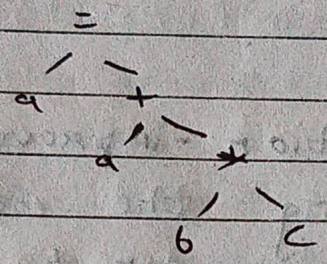
1) Analysis - break the source code into parts and arranges these pieces
2) Synthesis - Generation of target code or machine code.

-> Analysis phase is divided into three :
1) lexical analysis - broke the source code into tokens.

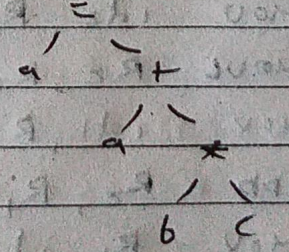
Ex: a := a+b*c ;

a -> identifier 1
:= -> (assignment) operator
a -> identifier 1
+ -> operator
b -> identifier 2
* -> operator
c -> identifier 3
; -> special symbol

2) Syntax analysis - It is called also as parsing.
- determines syntaxtical mistakes.
- if code is error free, then it generates the tree.



3) Semantic Analysis - determines meaning of a source string.
- For example Matching of expression, if else statement, or checking the scope of expression.
- Also generates tree.



-> Synthesis phase is divided into three:
1) Intermediate code generation
- converts code into intermediate phase
- one of the intermediate code example is "three address code"
- Example :-

a = a + b * c
id_1 id_2 id_3

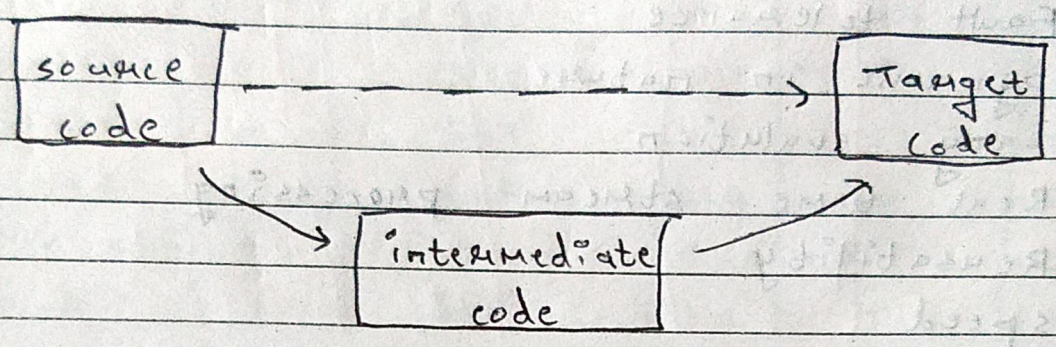
$t_1 = id_2 * id_3$
$id_1 = id_1 + t_1$

2) Code optimization - improves intermediate code
- by optimizing the code, overall running of program can be improved.

3) Code generation - Generation of target code
- The intermediate code instruction are translated into machine level instructions

```
MOV   id3, R1
MOV   id2, R2
MOVL  R2, R1
MOV   id1, R2
ADD   R2, R1
MOV   R1, id1
```

Q2) Need of Intermediate code generation phase :



- If a compiler translate the source lang. to its target machine without generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eleminates the need of new full complex for every unique machine by keeping analysis portion same for all the compilers.
- It becomes easier to apply the source code modification to improve code performance by applying code optimization techniques one the intermediate code.

→ Ways to implement three address code

1) Quadruple:
- It is structure which consists of 4 fields; OP, arg1, arg2 & result.
- OP denotes operator
- arg 1 & arg 2 are two operands
- Result is used to store result of the expression.

2) Triples:
- In this when a reference to another triple's values is needed, a pointer to that triple is used.
- So it consists of only three fields: OP, arg1 & arg 2

Q3) Conditions for an grammer to be LL(1)
i) if grammer is free from E productions, then
a is LL(1) if $A \to^* \alpha_1 | \alpha_2 | \ldots$
first(α_1) ∩ first(α_2) ∩ first(α_3) ... = ϕ
ii) if ϕ has G productions
$A \to \alpha | G$
if first(α) ∩ follow(A) = ϕ
then givin grammer is LL(1)
iii) if G has only one alternative on R.H.S variable then G is LL(1)
ex: $A \to \alpha$
$B \to \gamma$
$C \to \beta$

given grammer

$E \to TE'$
$E' \to +TE' | G$
$T \to FT'$
$T' \to *FT' | E$
$F \to id | (E)$

Symbol	first()	Follow
E	{(, id}	{$,)}
E'	{+, G}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, G}	{+, $,)}
F	{(, id}	{*, +, $,)}

Q5) Symbol table is an important data structure created & maintained by the compilers in order to keep of semantics of variable i.e. it stores information about the scope and binding information about the scope & names.

Information stored in symbol table are:
- variable names and constants
- procedure and function names
- literal constants & strings
- compiler generated temporaries
- labels in source languages
- associated attributes with identifiers.

→ Data structure used to implement symbol tabale are:-

1) List - In this method an array is used to store names & associated information

pros:- insertion is fast O(1)
- it takes minimum amount of space

cons:- Lookup is slow for large tables O(n)

IRDCE115

2) Binary search tree - In this generally we add two link field i.e. left & right child.

Pros :- Can grow dynamically
Cons :- insertion & lookup are $O(\log_2 n)$ on average.

3) Hash Tree - In this an array is used with an index range

Pros :- Insertion & lookup can be made very fast $O(1)$
- quick to search is possible
Cons :- hashing is complicated to implement
- average time complexity $O(n)$

4) Linked list - this implementation is using linked list by adding link field to each record.

pros :- insertion is fast $O(1)$
Cons :- lookup is slow for large tables $O(n)$ on average.

Q7) LL(1) grammer must be unambiguous + No left recursive + left factored.

If G is a grammer free from e-productions then G is LL(1) if $\forall A \to \alpha_1 | \alpha_2 | \ldots$

$S \to \overset{\alpha_1}{Aa} | \overset{\alpha_2}{bAc} | \overset{\alpha_3}{Bc} | \overset{\alpha_4}{bBa}$

first (α_1) = first (Aa) = $\{a\}$

first (α_3) = first (Aa) = $\{a\}$

$\therefore$ first $(\alpha_1) \cap$ first $(\alpha_3) \neq \phi$

$\therefore$ Given grammer is not LL(1)

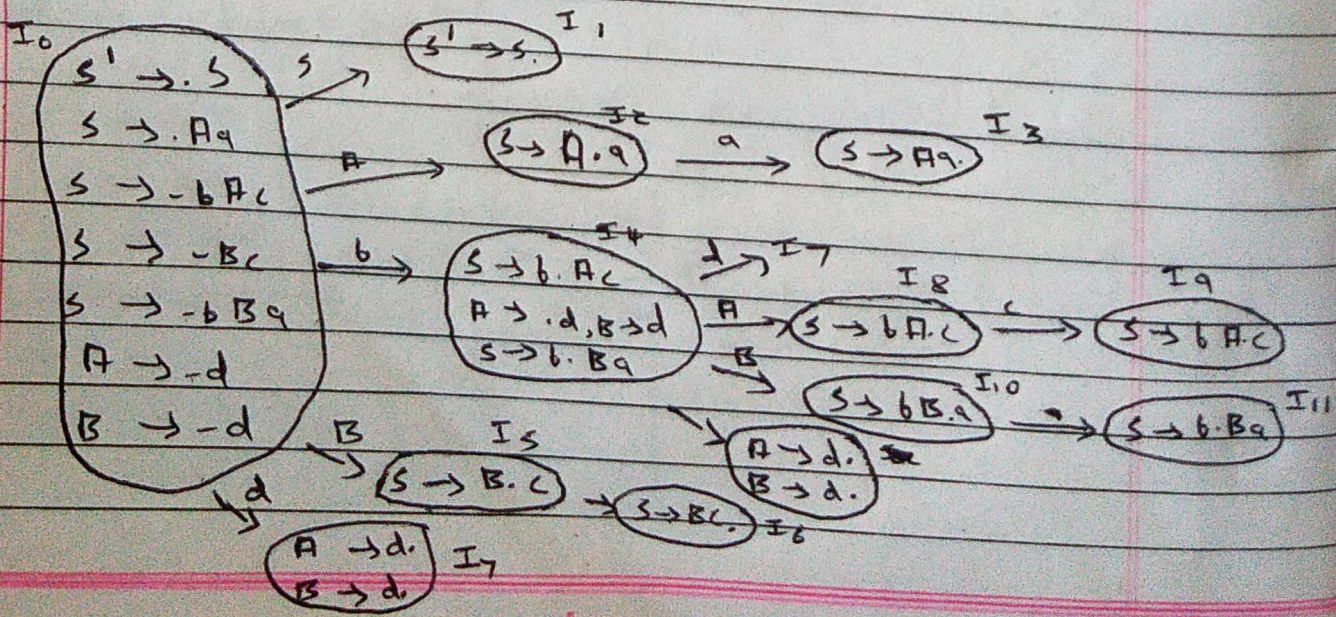
Grammer is SLR if SLR() parse table has no conflicts.

G : $S' \to S$

$S \to Aa | bAc | Bc | bBa$

$A \to d$

$B \to \lambda$



18DCE115

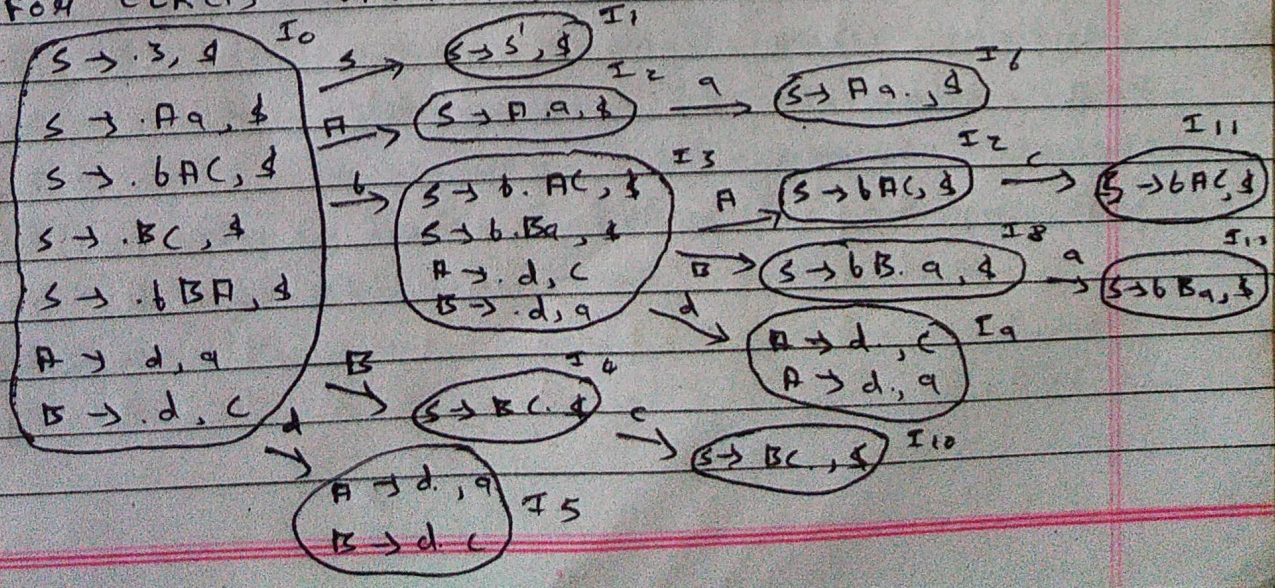
PAGE NO.:
DATE / /

Parse table for SLR()

	a	b	c	d	$	S	A	B
I_0		S_4				1	2	3
I_1					Accept			
I_2	S_3							
I_3					γ_1			
I_4				S_7			8	10
I_5			S_6					
I_6					γ_3			
I_7					γ_5/γ_6			
I_8								
I_9								
I_{10}								
I_{11}								
I_{12}								

Here we got R-R conflict, so grammer is not in SLR(1)

=> For CLR(1) Grammer



	a	b	c	d	$	S	A	B
I_0		S_3		S_4		1	2	4
I_1					Accept			
I_2	S_6							
I_3				S_9			7	8
I_4			S_{10}					
I_5	γ_5		γ_6					
I_6								
I_7			S_{11}					
I_8	S_{10}							
I_9	γ_6		γ_5					
I_{10}				γ_3				
I_{11}				γ_2				
I_{12}				γ_4				

In CLR(1) we get not any S-R / R-R conflict
∴ Given grammer is CLR(1) Grammer.

Q10) Basic block - it is a sequence of three address statements where control enters at the beginning and leaves only at the end without any jumps or halts

Three address code of c code

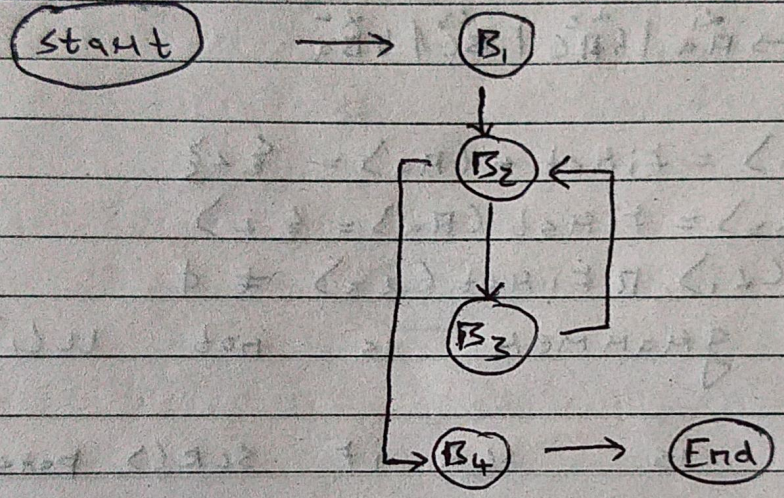
```
1   F=1;
2   i=2;
3   if (i<x) goto(9);
4   t1 = F*i;
5   F=t1;
6   t2 = i+1;
7   i= t2;
8   goto (3);
9   goto calling program
```

Basic blocks

```
F =1;*
i =2                    ] B1
if (i>x) goto (9)*      ] B2
t1 = F*i;*
F = t1;
t2 = i+1;
i = t2;
```

goto (3)
goto calling program*] B4

Control flow graph :



Q13) The main function of linker is to generate executable files. The linker takes input of object code generated by compilers / assemblers. It is a tool that merges the object files proceed by seperate compilation or assembly & creates an executable.

Whereas the main function of loader is to load executable file to main memory. Also the loader takes input of executable files generated by linker.

=> There are basically four types of loaders:-

1) Absolute loader : It is a simplaced type of loader. In this simply accepts the machine language code produced by assembler & place it into main memory at the location specified by assembler.

2) Bootstrap loader : When a computer is first turned on or restarted a special type of absolute loader is executed, called bootstrap loader.

It loads the operation system into the main memory and executes related program. It is added to the beggining of all object programs that are to be loaded into empty ideal system.

3) Relocating loader : To avoid possible reassembling of all sub-routines, it is changed to perform the task of allocation & linking of programm-er, so relocating loader was introduced Relocation provides efficient sharing of machine with larger memory & when several independent programs are to be run together.

4) Direct linking loader : It is general relocatable loader & most popular loading scan presently used. It allows programmer multiple procedure segments & multiple data segments. It gives programmer complete freedom in referencing data or instructions content in other segments.

Date :
Page No.

Q4) The error handler in a parser has the following goals :-

- Report the presence of errors clarity & accurately.
- Recover from each error quick enough to detect subsequent errors.
- add minimal over head to the processing of correct programs.

=> Error recovery strategies used by parser :-

i) Panic Mode Recovery - On discovering an error that parser discards input symbols one at a time until one of a designated set of synchronizing tokens are found while it often skips a considerable amount of input, it has the advantage of simplicity.

ii) Phrase level Recovery - On discovering an error, parser may perform local correction on the remaining input, it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which actual error has occured before the point of detection.

iii) Error Production - A parser constructed from a grammer augmented by common error detects the anticipated errors diagnostic about the erroneous constructs that has been recognized in the input.

Q6) Because of large amount of time consumption in moving characters, specialised buffering techniques have been developed to reduce the amount of overhead required to process in input characters.

=> The techniques are as follows:

1) Scheme - Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-number of characters on one disk block, eg. 4096
- N-characters are read from the input file to the buffer using one system read command.
- eof is inserted at the end if the number of characters is less than N.

2) Pointers - Two pointers lexemeBegin and ~~Proceed~~ forward are maintained
- lexemeBegin points to the beginning of the current lexeme which is get to be found.
- Forward scans ahead until a match for a pattern is found.
- Once a lexeme is found, lexemBegin is set to the characters immediately after the lexeme which is ~~part~~ first found and forward is set

to the characters at its right end.
- Current lexeme is the set of two pointers.

Date :
Page No.

Q12) The runtime environment is the environment in which a program or application is executed.
It is the hardware & software infrastructure of a particular codebase in real time

=> Memory allocated during compile time means the compiler resolves at compile time where certain things will be allocated inside the process memory map.
For example, consider a global array
int array [100];

- The compiler knows at compile time the size of array & the size of int, so knows the entire size of the array at compile time
- Also a global variable has static storage duration by default it is allocated in static memory area of the process memory space
- The compiler decides during compilation in what address of that static memory area the array will be.

=> Limitations of stack based memory allocation
- Each process is allocated a fixed amount of stack space.

- Amount of memory available through stack based allocation is commonly far less

Date :
Page No.

than the amount of memory available through heap-based memory allocation

- once you are out of stack-based variable memory your process will crash.