Introduction to language processing, Structure of a Compiler, Evaluation of programming language, The science of building a compiler, Applications of compiler technology, Programming language basics.

Lexical analysis – Role of lexical analysis, buffering, specifications of tokens, Recognition of tokens, lexical analyzer generator.

Language processing

1. Q: Define Language processor. Give and explain the diagrammatic representation of a language processing system.

Q: Explain briefly the need and functionalities of linkers, assemblers and loaders.

Q: Mention the functions of linkers and loaders in preprocessing.

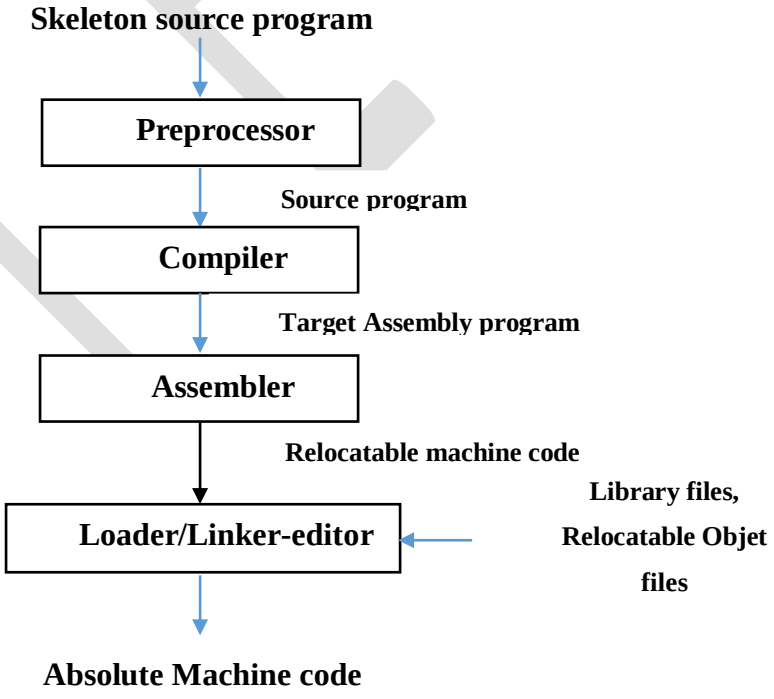
Q: Describe functionality of compilers in language processing.

Q: What are the functions of preprocessing?

Language processor –

An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

Language processing system:



Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. ***Macro processing:*** **A preprocessor may allow a user to define macros that are shorthands for longer constructs.**

   Eg: #define PI 3.14

   Whenever the PI is encountered in a program, it is replaced by the value 3.14
2. ***File inclusion:*** **A preprocessor may include header files into the program text.**

   Eg: #include<stdio.h>

   By this statement, the header file stdio.h can be included and user can make use of the functions in this header file. This task of preprocessor is called file inclusion.
3. ***Rational preprocessor:*** **these preprocessors augment older languages with moremodern flow-of-control and data structuring facilities.**
4. ***Language Extensions:*** **These preprocessor attempts to add capabilities to the languageby certain amounts to build-in macro**

Compiler

- **Compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language;**
- **If some errors are encountered during the process of translation, then compiler displays them as error messages.**
- **The basic model of compiler can be represented as follows:**

- **The compiler takes the source program in high level language such as C, PASCAL, FORTRAN and converts into low level language or machine level language such as assembly language.**

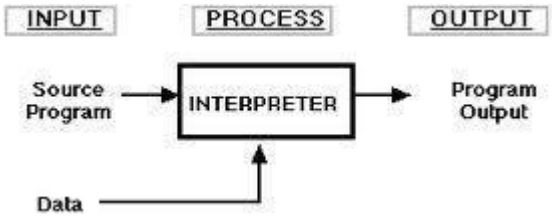
Assembler

- **Programmers found difficult to write or read programs in machine language.**
- **They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language.**

- **Programs known as assembler were written to automate the translation of assembly language into machine language.**
- **The input to an assembler program is called source program, the output is a machine language translation (object program).**

Interpreter

- **Interpreter is a program that appears to execute a source program as if it were machine language.**



- **Languages such as BASIC, SNOBOL, LISP can be translated using interpreters.**
- **JAVA also uses interpreter. The process of interpretation can be carried out in following phases.**
  1. **Lexical analysis**
  2. **Syntax analysis**
  3. **Semantic analysis**
  4. **Direct Execution**

Loader/Linker – editor

- **Loader is a program which performs two functions, loading and link editing.**
- **Loading is a process in which the relocatable machine code is read and the relocatable addresses are altered.**
- **Then that code with altered instructions and data is placed in the memory at proper location.**
- **The job of link editor is to make a single program from several files of relocatable machine code.**
- **If code in one file refers the location in another file, then such a reference is called external reference.**
- **The link editor resolves such external references also.**

2 Q: Differentiate between Compilers and Interpreters.

Compiler	Interpreter
1. It checks the entire high level program at once.	1. It checks one statement at a time.
2. If the program is error free, it translates the program into object program which is to be executed by interpreter.	2. If the program is error free, it executes the program and continuous till the last statement.
3. The translation process is carried out by a compiler. So it is termed as compilation.	3. The translation process is carried out the interpreter. So it is termed as interpretation
4. It process the program statements in their physical input sequence.	4. It process according the logical flow of control through the program.
5. Processing time is less.	5. Processing time high.
6. Compilers are larger in size and occupy more memory.	6. Interpreters are smaller than compilers.
7. It is also called software translation.	7. It is also called software simulation.
8. Eg: C, C++, FORTRAN, PASCAL etc	8. Eg: COBOL, LISP, Smalltalk etc

3Q: Describe the phases of compiler. Write the output of all phases for the following statement, position := initial + rate * 60

Q: Describe analysis-synthesis model of compilation.

Q: Explain the structure of compiler.

- **A compiler operates in phases. A phase is a logical interrelated operation that takes the source program in one representation and produces output in another representation.**
- **They communicate with error handlers and symbol table.**
- **There are two major parts of compilation.**
  1. **Analysis (Machine Independent / Language dependent)**
  2. **Synthesis (Machine Dependent / Language independent)**

<u>Analysis</u>

The analysis part consists of three phases.

1. **Lexical analysis or Linear analysis or Scanning.**
2. **Syntax analysis or Hierarchical analysis or Parsing.**
3. **Semantic analysis.**

Synthesis

The synthesis part consists of three phases.

4. **Intermediate code generation.**
5. **Code optimization.**
6. **Code generation.**

- **The phases of compiler are shown in the below figure.**

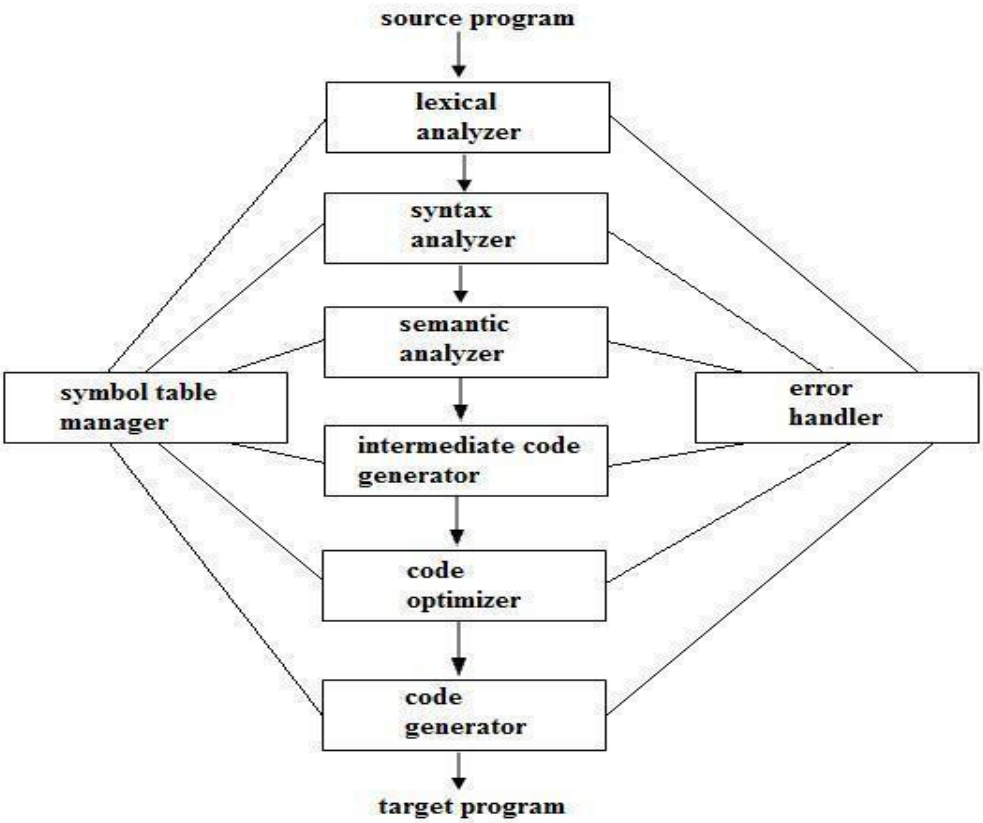


Fig 1.5 Phases of a compiler

Lexical analysis

- **Lexical analyzer reads the source program character by character and groups them into a stream of tokens.**
- **Consider the statement, position := initial + rate * 60**

  When lexical analyzer finds the identifier position, then it generates a token say id. Hence, the output of lexical analysis is given by,

  id1 := id2 + id3 * 60

  where id1, id2 and id3 are the tokens for position, initial and rate respectively.

Syntax analysis

- **In this phase, the tokens generated by the lexical analysis are grouped together to form a hierarchical structure.**
- **It is called parse tree or syntax tree.**
- **A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes and the operands of an operator are the children.**
- **For the input string, id1 := id2 + id3 * 60 the parse tree can be generated as follows:**

Semantic analysis

- **Type checking is an important part of semantic analysis. It checks all the operands in an expression are type compatible or not.**
- **Semantic analysis checks the source program for semantic errors.**
- **Suppose all the identifiers are declared as real but 60 is integer. So it is converted into real using inttoreal operator. The output is semantic tree.**

Intermediate code generation

- **This representation is easy to produce and easy to translate into the target program.**
- **This phase transforms the parse tree into an intermediate language representation of the source program.**
- **One of the popular type of intermediate language is called three address code.**
- **The output of intermediate code generation is given by**

  temp1 := inttoreal(60)

  temp2 := id3 * temp1

  temp3 := id2 + temp2

  id1 := temp3

Code optimization

- **It improves the intermediate code. It reduces the code by removing the unwanted instructions from the intermediate code.**
- **This is necessary for faster executing code or less consumption of memory.**
- **The above intermediate code can be optimized into the following code.**

  temp1 := id3 * 60.0

  id1 := id2 + temp1

- **Compiler can reduce the conversion of 60 from integer to real. So, inttoreal operation can be eliminated.**

Code generation

- **In this phase, the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.**
- **The above optimized code can be written using registers.**

```
MOVF id3, R1
MULF #60.0, R1
MOVF id2, R2
ADDF R2, R1
MOVF R1, id1
```

- **The "F" in each instruction instructs to deal with floating point numbers.**

Symbol table

- **Symbol table is data structure used to store the information about identifiers in the program.**
- **The symbol table also stores the information about datatype, its scope and information about storage of all identifiers.**

Error handlers

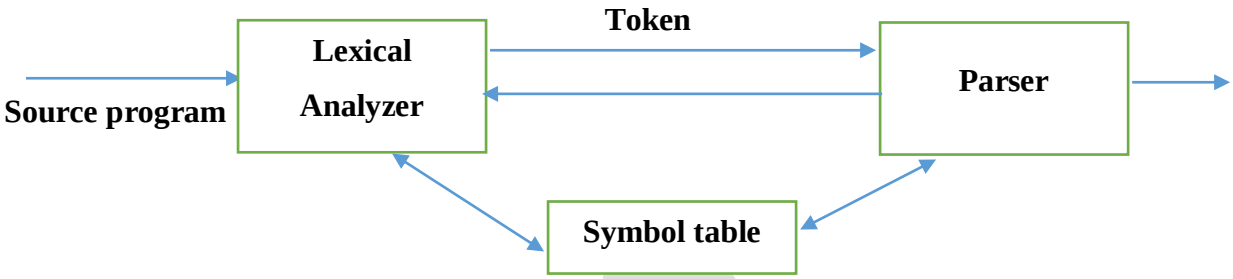
- **Error handler is invoked when a fault in the source program is detected.**
- **In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that compilation can proceed.**

4 Q: Explain the role of lexical analysis.

Role of lexical analysis

- **Lexical analysis is the first phase of compiler.**
- **It reads the source program from left to right character by character and generates the sequence of tokens as output.**
- **A token describes the pattern of characters having the same meaning in the source program. Token may be identifiers, operators, keywords, numbers, delimiters and so on.**
- **Lexical analyzer puts information about identifiers into the symbol table.**
- **Regular expressions are used to describe the tokens.**
- **A finite state machine is used in the implementation of lexical analyzer.**

- **Then the parser determines the syntax of source program. After receiving "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.**
- **The role of lexical analysis in the process of compilation is shown below:**



- **Apart from token identification, lexical analyzer performs the following functions:**
  1. **It eliminates blank spaces and comments.**
  2. **It communicates with symbol table which stores the information about identifiers, constants occurred in the input.**
  3. **It keeps track of line numbers.**
  4. **It reports errors while generating the tokens.**

5 Q: Explain Lexical analysis Vs Parsing

Lexical analysis	Parsing
1. It is one of the phase in compilation process in which the stream of tokens is generated by scanning the source code.	1. It is one of the phase in the compilation process in which the stream of tokens is obtained from lexical analysis phase for building the parse tree.
2. It is also recognized as scanning phase.	2. It is also recognized as syntax analyzing phase.
3. The input buffering scheme is used for scanning the source code.	3. The top down and bottom up parsing techniques are used for syntax analysis.
4. The regular expressions and finite automata are used in the design of lexical analysis.	4. The context free grammars are used in the design of parsing.
5. The LEX is an automated tool which is used to generate lexical analyzer.	5. The YACC tool is an automated tool which is used to generate syntax analyzer.

6 Q: Explain about tokens, patterns and lexemes with example.

- **Lexical analysis deals with the terms tokens, patterns and lexemes.**
- **A token describes the pattern of characters having the same meaning in the source program. Token may be identifiers, operators, keywords, numbers, delimiters and so on.**
- **Pattern is a set of rules that describes the token.**
- **For example, the following are the rules for the valid identifiers.**
  1. **The identifier must begin with alphabet.**
  2. **The only allowed special character in identifier name is underscore. eg: total_amout**
  3. **The identifier name should not contain any blank spaces. Example: int total amount is invalid identifier.**
  4. **The identifier name must not be a reserve keyword.**
- **Lexeme is a sequence of characters in the source program that are matched with the pattern of token. For example, int, num, choice.**
- **The following is the example which differentiates lexemes and tokens.**

```
int MAX(int a, int b)
{
        if( a > b )
                return a;
        else
                return b;
}
```

Lexeme	Token
Int	Keyword
MAX	Identifier
(	Operator
Int	Keyword
A	Identifier
,	Operator
int	Keyword
B	Identifier
)	Operator
…	…

- **The blank spaces and new line characters can be ignored.**
- **These stream of tokens will be given to the syntax analyzer.**

7 Q: Briefly explain the attributes for tokens.

- **Lexical analyzer provides additional information to distinguish between the similar type of patterns that match the lexeme.**
- **For example, a pattern "num" match both strings 0 and 1. But the code generator has to know what it matches exactly.**
- **Lexical analyzer collects the attributes of tokens as the additional information.**
- **For example, the tokens and associated attribute values for the FORTRAN statement**

  E = M * C ** 2 is given below

  <id, pointer to the symbol entry for E>

  <assign_op>

  <id, pointer to the symbol entry for M>

  <mult-op>

  <id, pointer to the symbol entry for C>

  <exp_op>

  <num, integer value 2>

8 Q: Briefly explain about lexical errors.

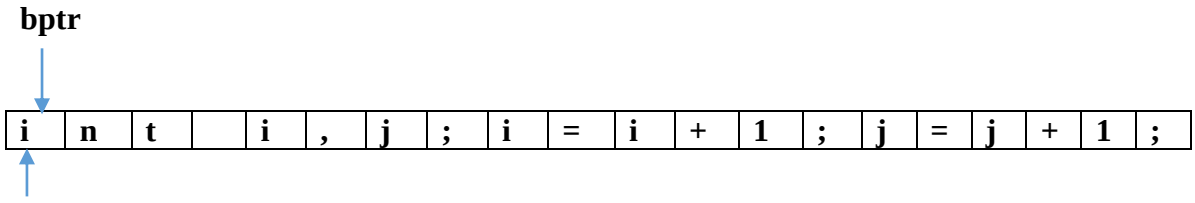
- **If the lexical analyzer discovers the prefix which is not matching the specification of any token, it invokes an error to get the remedial actions.**
- **For example, the string if is encountered as fi in a C program,**

  fi (a = = b)
- **A lexical analyzer cannot inform whether fi containing spelling mistake of the keyword.**
- **Under some situations, the lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input.**
- **The simplest recovery is to delete the successive characters from the remaining input until the lexical analyzer can find a well formed token.**

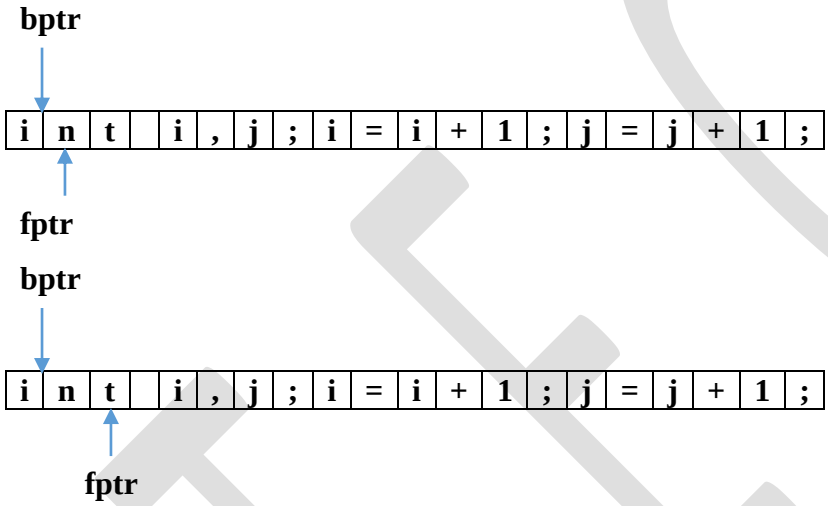
9 Q: Explain briefly about input buffering

- **Lexical analyzer scans the input string from left to right one character at a time.**
- **It uses two pointers, begin_ptr (bptr) and forward_ptr (fptr) to keep track of the portion of the input scanned.**

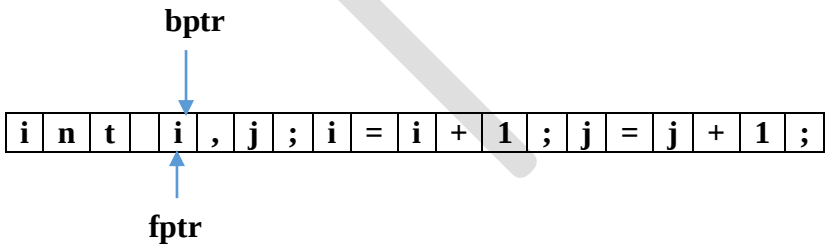
- **Initially both the pointers point to the first character of the input string which is shown below with the source code inti,j; i = i +1; j = j + 1;**



- **The fptr moves ahead to search for the end of lexeme. As soon as the blank space is found, it indicates the end of lexeme. In the above example, as soon as fptr found a blank space, the lexeme "int" is identified.**



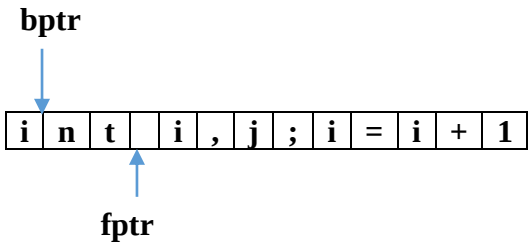
- **When fptr finds white space, it ignore and moves ahead. Then both bptr and fptr are set at next token "i".**



- **Thus, the input character is scanned from secondary storage. But reading in this way from secondary storage is costly. Hence buffering technique is used.**
- **A block of data is first read into a buffer and then scanned by lexical analyzer.**
- **There are two methods used, one buffer scheme and two buffer scheme.**

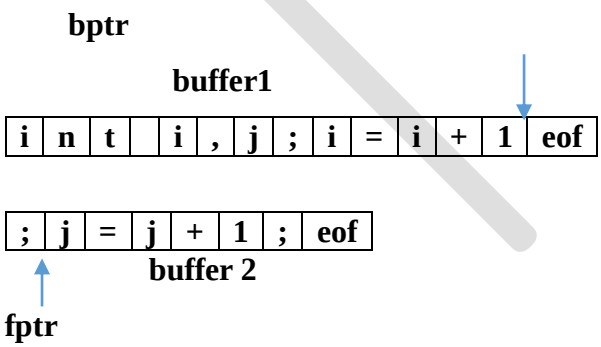
One buffer scheme:-

- **In this scheme, only one buffer is used to store the input string.**
- **But the problem with this scheme is that if lexeme is very long, then it crosses the buffer boundary.**
- **To scan the remaining lexeme, the buffer has to be refilled that makes overwriting the first part of lexeme.**



Two buffer scheme:-

- **To overcome the problem of one buffer scheme, two buffers are used to store the input string.**
- **The first and second buffers are scanned alternatively.**
- **When fptr finds EOF of current buffer, filling up of second buffer is started. In the same way, when second EOF is obtained, then it indicates the end of second buffer.**
- **Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. The EOF is called sentinel which is used to identify the end of buffer.**



10 Q: Explain briefly about Token specification.

- **Token type and its attribute uniquely identifies a lexeme.**
- **Regular expressions are used to specify tokens.**

Strings and languages

- **An alphabet is a fine non-empty set of symbol denoted by ∑**
  Eg: ∑ = { 0, 1 } represents binary alphabet.
- **String is finite sequence of symbols on an alphabet. € is an empty string.**

Eg: If ∑ = { 0, 1 } then possible strings are 0, 1, 00, 11, 01, 10, …

- **Language is a set of strings over some alphabet.**

Operations on languages

- **Union – If L1 and L2 are languages then L1 U L2 = { S / S € L1 or S €L2 }**
- **Concatenation – L1.L2 = { S1S2 / S1 € L1 and S2 €L2 }**
- **Kleen Closure – If R1 is a regular expression then $R = R^*$ is also a regular expression which represents Kleen closure.**

Regular grammar

- **Regular grammar G is defined as**

  G = (V, T, P, S) where

  V = Set of variables or non-terminals

  T = Set of terminals

  P = Set of productions

  S = Start symbol

- **For example,**

  A→aB

  B → b | €

  Then the grammar G is defined as

  V = { A, B }

  T = { a, b }

  P = { A→aB , B →b | € }

  S = { A }

- **Regular grammar is a grammar which can be represented by using finite automata.**

Regular Expressions (RE)

- **RE is used to describe tokens of a programming language.**
- **RE is defines as follows:**
  1. **is a RE denoting an empty language**
  2. **€ (epsilon) is RE denoting a language which has an empty string.**
  3. **‘ a ‘ is RE denoting a language containing only { a }**
  4. **If R and S are RE, then L_R and L_S denote language for RE R and S. Then:**
     - i. **R + S is RE for language L_R U L_S**
     - ii. **R . S is RE for language L_R . L_S**
     - iii. **R^* is RE for language L^*_R**

11 Q: Explain the procedure for recognition of tokens.

Using token type and Token value:

- **For a programming language, there are various types of tokens such as identifier, keywords, constants and operators and so on.**
- **The token is usually represented by a pair of token type and token value.**

Token type	Token value

- **The token type tells us the category of token and token value gives us the information regarding token. The token value is also called token attribute.**
- **The token value can be a pointer to symbol table in case of identifier and constants.**
- **The lexical analyzer reads the input program and uses symbol table for tokens.**
- **For example, consider the following symbol table,**

Token	Code	Value
if	1	-
else	2	-
while	3	-
for	4	-
identifier	5	ptr to symbol table
constant	6	ptr to symbol table
<	7	1
< =	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
-	9	2
=	10	-

- **Consider a program code as,**

```
if ( a < 10 )
    i = i + 2;
else
    i = i – 2;
```

- **The corresponding symbol table for identifiers and constants will be,**

Location counter	Type	Value
100	identifier	a
…	…	…
105	constant	10
…	…	…
107	identifier	b
…	…	…
110	constant	2

- **Our lexical analyzer will generate the following token stream**
  1, (8, 1), (5, 100), (7, 1), (6 , 105), (8, 2), (5, 107), 10, (5, 107), (9, 1) (6, 110), 2 , (5, 107) , 10, (5, 107), (9, 2), (6, 110)

Using transition diagram

- **The transition diagram is a directed graph with states drawn as circles (nodes) and edges representing transitions on input symbols. Edges are the arrows connecting the states.**
- **The transition diagram has a start state which indicates the beginning of token and final state which indicates the end of token.**
- **The fptr scans the input character by character from left to right.**
- **For example, Write a regular expression for identifier and keyword and design a transition diagram for it.**
  Ans: RE = letter (letter + digit) *

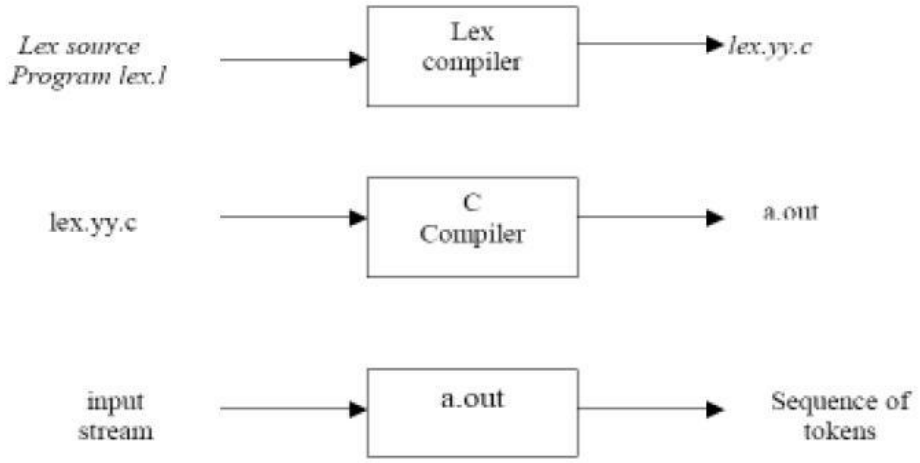
- **The installID ( ) checks if lexeme is already in table. If it is not present then it will install it. The gettoken( ) examines the lexeme and returns the token name as id or a reserve keyword.**

12 Q: Explain LEX lexical analyzer generator.

- **Basically LEX is a unix utility which generates lexical analyzer.**

Creating a lexical analyzer with LEX

-



The LEX specification file can be created using the extension .l (dot l) . For example, lex.l file is given to LEX compiler to produce lex.yy.c

This lex.yy.c is a C program which is actually a lexical analyzer program. The LEX specification file stores the regular expressions for the tokens and the lex.yy.c file consists of tabular representation of the transition diagram constructed for regular expression.

Finally lex.yy.cis run through the C compiler to produce object program a.out which is the lexical analyzer that transforms input stream into a sequence of tokens.

LEX specification

- **LEX program consists of three parts**

  % {

  Declaration section

  % }

  % %

  Translation rules section

  % %

  Auxiliary procedures section

- **The *declarations* section includes declarations of variables, constants and regular definitions.**

- **The *translation rules* of a Lex program are statements of the form :**

  p1 ***{action 1}***

  p2 ***{action 2}***

  p3 ***{action 3}***

  … **…**

  … **…**

  where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

- **The third section holds whatever *auxiliary procedures* are needed by the *actions. This* means, if any actions requires procedure, then that procedure will be defined.**

13 Q: Explain the evolution of programming languages

- **Programming language is helpful in describing what its programs look like (Syntax of the language) and what its program meaning is (Semantic of the language).**
- **The first electronic computers evolved in 1940. They performed very less operations. They were programmed in machine language by sequence of 0's and 1's**
- **The initial step towards development of high level languages started during late 1950's with the development of FORTRAN, COBOL and LISP.**
- **Programming languages can be classified into the following ways:**

I generation languages	**Machine language**
II generation languages	**Assembly language**
III generation languages	**HLL like FORTRAN, COBOL, LISP, C, C++**
IV generation languages	**Languages for specific applications like SQL for DB queries**
V generation languages	**OPS5**

14 Q: Briefly explain the science of building a compiler

- **Building a compiler is a challenging task.**
- **The main job of compiler is to accept the source program of any size and convert into suitable target program.**
- **Compilers mainly focus on how to design the correct mathematical model and choose correct algorithms, keeping generality and efficiency.**
- **Finite state machines and regular expressions are used to describe the tokens during lexical analysis of a compiler.**
- **The objectives of building a compiler is given below:**
  1. **"The Meaning" of the compiled program must be preserved.**
  2. **Optimization should improve programs performance.**
  3. **Time required for compilation should be reasonable.**
  4. **Management of engineering effort is a must.**

15 Q: Explain the applications of compiler technology.

<u>1. Implementation of High level programming languages:</u>

- **In HLL, the algorithm is expressed by the programmer using a language, and the job of compiler is to translate the program into the target language.**

- **Lower Level Language (LLL) is hard to write and is more error prone. HLL are easy to write but are less efficient. LLL produce more efficient code.**
- **The task of optimizing compilers is to improve the performance of the generated code and offsetting the efficiency produced by high level abstractions.**
- **Thus compiler optimizations have been developed to reduce the overhead.**

2. Optimizations for Computer Architecture:

- **There is a demand for new compiler technology due to rapid evolution of computer architectures. This leads to parallelism and memory hierarchies.**

Parallelism

- **Parallelism can exist as following levels:**
- **Instruction level includes execution of multiple operations.**
- **Processor level. Here different threads of the same applications run on the different processors.**

Memory hierarchies

- **They are found in all machines. They consist of several levels of storage with different speeds and sizes.**
- **Memory hierarchy can be improved by changing the layout of the data or by changing the order of instructions accessing the data.**

3. Program translation:

- **We already know that compiler converts HLL into machine level. The same technology can be applied to translate between different kinds of languages. Few important applications of program translation techniques include :**

Binary translation – Here the binary code for one machine can be translated to another, allowing a machine to run program that was originally compiled for another instruction set.

Hardware synthesis–Hardware – synthesis tools translate register transfer level (RTL) descriptions automatically into gates, which are then mapped to transistor and finally to physical layout.

Database query interprets – Database queries consist of predicates containing relational and Boolean operator. They can be compiled into commands to search the database for records satisfying that predicate. Eg: Structured Query Language (SQL) is used to search database.

4. Software productivity tools

- **Dataflow analysis can find errors along all the possible execution paths.**
- **Dataflow analysis design warns the programmers of all those statements that are violating a particular category of errors.**
- **Static analysis has been developed to find errors.**
- **Therefore, optimizers should not alter the semantics of the program under any situation.**
- **Type checking, Bonds checking and memory management tools can be used in order to detect errors.**

16 Q: Write about programming language basics.

In this, important basics related to programming languages like compile time, run time, static binding, dynamic binding, static scope, keywords like public, private, protected, dynamic scope, parameter passing techniques – call by value, call by reference, call by name and finally aliasing. These concepts are based on C, C++ or Java.

17 Q: What is bootstrapping? Explain.

- **Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can handle even more complicated program and so on.**
- **Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.**
- **To clear understanding bootstrapping technique, consider the following scenario.**
- **Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is we create X_YZ.**
- **Now if existing compiler Y runs on machine M and generates code for another machine M then it is denoted as Y_MM,**
- **Now if we run X_YZ using Y_MM then we get a compiler X_MZ.**
- **That means, a compiler for source language X that generates a target code in a language Z and which runs on machine M**

- **Bootstrap includes the initialization of operating system (OS) of a computer.**

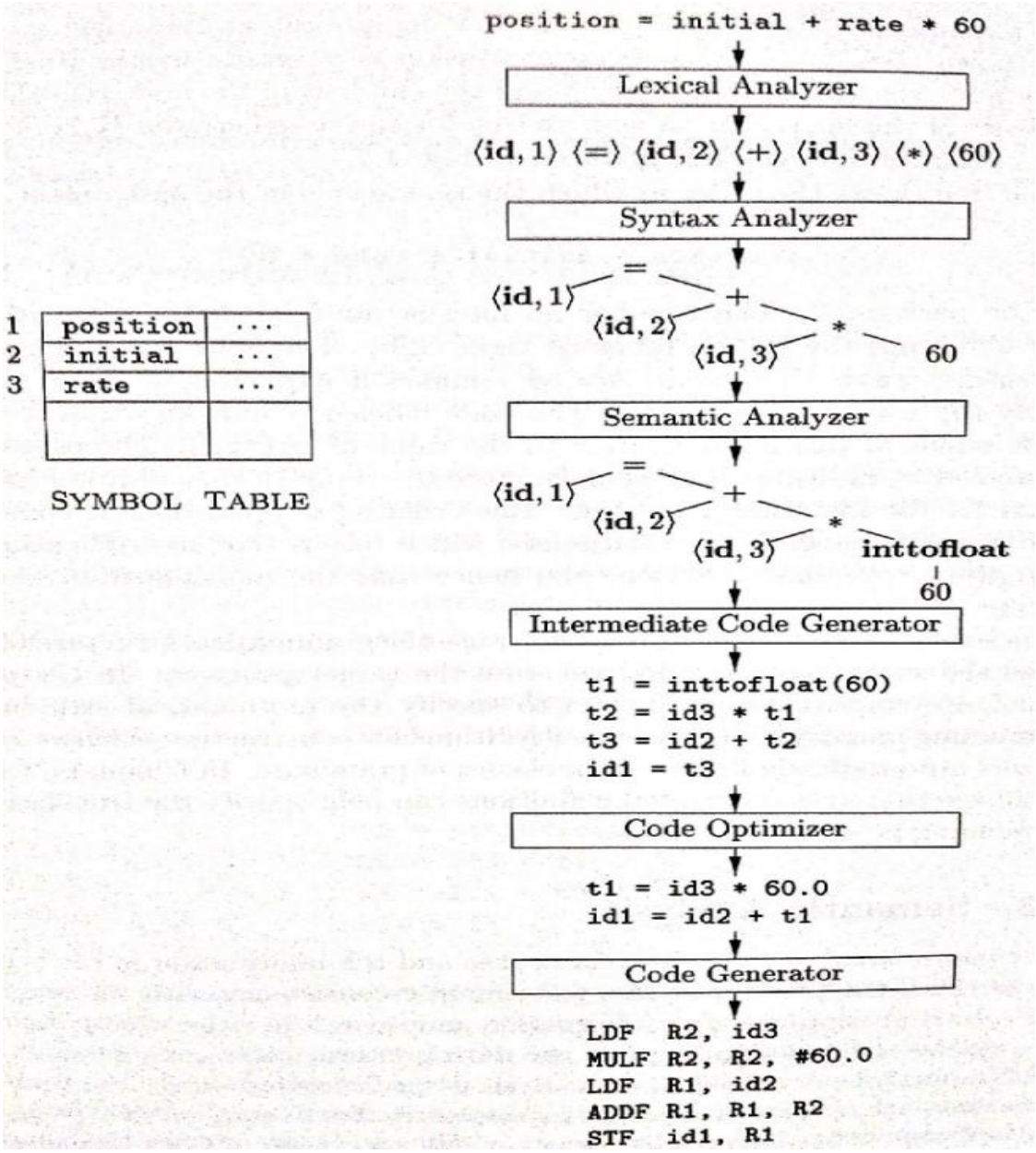
Advantages

- **It is easier to develop a compiler in the high level language being compiled, backend of a compiler can be improved.**
- **It is sufficient for a compiler developer to know only the language being compiled and it is non-trivial test of the language that is being compiled.**

18 Q: Write the differences between phases and pass.

Phase	Pass
1. Phase is a logically interrelated operation which takes the source program in one representation and produces output in another representation.	**1. The number of iterations to complete the execution is called pass.**
2. The process of compilation is carried out in various steps. These steps are referred as phases. The phases of compilation are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, code generation.	**2. Various phases are logically grouped together to form pass. The process of compilation is carried out in one pass or two pass.**
3. The phases such as lexical, syntax and semantic analysis are machine independent and language dependent. And the phases such as intermediate code generation, code optimization, code generation are machine dependent and language independent.	**3. Due to execution of program in passes, the compilation model can be viewed as front end and backend model.**

Q3: <u>Flow chart for phases of compiler</u>



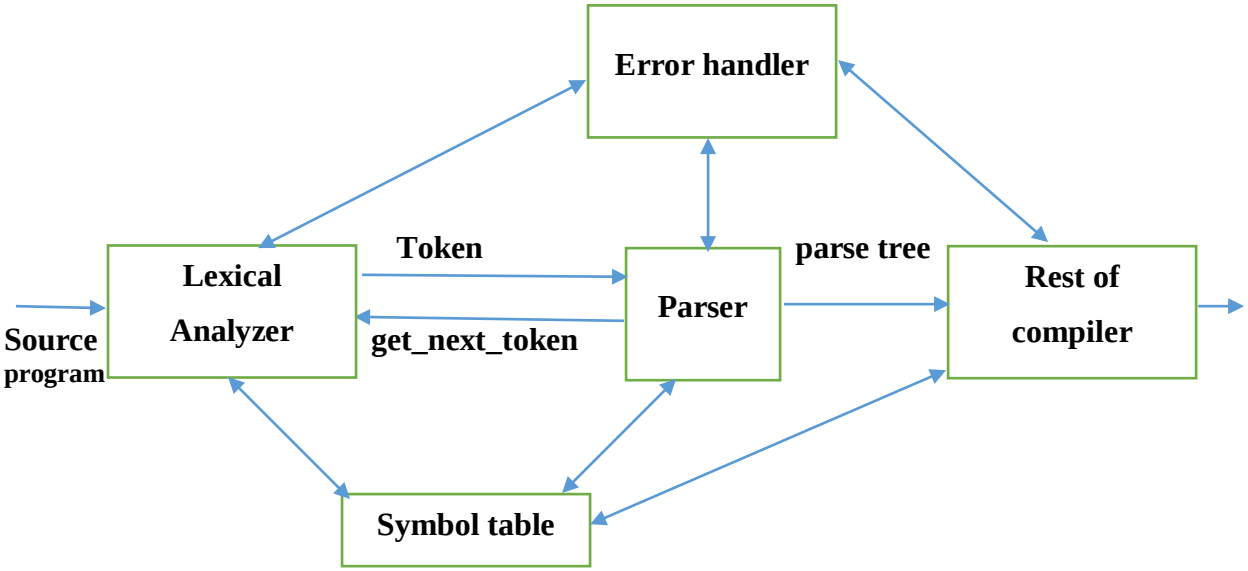
Syntax analysis – The role of a parser, Context free grammars, Writing a grammar, Top down parsing, bottom up parsing, Introduction to LR parser.

1 Q: Define parser. What is the role of a parser?

Def:

A parsing or syntax analysis is a process which takes the input string w and produces either a parse tree or generates the syntactic errors.

Role of a parser:



- **In the process of compilation, the parser and lexical analyzer works together. That means, lexical analyzer generates tokens to syntax analyzer.**
- **The parser collects sufficient number of tokens and builds the parse tree as output.**
- **The job of parser is to report syntax errors and to recover from commonly occurring errors to continue processing the remainder of the program.**
- **There are two types of parser, Top down parser and bottom up parser.**
- **When the parse tree is constructed from root to leaves, then it is said to be top down parsing.**
- **When the parse tree is constructed from leaves to root, then it said to be bottom up parsing.**

2 Q: What is Context Free Grammar (CFG)? Explain

Formal definition of CFG:

- **Context Free Grammar CFG is defined as**

  G = (V, T, P, S) where

V = Set of variables or non-terminals

T = Set of terminals

P = Set of productions

S = Start symbol

- **For example,**

  A→aB

  B →b | €

  Then

  V = { A, B }

  T = { a, b }

  P = { A→aB , B →b | € }

  S = { A }

Notational conventions:

1. The following symbols are terminals (T)

- **Lower case letters like a, b, c …**
- **Digits like 0, 1, 2, …**
- **Operators like +, -, *, / , …**
- **Punctuation symbols like parentheses, comma and so on.**
- **Strings like id.**

2. The following symbols are non-terminals / variables (V)

- **Upper case letters like A, B, C.**
- **Start symbol is S.**

3. The first symbol of the first production is the start symbol.

Derivations:

- **There are two types of derivations.**
  1. **Left Most Derivation (LMD) - If a production is applied at each step in a derivation to the left most variable, then the derivation is said to be left most.**
  2. **Right Most Derivation (RMD) - If a production is applied at each step in a derivation to the right most variable, then the derivation is said to be right most.**

Parse trees

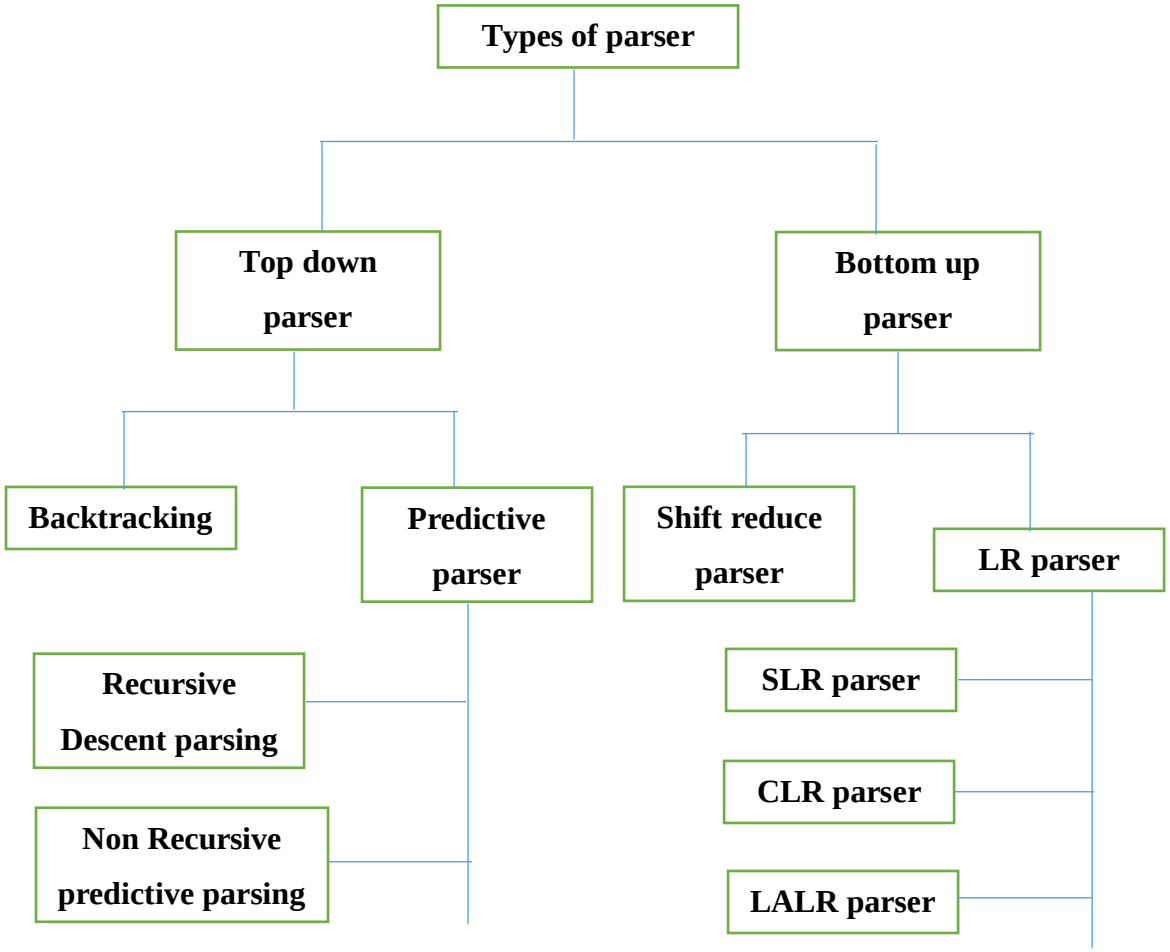
Parse trees are graphical representations of derivations. The interior nodes are labelled with non terminals and child nodes are labelled with terminals.

Ambiguity

A grammar is said to be ambiguous grammar if there are two or more possible left most derivations or right most derivations are constructed for the same input string.

3 Q: Explain about classification of parsing techniques.

- **There are two types of parser, Top down parser and bottom up parser.**
- **When the parse tree is constructed from root to leaves, then it is said to be top down parsing.**
- **When the parse tree is constructed from leaves to root, then it said to be bottom up parsing.**
- **These parsing techniques work on the following principle.**
  1. **The parser scans the input from left to right and identifies that the derivation is left most or right most.**
  2. **The parser makes use of production rules for choosing the appropriate derivation. The different parsing techniques use different approaches in selecting the appropriate rules for derivation. And finally a parse tree is constructed.**
- **The following diagram gives the classification of parsing techniques.**



4 Q: What is RD parser? Explain. What are its advantages and limitations?

Recursive Descent parser (RD parser)

A parser that uses a set of recursive procedures to recognize its input with no back tracking is called recursive descent parser.

It is a top down method of syntax analysis in which set of recursive procedures are executed to process the output. In this parser, the CFG is used to build the recursive routines.

Basic steps for the construction of RD parser:

1. **The RHS of the production rule is directly converted into program code symbol by symbol.**
2. **If the input symbol is a non-terminal, then call to the procedure corresponding to the non-terminal is made.**
3. **If the input symbol is a terminal, then it is matched with the lookahead symbol from the input. The lookahead pointer has to be advanced on matching of the input symbol.**
4. **If the production rule has many alternatives, then all these alternatives has to be combined into a single procedure.**
5. **The parser should be activated by a procedure corresponding to the start symbol.**

Advantages of RD parser:

1. **Recursive descent parsers are simple to build.**
2. **Recursive descent parsers can be constructed with the help of parse tree.**

Limitations or drawbacks of RD parser:

1. **Recursive descent parsers are not very efficient as compared to other parsing techniques. Because RD parses sometimes uses backtracking in few cases.**
2. **There are chances that the program for RD parser may enter in an infinite loop for some input.**
3. **RD parser cannot provide good error messaging.**
4. **It is difficult to parse the string if lookahead symbol is arbitrarily long.**

5 Q: Explain the process of eliminating ambiguity with example.

- **Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.**
- **For example, we shall eliminate the ambiguity from the following "dangling-else" grammar:**

  stmt*—>if *expr* then *stmt

  | if *expr* then *stm t*else *stmt*

| other

Here "other" stands for any other statement.

- **According to this grammar, thecompound conditional statement**

if*E1*then *S1*else if *E2* then *S2* else S3

- **The given grammar is ambiguous since the string if *E1* then S1 else if *E2* t hen S2 else S3 has the two parse trees shown below.**

First parse tree

Second parse tree

- **In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, "Match each else with the closest unmatched then".**

- **We can rewrite the dangling-else grammar as the following unambiguous grammar. The idea is that a statement appearing between then and else must be "matched";**

  stmt*→*matchedstmt

  | *openstmt*

  matchedstmt*→ if *expr* then *matchedstmt* else *matchedstmt

  | other

  ***openstmt*→ if *expr* then stmt**

  | if *expr* then *matchedstmt* else *openstmt*

6 Q: What is left factoring? Explain the process of eliminating left factoring with eg.

Left factoring

Left factoring is a grammar of transformations in which the common parts of two productions are isolated into a single production. It is suitable for predictive parsing.

Process of eliminating left factoring

- **Any production of the form A → $\alpha\beta_1 | \alpha\beta_2$ (where α is common) can be replaced by the following productions,**

  A → αA^1

  A^1→ $\beta_1 | \beta_2$

- **Left factoring is required because it is difficult to decide which production is to select either $\alpha\beta_1$ or $\alpha\beta_2$ to expand A.**
- **In left factoring, we defer this decision by expanding A to αA^1 until we have seen enough input to make the right choice.**
- **For example, consider CFG,**

  S →iEtS | iEtSeS

  E → b

  In this grammar, the common part is iEtS. Then after elimination of left factoring, the productions are

  S → $iEtSS^1$

  S^1 →eS | €

  E → b

7 Q: Explain the reasons for separating lexical analysis phase from syntax analysis.

The lexical analyzer scans the input program and collects the tokens from it. On the other hand, parser builds a parse tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages – Firstly it accelerates the process of compilation and secondly the errors in the source input can be identified precisely.

8 Q: What are the problems or difficulties in top-down parsing? Explain

The following are the problems.

1. **Backtracking.**
2. **Left recursion.**
3. **Left factoring.**
4. **Ambiguity.**

Backtracking

- **Def: Backtracking is a technique in which for the expansion of non-terminal symbol, we choose one alternative and if any mismatch occurs, and then we try another alternative.**
- **For example,**

  S → x P z

  P →yw | y

  then,

- **If for a non-terminal, there are multiple productions beginning with the same input symbol. We need to try all the alternatives to get the correct derivation.**
- **In backtracking, we need to move some levels upward inorder to check the possibilities. This increases lot of overhead in the implementation of parsing.**
- **Hence it is necessary to eliminate backtracking by modifying the grammar.**

Left recursion

- **Def: The grammar G is said to be left recursive if it has a non-terminal A such that there is a derivation A →Aα , for some α. Here means, deriving the input in one or more steps. A denotes non-terminal and α denotes some input string.**

- **If left recursion is in the grammar, then it causes a top down parser to go into an infinite loop. This is shown in the following figure.**

- **Thus, expansion of A causes further expansion of A and due to generation of A, Aα, Aαα, Aααα, … the input pointer will not be advanced. This causes major problem in top down parsing.**
- **Hence, it is necessary to eliminate left recursion by modifying the grammar.**

Left factoring

- **Basically left factoring is used when it is not clear that which of the two alternatives is used to expand the non-terminal.**
- **With left factoring, we may able to take a right decision by rewriting the productions.**
- **In general, if A → $\alpha\beta_1$ | $\alpha\beta_2$(where α is common) then it is not possible to take a decision whether to choose first rule or second.**
- **Hence, it is necessary to eliminate left factoring by modifying the grammar.**

Ambiguity

- **Def: The grammar G is said to be ambiguous grammar if it generates more than one parse tree for the sentence of a language L(G)**
- **The ambiguous grammar is not desirable in top down parsing.**
- **Hence it is necessary to eliminate ambiguity by modifying the grammar.**

9 Q: Explain about Handle pruning.

- **Handle is a substring of string w that matches the right side of production which when reduced to non-terminal on the left side of the production, represents one step of the reverse right most derivation.**

- **If there is a production A → β then β is said to be handle since it can be reduced to A in the string αβw.**
- **Reducing β to A in αβw is said to be pruning the handle. Handle pruning is nothing but reducing the handle by non-terminal which is towards the left of the production.**
- **Consider the grammar, E →E + E**

  E → id

  Now take the string id + id + id and the right most derivation is,

  E ----->E + E

  E -----> E + E + E

  E -----> E + E + id

  E -----> E + id + id

  E ----->id + id + id

  The underline strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	E → id
E + id + id	id	E → id
E + E + id	id	E → id
E + E + E	E + E	E →E + E
E + E	E + E	E →E + E
E		

- **Thus bottom up parser is essentially a process of detecting handles and using them in reduction. This process is called handle pruning.**

10 Q: Describe about Shift-Reduce parser

- **Shift reduce parser attempts to construct parse tree from leaves to root. Thus it works on the same principle of bottom up parser.**
- **Shift reduce parser requires the following data structures:**
  1. **The input buffer, storing the input string.**
  2. **Stack, for storing and accessing LHS and RHS of rules.**
- **The initial configuration of shift reduce parser is shown below:**

  Stack Input buffer

- **This parser performs following basic operations:**
  1. **Shift : Moving the symbols from input buffer to stack. This action is called shift.**
  2. **Reduce :If the handle appear on the top of stack then reduce it by appropriate rule. That means, RHS of the rule is popped and LHS is pushed in. This action is called reduce.**
  3. **Accept : If stack contains only start symbol and input buffer is empty at the same time then the action is called accept. When accept state is obtained in the process of parsing then it means a successful parsing is done.**
  4. **Error : A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as error.**

11 Q: Explain conflicts during shift reduce parsing.

- **When shift reduce parser is applied to some context free grammar, it leads to some conflicts. The conflicts are,**
  1. **Shift / Reduce conflict :**

     The parser even after knowing the entire stack contents and the next input symbol,

     cannot decide whether to shift or reduce. This is called as shift/reduce conflict.
  2. **Reduce / Reduce conflict :**

     The parser knowing entire stack contents and the next input symbol, cannot decide which reductions to use. This is called reduce/reduce conflict.
- **For example, consider dangling-else grammar,**

  stmt→ if expr then stmt

  | ifexpr then stmt else stmt

  | other

  where "other" stands for any other statements.
- **When shift reduce parser is applied to this grammar, we reach the configuration**

Stack	Input
… ifexpr then stmt	else … $

- **Under this configuration, we cannot tell whether "if expr then stmt" is handle or not. This leads the parser confusion whether to shift else or reduce the stack top element. Hence, there is a shift/reduce conflict.**

- **Another conflict occurs is when we have a handle but stack contents and the next input symbol are not sufficient to determine which production should be used in a reduction. This is called reduce/reduce conflict.**

12 Q: What are LR parsers? Explain briefly.

- **This is the most efficient method of bottom up parsing which can be used to parse large class of context free grammars. This method is also called LR(k) parsing. Here,**
  L stands for left to right scanning.
  R stands for right most derivation in reverse.
  k is number of input symbols. When k is omitted, k is assumed to be 1.
- **The LR(1) can also be written as LR.**
- **The following are the types of LR parsers.**
  1. **Simple LR (SLR) parser.**
  2. **Canonical LR (CLR) parser.**
  3. **Look Ahead LR (LALR) parser.**
- **The relationship among these parsers is expressed as,**
  SLR ≤ LALR ≤ CLR

<u>Structure of LR parser</u>

- **The structure of all these same. They consists of stack, input buffer, output stream, a driver program and a parsing table consists of two columns, action and goto.**

- **All LR parsers have the same driver program but the contents of the parsing table changes.**
- **The stack contains states in addition to the grammar symbols. The contents of stack are $S_0A_1S_1A_2S_2A_3\ldots A_mS_m$ where A is the grammar symbol and S is the state symbol.**

- **The driver program takes the current input from the input buffer and the state symbol on the top of stack to access the corresponding entry in the parsing table to determine the shift reduce decisions.**
- **The parsing table consists of two parts, action and goto. The entry action [S_m, a] in the action column contains one of the following values:**
  1. **Shift to state S_i.**
  2. **Reduce by production A → α of the grammar.**
  3. **Accept the input.**
  4. **Error recovery.**
- **The arguments to goto function is a state S_i on top of the stack and a grammar symbol A and returns the new state S_j.**

LR parsing algorithm

- **The LR parser driver program initializes the stack with S_0 and input buffer contains the input string as w$. The driver program takes the state S_m on top of the stack and current input symbol a_i and consult the parsing action table entry, action [S_m , a_i].**
- **Perform one of the moves depending upon the resulted action.**
  1. **If action [S_m , a_i] = S_i , it pushes a_i then S_i on top of the stack and advances the pointer to next input symbol and continues.**
  2. **If action [S_m , a_i] = $r_i$ie reduce be using A → α, it pops 2r symbols from the stack where r = length(α). And pushes A then goto [Si, A] on top of the stack and continues.**
- **If action [S_m , a_i] = accept, it stops and announces the successful parsing.**
- **If action [S_m , a_i] = error, it calls an error recovery routine.**

13 Q: Discuss in details model of LR parser. (OR) Explain the construction of SLR parsing table.

It is the weakest of the three parsers but it is easiest to implement. The parsing can be done as follows:

<u>Definition of LR(0) items and related terms:</u>

1. The LR(0) item for grammar G is production rule in which symbol . (dot) is inserted at some position in RHS of the rule. For example,

S → .ABC

S → A.BC

S → AB.C

S → ABC.

The production S → € generates only one item S → .

<u>2. Augmented grammar</u> – If a grammar G is having start symbol S then augmented grammar is a new grammar G^1 in which S^1 is a new start symbol such S^1→ S. The purpose of this grammar is to indicate the acceptance of input. That is, when parser is about to reduce S^1→S, it reaches to acceptance state.

<u>3. Kernel items</u> – It is the collection of items S^1 → . S and all the items whose dots are not at the left most end of RHS of the rule.

Non kernel items – The collection of all the items in which .are at the left end of RHS of the rule.

4. Function of closure and goto – These are two important functions required to create collection of canonical set of items.

Closure operation:-

For a context free grammar G, if I is the set of items then the function closure(I) can be constructed using following rules.

1. Consider I is a set of canonical items and initially every item I is added to closure(I).

2. If rule A → α.Bβ is a rule in closure(I) and there is another rule for B such as B → γ then,

closure(I) : A → α.Bβ

B → .γ

This rule has to be applied until no more new items can be added to closure(I).

goto operation:-

The function goto can be defined as follows:

If there is a production A → α.Bβ then goto(A→α.Bβ, B) = A → αB.β That means, simply shifting of . (dot) one position ahead over the grammar symbol (may be terminal or non-terminal). The rule A→α.Bβ is in I then the same goto function can be written as goto(I,B)

5. Viable prefix – It is the set of prefixes in the right sentential form of production A →α. This set can appear on the stack during shift/reduce action.

14 Q: Write the differences between LR and LL parsers (OR) LR and predictive parsers.

LR parsers	LL parsers
1. These are bottom up parsers.	1. These are top down parsers.
2. This is complex to implement.	2. This is simple to implement.
3. For LR(1), the first L means the input is scanned from left to right. The second R means it uses rightmost derivation in reverse for the input string. The number 1 indicates that one lookahead symbol to predict the parsing process.	3. For LL(1), the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for the input string. The number 1 indicates that one lookahead symbol to predict the parsing process.
4. These are efficient parsers.	4. These are less efficient.
5. It is applied to the large class of programming languages.	5. It is applied to small class of programming languages.

More powerful LR parser (LR(1), LALR), Using ambiguous grammars, Error recovery in LR parsing, Syntax Directed Translation Definition, Evaluation orders for SDDs, Applications of SDTS, Syntax Directed Translation Schemes

More powerful LR parser

1 Q: Write the algorithm for constructing CLR parsing table.

Algorithm – Construction of CLR parsing table

Input – Augmented grammar G^1

Output – CLR parsing table

Method

1. **Construct C = { I_0, I_1, …I_n}, the collection of LR(1) items.**
2. **State I of the parser is constructed from I_i. The parsing action for state I is determined as shown below:**
   a) **If [A → α . aβ, b] is in I_i and goto(I_i, a) = I_j, then set action[i,α]=Shift j where a=terminal.**
   b) **If [A→α. , a] is in I_I & A ≠ S^1 , then set action[I,a]=reduce(A→α.)**
   c) **If [S^1→ S. , $] is in I_i then set action[I,$]=accept**

   The grammar is not CLR(1) LR(1) if there exists any confliction action after applying rules 2(a), 2(b), 2(c).
3. **The goto transition for state I are constructed for all non-terminals A using the rule: if goto(I_i , A) = $I_j$then goto[I,A]=j**
4. **All the entries not defined by rules 2 and 3 are error entries.**
5. **[S^1→ .S, $] is the initial state of the parser that is constructed from the set of items.**

Error recovery in LR parsing

2 Q: Explain Error recovery in LR parsing.

Error is detected by LR parser after constructing the parsing "action" table and finds an error entry. "goto" table is never consulted for detecting errors. If there exists invalid entry in "action" table, then error is announced by the LR parser. CLR(1) parser detects error earlier than SLR or LALR parser.

There are two modes for error recovery in LR parsing.

1. **Panic mode error recovery.**
2. **Phrase level error recovery.**

1. Panic mode error recovery

The stack is checked from top to bottom until as state P with goto on non-terminal B is found. After this, zero or more input symbols are removed until symbol 'b' is found. Then the parser will push the state goto(P,B) and gets back to normal parsing. There could be more than one yield for the non-terminal B. Thus, this method tries to isolate or remove the phrase containing an error. (Syntactic error).

2. Phrase-level error recovery

Phrase level error recovery uses the strategy of checking the possible programmer errors list (based on the language) to decide what could have been the reason for the occurrence of an error.

After deciding the reason for error, a procedure can be constructed to handle the error. These procedures may cause an insertion, deletion of input symbols. However, we must be careful that the procedure used should not,

a) **Enter into an infinite loop.**
b) **Cause a state that is already reached successfully to be popped off.**

3Q: Write the comparisons of LR parsers.

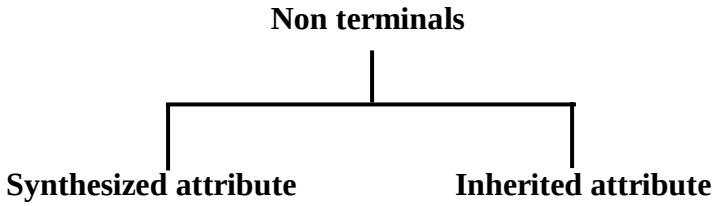
SLR parser	CLR parser	LALR parser
1. It is easier to implement.	1. It is expensive in terms of time and space.	1. It is expensive in terms of time and space.
2. The table obtained is smaller in size.	2. The table obtained is larger in size.	2. The table obtained is same as of SLR parser.
3. It uses FOLLOW function.	3. It uses FIRST function.	3. It also uses FIRST function.
4. It can handle only few classes of grammar.	4. It can handle large classes of grammar.	4. It can handle classes of grammar than SLR.
5. It is the least powerful parser.	5. It is the most powerful parser.	5. It lies between SLR and CLR parsers.
6. Before announcing error, it may make several reductions. But it never shift erroneous input symbol onto the stack.	6. It can detect syntactic error as soon as possible while scanning the input from left to right.	6. It behaves like SLR.

Syntax Directed Definition –SDD

4 Q: Explain about SDD or SDT – Syntax Directed Translation.

Syntax Directed Definitions

- **Syntax Directed Definition is a context free grammar with semantic rules attached to the productions. The semantic rules define values for attributes associated with the non-terminal symbols of the production.**
- **SDD is a CFG which consists of attributes and rules.**
  - **i) Attributes – They are associated with grammar symbol.**
  - **ii) Rules – They are associated with productions.**
- **For example, if X is grammar symbol and 'b' is one of its attributes, then X.b denotes the value of 'b' at a particular parse tree node labeled as 'X'. Attribute can be number, types table references or strings.**
- **The following are the two types of attributes for non-terminals.**
  1. **Synthesized attributes.**
  2. **Inherited attributes.**



Synthesized attribute

- **An attribute is synthesized if all its dependencies point from child to parent in the parse tree.**
- **The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.**
- **Synthesized attribute for a nonterminal S at the parse tree with node N is defined by a semantic rule with production at N.**
- **Here S is the head of the production.**

Inherited attribute

- **The value of inherited attribute is computed from the values of attributes at the siblings and parent of that node.**
- **Inherited attribute for nonterminal A at the parse tree with node N is defined by a semantic rule associated with the production at the parent node N.**

The following example shows the SDD for a simple desk calculator, 'n' is end marker. In SDD, each of the non-terminals or variables consists of a single synthesized attribute 'val'.

SNO	PRODUCTION	SEMANTIC RULE
1.	L → E n	L.val = E. val
2.	E →E + T	E.val = E.val + T.val
3.	E → T	E.val = T.val
4.	T → T* F	T.val = T.val * F.val
5.	T → F	T.val = F.val
6.	F → (E)	F.val = E.val
7.	F → digit	F.val = digit.lexvalue

Evaluation orders for SDDs

5 Q: Explain about evaluation orders for SDDs.

- **The useful tool used for determining an evaluation order for the attribute instances in a given parse tree is "Dependency graph". Annotated parse tree indicates attributes values and dependency graph helps to determine how those values can be computed.**

Dependency graph

- **The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency graph.**
- **Dependency graph indicates the flow of information among the attribute instances in a particular parse tree. If an edge exists from one attribute instance to another, it means that the value of the first is needed to compute the second.**

For example,

E →E + T is the production and E.val = E_1 + T.val is semantic rule. Then the corresponding dependency graph will be shown below.

Step 1

First let us consider the parse tree for E →E + T

Step 2

The dependency graph for synthesized attribute (ieval) at N is,

The synthesized attribute 'val' at N is computed using the values of 'val' at two children ie E and T. Here dotted lines indicates parse tree edges and solid lines indicates dependency graph edges.

- **The following are the cases of different kinds of inherited dependencies.**

Case 1

Case 2

Case 3

- **Dependency graphs are helpful in checking the possible orders that can be used to evaluate the attributes at various nodes of a parse tree.**

S – attributed definitions

- **The syntax directed definition SDD that uses synthesized attributes is called S-attributed definition.**
- **In a parse tree, the semantic rule at each node is evaluated for annotating (computing) the S-attributed definition.**

- **This process is bottom up ie from leaves to the root.**
- **For example, let us take the input string 5 * 6 + 7 for computing S-attributed definition (Syntax tree, Parse tree, Annotated parse tree)**

L – attributed definitions

- **If there exists attributes associated with a production body, dependency graph edges that goes from left to right but not from right to left. Then they are called as L-attributed definitions.**
- **Each attribute can be either Synthesized or Inherited, but with the following rules:**

  If there is a production of the form $Y \rightarrow X_1X_2 \ldots X_n$, and if there exists inherited attribute $X_i.b$ computed by a rule associated with this production. Then use the following:-

  a) **Inherited attributes associated with the head Y**
  b) **Either inherited or synthesized attributes associated with the occurrences of symbols $X_1X_2 \ldots X_{i-1}$ located to X_i's left.**
  c) **Synthesized or Inherited attributes associated with these occurrences of X_i itself, but there should not be cycles in the dependency graph formed by the attributes of this X_i.**
- **For example, The L-attributes are given in the following table.**

$T \rightarrow FT^1$	$T^1.inh = F.val$
$T^1 \rightarrow *FT^1$	$T^1.inh = T^1.inh * F.val$

Syntax Directed Translation (SDT)

- **SDT is defined as a method for compiler implementation where the source language translation is completely driven by the parser.**
- **Here the parsing process and parse tree are used to direct semantic analysis and the translation of the source program.**
- **SDT scheme is CFG with program fragments called semantic actions, embedded with in production bodies.**

For example,

		SDD production	Semantic rules
SDD	1.	$T \rightarrow T_1*F$	$T.val = T_1.val * F.val$
	2.	$T \rightarrow F$	T.val = F.val
	3.	$F \rightarrow$ digit	F.val = digit.lexval

		SDD production	Semantic rules
SDT	1.	$T \rightarrow T_1*F$	{ $T.val = T_1.val * F.val$; }
	2.	$T \rightarrow F$	{ T.val = F.val ; }
	3.	$F \rightarrow$ digit	{ F.val = digit.lexval ; }

- **SDTs are implemented during parsing without building a parse tree.**
  1) **S-attributed SDD is based on LR-parsing table grammar.**
  2) **L-attributed SDD is based on LL-parsing table grammar.**

Applications of SDT

6 Q: Explain the applications of SDT (OR)

Explain about

a) **Construction of syntax trees**

b) **The structure of a Type**

Construction of Syntax trees

- **The syntax tree is an abstract representation of the language constructs.**
- **The syntax trees are used to write the translation routines using syntax directed definitions.**
- **Constructing syntax tree for an expression means translation of expression into postfix form.**
- **The following functions are used in syntax tree for the expression.**
  1. **mknode (op, left, right)**

     This function creates a node with the field operator having operator as label, and the two pointers to left and right.
  2. **mkleaf (id, entry)**

     This function creates an identifier node with label "id" and a pointer to symbol table is given by "entry".
  3. **mkleaf (num, val)**

     This function creates a node for number with label "num" and "val" is for the value of that number.
- **For example, construct the syntax tree for the expression x * y – 5 + z**

  Step 1 – Convert the expression from infix to postfix x y * 5 – z +

  Step 2 – Make use of the functions mknode( ), mkleaf(id, ptr) and mkleaf(num, val)

  Step 3–The sequence of function calls is given.

Step 1:

Let us consider the given infix expression x * y – 5 + z and the corresponding syntax tree is,

The postfix expression is x y * 5 – z +

Step 2:

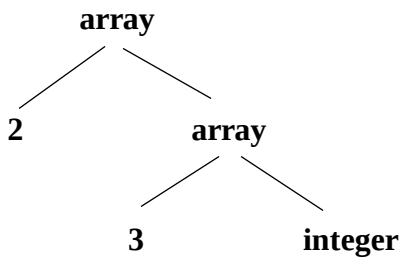
Symbol	Operation
x	P_1 = mkleaf(id, ptr to entry x)
y	P_2 = mkleaf(id, ptr to entry y)
*	P_3 = mknode(*, P_1 , P_2)
5	P_4 = mkleaf(num, 5)
-	P_5 = mknode(-,P_3 , P_4)
z	P_6 = mkleaf(id, ptr to entry z)
+	P_7 = mknode(+, P_5 , P_6)

Step 3:

The structure of Type

Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input. Then attributes can be used to carry information from one part of the parse tree to another.

For example, In C, the type int[2][3] can be read as, "array of 2 arraysof 3 integers." The corresponding type expression *array(2,* array(3, *integer))* isrepresented by the tree.



Syntax Directed Translation (SDT) Schemes

7 Q: Explain SDT schemes.

- **SDTs are implemented during parsing without building a parse tree.**
  - **a) S-attributed SDD is based on LR-parsing table grammar.**
  - **b) L-attributed SDD is based on LL-parsing table grammar.**

Postfix Translation Schemes

Postfix SDTs are SDTs with all actions at the right ends of the production bodies. For example, let us construct postfix SDT for desk calculator.

1.	**L → E n**	**{ print (E. val); }**
2.	**E →E + T**	**{ E.val = E.val + T.val ; }**
3.	**E → T**	**{ E.val = T.val ; }**
4.	**T → T* F**	**{ T.val = T.val * F.val ; }**
5.	**T → F**	**{ T.val = F.val ; }**
6.	**F → (E)**	**{ F.val = E.val ; }**
7.	**F → digit**	**{ F.val = digit.lexvalue ; }**

The above grammar is LR and SDD is S-attributed.

Parser stack implementation of postfix SDTs

	State / grammar symbol	Synthesized attributes
	:	:
	X	X.x
	Y	Y.y
TOP →	Z	Z.z
	..	..

A stack is being used by the bottom up parser to hold information of the parsed subtrees. During LR parsing, postfix SDTs can be implemented by executing the actions when reductions occur. Postfix SDTs are implemented by considering the attributes of each grammar on the stack in a place where grammar symbol is found during reduction.

The above table contain three grammar symbols XYZ. They are to be reduced based on the production A → XYZ. Here, X.x is one attribute of X and so on.

Eliminating left recursion from SDTs

- **During grammar transformation, the actions are treated as if they were terminal symbols. This preserves the order of the terminals in the generated string.**
- **In order to eliminate left recursion if**

  $$A \rightarrow A\alpha \mid \beta \text{ then}$$

  replace them by following productions,

  $$A \rightarrow \beta A^1$$

  $$A^1 \rightarrow \alpha A^1 \mid \epsilon$$

- **For example, E → E + T | T is left recursive. Then the productions after elimination of left recursion are**

  $$E \rightarrow TE^1$$

  $$E^1 \rightarrow +TE^1 \mid \epsilon$$

Intermediate code generation – Variants of syntax trees, Three address code, Types and declarations, Translation of expressions, Type checking, Control flow, Back patching.

Variants of Syntax trees

1 Q: Explain briefly about variants of syntax trees.

- **Syntax tree consists of nodes and children.**

  i) Nodes – They represent constructs in the source program.

  ii) Children – They represent the meaningful components of a construct.

Directed Acyclic Graphs (DAG) for expressions (OR) Procedure for constructing DAG

- **In a given expression, the common sub-expressions (sub-expressions occurring more than one time) are identified by Directed Acyclic Graph (DAG)**
- **DAG and syntax tree construction is similar.**
- **The difference between them is : a node N in DAG can have more than one parent if N represents a common sub-expression. But in case of syntax tree, the common sub-expression would be represented repeatedly as many times the sub-expression appears in the original expression.**
- **For example, let us construct DAG for the expression:**

  e * (b – c) + (b – c) * a

  Here, expression (b – c) is repeated twice, called common sub-expression.

  Step 1

  First let us construct DAG for the common sub-expression, (b –c)

  Step 2

  Construct DAG for e * (b – c)

Step 3

Again no need to construct DAG for (b – c). Hence consider (b – c) * a

Step 4

Finally draw DAG for the expression e * (b – c) + (b – c) * a

- **The steps for constructing DAG are as follows.**

Symbol	Operation
b	P_1 = mkleaf(id, ptr to entry b)
c	P_2 = mkleaf(id, ptr to entry c)
-	P_3 = mknode(-, P_1 , P_2)
e	P_4 = mkleaf(id, ptr to entry e)
*	P_5 = mknode(*, P_3 , P_4)
a	P_6 = mkleaf(id, ptr to entry a)
*	P_7 = mknode(*, P_3, P_6)
+	P_8 = mknode(*, P_5 , P_7)

- **Consider the following SDD to produce syntax tree or DAG**

SNO	PRODUCTION	SEMANTIC RULE
1.	E → E + T	E.Node = new Node (+, E.Node, T.Node)
2.	E → E - T	E.Node = new Node (-, E.Node, T.Node)
3.	E → T	E.Node = T.Node
4.	T →(E)	T.Node = E.Node
5.	T →id	T.Node = new Lean(id, id.entry)
6.	T→ digit	T.Node = new Leaf(digit, digit.val)

Value Number method for constructing DAGs

- **Array of records can be used to store nodes of syntax trees or DAGs.**
- **Each row of the array represents one record, and therefore one node.**
- **The first field is an operation code which indicates the label of a node. This is present in each record.**
- **Leaves consist of one additional field to hold the lexical value (either a pointer to symbol table or constant).**
- **Interior nodes consist of two additional fields to represent the left and right child.**

2 Q: Explain about Three address code (TAC or 3AC)

- **TAC is a sequence of statements of the general form**

  x = y op z

  Where x,y,z are names, constants or compiler generated temporaries. op stands for any operator such fixed or floating point arithmetic operator or logical operator on Boolean valued data. Here three address refers to three addresses ie addresses of x, y and z.

- **For example, if x = a + b * c then the TAC is,**

  t1 = b * c

  t2 = a + t1

  x = t2

Where t1 and t2 are compiler generated temporary variables.

Addresses and Instructions

These are different types of TAC:

- **Assignment statements are of the form x = y op z**
- **Assignment instructions are of the form x = op z (is unary operation like unary minus)**
- **Copy statements are of the form x = y (value of y is assigned to x).**
- **Unconditional jumps such as**

  if x relop y goto L

  (relational operators <>, <, >, <=, >=, =)

- **Param x and call p, n for procedure calls and return y where y represents a returned value (optional).**

  Param X1

  Param X2

  Param Xn

- **The following are the different types of TAC**

1.	**x = y op z**	**Assignment (op = binary arithmetic or logical operation)**
2.	**x = op y**	**Unary assignment (op = unary operation)**
3.	**x = y**	**Copy (x is assigned the value of y)**
4.	**goto L**	**Unconditional jump**

5.	if x relop goto L	Conditional jump (relop is <>, <, >, <=, >=, =)
6.	Param x	Procedure call
7.	call p, n	Procedure call
8.	return z	Procedure call
9.	a = b [i]	Index assignment
10.	x [i] = y	Index assignment
11.	x = &y x = *y	Pointer assignment
12.	*x = y	Pointer assignment

3 Q: Briefly discuss about the implementations of TAC or Structure of 3AC.

- **Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields.**
- **There are three representations used for three address code.**
- **They are**
  1. **Quadruple representation.**
  2. **Triple representation.**
  3. **Indrect triple representation.**

Quadruple representation

- **Quadruple is a structure with the atmost four fields such as,**

op	arg1	arg2	result

- **The field op is sued to represent the internal code for operator.**
- **arg1 and arg2 represents two operands used and result field is used to store the result of an expression.**
- **For example, consider the input string x = - a * b + - a * b**
- **The following is the three address code.**

  t1 = uminus a

  t2 = t1 * b

  t3 = uminus a

  t4 = t3 * b

  t5 = t2 + t4

  x = t5

- **The following is the quadruple representation.**

op	arg1	arg2	result
uminus	a		t1
*	t1	b	t2
uminus	a		t3
*	t3	b	t4
+	t2	t4	t5
=	t5		x

Triple representation

- **In this, the use of temporary variables is avoided by referring the pointers in the symbol table.**
- **Triple is a structure with the atmost three fields such as,**

op	arg1	arg2

- **For example, consider the input string x = - a * b + - a * b**
- **The following is the three address code.**

  t1 = uminus a

  t2 = t1 * b

  t3 = uminus a

  t4 = t3 * b

  t5 = t2 + t4

  x = t5

- **The following is the triple representation.**

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

Indirect triple representation

- **In this, it uses additional array to list the pointers of triple in the desired order.**
- **For example, consider the input string x = - a * b + - a * b**
- **The following is the three address code.**

  t1 = uminus a

  t2 = t1 * b

  t3 = uminus a

  t4 = t3 * b

  t5 = t2 + t4

  x = t5
- **The following is the Indirect triple representation.**

(0)	**(11)**
(1)	**(12)**
(2)	**(13)**
(3)	**(14)**
(4)	**(15)**
(5)	**(16)**

op	arg1	arg2
uminus	a	
*	(11)	b
uminus	a	
*	(13)	b
+	(12)	(14)
=	x	(15)

4 Q: Briefly explain about Types and Declarations

- **A complier has to perform semantic checking along with syntactic checking.**
- **Semantic checks can be static (during compilation) or dynamic (during run time).**
- **The application of types is grouped under checking and translation.**

Type checking

- **The process of verifying and enforcing the constraints of types – type checking – may occur either at compile time (static check) or run time (dynamic check).**
- **A "type checker" implements the type system.**
- **A "type system" is a collection of rules for assigning type expressions to the parts of a program.**

Translation applications

- **Compiler can determine the storage required from the type of a name at run time.**

Type expressions

- **Type expression denotes the type of a language construct. It can be a basic type Eg: primitive data type like int, real, char, boolean etc., or a type error or a void: no type.**

Type equivalence

- **Type equivalence contains two important concepts.**
- **Name equivalence have the same name. For example, a, b have same type and c has different type.**
- **Structural equivalence have the same structure. For example, a, b and c have the same type.**

Declaration

- **Whenever a declaration is encountered, then create a symbol entry for every local name in a procedure.**
- **The symbol table entry should contain type of name, how much storage the name requires and a relative offset.**

Storage layout for local names

- **If we know the type of a name, we can determine the amount of storage required for the name at run time.**
- **Type and relative address are saved in the symbol table entry for the name.**
- **The 'width' of type is defined as the number of storage units required for objects of that type.**
- **SDT is used to compute types and their width for basic and array types.**

5 Q: Explain about Type checking.

Rules for Type Checking (TC)

- **Type Checking can be Type Synthesis (TS) and Type Inference (TI).**
- **Type Synthesis (TS) uses the types of its subexpressions in order to build the type of expression. TS require names to declare before they are used.**
- **Type Inference (TI) is used to determine the type of a language construct from the way it is used.**

Type conversions

- **Type conversion rules differ from one language to another. Java uses widening and narrowing conversions between primitive types.**

Unification

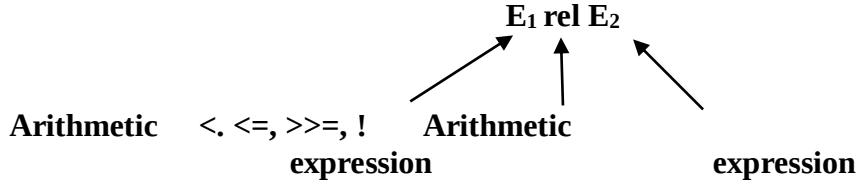
- **Testing equality of expressions is the concept of unification.**

6 Q: Explain about Control Flow in Intermediate code generation.

- **The main idea of converting any flow of control statements to 3AC is to stimulate the "branching" of the control flow.**
- **Boolean expression in programming languages are often used to:**
  1. **Alter the flow of control.**
  2. **Compute logical value.**
- **Flow of control statements may be converted to 3AC by using following functions:**
  1. **new label – returns a new symbolic label each time it is called.**
  2. **gen ( ) – generates the code (string) passed as parameter to it.**
- **The following attributes are associated with non-terminals for the code generation:**
  1. **code– contains the generated 3AC.**
  2. **true – contains the label to which a jump takes place if the Boolean expression associated evaluates to true.**
  3. **false – contains the label to which a jump takes place if the Boolean expression associated evaluates to false.**
  4. **begin – contains the label / address pointing to the beginning of the code block for the statement 'generated' by the non-terminal.**

Boolean expressions

- **Boolean expressions consist of Boolean operations like AND (&&), OR (||) and NOT (!) as in C applied to the elements that are Boolean variables or relational expressions of the form E_1 rel E_2**



- **For example, E → E && E | E || E | !E | (E) | E_1 rel E_2 | true | false**

Short circuit code

- **Given Boolean expression can be translated into 3AC without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This is called as Jumping code or short circuit code.**

- **For example, if (a < 10 || a > 15 && a != b)**

x = 1;

Translate this to jumping code or short circuit code

if a < 10 goto L1

if false a > 15 goto L2

if false a != b goto L2

L1 : x = 1

L2 :

Flow of control statements

- **The main idea of converting any flow of control statement to the 3AC is to stimulate the "branching" of the flow of control using the goto statement.**
- **Consider the following grammar,**

S → if E then S_1 | if E then S_1 else S_2 | while E do S_1 ;

- **The simulation of the flow of control branching for each statement is depicted pictorially as follows:**

if – then

if – then else

while – do

7 Q: Explain about Backpatching.

- **The main problem in generating 3AC in a single pass is that we may know the labels that control must go to at the time jump statements are generated.**
- **In order to get rid of this problem, a series of branching statements with the targets of the jumps temporarily left unspecified is generated.**
- **Back patching is putting the address instead of labels when the proper label is determined.**

One pass code generation using back patching:

- **Back patching algorithms perform three types of operations:**
  1. **Makelist (i) – Creates a new list containing only ' i ', an index into the array of quadruples and returns pointer to the list it has made.**
  2. **Merge (i, j) – Concatenates the lists pointed by 'i' and 'j' and returns a pointer to the concatenated list.**
  3. **Back patch(P, i) – Inserts 'i' as the target label for each of the statements on the list pointed by P.**

Runtime environments – Stack allocation of space, Access to non-local data on the stack, Heap management.

Code generation – Issues in the design of code generation, The target language, Addresses in the target code, Basic blocks and flow graphs, A simple code generator.

---

1 Q: Stack allocation of space

- **Stack area is used to allocate space to the local variables whenever the procedure is called. This space is popped off the stack when the procedure terminates.**

Activation Trees

- **Activation trees are used to describe the nesting of procedure calls to make the stack allocation feasible efficiently.**
- **The nesting of procedure calls can be illustrated using the following quicksort example.**

```
1. program sort(input,output)
2. var  a : array[ 0 .. 10 ] of integers
3. procedure readarray
4. var i : integer
5. begin
6. for i = 1 to 9 do read array( a [ i ] )
7. end
8. function partition( y , z : integer ) : integer
9. var i , j , x , v : integer
10. begin ….
11. end
12. procedure quicksort( m ,n : integer )
13. var i : integer
14. begin
15. if ( n > m ) then begin
16. i = partition ( m , n )
17. quicksort( m, i – 1 )
18. quicksort( i + 1 , n )
19. end
20. end
```

21. begin

22. a [0] := -9999, a [10] := 9999

23. readarray

24. quicksort(1, 9)

25. end

- **The following is the activation tree corresponding to the output of quicksort.**

Activation Record (AR)

- **Activation Record is a memory block used for information management for single execution of a procedure.**
- **The following is the activation record (Read from bottom to top)**

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Temporaries – It hold temporary values, such as the result of mathematical calculation, a buffer value or so on.

Local data – It belongs to the procedure where the control is currently located.

Saved machine status- It provides information about the state of a machine just before the point where the procedure is called.

Access link – It is used to locate remotely available data. This field is optional.

Control link – It points to the activation record of the procedure which called it, ie the caller. This field is optional. This link is also known as dynamic link.

Return value – It holds any valued returned by the called procedure. These values can also be placed in a register depending on the requirements.

Actual parameters – These are used by the calling procedures.

2 Q: Access to non-local data on the stack

- **Non-local data is a parameter that is used within a procedure. There are various situations of access to non-local data. These include:**

Data access without nested procedures

For non-nested procedures, variables are accessed as follows:

- **Global variables are declared statically as their values and locations are known or fixed at compile time.**
- **Local variables are declared locally at the top of the stack using stack top pointer.**

A language with nested procedure declarations – ML

Properties of ML include:

- **The variables declared in ML are unchangeable once they are initialized. This means that ML is a functional language.**
- **Variable declaration is done using the statement:**
  val <name> = <exp>
- **The syntax for defining functions is,**
  fun <name> (<arg>) = <body>
- **Function bodies are defined as,**
  let <definitions> in <statements> end

Nesting depth

- **Nesting Depth defines the level from the start of a procedure at which a particular nested procedure is defined.**

Access link

- **It is used to locate any variable or procedure in the case of nested procedures.**

Displays

- **Displays are used for efficient and easy access of non-local data by avoiding the use of long chains of activation links, in situations having large network depths.**
- **Display uses auxiliary array called d.**

3 Q: Heap Management

- **Heap is the unused memory space available for allocation dynamically.**
- **It is used for data that lives indefinitely, and is freed only explicitly.**
- **The existence of such data is independent of the procedure that created it.**

The memory manager

- **Memory manager is used to keep account of the free space available in the heap area. Its functions include,**
  - **allocation**
  - **deallocation**
- **The properties of an efficient memory manager include:**
  - **space efficiency**
  - **program efficiency**
  - **low overhead**

The memory hierarchy of a computer

- **Typical sizes** **Typical access times**

2GB	**Virtual memory (disk)**	**3 – 15 ms**
256MB – 2 GB	**Physical memory (RAM)**	**100 – 150 ms**
128Kb – 4 MB	**2nd Level Cache**	**40 – 60 ns**
16 – 65KB	**1st Level Cache**	**5 – 10 ns**
32 words	**Registers**	**1 ns**

Locality in programs

Locality refers to the amount of data requirements for a particular program, and the times required to access or locate the data. Locality is of two types namely,

- **Temporal locality – This is present if the memory locations accessed by it are likely to be accessed again soon.**
- **Spatial locality – This is the condition when the memory locations close to the location accessed are likely to be accessed within a short period of time.**

4 Q: Issues in the design of code generation

Instruction selection

- **The job of instruction selector is to do a good job overall choosing which instructions to implement which operator in the low level intermediate representation.**
- **Issues here are:**
  - **level of the IR**
  - **nature of the instruction set architecture**
  - **desired quality of the generated code**
- **Target Machine:**
  - **n general purpose registers**
  - **instructions: load, store, compute, jump, conditional jump**
  - **various addressing mode**
  - **indexed address, integer indexed by a register, indirect addressing and immediate constant.**
- **For example, X = Y + Z we can generate,**

  LD R0, Y

  ADD R0, R0, Z

  ST X, R0

Register allocation

- **Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code.**
- **The most frequently used variables should be kept in process register for faster access.**
- **The use of registers if often subdivided into two sub problems:**
  - ❖ **During "register allocation", we select the set of variables that will reside in register at a point in the program.**
  - ❖ **During "register assignment", we pick the specific register that a variables will reside in.**
- **For example, consider** `t1 = a * b`
  `t2 = t1 + c`
  `t3 = t2 / d`

  optimal machine code sequence is,

```
L    R1, a
M    R1, b
A    R2, c
D    R2, d
ST   R1, t3
```

5 Q: The Target Language

- **The output of Code Generation (CG) is the target language.**
- **The output may take different forms like absolute machine language, relocatable machine language or assembly language.**

Simple target machine model

The possible kinds of operations are listed below:

1. Load operation (LD)

- **General format is LD dst, addr.**
- **LD loads the value in location 'addr' into location 'dst'. Here dst is destination.**
- **For example, LD R1, x → loads the value in location x into register R1.**

2. Store operation (ST)

- **General format is ST dst, src.**
- **For example, ST x, R1 → stores the value in register R1 into the location x.**

3. Computation operation

- **The general form is OP dst, src1, src2 where OP is a operator like ADD, SUB.**
- **For example, ADD R1, R2, R3 → adds R2 and R3 values and stores into R1.**

4. Unconditional jump (BR)

- **The general form is BR L where BR is branch. This causes control to branch to the machine instruction with label L.**

5. Conditional jump

- **The general form is Bcond R, L where R is a register, L is label and cond stands for any of the common tests on values in register R.**
- **For example, BGTZ R2, L → This instruction causes a jump to label L if the value in register R2 is greater than zero and allows control to pass to the next machine instruction if not.**

6 Q: Addresses in the target code

- **In this, it explains storage allocation strategies namely static and stack allocation strategies.**

Static allocation strategy

- **In static allocation names are bound to the storage as the program is compiled. Since bindings don't change at runtime, its names are bound to the same storage whenever the procedure is activated.**
- **The compiler must decide where the activation record is to go, relative to the target code. Once this decision is made, the position of each activation record and the storage for each name in the record is fixed.**
- **Compiler gives the address of code which is required by the target code.**

Limitations:

- **The size of data objects must be known at compile time.**
- **Recursive procedures are not allowed because all the activations of a procedure use the same bindings for local names.**
- **Data structures cannot be created dynamically since there is no mechanism for storage allocation at runtime.**

Stack allocation strategy

- **In stack allocation, storage is organized as stack and activation records are pushed and popped as the activation begins and ends respectively.**
- **Storage for locals in each call of a procedure is contained in the activation record.**

- **The values of locals are deleted when activation ends.**
- **Consider the stack in which size of all activation records are known at compile time. Consider a register "top" which points the top and increments at run time. Activation record is allocated and deallocated by incrementing and decrementing the top with size of the record.**

7. Basic blocks and Flow graph

Basic blocks

- **Basic block is sequence of constructive 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.**
- **Basic blocks are constructed by partitioning a sequence of three address instruction.**

Algorithm for partitioning into basic blocks:

INPUT:- sequence of three address instructions

OUTPUT:- list of basic blocks

METHOD:

Step1

The first step is to determine the set of leaders. The rules to obtain the leaders are

1. **The first statement is a leader.**
2. **Any statement which is the target of conditional or unconditional GOTO is a leader.**
3. **Any statement which immediately follows the conditional GOTO or unconditional GOTO is a leader.**

Step2

For each leader, construct the basic blocks which consists of the leader and all the instructions up to but not including the next leader or the end of the intermediate program.

- **For example, let us construct the basic blocks for the following:**
  1. **i = 0**
  2. **if (i > 10) goto 6**
  3. **a[i] = 0**
  4. **i = i + 1**
  5. **goto 2**
  6. **end**

- **Let us apply step1 and step2 of algorithm of algorithm to identify basic blocks.**

Step1

1. **i = 0**
2. **if (i > 10) goto 6**
3. **a[i] = 0**
4. **i = i + 1**
5. **goto 2**
6. **end**

- **Based on rule 1 of step1, 1 statement is leader.**
- **Based on rule 2 of step1, 2 and 5 are leaders.**
- **Based on rule 3 of ste1, 3 is a leader.**

Step2

- **For each leader, construct the basic blocks.**

L

```
1. i = 0     B1
```

L

```
2. if (i > 10 ) GOTO 6    B2
```

L

```
3. a[i] = 0
4. i = i + 1
5. GOTO 2         B3
```

L

```
6. end     B4
```

Flow graph

- **Flow graphs are used to represent the basic blocks and their relationship by a directed graph.**
- **There exists an edge from block 1 to block2 iff it is possible for the first instruction in block2 to immediately flow to the last instruction in block1.**
- **For example, Let us write the flow graph for the following basic blocks.**

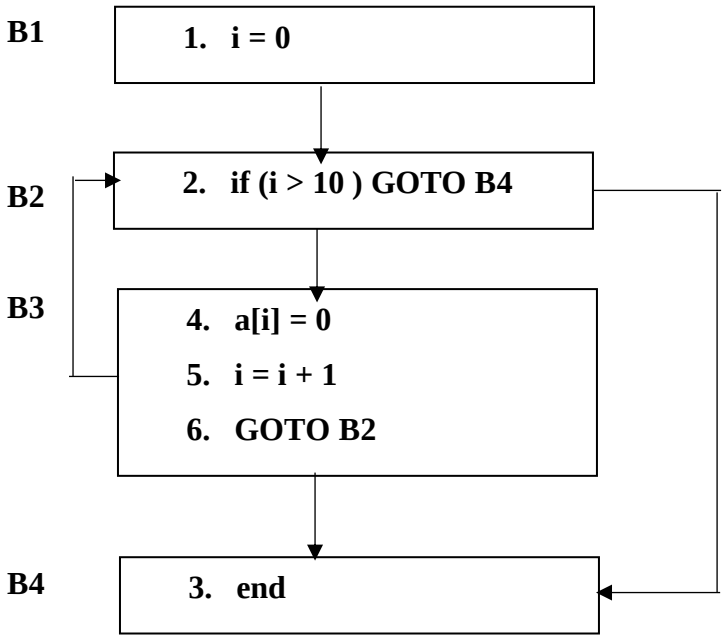
```
1. i = 0     B1
```

```
2.  if (i > 10 ) GOTO 6    B2
```

```
4.  a[i] = 0
5.  i = i + 1
6.  GOTO 2          B3
```

```
3.  end      B4
```

- **Flow graph for these basic blocks is:**



8 Q Code Generation algorithm

Machine instruction for operations

For X = Y + Z do the following:

- **Use getReg (X = Y + Z) to select registers for X, Y and Z. Let the registers be Rx, Ry and Rz.**
- **If Y is not in Ry then issue an instruction LD Ry, Y. Here Y is one of the memory location for Y.**

- **Similarly if Z is not in Rz, issue an instruction LD Rz, Z. Here Z is a memory location of Z.**
- **Issue the instruction ADD Rx, Ry, Rz**

Machine instructions for copy statements

- **A three address copy statement can be of the form x = y. This is a special case where we assume that 'getReg' will always choose the same register for both x and y.**

Ending the basic block

- **When the block ends, we need not have to remember the variable value if it is temporary. In this case, assume that its register is empty if the variable is temporary and used only within the block.**
- **Then generate the instruction ST x, R where R is a register in which x's value exists at end of the block.**

Managing the Register and Address descriptors

1. Load operation (LD)

- **General format is LD dst, addr.**
- **LD loads the value in location 'addr' into location 'dst'. Here dst is destination.**
- **For example, LD R1, x → loads the value in location x into register R1.**

2. Store operation (ST)

- **General format is ST dst, src.**
- **For example, ST x, R1 → stores the value in register R1 into the location x.**

3. Computation operation

- **The general form is OP dst, src1, src2 where OP is a operator like ADD, SUB.**

For example, ADD R1, R2, R3 → adds R2 and R3 values and stores into R1.

Machine Independent code optimization – Principle sources of optimization, Peephole optimization, Introduction to data flow analysis.

Principle sources of optimization

1 Q: Explain Principle sources of optimization techniques (OR) Function preserving transformations).

The following are the principle sources of optimization techniques or function preserving transformations:

a) **Elimination of common sub expression**
b) **Copy propagation**
c) **Dead code elimination**
d) **Constant folding**
e) **Loop optimizations**

Elimination of common sub expression

- **Common sub expressions can be either eliminated locally or globally.**
- **Local common sub expressions can be identified in a basic block.**
- **Hence first step to eliminate local common sub expressions is to construct a basic block.**
- **Then the common sub expressions in a basic block can be deleted by constructing a directed acyclic graph (DAG).**
- **For example, x = a + b * (a + b) + c + d**

  The following is the basic block.

```
t1 = a + b
t2 = a + b
t3 = t1 * t2
t4 = t3 + c
t5 = t4 + d
x = t5
```

- **The local common sub expression in the above basic block are**

  t1 = a + b

  t2 = a + b

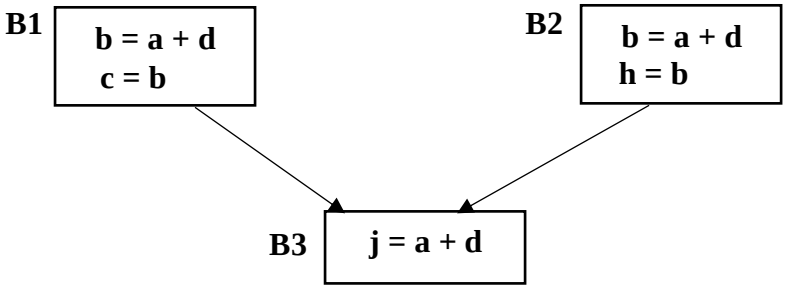
- **Hence these local common sub expressions can be eliminated. The block obtained after eliminating local common sub expression is shown below.**

```
t1 = a + b
t2 = t1 * t1
t3 = t2 + c
t4 = t3 + d
x = t4
```

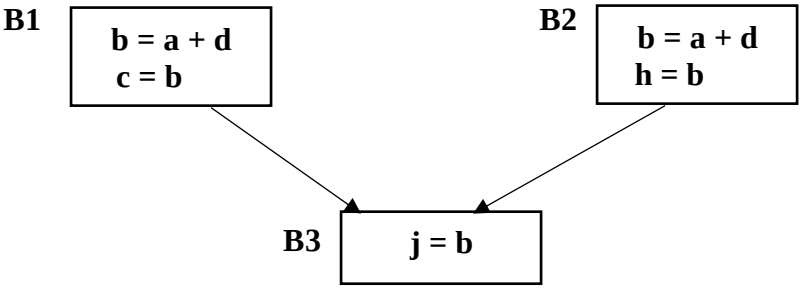
Copy or Variable propagation

- **Consider the assignment statement of the form x = y. The statement x = y is called as copy statement.**
- **To explain copy propagation, take the following example.**

a)



b)



- **Here the common sub expression is j = a + d. When a + d is eliminated, it uses j = b.**

Dead code elimination

- **A piece of code which is not reachable and never used anywhere in the program, then it is said to be dead code, and can be removed from the program safely.**
- **Generally copy statements may lead to dead code. For example,**

  x = b + c

  z = x

  …

  d = x + y

- **Suppose z variable is not used in the entire program, the z =x becomes the dead code. Hence, it can be optimized as,**

```
x = b + c
…
d = x + y
```

Constant folding

- **In folding technique, the computation of a constant is done at compile time instead of execution time and further the computed value of the constant is used.**
- **For example, k = (22 / 7) * r * r**

  Here folding is done by performing the computation of (22 / 7) at compile time. So it can be optimized as,

  k = 3.14 * r * r

Loop optimizations

- **The major source of code optimization is loops, especially the inner loops.**
- **Most of the run time is spent inside the loops which can be reduced the number of instructions in the inner loop.**
- **The following are the loop optimization techniques.**
  1. **Code motion.**
  2. **Elimination of induction variables.**
  3. **Strength reduction.**

1. Code motion:-

- **Code motion is a technique which is used to move the code outside the loop.**
- **If there exists any expression outside the loop which the result is unchanged even after executing the loop many times, then such expression should be placed just before the loop.**
- **For example,**

```
while ( x ! = max =3 )
{
        x = x + 2;
}
```

Here the expression max -3 is a loop invariant computation. So this can be optimized as follows:

```
k = max -3;
while ( x ! = k )
{
        x = x + 2;
}
```

2. Elimination of induction variables:-

- **An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way.**
- **It can also be defined as a variable which is incremented or decremented by a fixed number in the loop each time is executed.**
- **For example,**

```
for( i = 0, j = 0, k =0; i < n; i++ )
{
        printf("%d", i );
}
```

- **There are three induction variables i, j, k used in the for loop. So each time i is used but j and k are not used. Hence the code can be optimized after elimination of unused induction variables is given below.**

```
for( i = 0; i < n; i++ )
{
        printf("%d", i );
}
```

3. Strength reduction:-

- **It is the process of replacing expensive operations by the equivalent cheaper operations on the target machine.**
- **For example,**

```
for( k = 1; k < 5; k++)
{
        x = 4 * k;
}
```

- **On many machines, multiplication operation takes more time than addition or subtraction. Hence, the speed of the object code can be increased by replacing multiplication with addition or subtraction.**

```
for( k = 1; k < 5; k++)
{
        x = x + 4;
}
```

2 Q: Explain briefly about peephole optimization techniques.

- **Peephole optimization is simple but effective method used to locally improve the target code by examining a short sequence of target instructions known as peephole and then replace these instructions by a shorter and/or faster sequence of instructions whenever required.**
- **The following are peephole optimization techniques:**
  1. **Elimination of redundant instructions.**
  2. **Optimization of flow of control or elimination of unreachable code.**
  3. **Algebraic simplifications.**
  4. **Strength reduction.**

1. Elimination of redundant instructions:-

- **This includes elimination of redundant load and store instructions.**
- **For example,** **MOV R1, A**
  MOV A, R1
- **Here first instruction is storing the value of A into register R1 and second instruction is loading R1 value into A.**
- **These two instructions are redundant so eliminate instruction (2) because whenever instruction (2) is executed after (1), it is ensured that the register R1 contains A value.**

2. Optimization of flow of control or elimination of unreachable code:-

- **An unlabeled instruction that immediately follows an unconditional jump can be removed.**
- **For example,**

  i = j
  if k = 2 goto L1
  goto L2
  L1: k is good
  L2:

  Here L1 immediately follows unconditional jump statement goto L2. Then the code after elimination of unreachable code is

  i = j
  if k ≠ 2 goto L2
  k is good
  L2:

3. Algebraic simplifications:-

- **Algebraic identities that occur frequently and which is worth considering them can be simplified.**
- **For example, X = X * 1 or X = 0 + X is often produced by straight forwards intermediate code generation algorithms. Hence they can be eliminated easily through peephole optimization.**

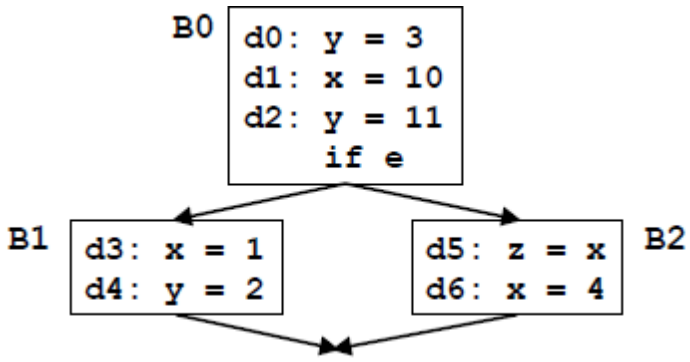
4. Strength reduction:-

- **Replace expensive statements by a cheaper one.**
- **For example, X^2 is expensive operation. Hence replace this by X * X which is cheaper one.**

3 Q: Explain about Data flow analysis

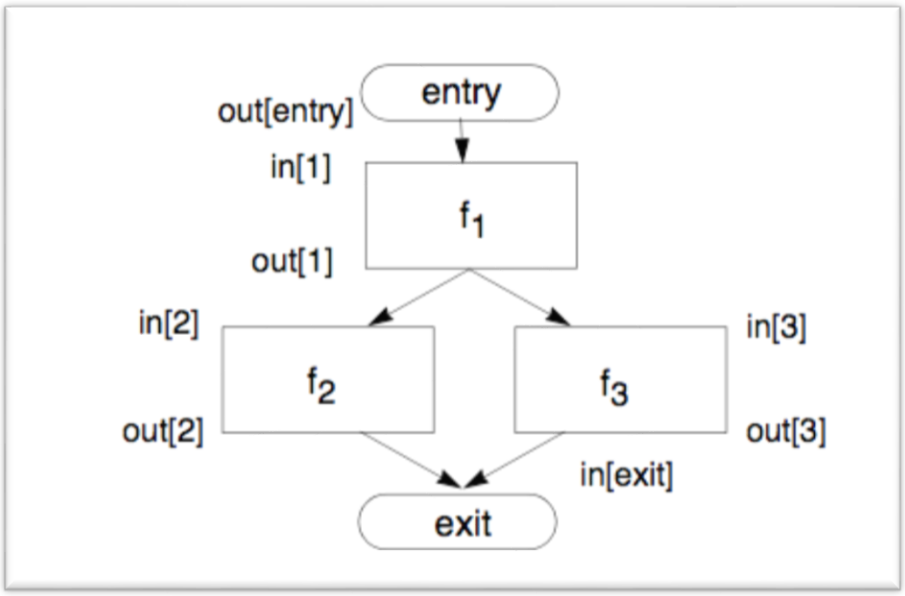
- **Data flow analysis:**
  - **– Flow-sensitive: sensitive to the control flow in a function**
  - **– Intra procedural analysis**
- **Examples of optimizations:**
  - **– Constant propagation**
  - **– Common sub expression elimination**
  - **– Dead code elimination**
- **Data flow analysis abstraction:**
  - **– For each point in the program, combines information of all the instances of the same program point.**

- **Reaching Definitions**



- **Every assignment is a definition**
- **A definition d reaches a point p if there exists path from the point immediately following d to p such that d is not killed (overwritten) along that path.**

- **Data Flow Analysis Schema**



- **Build a flow graph (nodes = basic blocks, edges = control flow)**
- **Set up a set of equations between in[b] and out[b] for all basic blocks b**

Live Variable Analysis

- **Definition**

– A variable v is *live* at point p if the value of v is used along some path in the flow graph starting at p.

– Otherwise, the variable is *dead*.

- **For each basic block, determine if each variable is live in each basic block**