## UNIT – I

## PART – A

## QUESTIONS AND ANSWERS

1. **How will you group the phases of compiler? [MAY-2016]**

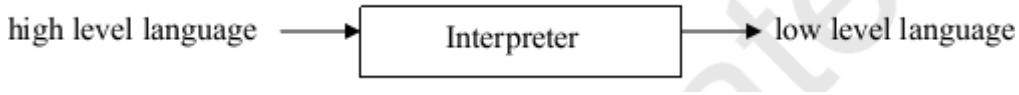
   Compiler can be grouped into front and back ends:

   **Front end:** analysis (machine independent) These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

   **Back end:** synthesis (machine dependent) It includes code optimization phase and code generation along with the necessary error handling and symbol table operations.

2. **What is an interpreter? [MAY-2011][NOV DEC 2017]**

   An interpreter reads a statement from the input converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it.




   Example: Basic, Lower version of Pascal, Perl, Python, MATLAB and Ruby.

3. **Mention few cousins of compiler. [MAY-2012] (Apr May 2017)**

   Cousins of the compiler are:

   - Preprocessors
   - Assemblers
   - Loaders and Link-Editors

4. **What are the possible error recovery actions in lexical analyzer? [MAY-2012]**

1) Deleting an extraneous character.

2) Inserting a missing character.

3) Replacing an incorrect character by a correct character.

4) Transforming two adjacent characters.

5) Panic mode recovery

5. **What is a symbol table?**

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

6. **What are the two parts of a compilation? Explain briefly. [MAY- 2016] (Apr May 2017) (Apr May 2018)**

There are two parts to compilation: ***analysis*** and ***synthesis***.

- The ***analysis part*** breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The ***synthesis part*** constructs the desired target program from the intermediate representation.

7. **What are the various ways to pass a parameter in a function? (Apr May 2018)**

- Call by value
- Call by reference
- Copy-restore
- Call by name

8. **What do you mean by a cross-compiler? (APR-MAY' 14) [NOV DEC 2017]**

A *cross compiler* is a *compiler* capable of creating executable code for a platform other than the one on which the *compiler* is running. For example, a *compiler* that runs on a Windows 7 PC but generates code that runs on Android smart phone is a *cross compiler*.

## PART – B

## QUESTIONS AND ANSWERS

1. **Describe the various phases of compiler and trace it with the program segment (position: = initial + rate * 60). [NOV-2011,2012,2013,MAY-2012,MAY-2013] [nov dec 2014][MAY-2016] (Apr May 2017) [NOV DEC 2017] (Apr May 2018)**

PHASES OF COMPILER

A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:

**1)** Lexical analysis

**2)** Syntax analysis

**3)** Semantic analysis

**4)** Intermediate code generation

**5)** Code optimization

**6)** Code generation

Sub-Phases:

1) Symbol table management

2) Error handling

ANALYSIS OF THE SOURCE PROGRAM

Analysis consists of 3 phases:

- Linear/Lexical Analysis:

It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning.
For example, in the assignment statement a=b+c*2, the characters would be grouped into the following tokens:

i) The identifier1 ‘a’

ii) The assignment symbol (=)

iii) The identifier2 ‘b’

iv) The plus sign (+)

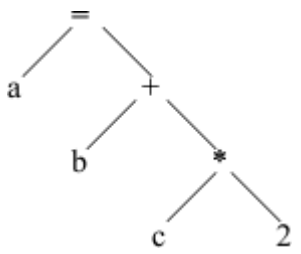
v) The identifier3 ‘c’

vi) The multiplication sign (*)

vii) The constant ‘2’

- Syntax Analysis

- It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- They are represented using a syntax tree as shown below:




A syntax tree is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands.

- This analysis shows an error when the syntax is incorrect.

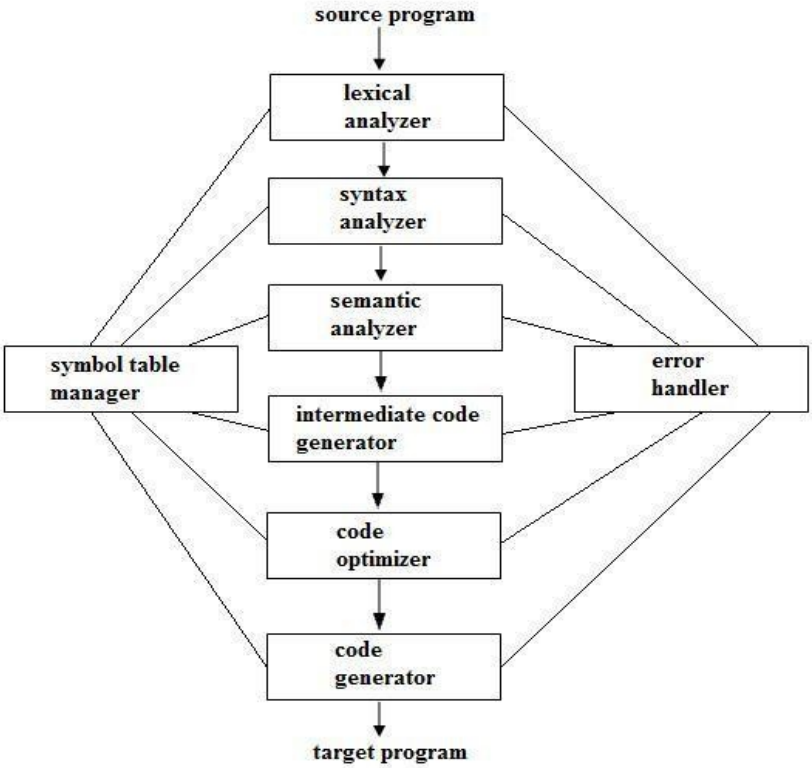
- Semantic Analysis
    - It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase. It uses the syntax tree to identify the operators and operands of statements.
    - An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.
- Intermediate Code Generation
    - ✓ It is the fourth phase of the compiler.
    - ✓ It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
    - ✓ The three-address code consists of a sequence of instructions, each of which has atmost three operands.

    Example: t1=t2+t3

- Code Optimization
    - ✓ It is the fifth phase of the compiler.
    - ✓ It gets the intermediate code as input and produces optimized intermediate code as output.
    - ✓ This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
    - ✓ During the code optimization, the result of the program is not affected.
    - ✓ To improve the code generation, the optimization involves deduction and removal of dead code (unreachable code).
        - calculation of constants in expressions and terms.
        - collapsing of repeated expression into temporary string.
        - loop unrolling.
        - moving code outside the loop.
        - removal of unwanted temporary variables.

Code Generation

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
    - allocation of register and memory
    - generation of correct references
    - generation of correct data types
    - generation of missing code



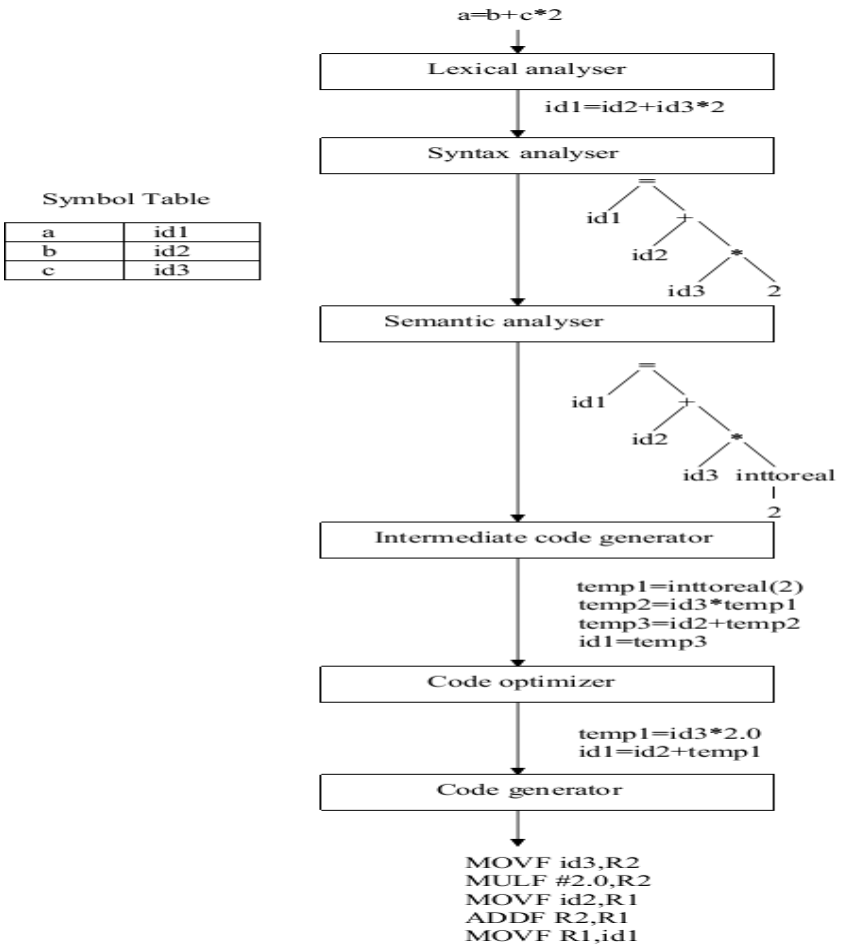

**SYMBOL TABLE MANAGEMENT**

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier , with fields for the attributes of the identifier. It allows to find the record for each identifier quickly and to store or retrieve data from that record. Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

**ERROR HANDLING**

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2




2. **State the complier construction tools. Explain them.** [NOV-2011,MAY-2011,2012] **[nov dec 2014] (Apr May 2017) [NOV DEC 2017] (Apr May 2018)**

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler construction tools:

**1) Parser Generators:**

These produce syntax analyzers, normally from input that is based on a context-free grammar.

-It consumes a large fraction of the running time of a compiler.

-Example-YACC (Yet Another Compiler-Compiler).

**2) Scanner Generator:**

-These generate lexical analyzers, normally from a specification based on regular expressions.

-The basic organization of lexical analyzers is based on finite automation.

**3) Syntax-Directed Translation:**

-These produce routines that walk the parse tree and as a result generate intermediate code.

-Each translation is defined in terms of translations at its neighbor nodes in the tree.

**4) Automatic Code Generators:**

-It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

**5) Data-Flow Engines:**

-It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

**3. Discuss the cousins of compiler. [MAY/NOV-2013] (Apr May 2017)**

COUSINS OF COMPILER

a. Preprocessor
b. Assembler
c. Loader and Link-editor

PREPROCESSOR

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions:

1. Macro processing
2. File Inclusion
3. Rational Preprocessors
4. Language extension

1. Macro processing:

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

2. File Inclusion:

Preprocessor includes header files into the program text. When the preprocessor finds an
#include directive it replaces it by the entire content of the specified file.

3. Rational Preprocessors:

These processors change older languages with more modern flow-of-control and data-structuring facilities.

4. Language extension:

These processors attempt to add capabilities to the language by what amounts to built-in macros.

ASSEMBLER

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code.

**LINKER AND LOADER**

A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

**Three tasks of the linker are:**

1. Searches the program to find library routines used by program, e.g. printf(), math routines.

2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references

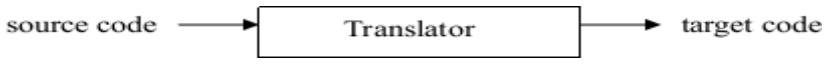
3. Resolves references among files.

A loader is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

4. **Define the following terms: Compiler, Interpreter and Translator and differentiate between them. [**MAY-2014]

Translator:

It is a program that translates one language to another**.**

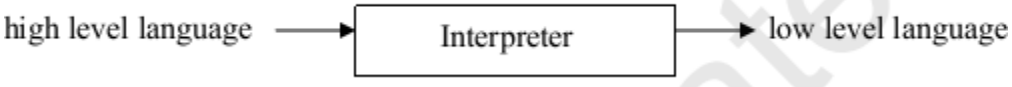



Types of Translator:

1. Interpreter
2. Compiler
3. Assembler

1. Interpreter:

It is one of the translators that translate high level language to low level language .

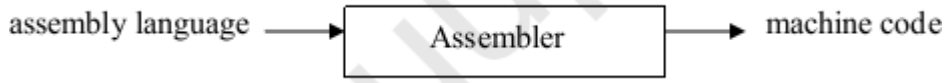



During execution, it checks line by line for errors.

Example: Basic, Lower version of Pascal.

2. Assembler:

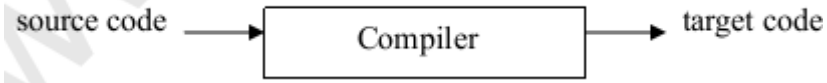
It translates assembly level language to machine code .




Example: Microprocessor 8085, 8086.

3. Compiler:

It is a program that translates one language(source code) to another language (target code).



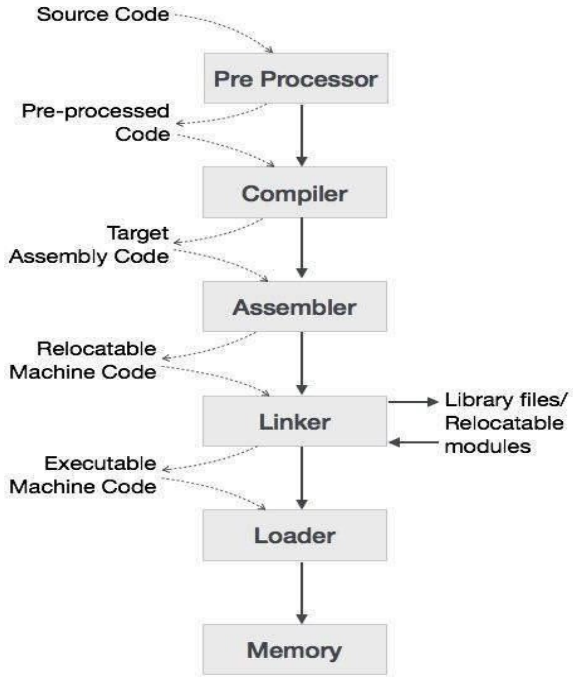

It executes the whole program and then displays the errors.

Example: C, C++, COBOL, higher version of Pascal .

| **Compiler** | **Interpreter** |
|---|---|
| It is a translator that translates high level to low level language | It is a translator that translates high level to low level language |
| A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. | An interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. |
| A compiler reads the whole program even if it encounters several errors. | If an error occurs, an interpreter stops execution and reports it |
| Examples: Basic, lower version of Pascal,Perl,Python,MATLAB and Ruby | Examples: C, C++, Cobol, higher version of Pascal. |

**5.** Explain language processing system with neat diagram. **[MAY-2016] (Apr May 2018)**

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.




The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

How a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. Whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader

Loader is a part of operating system and is responsible for loading executable files into memory and executes them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

**6.Explain about various parameter passing methods in procedure calls (dec 2013)**

Parameters are used to provide the communication between the caller and callee

There are four methods for associating actual and formal parameters.They are,

1.Call by value

2.Call by reference

3.Copy by restore

4.Call by name

**Call by value :**

It is the simplest method of passing parameters

The actual parameters are evaluated and their r-values are passed to the called procedure

This method is used in pascal and c

Call by value can be implemented as follows :

I. A formal parameter is treated just like a local name.So the storage for the formals is in the activation record of the called procedure

II. The caller evaluates the actual parameters and places their r-values in the storage for the formals.

**Call by reference** :

This method is otherwise called as call by address or call by location

If an actual parameter is a name or an expression having an I-value then that I-value itself is passed. However, if the actual parameter is an expression like a+b or constant, that has no I-value, then the expression is evaluated in a new location and the address of that location is passed

**Copy restore**:

This method is a hybrid between (i) and (ii).Also known as copy in copy out or value reset

This calling procedure calculates the value of the actual parameter and it is then copied to activation record for the called procedure. The I-values of these actual parameters having I-values are

determined before the call. When control returns, the current r-values of the formal parameters are copied back into the I-values of the actual, using the i-values computed before the call

**Call by name** :

This procedure is treated like a macro, that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. such a literal substitution is called macro expansion or in line expansion. The local names of the called procedure are kept distinct from the names of the calling procedure. The actual parameters are surrounded by parenthesis if necessary to presence their integrity

## UNIT – II

## PART – A

## QUESTIONS AND ANSWERS

1. **Define tokens, patterns and lexeme. [MAY-2011,2013] [MAY-2016] [Apr/ May 2017] [Nov/Dec 2017]**

**Token:** Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

**Pattern:** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

**Lexeme:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

| Token | lexeme | pattern |
|---|---|---|
| const | const | const |
| if | if | if |
| relation | <,<=,= ,< >,>=,> | < or <= or = or < > or >= or letter followed by letters & digit |
| i | pi | any numeric constant |
| nun | 3.14 | any character b/w “and “except" |
| literal | "core" | pattern |

2. **Mention issues in lexical analyzer. [MAY-2013, MAY-2012,2014] [MAY-2016] (Apr May 2017) [NOV DEC 2017]**

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.

- To enhance the computer portability.

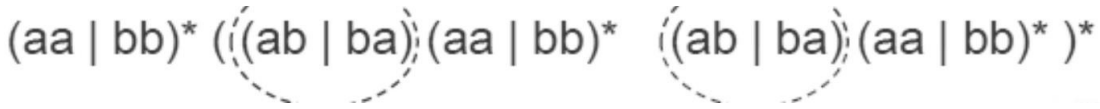
3. **Why is buffering used in lexical analysis? What are the commonly used buffering methods?[MAY-2014]**

   As characters are read from left to right, each character is stored in the buffer to form a meaningful token.
   Types: one buffer scheme, two buffer scheme

4. **What is Sentinel?**

   The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

5. **Write regular expression to describe a language consists of string made of even number a and b [nov dec 2014]**



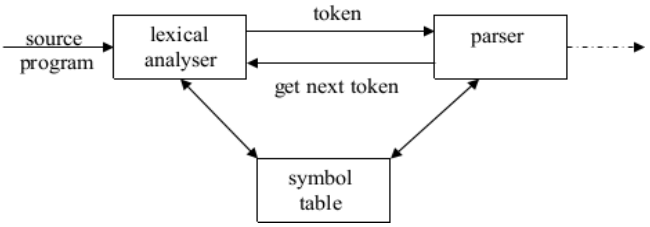

6. **List the operations on languages. [MAY-2016]**

   The various operations on languages are:

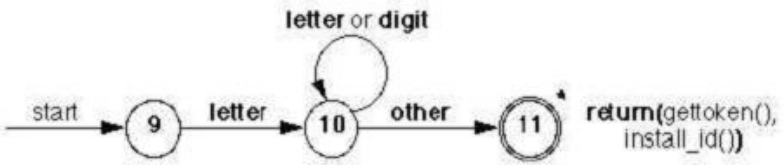
   - Union of two languages L and M is written as L U M = {s | s is in L or s is in M}
   - Concatenation of two languages L and M is written as LM = {st | s is in L and t is in M}
   - The Kleene Closure of a language L is written as L* = Zero or more occurrence of language L.

7. **What is the role of lexical analyzer?[DEC-2011] [NOV DEC '14] [NOV DEC 2017]**

   The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.




**8. Give the transition diagram for an identifier.[DEC-2011]**




**9. Write the Regular expressions for identifier and number.[DEC-2012][APR MAY 2017]**

Identifier :R.E =letter(letter+digit)*

Number : R.E = $digit^{+}(.digit^{+})?(E(+|-)?digit^{+})?$

**10. What are the various parts in LEX program ?[Apr/May 17]**

declarations

%%

pattern specifications

%%

support routines

**11. Compare the features of NFA and DFA.(APR/MAY-2014)(Nov/Dec 2017)**

NFA is a mathematical model that consists of five tuples denoted by

M = {Qn, Σ, δ, q0, fn} ,Qn – finite set of states, Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states, q0 – starting state

fn – final state.

DFA is a special case of a NFA in which

i) no state has an ε-transition.

ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

M = {Qd, Σ, δ, q0, fd},Qd – finite set of states,Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states,q0 – starting state

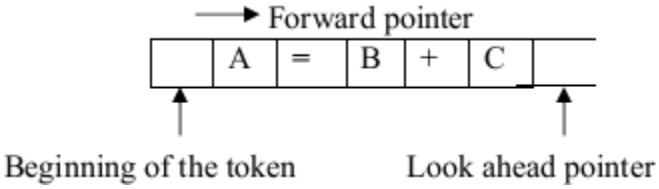
fd – final state

## PART – B

## QUESTIONS AND ANSWERS

**1. Discuss input buffering techniques in detail. [NOV-2013,NOV-2011]**

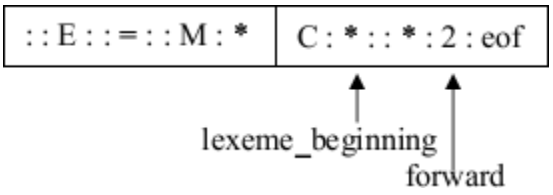
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:




We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

**BUFFER PAIRS**

A buffer is divided into two N-character halves, as shown below




Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.

- Using one system read command we can read N characters into a buffer.

- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file.
- Two pointers to the input are maintained:
- 1. Pointer lexeme_beginning, marks the beginning of the current lexeme, whose extent we are attempting to determine.
- 2. Pointer forward scans ahead until a pattern match is found. Once the next lexeme is determined, forward is set to the character at its right end.
- The string of characters between the two pointers is the current lexeme. After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found.
- **Advancing forward pointer:**
- Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.
- **Code to advance forward pointer:**

```
if
        forward at end of first half then
begin
        reload second half;
        forward := forward + 1
end
else if
        forward at end of second half then
        begin
reload second half;
move forward to beginning of first half
end
else
        forward := forward + 1;
```

2. **Draw the transition diagram for relational operators and unsigned numbers [MAY-2011] (Apr May 2017) (Apr May 2018)**

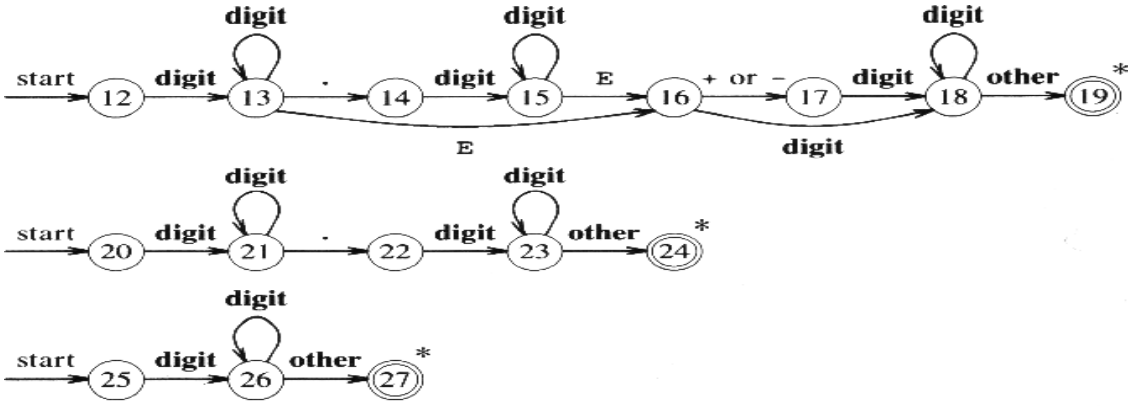
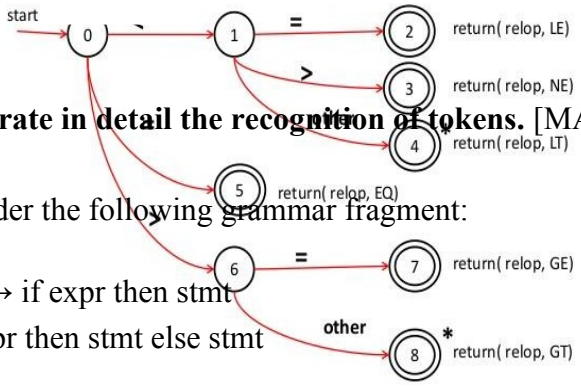



**Fig. 3.14.** Transition diagrams for unsigned numbers in Pascal.




3. **Elaborate in detail the recognition of tokens.** [MAY-2012] **[nov dec 2014]**

Consider the following grammar fragment:

stmt → if expr then stmt
| if expr then stmt else stmt
| ε
expr → term relop term

| term

term → id

| num

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → <|<=|=|<>|>|>=

id → letter(letter|digit)*

num → digit+(.digit+)?(E(+|-)?digit+)?

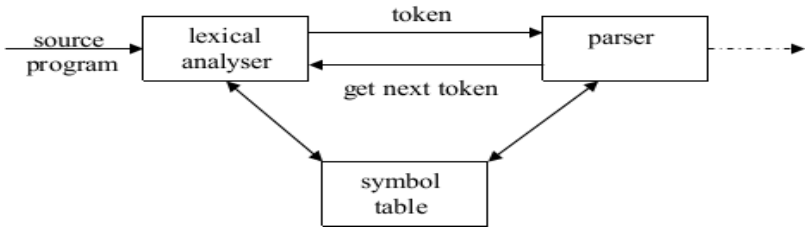
For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers

4. **Explain in detail about the role of lexical analyzer with the possible error recovery actions.[MAY-2013, MAY-2011] (Apr May 2018)**

THE ROLE OF THE LEXICAL ANALYZER

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.



ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

The following are the error-recovery actions in lexical analysis:

- Deleting an extraneous character.
- Inserting a missing character.
- Replacing an incorrect character by a correct character.
- Transforming two adjacent characters.
- Panic mode recovery: Deletion of successive characters from the token until error is resolved.

5. **Elaborate specification of tokens.[**MAY-2013] [MAY-2016]

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1) Strings
2) Language
3) Regular expression

**Strings and Languages**

An alphabet or character class is a finite set of symbols.

A string over an alphabet is a finite sequence of symbols drawn from that alphabet.

A language is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s.

For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

**Operations on strings**

The following string-related terms are commonly used:

1. A prefix ofstring s is any string obtained by removing zero or more symbols from the end of string s.

2. A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example; nana is a suffix of banana.

3. A substring of s is obtained by deleting any prefix and any suffix from s.

For example, nan is a substring of banana.

4. The proper prefixes, suffixes, and substrings of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.

5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s.For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1.Union
2.Concatenation
3.Kleene closure
4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}

2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}

3. Kleene closure : L*={ ε,0,1,00….}

4. Positive closure : L+={0,1,00….}

**Regular Expressions**

Each regular expression r denotes a language L(r).

**Here are the rules that define the regular expressions** over some alphabet Σ and the languages that those expressions denote: **(Apr May 2018)**

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a' in its one position.

3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

a) (r)|(s) is a regular expression denoting the language L(r) U L(s).

b) (r)(s) is a regular expression denoting the language L(r)L(s).

c) (r)* is a regular expression denoting (L(r))*.

d) (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.

5. Concatenation has second highest precedence and is left associative.

6. | has lowest precedence and is left associative.

**Regular set**

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms. For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

**Regular Definitions**

Giving names to regular expressions is referred to as a Regular definition.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter ( letter | digit ) *

**Shorthands**

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them.

1. One or more instances (+):

- The unary postfix operator + means " one or more instances of" .

- If r is a regular expression that denotes the language L(r), then ( r )+is a regular expression that denotes the language (L (r ))+

- Thus the regular expression a+denotes the set of all strings of one or more a's.

- The operator+ has the same precedence and associativity as the operator *.

2. Zero or one instance ( ?):

- The unary postfix operator ? means "zero or one instance of".

- The notation r? is a shorthand for r | ε.

- If 'r' is a regular expression, then ( r )? is a regular expression that denotes the language

L( r ) U { ε }.

3. Character Classes:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression

a | b | c.

- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.

- We can describe identifiers as being strings generated by the regular expression,

[A–Za–z][A–Za–z0–9]*

**Non-regular Set**

A language which cannot be described by any regular expression is a non-regular set.

Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

**6. Differentiate between lexeme, token and pattern. [**MAY-2014][MAY-2016]

TOKENS

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called tokenization. A token can look like anything that is useful for processing an input text stream or text file.

| Lexeme | Token type |
|---|---|
| sum | Identifier |
| = | Assignment operator |
| 3 | Number |
| + | Addition operator |
| 2 | Number |
| ; | End of statement |

LEXEME: **[NOV DEC 2017]**

Collection or group of characters forming tokens is called Lexeme.

PATTERN: A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

## UNIT – III

## PART – A

## QUESTIONS AND ANSWERS

1. **Define handle pruning.[MAY/NOV-2011] (Apr May 2018)**

   A rightmost derivation in reverse can be obtained by handle pruning.If w is a sentence of the grammar at hand, then w = γn, where γn is the nth right-sentential form of some as yet unknown rightmost derivationS = γ0 => γ1…=> γn-1 => γn = w

   For eg, $id_1 + id_2 * id_3$

| RIGHT-SENTENTIAL FORM | HANDLE | REDUCING PRODUCTION |
|---|---|---|
| $id_1 + id_2 * id_3$ | $id_1$ | $E \rightarrow id$ |
| $E + id_2 * id_3$ | $id_2$ | $E \rightarrow id$ |
| $E + E * id_3$ | $id_3$ | $E \rightarrow id$ |
| $E + E * E$ | $E * E$ | $E \rightarrow E * E$ |
| $E + E$ | $E + E$ | $E \rightarrow E + E$ |
| $E$ | | |

2. **Write the rule to eliminate left recursion in a grammar (NOV/DEC- 2012)**

   If a grammar is left recursive (i.e) of the form A => Aα | β Then the left recursion can be eliminated by rewriting the given grammar as

   A => βA'

   A' => αA' | ε

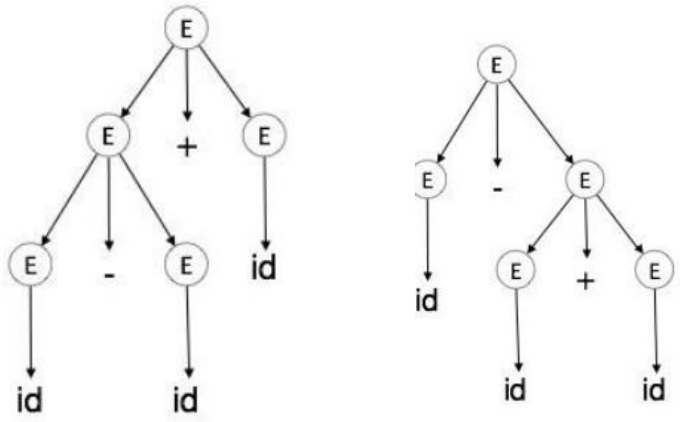
3. **Define an ambiguous grammar. [APR/MAY-2012]**

   A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G). i.e. both leftmost and rightmost derivations are same for the given sentence.

Eg, $E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$




**4. What is dangling reference? [APR/MAY-2012] [MAY-2016]**

Ambiguity can be eliminated by means of dangling-else grammar which is show below:

stmt → if expr then stmt

| if expr then stmt else stmt

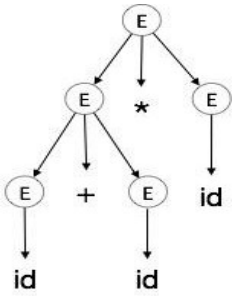
| other

**5. Define parse tree.**

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

**E → id + id * id**




**6. Define left factoring**

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

If there is any production A → αβ1 | αβ2 , it can be rewritten as

A → αA'

A' → β1 | β2

**7. What is recursive descent parsing?**

Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.** This parsing method may involve backtracking, that is, making repeated scans of the input.

**8. Write a CF grammar to represent palindrome [nov dec 14]**

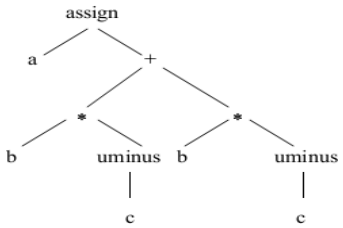
$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$

**9. Eliminate the left recursion for the grammar A->Ac | Aad | bd**

*A -> bdA' | A'*

*A'->cA' | adA' |*ε

**10. Construct the syntax tree for the following assignment statement: a:=b*-c+b*-c. [NOV-2011,2012] [NOV DEC 2017]**




**11. Compare syntax tree and parse tree.[NOV/DEC 2017]**

**syntax tree:** interior nodes are operators and leaves are operands.

Represents the abstract syntax of a program (the semantics)

**Parse tree:** interior nodes are non-terminals, leaves are terminals .Represents the concrete syntax of a program

## PART – B

## QUESTIONS AND ANSWERS

**1. Construct non recursive predictive parsing table for the following grammar**

**E -> E | E and E | not E | (E) | 0 | 1 (dec 2012) [MAY-2016] [NOV DEC 2017]**

**(or)**

**E -> E+T |T**

**T->T*F | F**

**F->(E) | id**

Steps :

1. Compute FIRST and FOLLOW function for the given grammar
2. Construct the predictive parsing table using FIRST and FOLLOW functions
3. Parse the input string with the help of predictive parsing table

Computing FIRST(a) :

FIRST(a) is the set of terminal symbols that are first symbols appearing at right hand side in derivation of a

1. If a=> e, then e is also inc.uded in FIRST(a)
2. If the terminal symbol at RHS is X->a,then FIRST(X)={a}
3. If there is a rule X=>e,then FIRST(X)={e}

4.If A->X1,X2,…$X_k$ ,then FIRST(A)=FIRST(X1) U FIRST(X2) U FIRST($X_k$)

Computing FOLLOW(a):

1. For the start symbol S,place $ in FOLLOW(S)
2. If there is a production A->αBβ,then everything in FIRST(β) without e is to be placed in FOLLOW(B)
3. If there is a production A->αBβ or A->αB where FIRST(β) =e,then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A)

**2. Construct SLR parsing table for the grammar**

**E->E+T|T**

**T_>TF|F**

**F->F*|a|b (dec 2012) (Apr May 2017)**

Steps :

Constructing SLR Parser table

1. Define an augumented grammar S'->S
2. If I is the set of items for a grammar G,then closure(I) is the set of items constructed from I by the following two rules
   i. Initially every item in I is added to closure(I)
   ii. If A->a.Bb is in closure of I and B->c is a production,then add the item B->c to I,if it is not already there
3. Apply the above rule until no more new items can be added to closure(I)

Determination of parsing actions

1.If [A->α.aβ] is in $I_i$ and goto($I_i$,a)=$I_j$,then set action[i,a] to "shift j",where a must be a terminal

if [A->a.] is in $I_i$,then set action [I,a] to "reduce by A->a' " for all a in Follow(A),here A may not be S'

if [S'->S.] is in $I_i$,then set action [i,$] to "accept".

If any confliction actions are generated by above rules,we say the grammar is not SLR

3. The goto transitions for state I are constructed for all Nonterminal using the rule

If goto($I_i$,A)=$I_j$

Then

Goto [i,A]=j

4. All entries that are not defined by rule 2 and 3 are made "Error"

**3. Design an LALR parser for the following grammar and parse the input id=id**

**S->L=R | R (dec 2013) [nov dec 2014]**

**L->*R | id**

**R->L**

LALR stands for lookahead LR parser.

• This is the extension of LR(O) items, by introducing the one symbol of lookahead on the input.

• It supports large class of grammars.

• Most syntactic constructs of programming language can be stated conveniently.

Steps to construct LALR parsing table

• Generate LR(l) items.

• Find the items that have same set of first components (core) and merge these sets into one.

• Merge the goto's of combined itemsets.

• Revise the parsing table of LR(l) parser by replacing states and goto's with combined states and combined goto's respectively.

**4. Explain the error recovery strategies in syntax analysis. (may 2011)**

**Panic Mode**

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

**Statement Mode**

When a parser encounters an error, it tries to take corrective measures so that the rest of the inputs of the statement allow the parser to parse ahead. For example, inserting a missing

semicolon, replacing comma with a semicolon, etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

**Error Productions**

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

**Global Correction**

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

5. **Construct a predictive parser for the following grammar (may 2011) (Apr May 2017)**

S->(L) | a  L->L,S | **S**

Steps :

1. Compute FIRST and FOLLOW function for the given grammar
2. Construct the predictive parsing table using FIRST and FOLLOW functions
3. Parse the input string with the help of predictive parsing table

Computing FIRST(a) :

FIRST(a) is the set of terminal symbols that are first symbols appearing at right hand side in derivation of a

1. If a=> e, then e is also inc.uded in FIRST(a)
2. If the terminal symbol at RHS is X->a,then FIRST(X)={a}
3. If there is a rule X=>e,then FIRST(X)={e}
4. If A->X1,X2,…$X_k$ ,then FIRST(A)=FIRST(X1) U FIRST(X2) U FIRST($X_k$)

Computing FOLLOW(a):

1. For the start symbol S,place $ in FOLLOW(S)

2.If there is a production A->αBβ,then everything in FIRST(β) without e is to be placed in FOLLOW(B)

3.If there is a production A->αBβ or A->aB where FIRST(β) =e,then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A)

6. **Describe the conflicts that may occur during shift reduce parsing (may 2012) (Apr May 2017)**

   There are context free grammars for which shift reduce parsers cannot be used
   Stack contents and the next input symbol may not decide action :

   - Shift/reduce conflict : whether to make a shift operation or a reduction
   - Reduce/reduce conflict : the parser cannot decide which of several reductions to make

7. **Construct a predictive parser for the following grammar (may 2013)**

   **S->(L) | a**
   **L->L,S|S**

Steps :

1.Compute FIRST and FOLLOW function for the given grammar

2.Construct the predictive parsing table using FIRST and FOLLOW functions

3.Parse the input string with the help of predictive parsing table

Computing FIRST(a) :

FIRST(a) is the set of terminal symbols that are first symbols appearing at right hand side in derivation of a

1.If a=> e, then e is also inc.uded in FIRST(a)

2.If the terminal symbol at RHS is X->a,then FIRST(X)={a}

3.If there is a rule X=>e,then FIRST(X)={e}

4.If A->X1,X2,…$X_k$ ,then FIRST(A)=FIRST(X1) U FIRST(X2) U FIRST($X_k$)

Computing FOLLOW(a):

1. For the start symbol S,place $ in FOLLOW(S)
2. If there is a production A->αBβ,then everything in FIRST(β) without e is to be placed in FOLLOW(B)
3. If there is a production A->αBβ or A->αB where FIRST(β) =e,then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A)

**8. List all LR(0) items for the following grammar**

S->AS | b

A->SA | a (may 2013)

Steps :

1. Define an augumented grammar S'->S
2. If I is the set of items for a grammar G,then closure(I) is the set of items constructed from I by the following two rules
   i. Initially every item in I is added to closure(I)
   ii. If A->α.Bβ is in closure of I and B-> γ is a production,then add the item B-> γ to I,if it is not already there
3. Apply the above rule until no more new items can be added to closure(I)

**9. Consider the following grammar**

**S->AS | b**

**A->SA |a**

**Construct the SLR parse table for the grammar.Show the actions of the parser for The input string "abab" (may 2014)**

Steps :

Constructing SLR Parser table

1. Define an augumented grammar S'->S
2. If I is the set of items for a grammar G,then closure(I) is the set of items constructed from I by the following two rules

i. Initially every item in I is added to closure(I)

ii. If A->α.Bβ is in closure of I and B-> γ is a production,then add the item B-> γ to I,if it is not already there

3. Apply the above rule until no more new items can be added to closure(I)

Determination of parsing actions

1.If [A->α.aβ] is in $I_i$ and goto($I_i$,a)=$I_j$,then set action[i,a] to "shift j",where a must be a terminal

if [A->α.] is in $I_i$,then set action [I,a] to "reduce by A->a' " for all a in Follow(A),here A may not be S'

if [S'->S.] is in $I_i$,then set action [i,$] to "accept".

If any confliction actions are generated by above rules,we say the grammar is not SLR

3.The goto transitions for state I are constructed for all Nonterminal using the rule

If goto($I_i$,A)=$I_j$

Then

Goto [i,A]=j

4.All entries that are not defined by rule 2 and 3 are made "Error"

**10. What is an ambiguous grammar?Is the following grammar ambiguous? Prove E->E+E | E*E | (E) | id.The grammar should be moved to the next line,centered. (may 2014) [MAY-2016]**

**Ambiguous grammar :**

A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language L(G).i.e. both leftmost and rightmost derivations are same for the given sentence.Yes the given grammar is ambiguous since it generates two parse for the same input id+id*d

**Removing Ambiguity :** If the grammar has left associative operators such as +,-,* and / , then induce left recursion,else if the grammar has right associative operators such as ^ ,then induce right recursion E->E+E E->E*E E->(E) E->id

E->E+T T->T*F

E->T T->F

**11.** Construct stack implementation of shift reduce parsing for the following grammar.

**[MAY-2016]**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow ( E )$

$E \rightarrow id$ and the input string $id_1 + id_2 * id_3$. Use the shift-reduce parser to check whether the input string is accepted.

| Stack | Input | Action |
|---|---|---|
| $ | $id_1 + id_2 * id_3$ | shift |
| $id_1 | $+ id_2 * id_3$ | reduce by E→id |
| $E | $+ id_2 * id_3$ | shift |
| $E + | $id_2 * id_3$ | shift |
| $E + id_2 | $* id_3$ | reduce by E→id |
| $E + E | $* id_3$ | shift |
| $E + E* | $id_3$ | shift |
| $E + E * id_3 | $ | reduce by E→id |
| $E + E * E | $ | reduce by E→E * E |
| $E + E | $ | reduce by E→ E + E |
| $E | $ | accept |

## UNIT IV- SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

### PART – A

**1. Define Syntax Directed Translation.**

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars. – We associate Attributes to the grammar symbols representing the language constructs. – Values for attributes are computed by Semantic Rules associated with grammar productions.

**2. What are the limitations of static allocation? [APR/MAY-2011]**

1. No recursive procedures
2. No dynamic data structures

**3. What are the types of three address statements? [NOV-2011]**

(i). Assignment statements of the form x : = y op z.

(ii). Assignment instructions of the form x : = op y, where op is a unary operation.

(iii).Copy statements of the form x : = y where the value of y is assigned to x.

(iv). The unconditional jump goto L.

(v).Conditional jumps such as if x relop y goto L.

**4.Define Backpatching [NOV-2009,2012,2013,MAY-2007,2009,2013]**

Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process.In the semantic actions the functions used are mklist(i),merge_list(p1,p2) and backpatch(p,i)

**5. List out the benefits of using machine-independent intermediate forms. [MAY-2011]**

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.

2. A machine-independent code optimizer can be applied to the intermediate representation.

**6. Why are quadruples preferred over triples in an optimizing compiler? [MAY-2012]**

The advantage of Quadruple representation is that one can quickly access the value of temporary variables using symbol table

**7. List out the motivations for back patching. [MAY-2012]**

In Boolean expression suppose there are two expressions E1 and E2.If E1 is false then there is no need to compute rest of the expression. But if we generate the code for E1 we must know where to jump on encountering the false E1 expression. To overcome this backpatching is used.For the flow of control statements the jumping location can be identified using backpatching.

**8. What are the various ways of representing intermediate languages? [NOV-2006,2007,MAY-2013] [nov dec 2014]**

There are mainly three types of intermediate code representations.

Syntax tree, Postfix, Three address code

**9. What is the significance of intermediate code.[MAY-2014]**

The generation of an intermediate code leads to efficient code generation and acts as a effective mediator between front end and back end.

**10. What is the intermediate code representation for the expression a or b and not c?**

t1 := not c
t2 := b and t1
t3 := a or t2

**11. Give the syntax directed definition for if-else statement.[MAY-2006]**

1. S → if E then S1

E.true := new_label()

E.false :=S.next

S1.next :=S.next

S.code :=E.code | | gen_code(E.true ‘: ‘) | | S1.code

2. S → if E then S1 else S2

E.true := new_label()

E.false := new_label()

S1.next :=S.next

S2.next :=S.next

S.code :=E.code||gen_code(E.true‘: ‘)||S1.code|| gen_code(‘go to’,S.next)||gen_code(E.false‘:’) || S2.code

**12. Write the properties of intermediate language.[MAY-2007]**

(i) It is an easy form of source language which can be generated efficiently by the compiler.

(ii) The generation of intermediate language should lead to efficient code generation.

(iii) The intermediate language should act as effective mediator between front and back end.

**13. Write down the equation for two dimensional arrays.[MAY-2007,NOV-2009]**

A[i,j]=((i*n2)+j)*w+(base-((low1*n2)+low2)*w)

**14. Define short circuit code.[NOV-2010]**

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called " short-circuit" or "jumping" code.

**15. Define three address code.[NOV-2009]**

Three-address code is a sequence of statements of the general form x : = y op z where x, y and z are names, constants, or compiler -generated temporaries; op stands for anyoperator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence t1: = y * z t2: = x + t1 where t1and t2 are compiler-generated temporary name

**16. Define static check.(APR/MAY-13)**

A compiler must check that the source program follows both the syntactic and semantic conventions of the source language. This checking, called static checking, ensures that certain kinds of programming errors will be detected and reported Examples of static checks include:

- Type checks
- .Flow-of-control checks
- Uniqueness checks
- Name-related checks

**17. Evaluation of Semantic Rules.**

- Generate Code
- Insert information into the Symbol Table
- Perform Semantic Check
- Issue error messages , etc.

**18. Write the notations of semantic rules.**

There are two notations for attaching semantic rules:

1. Syntax Directed Definitions. High-level specification hiding many implementation details (also called Attribute Grammars).

2. Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

**19. How would you calculate the cost of an instruction?**

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one,since it occupies only one word of memory. The three-address statement a : = b + c can be implemented by many different instruction

sequences :

i) MOV b, R0
ADD c, R0 cost = 6
MOV R0, a

**20. Define symbol table.(APR-MAY'14)**

Symbol table is used to store all the information about identifiers used in the program.It is a data structure containing a record for each identifier, with fields for the attributes of the identifier. It allows to find the record for each identifier quickly and to store or retrieve data from that record. Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

**21. List the fields in an activation record.(NOV-DEC'13)**

The activation record is a block of memory used for managing the information needed by a single execution of a procedure. Various fields f activation record are:

- Temporary variables
- Local variables
- Saved machine registers
- Control link

- Access link
- Actual parameters
- Return values

**22. What are the three storage allocation strategies? [NOV DEC 2017] (Apr May 2018)**

- Static Allocation
- Stack Allocation
- Heap Allocation

**23. What are the functions used to create the nodes of syntax trees?**

- Mknode (op, left, right)
- Mkleaf (id,entry)
- Mkleaf (num, val)

**24. What are the functions for constructing syntax trees for expressions?**

i) The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.

ii) Each node in a syntax tree can be implemented as a record with several fields.

**25. Define static allocations and stack allocations**

Static allocation is defined as lays out for all data objects at compile time. Names are bound to storage as a program is compiled, so there is no need for a run time support package.

Stack allocation is defined as process in which manages the run time as a Stack. It is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end.

**26. Define an attribute. Give the types of an attribute?**

An attribute defines the various properties corresponding with each grammar symbol.Each and production is associated with a set of semantic rules for computing values of the attributes associated with the symbol appearing in that production.

**e.g.,**a type,a value, a memory location,etc..

**Types:**

- ✓ Synthesized attributes.
- ✓ Inheritance attributes.

**27. Define annotated parse tree?**

A parse tree showing the values of attributes at each node is called an annotated parse tree .The process of computing attribute values at the nodes is called annotating parse tree.

**e.g.,** An annotated parse tree for the input 3*5+4n.

construct the annotated parse tree for the input 3*5+4n.

**28. Define dependency graph?**

Dependency graph is a directed graph that depicts the interdependencies among the inherited and synthesized attributes at the nodes in a parse tree.

**e.g.,** Production E->E1+E2

Semantic rule E.val:=E1.val+E2.val

**29. Define Syntax tree.Give an example?**

An (Abstract)syntax tree is a condensed/compressed form pf parse a tree having operators as the root node and operands as leaf nodes.It is useful for representing language constructs.

**e.g.,**Syntax tree for a-4+c

**30. Define DAG.Give an example?**

DAG is a directed acyclic graph.It identifies the common sub-expression in the expression.

**e.g.,** DAG for the expression a+a*(b-c)+(b-c)*d

**Steps:**

p1=make_leaf(**id,**entry-a)

p2=make_leaf(**id,**entry-a)=p1

p3=make_leaf(**id,**entry-b)

p4=make_leaf(**id,**entry-c)

p5=make_node**(‘-‘,**p3,p4)

p6=make_node**(‘*‘,**p1,p5)

p7=make_node**(‘+‘,**p1,p6)

p8=make_leaf(**id,**entry-b)=p3

p9=make_leaf(**id,**entry-c)=p4

p10=make_node(‘-‘,p3,p4)=p5

p11=make_leaf(**id,**entry-d)

p12=make_node(‘*’,p5,p11)

p13=make_node(‘+’,p7,p12)

**31. What are the advantages of generating an intermediate representation?**

- Conversion of the intermediate code from the source program is easy.
- Subsequent processing from the intermediate code is easy.

**32. Define translation scheme?**

- A translation scheme is a CFG in which program fragments called semantic action are embedded within the right sides of productions.
- A translation scheme is like a syntax-directed definition,except that the order of evaluation of the semantic rules is explicitly shown.

**33. What are the parameter transmission mechanisms?**

- Call-by-value

- Call-by-reference
- Call-by-name

**35. What are the various data structures used for implemention the symbol table?**

- Linear list
- Binary tree
- Hash table

**36. Give short note about call-by-name?**

Call by name textually substitutes the argument expression in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters,much like call-by-reference.

**37. How parameters are passed to procedure in call-by-value method?**

The calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record.Formal parameter then hold the values passed by the calling procedure.If the values held by the formal parameters are changed,it should have no impact on the actual parameters.

**38. What are the semantic rules that are defined in the declarations operation?**

- Make_table(previous)
- Enter (table,name,type,offset)
- Add width(table,width)
- Enter proc(table,name new table).

**39. Give the two attributes of syntax-directed translation into three-address code?**

- E.place,the name that will hold the value of E and
- E.code,the sequence of three-address statements evaluating E.

**40. Give the two parts of basic hashing scheme?**

- A hash table consisting of a fixed array of m pointers to table entries.

- Table entries are organized into m separate linked lists, called buckets. Each record in the symbol table appears on exactly one of these lists.

## PART – B

## QUESTIONS AND ANSWERS

**1. What are the different storage allocation strategies ?Explain (dec 2011) [may 2011,dec 2013] [NOV DEC 2014] [MAY-2016] (Apr May 2017) (APR/MAY 2018)**

**Storage allocation Strategies:**

Runtime environment manages runtime memory requirements for the following entities:

- Code : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- Procedures : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- Variables : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

**Static Allocation**

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage are known in advance, runtime support package for memory allocation and de-allocation is not required.

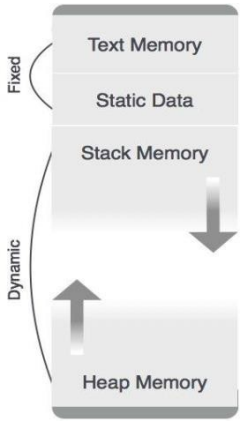
**Stack Allocation**

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

**Heap Allocation**

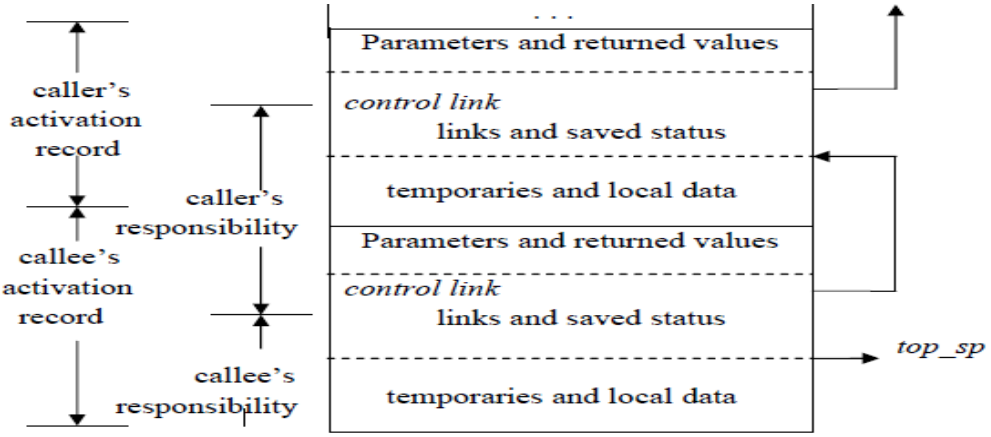
Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.




Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.

Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.



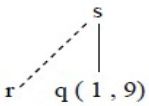
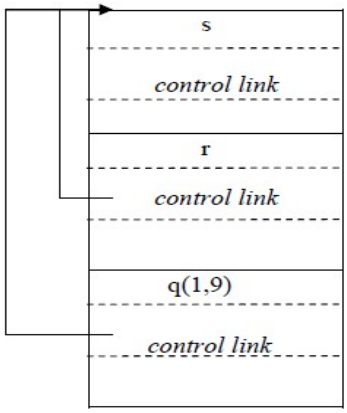

**Division of tasks between caller and callee**

**HEAP ALLOCATION**

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

| Position in the activation tree | Activation records in the heap | Remarks |
|---|---|---|
|   |   | Retained activation record for r |

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

**2.Give the translation scheme for converting the assignments into three address code. [MAY/DEC-2011,MAY-2013]**

Translation scheme to produce three-address code for assignments

S ->id : = E { p : = lookup ( id.name);

if p ≠ nil then

emit( p ‘ : =’ E.place)

else error }

E -> E1 + E2 { E.place : = newtemp;

emit( E.place ‘: =’ E1.place ‘ + ‘ E2.place ) }

E ->E1 * E2 { E.place : = newtemp;

emit( E.place ‘: =’ E1.place ‘ * ‘ E2.place ) }

E -> - E1 { E.place : = newtemp;

emit ( E.place ‘: =’ ‘uminus’ E1.place ) }

E -> ( E1 ) { E.place : = E1.place }

E ->id { p : = lookup ( id.name);

if p ≠ nil then

E.place : = p

else error }

**3. Describe the various methods of implementing three-address statements. [MAY/DEC-2011,MAY/NOV-2013] [MAY-2016]**

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.

Three such representations are:

- Quadruples
- Triples
- Indirect triples

**Quadruples:**

A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result.The op field contain an internal code for the operator. The three -address statement x: =y op z is represented by placing y in arg1, z in arg2 and x in result.

The contents of fields arg1, arg2 and result are normally pointers to the symbol -table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

**Triples**

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.If we do so,three-address statements can be represented by records with only three fields:

op, arg1 and arg2.

The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).Since three fields are used, this intermediate code format is known as triples.

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

**(a) Quadruples**

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

**(b) Triples**

Quadruple and triple representation of three-address statements given above.

**Indirect Triples**

Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be

represented as follows

| | *statement* |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

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

**Indirect triples representation of three-address statements**

**4.How can Backpatching be used to generate code for Boolean expressions and flow of control statements? [DEC-2011, MAY-2013,2014]**

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth –first order, computing the translations. The main problem with generating code for Boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

- makelist(i) creates a new list containing only i, an index into the array of quadruples;
- makelist returns a pointer to the list it has made.
- merge(p1,p2) concatenates the lists pointed to by p1and p2
- backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

**Boolean Expressions**

We now construct a translation scheme suitable for producing quadruples for Boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) E ->E1or M E2

(2) | E1 and M E2

(3) | not E1

(4) | ( E1)

(5) | id1 relop id2

(6) | true

(7) | false

(8) M -> ε

Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping cod for Boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by E.truelist and E.falselist. Attribute M.quad records the number of the first statement of E2.code. With the production M ->ε we associate the semantic action { M.quad : = nextquad }The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E1.truelist when we have seen the remainder of the production E ->E1 and M E2.The translation scheme is as follows:

(1) E -> E1or M E2 { backpatch ( E1.falselist, M.quad);

E.truelist : = merge( E1.truelist, E2.truelist);

E.falselist : = E2.falselist }

(2) E -> E1 and M E2 { backpatch ( E.truelist, M.quad);

E.truelist : = E2.truelist;

E.falselist : = merge(E1.falselist, E2.falselist) }

(3) E -> not E1 { E.truelist : = E1.falselist;

E.falselist : = E1.truelist; }

(4) E -> ( E1) { E.truelist : = E1.truelist;

E.falselist : = E1.falselist; }

(5) E -> id1 relop id2 { E.truelist : = makelist (nextquad);

E.falselist : = makelist(nextquad + 1);

emit(‘if’ id1.place relop.op id2.place ‘goto_’)

emit(‘goto_’) }

(6) E ->true { E.truelist : = makelist(nextquad);

emit(‘goto_’) }

(7) E -> false { E.falselist : = makelist(nextquad);

emit(‘goto_’) }

(8) M -> ε { M.quad : = nextquad }

**5. Write short notes of Flow-of-Control Statements: [MAY-2016]**

A translation scheme is developed for statements generated by the following grammar :

(1) S -> if E then S

(2) | if E then S else S

(3) | while E do S

(4) | begin L end

(5) | A

(6) L -> L ; S

(7) | S

Here S denotes a statement, L a statement list, A an assignment statement, and E a Boolean expression.

Scheme to implement the Translation:

The non terminal E has two attributes E.truelist and E.falselist. L and S also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes L..nextlist and S.nextlist.

The semantic rules for the revised grammar are as follows:

(1) S -> if E then M1S1 N else M2S2

{ backpatch (E.truelist, M1.quad);

backpatch (E.falselist, M2.quad);

S.nextlist : = merge (S1.nextlist, merge (N.nextlist, S2.nextlist)) }

We backpatch the jumps when E is true to the quadruple M1.quad, which is the beginning of the code for S1.Similarly, we backpatch jumps when E is false to go to the beginning of the code for S2. The list S.nextlist includes all jumps out of S1 and S2, as well as the jump generated by N.

(2) N -> ε { N.nextlist : = makelist( nextquad );
emit('goto _') }

(3) M -> ε { M.quad : = nextquad }

(4) S -> if E then M S1

{ backpatch( E.truelist, M.quad);

S.nextlist : = merge( E.falselist, S1.nextlist) }

(5) S -> while M1 E do M2S1

{ backpatch( S1.nextlist, M1.quad);

backpatch( E.truelist, M2.quad);

S.nextlist : = E.falselist

emit( 'goto' M1.quad ) }

(6) S -> begin L end { S.nextlist : = L.nextlist }

(7) S -> A { S.nextlist : = nil }

The assignment S.nextlist : = nil initializes S.nextlist to an empty list.

(8) L -> L1 ; M S { backpatch( L1.nextlist, M.quad);

L.nextlist : = S.nextlist }

The statement following L1in order of execution is the beginning of S. Thus the L1.nextlist list is backpatched to the beginning of the code for S, which is given by M.quad.

(9) L ->S { L.nextlist : = S.nextlist }

**6.Write a short note on procedures calls. [NOV-2011,2012]**

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run –time routines that handle procedure argument passing, calls and returns are part of the run –time support package.

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence, when a procedure call occurs, space must be allocated for the activation record of the called procedure. The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.

Environment pointer must be established to enable the called procedure to access data in enclosing blocks. The state of the calling procedure must be saved so it can resume execution after the call.

Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished. Finally a jump to the beginning of the code for the called procedure must be generated.

**7. Generate 3 address code for the Boolean expression a<b or c<d and e<f. [NOV-2012]**

Translation of a < b or c < d and e < f

100 : if a < b goto 103

101 : t1: = 0

102 : goto 104

103 : t1: = 1

104 : if c < d goto 107

105 : t2: = 0

106 : goto 108

107 : t2: = 1

108 : if e < f goto 111

109 : t3: = 0

110 : goto 112

111 : t3: = 1

112 : t4: = t2and t3

113 : t5: = t1or t4

**8.Translate the executable statements of the following c program into 3 address code[NOV-2012]**

```
main()
{
int i,a[10];
i=1;
while(i<=10)
{
a[i]=0;i=i+1;}}
```

100: i:=1

101: if i<=10 then goto 103

102: goto 108

103: t1 :=i

104: t2;=4*t1

105: a[t2]:=0

106: i:=i+1

107: goto 101

108: stop

**9.Translate the following switch and while statement into intermediate code. [NOV-2012,MAY-2014] [nov dec 2014]**

**Switch E**

**Begin**

**case V1:S1**

**case V2:S2**

**case Vn-1:Sn-1**

**default:Sn**

**end**

Translation of a case statement

code to evaluate E into t

goto test

L1: code for S1

goto next

L2: code for S2

goto next

. . .

Ln-1: code for Sn-1

goto next

Ln: code for Sn

goto next

test : if t = V1 goto L1

if t = V2 goto L2

. . .

if t = Vn-1

goto Ln-1

goto Ln

next :

To translate into above form :

When keyword switch is seen, two new labels test and next, and a new temporary t are generated. As expression E is parsed, the code to evaluate E into t is generated. After processing E ,the jump goto test is generated. As each case keyword occurs, a new label Li is created and entered into the symbol table.A pointer to this symbol-table entry and the value Vi of case constant are placed on a stack (used only to store cases). Each statement case Vi : Si is processed by emitting the newly created label Li , followed by the code for Si, followed by the jump goto next. Then when the keyword end terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack fromthe bottom to the top, we can generate a sequence of three -address statements of the form

case V1 L1

case V2 L2

. . .

case Vn-1 Ln-1

case t Ln

label next

where t is the name holding the value of the selector expression E, and Ln is the label for the default statement.

**10.Write the grammar and translation scheme for procedure call statements. [MAY-2012,NOV-2013,MAY-2008]**

Let us consider a grammar for a simple procedure call statement

(1) S ->call id ( Elist )

(2) Elist ->Elist , E

(3) Elist ->E

For example, consider the following syntax-directed translation

(1) S ->call id ( Elist ) { for each item p on queue do

emit (‘ param’ p );

emit (‘call’ id.place) }

(2) Elist ->Elist , E { append E.place to the end of queue }

(3) Elist -> E { initialize queue to contain only E.place }

Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement.

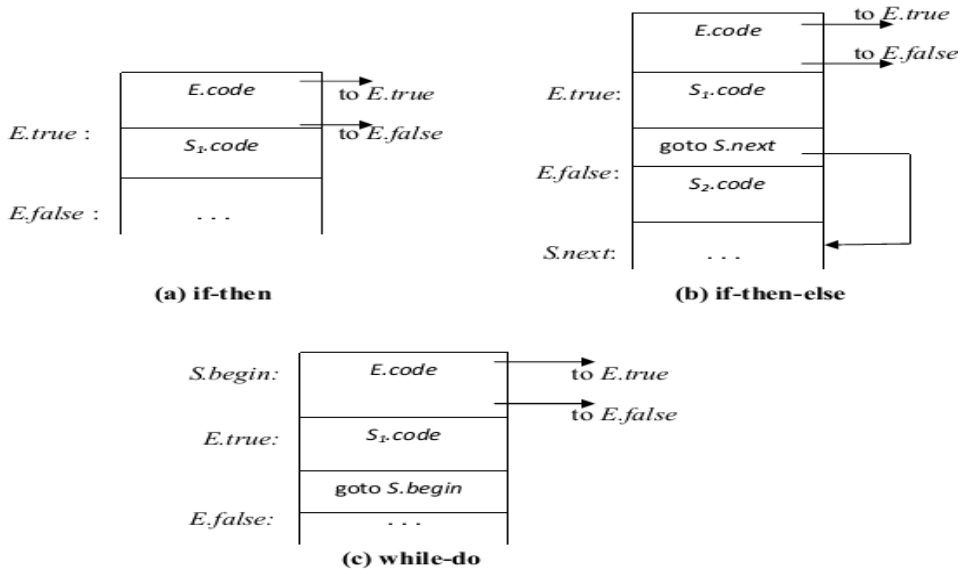
queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

**11.Write the translation scheme for flow of control statements. [NOV-2013, MAY-2013]**

Consider the translation of boolean expressions into three-address code in thecontext of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S -> if E then S1| if E then S1else S2| while E do S1

**Code for if-then , if-then-else, and while-do statements**




**Syntax-directed definition for flow-of-control statements**

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $S \rightarrow$ **if** $E$ **then** $S_1$ | $E.true := newlabel;$<br>$E.false := S.next;$<br>$S_1.next := S.next;$<br>$S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$ |
| $S \rightarrow$ **if** $E$ **then** $S_1$ **else** $S_2$ | $E.true := newlabel;$<br>$E.false := newlabel;$<br>$S_1.next := S.next;$<br>$S_2.next := S.next;$<br>$S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$<br>$gen('\textbf{goto}\ ' S.next) \mid\mid$<br>$gen(E.false\ ':') \mid\mid S_2.code$ |
| $S \rightarrow$ **while** $E$ **do** $S_1$ | $S.begin := newlabel;$<br>$E.true := newlabel;$<br>$E.false := S.next;$<br>$S_1.next := S.begin;$<br>$S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$<br>$gen(E.true\ ':') \mid\mid S_1.code \mid\mid$<br>$gen('\textbf{goto}\ ' S.begin)$ |

**12.Discuss the various methods for translating Boolean expression.[MAY-2011,2012]**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false. To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow -of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

The translation for a or b and not c is the three-address sequence t1 : = not c

t2: = b and t1

t3: = a or t2

A relational expression such as a < b is equivalent to the conditional statement

if a < b then 1 else 0 which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

100 : if a < b goto 103

101 : t : = 0

102 : goto 104

103 : t : = 1

104 :

**Translation scheme using a numerical representation for booleans**

| | |
|---|---|
| $E \rightarrow E_1$ **or** $E_2$ | { *E.place : = newtemp;*<br>*emit( E.place ': =' E1.place* **'or'** *E2.place ) }* |
| $E \rightarrow E_1$ **and** $E_2$ | { *E.place : = newtemp;*<br>*emit( E.place ': =' E1.place* **'and'** *E2.place ) }* |
| $E \rightarrow$ **not** $E_1$ | { *E.place : = newtemp;*<br>*emit( E.place ': ='* **'not'** *E1.place ) }* |
| $E \rightarrow ( E_1 )$ | { *E.place : = E1.place }* |
| $E \rightarrow$ **id**$_1$ **relop id**$_2$ | { *E.place : = newtemp;*<br>*emit( 'if'* **id**$_1$*.place* **relop***.op* **id**$_2$*.place* **'goto'** *nextstat + 3);*<br>*emit( E.place ': ='* **'0'** *);*<br>*emit(***'goto'** *nextstat +2);*<br>*emit( E.place ': ='* **'1'** *') }* |
| $E \rightarrow$ **true** | { *E.place : = newtemp;*<br>*emit( E.place ': ='* **'1'** *') }* |
| $E \rightarrow$ **false** | { *E.place : = newtemp;*<br>*emit( E.place ': ='* **'0'** *') }* |

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called " short-circuit" or "jumping" code. It is possible to evaluate boolean expressions without generating code for the boolean operators and, or, and not if we represent the value of an expression by a posit ion in the code sequence.

**Control-Flow Translation of Boolean Expressions:**

**Syntax-directed definition to produce three-address code for booleans**

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $E \rightarrow E_1$ **or** $E_2$ | *E1.true : = E.true;*<br>*E1.false : = newlabel;*<br>*E2.true : = E.true;*<br>*E2.false : = E.false;*<br>*E.code : = E1.code \|\| gen(E1.false ':') \|\| E2.code* |
| $E \rightarrow E_1$ **and** $E_2$ | *E.true : = newlabel;*<br>*E1.false : = E.false;*<br>*E2.true : = E.true;*<br>*E2.false : = E.false;*<br>*E.code : = E1.code \|\| gen(E1.true ':') \|\| E2.code* |
| $E \rightarrow$ **not** $E_1$ | *E1.true : = E.false;*<br>*E1.false : = E.true;*<br>*E.code : = E1.code* |
| $E \rightarrow ( E1 )$ | *E1.true : = E.true;* |

**13. Describe the various types of three address statements. [MAY-2012]**

The common three-address statements are:

1. Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and con version operators that, for example, convert a fixed-point number to a floating-point number.
3. Copy statements of the form x : = y where the value of y is assigned to x.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc. ) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.
6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. For example, param x1 param x2. . .param xn call p,n generated as part of a call of the procedure p(x1, x2, …. ,xn).
7. Indexed assignments of the form x : = y[i] and x[i] : = y.
8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

**14. Specify a type checker which can handle expressions, statements and functions (or) Discuss specification of simple type checker. (dec 2013) [nov dec 2014] [MAY-2016] (Apr May 2017) [NOV DEC 2017]**

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E

D → D ; D | id : T

T → char | integer | array [ num ] of T | ↑ T

E → literal | num | id | E mod E | E [ E ] | E ↑

Translation scheme:

P → D ; E

D → D ; D

D → id : T { addtype (id.entry , T.type) }

T → char { T.type : = char }

T → integer { T.type : = integer }

T → ↑ T1 { T.type : = pointer(T1.type) }

T → array [ num ] of T1 { T.type : = array ( 1… num.val , T1.type) }

In the above language,

→ There are two basic types : char and integer ;

→ type_error is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example, ↑ integer leads to the type expression pointer ( integer ).

Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1. E → literal { E.type : = char }

E → num { E.type : = integer }

Here, constants represented by the tokens literal and num have type char and integer.

2. E → id { E.type : = lookup ( id.entry ) }

lookup ( e ) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 mod E2 { E.type : = if E1. type = integer and
E2. type = integer then integer
else type_error }

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type_error.

4. E → E1 [ E2 ] { E.type : = if E2.type = integer and
E1.type = array(s,t) then t
else type_error }

In an array reference E1 [ E2 ] , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

5. E → E1 ↑ { E.type : = if E1.type = pointer (t) then t else type_error }

The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

S → id : = E { S.type : = if id.type = E.type then void
else type_error }

2. Conditional statement:

S → if E then S1 { S.type : = if E.type = boolean then S1.type
else type_error }

3. While statement:

S → while E do S1 { S.type : = if E.type = boolean thenS1.type
else type_error }

4. Sequence of statements:

S → S1 ; S2 { S.type : = if S1.type = void and
S1.type = void then void
else type_error }

Type checking of functions

The rule for checking the type of a function application is :

E → E1 ( E2) { E.type : = if E2.type = s and

E1.type = s → t then t

else type_error }

**15. Explain Syntax Directed Definitions [MAY-2016]**

Dependency Graphs

S-Attributed Definitions

L-Attributed Definitions

Dependency Graphs

• Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes

– Each attribute value must be available when a computation is performed.

• Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.

• A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree. – There is a node for each attribute; – If attribute b depends on an attribute c there is a link from the node for c to the node for b ( b ← c).

• Dependency Rule: If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

S-Attributed Definitions

• Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.

• The parser keeps the values of the synthesized attributes in its stack.

• Whenever a reduction A → α is made, the attribute for A is computed from the attributes of α which appear on the stack.

• Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

L-Attributed Definitions

• L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build dependency graph to evaluate them.

• Definition. A syntax directed definition is L-Attributed if each inherited attribute of Xj in a production A → X 1 . . . Xj . . . X n, depends only on: 1. The attributes of the symbols to the left (this is what L in L-Attributed stands for) of Xj , i.e., X 1 X 2 . . . Xj − 1, and 2. The inherited attributes of A.

• Theorem. Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree

Evaluating L-Attributed Definitions

• L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.

• The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)

Input: Node of an annotated parse-tree.

Output: Attribute evaluation.

```
Begin
        For each child m of n, from left-to-right Do

Begin
        Evaluate inherited attributes of m;
         L-Eval(m)

 End;
         Evaluate synthesized attributes of n

End.
```

**16. Evaluation of S-Attributed Definitions**

• Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.

• The parser keeps the values of the synthesized attributes in its stack.

• Whenever a reduction A→ α is made, the attribute for A is computed from the attributes of α which appear on the stack.

• Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields: *state* and *val*

| *state* | *val* |
|---|---|
| $Z$ | $Z.x$ |
| $Y$ | $Y.x$ |
| $X$ | $X.x$ |
| ... | ... |

The current top of the stack is indicated by the pointer *top*.

- Synthesized attributes are computed just before each reduction:

Before the reduction A → XY Z is made, the attribute for A is computed:

A.a := f(val[top], val[top − 1], val[top − 2]). Extending

a Parser Stack: An Example

- Example. Consider the S-attributed definitions for the arithmetic expressions.To Evaluate attributes the parser executes the following code

| RODUCTION | ODE |
|---|---|
| → En | *rint*(val[top − 1]) |
| → E1 + T | al[ntop] := val[top] + val[top −2] |
| → T | |
| → T1 ∗F | l[ntop] := val[top] ∗val[top − 2] al[ntop] := |
| → F | |
| → (E) | val[top − 1] |
| → digit | |

The variable ntop is set to the *new top of the stack*. After a reduction is done *top* is set to *ntop*:

When a reduction A → α is done with |α| = r, then ntop = top− r + 1.During a shift action both the token and its value are pushed into the stack.

- The following Figure shows the moves made by the parser on input 3*5+4n.

– Stack states are replaced by their corresponding grammar symbol;

– Instead of the token digit the actual value is shown.

| INPUT | *state* | *val* | PRODUCTION USED |
|---|---|---|---|
| 3*5+4n | _ | _ | |
| *5+4n | 3 | 3 | |
| *5+4n | *F* | 3 | *F* → **digit** |
| *5+4n | *T* | 3 | *T* → *F* |
| 5+4n | *T* * | 3 _ | |
| +4n | *T* * 5 | 3 _ 5 | |
| +4n | *T* * *F* | 3 _ 5 | *F* → **digit** |
| +4n | *T* | 15 | *T* → *T* * *F* |
| +4n | *E* | 15 | *E* → *T* |
| 4n | *E* + | 15 _ | |
| n | *E* + 4 | 15 _ 4 | |
| n | *E* + *F* | 15 _ 4 | *F* → **digit** |
| n | *E* + *T* | 15 _ 4 | *T* → *F* |
| n | *E* | 19 | *E* → *E* + *T* |
| | *E* n | 19 _ | |
| | *L* | 19 | *L* → *E* n |

## PART – A

1. **Mention the issues to be considered while applying the techniques for code optimization**

- The semantic equivalence of the source program must not be changed.
- The improvement over the program efficiency must be achieved without changing the algorithm of the program.

2. **Name the techniques in loop optimization. (APR-MAY'14) (Apr May 2018)**

Three techniques are important for loop optimization:

- code motion, which moves code outside a loop.
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

3. **Define basic blocks and flow graphs.(APR-MAY'13)(APR-MAY'12)(NOV-DEC'12) (NOV-DEC'11)[nov dec 2014] (Apr May 2017)**

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end. Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks. It has a distinguished initial node.

4. **What are the steps involved in partitioning a Sequence of three address statements into basic blocks?**

**Algorithm:** Partition into basic blocks

**Input:** A sequence of three-address statements

**Output:** A list of basic blocks with each three-address statement in exactly one block

**Method:**

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:

a. The first statement is a leader.

b. Any statement that is the target of a conditional or unconditional goto is a leader.

c. Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

**5. Give any 3 applications of DAG.(APR-MAY'13)**

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

**6. What is dead code elimination? (APR-MAY'11)**

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : =y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

**7. Give the primary structure preserving transformations on Basic Blocks. (APR- MAY'11) (Apr May 2018)**

1. Common subexpression elimination
2. Dead code elimination
3. Renaming temporary variables.
4. Interchange of statements.

**8. What are DAGS and how are they useful in implementing transformations on basic blocks? (NOV-DEC'11) [MAY-2016] (Apr May 2017)**

A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks.It gives a picture of how the value computed by a statement is used in subsequent statements.It provides a good way of determining common sub - expressions.

**9. What is the use of Next-use information? (NOV-DEC'13)**

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Example:

| **Names** | **liveliness** | **Next use** |
|---|---|---|
| x | Not live | No next use |
| Y | Live | i |
| Z | Live | i |

**10. What are the uses of register and address descriptors? (NOV-DEC'12)**

A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

An address descriptor stores the location where the current value of the name can be found at run time.

**11. What are machine idioms?**

A target machine might have some special instructions called machine idioms.

Eg: x=x+1 is called auto increment and x=x-1 is called auto decrement instructions.

**12. What are the basic goals of code movement?**

- To reduce the size of the code i.e. to obtain the space complexity.
- To reduce the frequency of execution of code i.e. to obtain the time complexity.

**13. What do you mean by machine dependent and machine independent optimization?**

- The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.
- The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

**14. What are the different data flow properties?**

- Available expressions
- Reaching definitions
- Live variables
- Busy variables

**15. List the different storage allocation strategies. [MAY-2016]**

The strategies are:

- Static allocation
- Stack allocation
- Heap allocation

**16. What are the contents of activation record?(APR/MAY 2018)**

The activation record is a block of memory used for managing the information needed by a single execution of a procedure. Various fields f activation record are:

- Temporary variables
- Local variables
- Saved machine registers
- Control link
- Access link
- Actual parameters
- Return values

**17. What is dynamic scoping?**

In dynamic scoping a use of non-local variable refers to the non-local data declared in most recently called and still active procedure. Therefore each time new findings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

**18. What is code motion?**

Code motion is an optimization technique in which amount of code in a loop is decreased. This transformation is applicable to the expression that yields the same result independent of the number of times the loop is executed. Such an expression is placed before the loop.

**6. What are the properties of optimizing compiler? (APR-MAY' 13) , (NOV-DEC '13) [MAY-2016] [NOV DEC 2017]**

The source code should be such that it should produce minimum amount of target code. There should not be any unreachable code. Dead code should be completely removed from source language.The optimizing compilers should apply following code improving transformations on source language.

i) common subexpression elimination

ii) dead code elimination

iii) code movement

iv) strength reduction

**19. Suggest a suitable approach for computing hash function.**

Using hash function we should obtain exact locations of name in symbol table.The hash function should result in uniform distribution of names in symbol table.The hash function should be such that there will be minimum number of collisions. Collision is such a situation where hash function results in same location for storing the names.

**20. What is called constant folding? (APR-MAY' 13)**

We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example, a=3.14157/2 can be replaced by a=1.570 thereby eliminating a division operation.

**21. What is the use of algebraic identities in optimization of basic blocks? (APR-MAY' 12)**

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

x:=x+0 can be removed. x:=y**2 can be replaced by a cheaper statement x:=y*y

**22. List out two properties of reducible flow graph? (APR-MAY' 12)**

Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguous defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently. Exclusive use of structured flow-of- control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

**23. Define dead-code elimination. (APR-MAY' 11)**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

**24. What is loop optimization? (APR-MAY' 11)**

A very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of codeoutside that loop.

Three techniques are important for loop optimization:

- code motion, which moves code outside a loop;
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

**25. Define loop unrolling with example. (NOV-DEC '13)**

Loop unrolling, also known as loop unwinding, is *a loop transformation* technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler. The goal of loop unwinding is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration, reducing branch penalties; as well as "hiding latencies, in particular, the delay in reading data from memory". To eliminate this overhead, loops can be re-written as a repeated sequence of similar independent statements.

**26. Define live variable .(NOV-DEC'12)**

A variable is **live** if it holds a value that may be needed in the future.

L1: b := 3;
L2: c := 5;
L3: a := f(b + c);
goto L1; The live variables are a,b,c.

**27. What is Data flow analysis? (NOV-DEC'12)**

A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

**28. List down the criteria for code improving transformations.(NOV-DEC'11)**

Simply stated, the best program transformations are those that yield the most benefit for the least effort. The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program.

A transformation must, on the average, speed up programs by a measurable amount. The transformation must be worth the effort. It does not make sense for a compiler writer to expand the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

**29.** What is data flow analysis? [nov dec 2014]

A program's control **flow** graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program.

**30.** What do you mean by copy propogation? **(Apr May 2017)**

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another.

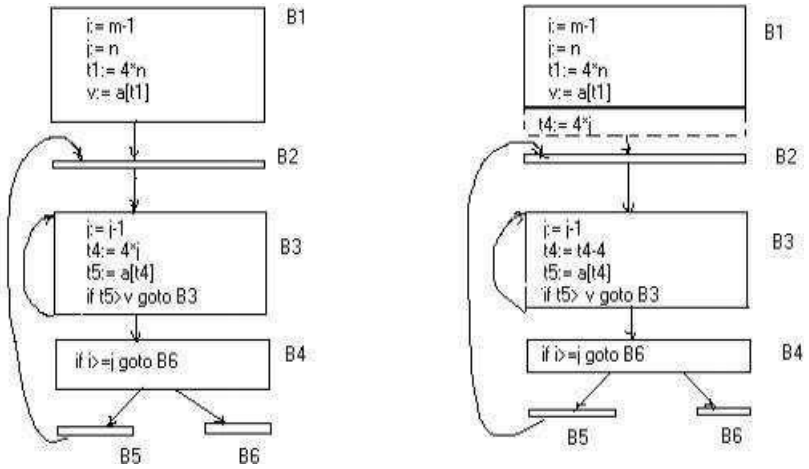
## PART – B

## QUESTIONS AND ANSWERS

**1. Explain the principle sources of optimization in detail. (APR-MAY’ 13)(MAY-JUN’12)**

**(Or) Discuss in detail the process of optimization of basic blocks. (APR-MAY’14)(MAY-JUN’11)(NOV-DEC’11) [nov dec 2014] [MAY-2016] (Apr May 2017) [NOV DEC 2017]**

A transformation of a program is called local if it can be performed by looking only at thestatements in a basic block; otherwise, it is called global.Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.




**Function-Preserving Transformations**

There are a number of ways in which a compiler can improve a program without changing the function it computes. The transformations Common sub expression elimination, Copy propagation, Dead-code elimination, and Constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

**Common Sub expressions elimination:**

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t5: = n

t6: = b [t1] +t5

The common sub expression t4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

**Copy Propagation:**

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example: x=Pi; …… A=x*r*r; The optimization using copy propagation can be done as follows:A=Pi*r*r; Here the variable x is eliminated

**Dead-Code Eliminations:**

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
a=b+5;
```

} Here, 'if' statement is dead code because this condition will never get satisfied.

**Constant folding**:

We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code. For example, a=3.14157/2 can be replaced by a=1.570 there by eliminating a division operation.

**Loop Optimizations:**

A very important place for optimizations namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- code motion, which moves code outside a loop;
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as

a multiplication by an addition.

**Code Motion:**

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/ Code motion will result in the equivalent of t= limit-2;

while (i<=t) /* statement does not change limit or t */

**Induction Variables:**

Loops are usually processed inside out. For example consider the loop around B3.Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables. When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4.

The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is before after initialized, shown by the dashed addition to block B1 in second Fig. The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

**Reduction in Strength:**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, $x^2$ is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

**2. Explain data flow analysis in detail. (APR-MAY'14) (APR-MAY'13)(APR-MAY'12) (NOV-DEC'13)(NOV-DEC'12)&(NOV-DEC'11)**

In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

Data-flow information can be collected by setting up and solving systems of equations of the form : **out [S] = gen [S] U ( in [S] – kill [S] )**

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

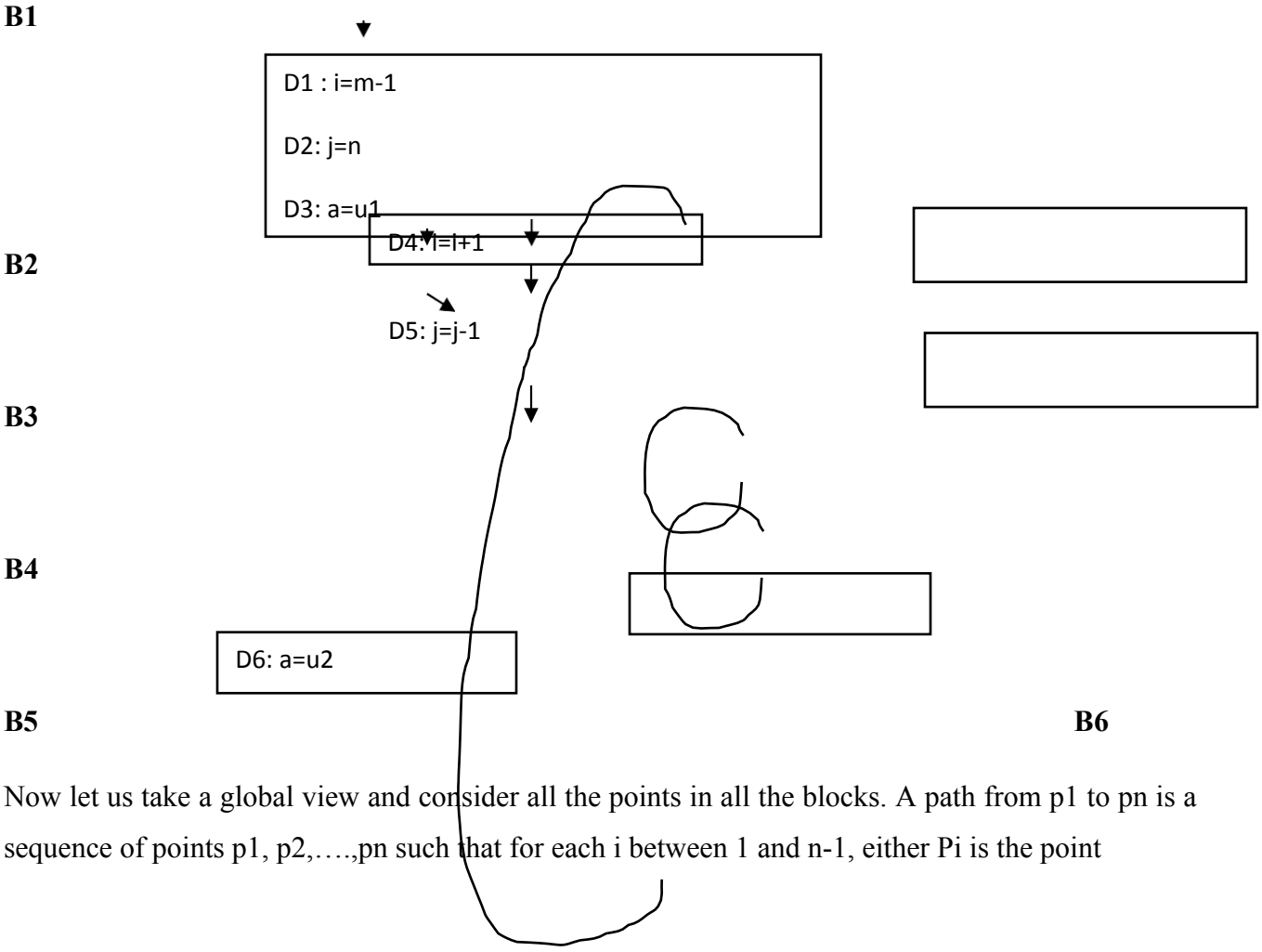
The details of how data-flow equations are set and solved depend on three factors. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to

be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

**Points and Paths:**

Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.




Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either Pi is the point

immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or Pi is the end of some block and pi+1 is the beginning of a successor block.

**Reaching definitions:**

A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure. An assignment through a pointer that could refer to x. For example, the assignment *q: =y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

**3. Discuss in detail the process of optimization of basic blocks. (APR-MAY'14)**

**(NOV-DEC'11) [nov dec 2014]**

There are two types of basic block optimizations. They are :

1. Structure-Preserving Transformations 2. Algebraic Transformations

**Structure-Preserving Transformations:**

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

**Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c

b: =a-d

c: =b+c

d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c

b: = a-d

c: = a

d: = b

**Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

**Renaming of temporary variables:**

A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u. In this we can transform a basic block to its equivalent block called normal-form block.

**Interchange of two independent adjacent statements:**

Two statements

t1:=b+c

t2:=x+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

**Algebraic Transformations:**

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28. The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c

e :=c+d+b

the following intermediate code may be generated:

a :=b+c

t :=c+d

e :=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

**4. Discuss briefly about Peephole Optimization.(APR-MAY'12)(NOV-DEC'13)&(NOV-DEC'11) [nov dec 2014]**

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program. A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible. The peephole is a small, moving window on the target program. The code in the peephole need not

contiguous, although some implementations do require this .it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. We shall give the following examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable Code

**Redundant Loads And Stores:**

If we see the instructions sequence

(1) MOV R0,a

(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

**Unreachable Code:**

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0
….
if ( debug ) {
Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
if debug =1 goto L2
goto L2
L1: print debugging information
```

L2: (a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

If debug ≠1 goto L2

Print debugging information

L2: (b)

As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: (c)

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

**Flows-Of-Control Optimizations:**

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

….

L1: gotoL2

by the sequence

goto L2

….

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1: goto L2

can be replaced by

If a < b goto L2

….

L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto.

Then the sequence

goto L1

……..

L1: if a < b goto L2

L3: (1)

May be replaced by

If a < b goto L2

goto L3

…….

L3: (2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time.

**Algebraic Simplification:**

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0

Or

x := x * 1

Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

**Reduction in Strength:**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, $x^2$ is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

X2 → X*X

**Use of Machine Idioms:**

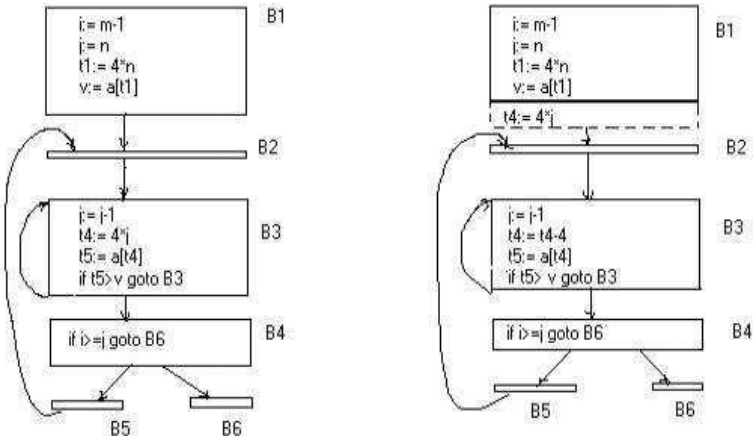
The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like

i :=i+1.

i:=i+1 → i++

i:=i-1 → i- -

**5. Explain loop optimization techniques in detail.**




Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables. When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must

maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is before after initialized, shown by the dashed addition to block B1 in second Fig. The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

**6.Discuss various issues in code generation in detail.(APR-MAY'14)(APR-MAY'12) (APR-MAY'11)(NOV-DEC'11) [nov dec 2014] [MAY-2016] (Apr May 2017) [NOV DEC 2017] (Apr May 2018)**

**ISSUES IN THE DESIGN OF A CODE GENERATOR**

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

**1. Input to code generator:**

The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

a. Linear representation such as postfix notation

b. Three address representation such as quadruples

c. Virtual machine representation such as stack machine code

d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

**2. Target program:**

The output of the code generator is the target program. The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

**3. Memory management:**

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name. Labels in three-address statements have to be converted to addresses of instructions.

For example,

*j* : **goto** *i* generates jump instruction as follows :

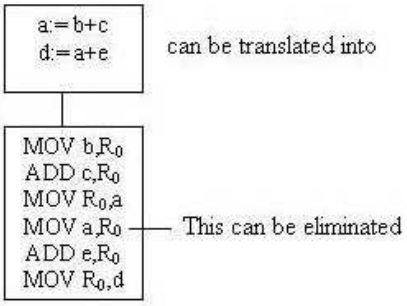
if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.

if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

**4. Instruction selection:**

The instructions of target machine should be complete and uniform. Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

The quality of the generated code is determined by its speed and size. The former statement can be translated into the latter statement as shown below:




**5. Register allocation**

Instructions involving register operands are shorter and faster than those involving operands in memory.The use of registers is subdivided into two subproblems :

***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

***Register assignment*** – the specific register that a variable will reside in is picked.Certain machine requires even-odd *register pairs* for some operands and results.

For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair

y – divisor even register holds the remainder odd register holds the quotient

**6. Evaluation order**

The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

**7.Explain how a DAG is constructed along with a suitable example.(APR-MAY'14) (APR-MAY'13)(NOV-DEC'13)[Nov/Dec 2017]**

**THE DAG REPRESENTATION FOR BASIC BLOCKS**

A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks. It gives a picture of how the value computed by a statement is used in subsequent statements.It provides a good way of determining common sub - expressions.

**Algorithm for construction of DAG**

**Input:** A basic block

**Output:** A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

**Method:**

**Step 1:** If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

**Step 2:** For the case(i), create a node(OP) whose left child is node(y) and right child is node(z).

( Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

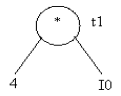
For case(iii), node n will be node(y).

**Step 3:** Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

**Example:** Consider the block of three- address statements:

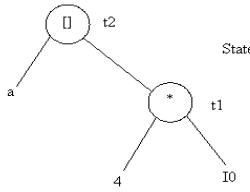
1. t1 := 4* I
2. t2 := a[t1]
3. t3 := 4* i
4. t4 := b[t3]
5. t5 := t2*t4
6. t6 := prod+t5
7. prod := t6
8. t7 := i+1
9. i := t7
10. if i<=20 goto (1)

(a)



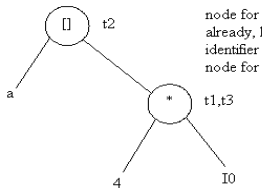

Statement (1)

(b)



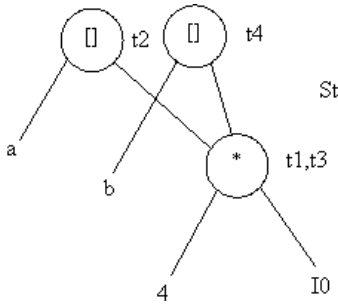

Statement (2)

(c)



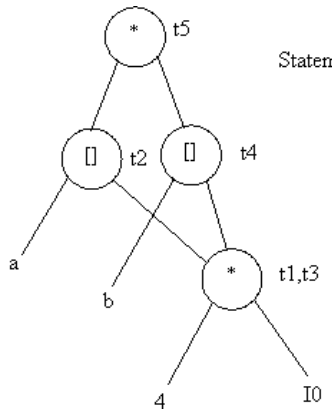

node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



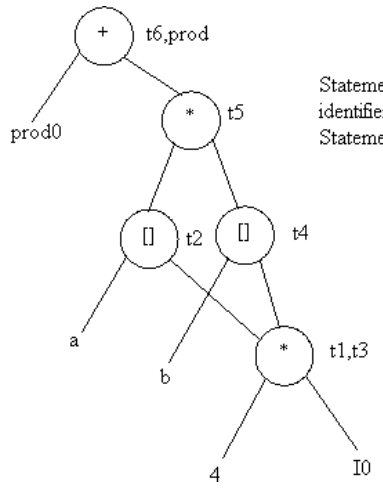

Statement (4)

(e)



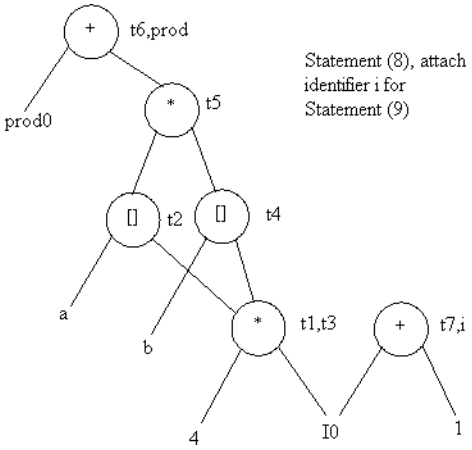

Statement (5)

(f)



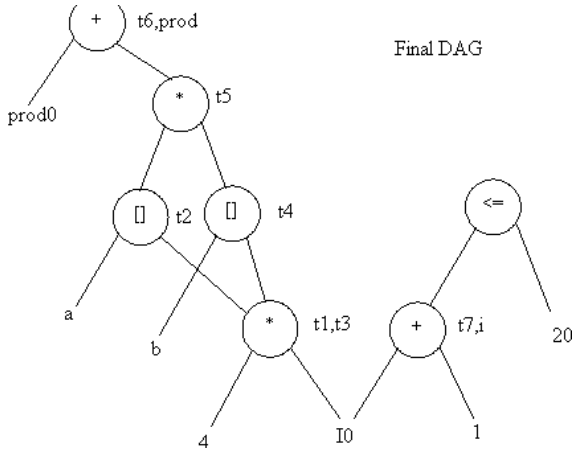

Statement (6), attach identifier prod for Statement (7)

(g)




Statement (8), attach identifier i for Statement (9)

(h)




Final DAG

**8. Explain a simple code generator algorithm in detail. (APR-MAY'13)(APR-MAY'12) [MAY-2016] (APR-MAY'11)(NOV-DEC'11) [nov dec 2014]**

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement **a := b+c** It can have the following sequence of codes:

ADD Rj, Ri Cost = 1 // if Ri contains b and Rj contains c

(or)

ADD c, Ri Cost = 2 // if c is in a memory location

(or)

MOV c, Rj Cost = 3 // move c from memory to Rj and add

ADD Rj, Ri

**Register and Address Descriptors:**

A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty. An address descriptor stores the location where the current value of the name can be found at run time.

**Input:** Basic block B of three-address statements

**Output:** At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

**Method:** We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

**A code-generation algorithm:**

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.
2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

**Generating Code for Assignment Statements:**

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three address code sequence:

t : = a – b

u : = a – c

v : = t + u

d : = v + u with d live at the end.

**9. Explain how code is generated from DAG. [nov dec 2014]**

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

**Rearranging the order**

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

t1 : = a + b

t2 : = c + d

t3 : = e – t2

t4 : = t1 – t3

**Generated code sequence for basic block:**

MOV a , R0

ADD b , R0

MOV c , R1

ADD d , R1

MOV R0 , t1

MOV e , R0

SUB R1 , R0

MOV t1 , R1

SUB R0 , R1

MOV R1 , t4

**Rearranged basic block:**

Now t1 occurs immediately before t4.

t2 : = c + d

t3 : = e – t2

t1 : = a + b

t4 : = t1 – t3

**Revised code sequence:**

MOV c , R0

ADD d , R0

MOV a , R0

SUB R0 , R1

MOV a , R0

ADD b , R0

SUB R1 , R0

MOV R0 , t4

In this order, two instructions **MOV R0 , t1** and **MOV t1 , R1** have been saved

**A Heuristic ordering for Dags**

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument. The algorithm shown below produces the ordering in reverse.

**Algorithm:**

**1) while** unlisted interior nodes remain **do begin**

2) select an unlisted node n, all of whose parents have been listed;

3) list n;

**4) while** the leftmost child m of n has no unlisted parents and is not a leaf **do**

**begin**

5) list m;

6) n : = m

**end**

**end**

**10.Explain structure preserving transformations on basic blocks.(NOV-DEC'11)**

**Transformations on Basic Blocks:**

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

**1. Structure preserving transformations:**

**a) Common subexpression elimination:**

a : = b + c a : = b + c

b : = a – d b : = a - d

c : = b + c c : = b + c

d : = a – d d : = b

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

**b) Dead-code elimination:**

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : =y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

**c) Renaming temporary variables:**

A statement **t : = b + c** ( t is a temporary ) can be changed to **u : = b + c** (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block. Such a block is called a *normal-form block*.

**d) Interchange of statements:**

Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is **t1** and neither **b** nor **c** is **t2**.

**2. Algebraic transformations:**

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

**11.Discuss specification of a simple type checker. [MAY-2016]**

Specification of a Simple Type Checker:

.1) A simple language

2) The part of a translation scheme that saves the type of an identifier.

- P→D;E
- D→D; D
- D→ id:T {addtype(id.entry, T.type)}
- T →char {T.type=char}
- T → integer {T.type=integer}
- T → array [num] of T {T.type=array(1..num,$T_1$.type)}
- T → ^$T_1$ {T.type=pointer($T_1$.type)}

2. Type Checking of Expressions

- E →literal {E.type=char}
- E →num {E.type=integer}
- E →id {E.type=*lookup*(id.entry)}
- E →$E_1$ mod $E_2$
- {E.type= if ($E_1$.type==integer) && ($E_1$.type==integer) integer else type_error}

2. Type Checking of Expressions

- E →$E_1$[$E_2$]
- {E.type= if ($E_2$.type==integer) && ($E_1$.type==array(s,t) t else type_error}
- E →$E_1$^
- {E.type= if ($E_1$.type==pointer(t)) t else type_error}

3. Type Checking of Statements

- S →id:=E
- {S.type= if (id.type==E.type) void else type_error}
- S →if E then $S_1$

- {S.type= if (E.type==boolean) $S_1$.type else type_error}

Type Checking of Statements

- S →while E do $S_1$
- {S.type= if (E.type==boolean) $S_1$.type else type_error}
- S → $S_1$;$S_2$
- {S.type= if ($S_1$.type==void) && ($S_2$.type== void) void else type_error}

4. Type Checking of Functions

- T → $T_1$ '→' $T_2$ {T.type= $T_1$ .type→ $T_2$ .type}
- E →$E_1$ ($E_2$)
- {E.type= if ($E_2$.type==s) &&($E_1$.type==s→t) t else type_error}

# UNIVERSITY QUESTION PAPERS

Reg. No. : | | | | | | | | | | | |

# Question Paper Code : 66165

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2011.

Sixth Semester

Computer Science and Engineering

CS 2352 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is the role of lexical analyzer?

2. Give the transition diagram for an identifier.

3. Define handle pruning.

4. Mention the two rules for type checking.

5. Construct the syntax tree for the following assignment statement: a:=b*-c+b*-c.

6. What are the types of three address statements?

7. Define basic blocks and flow graphs.

8. What is DAG?

9. List out the criterias for code improving transformations.

10. When does dangling reference occur?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Describe the various phases of compiler and trace it with the program segment (position: = initial + rate * 60). (10)

(ii) State the complier construction tools. Explain them. (6)

Or

(b) (i) Explain briefly about input buffering in reading the source program for finding the tokens.

(ii) Construct the minimized DFA for the regular expression (0+1)*(0+1) 10. (8)

12. (a) Construct a canonical parsing table for the grammar given below. Also explain the algorithm used. (16)

$E \rightarrow E + T$ $E \rightarrow T$ $T \rightarrow T * F$ $T \rightarrow F$

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

Or

(b) What are the different storage allocation strategies? Explain. (16)

13. (a) (i) Write down the translation scheme to generate code for assignment statement. Use the scheme for generating three address code for the assignment statement g: = a+b-c*d. (8)

(ii) Describe the various methods of implementing three-address statements. (8)

Or

(b) (i) How can Back patching be used to generate code for Boolean expressions and flow of control statements? (10)

(ii) Write a short note on procedures calls. (6)

14. (a) (i) Discuss the issues in the design of code generator. (10)

(ii) Explain the structure-preserving transformations for basic blocks. (6)

Or

(b) (i) Explain in detail about the simple code generator. (8)

(ii) Discuss briefly about the Peephole optimization. (8)

15. (a) Describe in detail the principal sources of optimization. (16)

Or

(b) (i) Explain in detail optimization of basic blocks with example. (8)

(ii) Write about Data flow analysis of structural programs. (8)

---

Reg. No. : | | | | | | | | | | | | |

# Question Paper Code : 31313

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2013.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Common to PTCS 2352 — Principles of Compiler Design for B.E. (Part-Time) Fifth Semester — Computer Science and Engineering — Regulation 2009)

(Regulation 2008/2010)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — ($10 \times 2 = 20$ marks)

1. How will you group the phases of compiler?
2. Write the regular expression for identifier and whitespace.
3. Eliminate the left recursion for the grammar

   $S \rightarrow A\,a \mid b$

   $A \rightarrow A\,c \mid S\,d \mid \in$.
4. What is meant by coercion?
5. Define backpatching.
6. Translate the arithmetic expression $a * -(b + c)$ into syntax tree and postfix notation.
7. What is the use of Next-use information?
8. List the fields in an activation record.
9. Define loop unrolling with example.
10. What is an Optimizing compiler?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain the different phases of a compiler in detail. (12)

(ii) Discuss the cousins of compiler. (4)

Or

(b) (i) Draw the DFA for the augmented regular expression $(a \mid b)^* \#$ directly using syntax tree. (12)

(ii) Discuss input buffering techniques in detail. (4)

12. (a) Design an LALR parser for the following grammar and parse the input $id = id$. (16)

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

Or

(b) (i) Discuss in detail about storage allocation strategies. (8)

(ii) Explain about various parameter passing methods in procedure calls. (8)

13. (a) (i) Write the translation scheme for flow of control statements. (8)

(ii) Explain and compare in detail the various implementation forms of three address code. (8)

Or

(b) (i) Write the grammar and translation scheme for procedure call statements. (12)

(ii) Draw the DAG for the following three address code. (4)

$d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$.

14. (a) (i) Discuss runtime storage management in detail. (8)

(ii) Write short notes on structure preserving transformation of basic blocks. (8)

Or

(b) Construct DAG and three address code for the following C program.

```
        i = 1;
s = 0;
        while (i < = 10)
{
s = s +a [i] [i]
i = i + 1
}
```

(16)

15. (a) (i) Write in detail about loop optimization. (8)

(ii) Discuss the characteristics of peephole optimization. (8)

Or

(b) Discuss in detail about global data flow analysis. (16)

**B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2011**

**Sixth Semester**

**Computer Science and Engineering**

**CS 2352 — PRINCIPLES OF COMPILER DESIGN**

**(Regulation 2008)**

---

Time : Three hours

Maximum : 100 marks

Answer ALL questions

---

**PART A — (10 × 2 = 20 marks)**

1. What is an interpreter?

2. Define token and lexeme.
3. What is handle pruning?
4. What are the limitations of static allocation?
5. List out the benefits of using machine-independent intermediate forms.
6. What is a syntax tree? Draw the syntax tree for the following statement:
. : c b c b a − * + − * =
7. List out the primary structure preserving transformations on basic block.
8. What is the purpose of next-use information?
9. Define dead-code elimination.
10. What is loop optimization?

---

**PART B — (5 × 16 = 80 marks)**

11. (a) (i) Describe the various phases of complier and trace the program segment 4 : * + = c b a for all phases. (10)
(ii) Explain in detail about compiler construction tools. (6)
Or
(b) (i) Discuss the role of lexical analyzer in detail. (8)

(ii) Draw the transition diagram for relational operators and unsigned numbers in Pascal. (8)
12. (a) (i) Explain the error recovery strategies in syntax analysis. (6)

(ii) Construct a SLR construction table for the following grammar.
T E E + →
T E →
F T T * →
F T →
( ) E F →
id F → (10)

Or

(b) (i) Distinguish between the source text of a procedure and its activation at run time. (8)
(ii) Discuss the various storage allocation strategies in detail. (8) [MAY-2016]

13. (a) (i) Define three-address code. Describe the various methods of implementing three-address statements with an example. (8)
(ii) Give the translation scheme for converting the assignments into three address code. (8)
Or
(b) (i) Discuss the various methods for translating Boolean expression. (8)

(ii) Explain the process of generating the code for a Boolean expression in a single pass using back patching. (8)
14. (a) (i) Write in detail about the issues in the design of a code generator. (10)
(ii) Define basic block. Write an algorithm to partition a sequence of three-address statements into basic blocks. (6)
Or
(b) (i) How to generate a code for a basic block from its dag representation? Explain. (6)
(ii) Briefly explain about simple code generator. (10)

15. (a) (i) Write in detail about function-preserving transformations. (8)
(ii) Discuss briefly about Peephole Optimization. (8)

Or

(b) (i) Write an algorithm to construct the natural loop of a back edge. (6)
(ii) Explain in detail about code-improving transformations. (10)

B.E/B.Tech DEGREE EXAMINATION, MAY/JUNE 2012

Sixth semester

Computer science and Engineering

CS2352/Cs62/10144 Cs602-PRINCIPLES OF COMPILER DESIGN

(Regulation 2008)

Answer ALL questions

PART- A

1. Mention few cousins of compiler.

2. What are the possible error recovery actions in lexical analyzer?

3. Define an ambiguous grammar.

4. What is dangling reference?

5. Why are quadruples preferred over triples in an optimizing compiler?

6. List out the motivations for back patching.

7. Define flow graph.

8. How to perform register assignment for outer loops?

9. What is the use of algebraic identities in optimization of basic blocks?

10. List out two properties of reducible flow graph?

PART B

11. (a) (i) What are the various phases of the compiler? Explain each phase in detail.

(ii) Briefly explain the compiler construction tools.

OR

(b) (i) What are the issues in lexical analysis?

(ii)Elaborate in detail the recognition of tokens.

12. (a) (i) Construct the predictive parser for the following grammar.

S->(L)/a

L->L,S/S

(ii)Describe the conflicts that may occur during shift reduce parsing.

OR

(b) (i)Explain the detail about the specification of a simple type checker.

(ii)How to subdivide a run-time memory into code and data areas. Explain

13. (a) (i) Describe the various types of three address statements.

(ii)How names can be looked up in the symbol table? Discuss.

OR

(b) (i)Discuss the different methods for translating Boolean expressions in detail.

(ii) Explain the following grammar for a simple procedure call statement S->call id (enlist).

14. (a) (i) Explain in detail about the various issues in design of code generator.

(ii)Write an algorithm to partition a sequence of three address statements into basic blocks.

OR

(b) (i)Explain the code-generation algorithm in detail.

(ii)Construct the dag for the following basic block.

d: =b*c

e: = a +b

b: =b*c

a: =e-d

15. (a) (i) Explain the principal sources of optimization in detail.

(ii)Discuss the various peephole optimization techniques in detail.

OR

(b) (i) How to trace data-flow analysis of structured program?

(ii) Explain the common sub expression elimination, copy propagation, and transformation for moving loop invariant computations in detail.

Reg. No. :

**Question Paper Code : 51353**

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2014.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352 – Principles of Compiler Design for B.E. (Part-Time) Fifth Semester – Computer Science and Engineering – Regulation 2009)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. State any two reasons as to why phases of compiler should be grouped.

2. Why is buffering used in lexical analysis? What are the commonly used buffering methods?

3. Define Lexeme.

4. Compare the features of DFA and NFA.

5. What is the significance of intermediate code?

6. Write the various three address code form of intermediate code.

7. Define symbol table.

8. Name the techniques in loop optimization.

9. What do you mean by Cross-Compiler?

10. How would you represent the dummy blocks with no statements indicated in global data flow analysis?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Define the following terms : Compiler, Interpreter, Translator and differentiate between them. (6)

(ii) Differentiate between lexeme, token and pattern. (6)

(iii) What are the issues in lexical analysis? (4)

Or

(b) Explain in detail the process of compilation. Illustrate the output of each phase of compilation for the input "a = (b + c) * (b + c) * 2".

12. (a) Consider the following grammar

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a.$

Construct the SLR parse table for the grammar. Show the actions of the parser for the input string "abab".

Or

(b) (i) What is an ambiguous grammar? Is the following grammar ambiguous? Prove $E \rightarrow E + E \mid E * E \mid (E) \mid id$. The grammar should be moved to the next line, centered.

(ii) Draw NFA for the regular expression ab*/ab.

13. (a) How would you convert the following into intermediate code? Give a suitable example.

(i) Assignment statements. (8)

(ii) 'Case' statements. (8)

Or

(b) (i) Write notes on backpatching.

(ii) Explain the sequence of stack allocation processes for a function call.

14. (a) Discuss the various issues in code generation with examples.

Or

(b) Define a Directed Acyclic Graph. Construct a DAG and write the sequence of instructions for the expression a + a * (b – c) + (b – c) * d.

15. (a) Discuss in detail the process of optimization of basic blocks. Give an example.

Or

(b) What is data flow analysis? Explain data flow abstraction with examples.

———————

**Question Paper Code : 91355**

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2014.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352–Principles of Compiler Design for B.E. (Part–Time) Fifth Semester–Computer Science and Engineering–Regulation 2009)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is the role of lexical analyzer?
2. Write regular expression to describe a languages consist of strings made of even numbers a and b.
3. List out the various storage allocation strategies.
4. Write a CF grammar to represent palindrome.
5. What are the types of intermediate languages?
6. Give syntax directed translation for case statement.
7. Differentiate between basic block and flow graph.
8. Draw DAG to represent $a[i] = b[i]$; $a[i] = \& t$;
9. Represent the following in flow graph
   $i = 1$; $sum = 0$; $while(i <= 10)\{sum+ = i; i + +;\}$
10. What is global data flow analysis?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain the need for grouping of phases of compiler. (8)

    (ii) Explain a language for specifying the lexical analyzer. (8)

    Or

    (b) (i) Write short notes on compiler construction tools. (8)

    (ii) Explain — specification and recognition of tokens. (8)

12. (a) (i) Explain the specification of simple type checker. (8)

(ii) Explain — runtime environment with suitable example. (8)

Or

(b) Find the LALR for the given grammar and parse the sentence (a+b)*c

$E \rightarrow E+T/T, T \rightarrow T*F/F, F \rightarrow (E)/id$. (16)

13. (a) Generate intermediate code for the following code segment along with the required syntax directed translation scheme.

While (i<10)

If (i % 2 = = 0)

Evensum = evensum + i;

Else

Oddsum =oddsum + i;

Or

(b) Generate intermediate code for the following code segment along with the required syntax directed translation scheme. (16)

s=s+a[i][j];

14. (a) (i) Explain register allocation and assignment with suitable example. (8)

(ii) Explain — code generation phase with simple code generation algorithm. (8)

Or

(b) (i) Generate DAG representation of the following code and list out the applications of DAG representation. (8)

i = 1; while (i<=10) do

sum = a[i];

(ii) Explain — Generating code from DAG with suitable example. (8)

15. (a) (i) Explain — principle sources of optimization. (8)

(ii) Illustrate optimization of basic blocks with an example. (8)

Or

(b) Explain peephole optimization and various code improving Transformations. (16)

---

# Question Paper Code : 71391

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2015.

Sixth Semester

Computer Science and Engineering

CS 2352/CS 62/10144 CS 602 — PRINCIPLES OF COMPILER DESIGN

(Regulation 2008/2010)

(Common to PTCS 2352 – Principles of Compiler Design for B.E. (Part-Time) Fifth Semester – Computer Science and Engineering – Regulation 2009)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Describe the error recovery schemes in the lexical phase of a compiler.

2. Write a regular Definition to represent date in the following format : JAN-5th 2014.

3. What is the role of a passer?

4. Construct a decorated parse tree according to the syntax directed definition, for the following input statement: (4 + 7.5 * 3) / 2

5. Write the 3-address code for ; $x = *y$ ; $a = \&x$ .

6. Place the above generated 3-address code in Triplets and Indirect Triplets.

7. What role does the target machine play on the code generation phase of the compiler?

8. How is Liveness of a variable calculated

9. Generate code for the following C statement assuming three registers are available : $x = a/(b+c) - d*(e+f)$.

10. Write the algorithm that orders the DAG nodes for generating optimal target code.

12. (a) (i) Explain the specification of simple type checker. (8)

(ii) Explain — runtime environment with suitable example. (8)

Or

(b) Find the LALR for the given grammar and parse the sentence (a+b)*c

$E \rightarrow E+T/T, T \rightarrow T*F/F, F \rightarrow (E)/id$. (16)

13. (a) Generate intermediate code for the following code segment along with the required syntax directed translation scheme.

While (i<10)

If (i % 2 = = 0)

Evensum = evensum + i;

Else

Oddsum =oddsum + i;

Or

(b) Generate intermediate code for the following code segment along with the required syntax directed translation scheme. (16)

s=s+a[i][j];

14. (a) (i) Explain register allocation and assignment with suitable example. (8)

(ii) Explain — code generation phase with simple code generation algorithm. (8)

Or

(b) (i) Generate DAG representation of the following code and list out the applications of DAG representation. (8)

i = 1; while (i<=10) do

sum = a[i];

(ii) Explain — Generating code from DAG with suitable example. (8)

15. (a) (i) Explain — principle sources of optimization. (8)

(ii) Illustrate optimization of basic blocks with an example. (8)

Or

(b) Explain peephole optimization and various code improving Transformations. (16)

———————

Reg. No. | | | | | | | | | | | | |

**Question Paper Code : 57263**

**B.E./ B.Tech. DEGREE EXAMINATION, MAY/JUNE 2016**

**Sixth Semester**

**Computer Science and Engineering**

**CS 6660– COMPILER DESIGN**

**(Regulations 2013)**

**Time : Three Hours** **Maximum : 100 Marks**

**Answer ALL questions.**

**PART – A (10 × 2 = 20 Marks)**

1. What are the two parts of a compilation ? Explain briefly.
2. Illustrate diagrammatically how a language is processed.
3. Write a grammar for branching statements.
4. List the operations on languages.
5. Write the algorithm for FIRST and FOLLOW in parser.
6. Define ambiguous grammar.
7. What is DAG ?
8. When does Dangling references occur ?
9. What are the properties of optimizing compiler ?
10. Write three address code sequence for the assignment statement

    d:= (a-b) + (a-c) + (a-c).

**PART – B (5 × 16 = 80 Marks)**

11. (a) Describe the various phases of compiler and trace it with the program segment ( position:= initial + rate * 60). **(16)**

**OR**

(b) (i) Explain language processing system with neat diagram. **(8)**
(ii) Explain the need for grouping of phases. **(4)**
(iii) Explain various Error encountered in different phases of compiler. **(4)**

12. (a) (i) Differentiate between lexeme, token and pattern. **(6)**
(ii) What are the issues in lexical analysis ? **(4)**
(iii) Write notes on regular expressions. **(6)**

**OR**

(b) (i) Write notes on regular expression to NFA. Construct Regular expression to NFA for the sentence ( a|b)* a. **(10)**
(ii) Construct DFA to recognize the language ( a/b)* ab. **(6)**

13. (a) (i) Construct Sack implementation of shift reduce parsing for the grammar **(8)**
E -> E+E
E -> E*E
E -> (E)
E -> id and the input string id1 + id2 *id3
(ii) Explain LL(1) grammar for the sentence S->iEts | iEtSeS | a E->b. **(8)**

**OR**

(b) (i) Write an algorithm for Non recursive predictive parsing. **(6)**
(ii) Explain Context free grammers with examples. **(10)**

14. (a) (i) Construct a syntax directed definition for constructing a syntax tree for assignment statements. **(8)**
S → **id** : = E
E → E1 + E2
E →E1 * E2
E →-E1
E →( E1)
E →**id**

(ii) Discuss specification of a simple type checker. **(8)**

**OR**

(b) Discuss different storage allocation strategies. **(16)**

15. (a) Explain Principal sources of optimization with examples. **(16)**

**OR**

(b) (i) Explain various issues in the design of code generator. **(8)**
(ii) Write note on simple code generator. **(8)**

———————

**Question Paper Code : 71689**

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2017.

Sixth Semester

Computer Science and Engineering

CS 6660 — COMPILER DESIGN

(Common to Information Technology)

(Regulations 2013)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Define the two parts of compilation.
2. List the cousins of the compiler?
3. Write a regular expression for an identifier and number.
4. What are the various parts in LEX program?
5. Eliminate the left recursion for the grammar A→ Ac | Aad | bd.
6. What are the various conflicts that occur during shift reduce parsing?
7. What do you mean by binding of names?
8. Mention the rules for type checking.
9. What is a basic block?
10. What do you mean by copy propagation?

PART B — (5 × 16 = 80 marks)

11. (a) What are the phases of the compiler? Explain the phases in detail. Write down the output of each phase for the expression a : = b + c *60. (16)

Or

(b) (i) Explain briefly about compiler construction tools. (6)

(ii) Describe in detail about Cousins of compiler? (4)

(iii) Draw the transition diagram for relational operators and unsigned numbers. (6)

12. (a) Convert the Regular Expression abb (a|b)* to DFA using direct method and minimize it. (16)

Or

(b) (i) Differentiate between lexeme, token and pattern. (6)

(ii) What are the issues in lexical analysis? (4)

(iii) Draw the transition diagram for relational operators and unsigned numbers. (6)

13. (a) Construct a predictive parsing table for the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S.$

and show whether the following string will be accepted or not. (a,(a,(a,a))). (16)

Or

(b) Consider the following Grammar

E → E+T | T

T→ TF | F

F→ F* | a | b

Construct the SLR parsing table for the above grammar. (16)

14. (a) What are the different storage allocation strategies? (16)

Or

(b) (i) Explain in detail about Specification of a simple type checker (10)

(ii) Explain about the parameter passing. (6)

15. (a) Discuss the various issues in design of Code Generator. (16)

Or

(b) (i) Explain in detail about optimization of Basic Blocks. (8)

(ii) Construct the DAG for the following Basic Block. (8)

1. t1 : = 4*i
2. t2 : = a[t1]
3. t3 : = 4*i
4. t4 : = b[t3]
5. t5 : = t2*t4
6. t6 : = prod+t5
7. prod : = t6
8. t7 : = i+1
9. i : = t7
10. if i < = 20 goto (1).

# Question Paper Code : 50398

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2017

Sixth Semester

Computer Science and Engineering

CS6660 – COMPILER DESIGN

(Common to : Information Technology)

(Regulations 2013)

Time : Three Hours | Maximum : 100 Marks

Answer ALL questions

PART – A (10×2=20 Marks)

1. What is an interpreter ?
2. What do you mean by Cross-Compiler ?
3. What is the role of lexical analysis phase ?
4. Define Lexeme.
5. Draw syntax tree for the expression a=b*– c+b*– c.
6. What are the three storage allocation strategies ?
7. Differentiate NFA and DFA.
8. Compare syntax tree and parse tree.
9. Draw the DAG for the statement a = (a*b+c)–(a*b+c).
10. What are the properties of optimizing compilers ?

PART – B (5×16=80 Marks)

11. a) What are compiler construction tools ? Write note on each Compiler Construction tool.

(OR)

b) Explain in detail the various phases of compilers with an example.

50398

12. a) i) Discuss the issues involved in designing Lexical Analyzer.

ii) Draw NFA for the regular expression ab*/ab.

(OR)

b) Write an algorithm to convert NFA to DFA and minimize DFA. Give an example.

13. a) Explain LR parsing algorithm with an example.

(OR)

b) Explain the non-recursive implementation of predictive parsers with the help of the grammar.

E-> E+T | T

T->T*F | F

F->(E) | id

14. a) Explain the specification of simple type checker for statements, expressions and functions.

(OR)

b) Explain about runtime storage management.

15. a) Discuss the issues in code generation with examples.

(OR)

b) Explain briefly about the principal sources of optimization.

Reg. No. :

**Question Paper Code : 40916**

08 MAY 2018 FN

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2018
Sixth Semester
Computer Science and Engineering
CS 6660 – COMPILER DESIGN
(Common to Information Technology)
(Regulations 2013)

Time : Three Hours | Maximum : 100 Marks

Answer ALL questions

PART – A **(10×2=20 Marks)**

1. State the two main parts of compilation and its function.
2. Describe the possible error recovery actions in lexical analyzer.
3. Apply the rules used to define a regular expression. Give example.
4. What do you mean by Handle Pruning ?
5. Summarize the merits and demerits of LALR parser.
6. Draw the activation tree for the following code.

```
int main ( )
{
   printf("Enter Your Name");
   scanf("%s", username);
   int show_data(username);
   printf("Press Any Key to Continue...");
   ...
   int show_data(char *user)
{
   printf(" Your Name is %s", username);
   return 0;
}
}
```

7. How do you identify predictive parser and non-recursive predictive parser ?
8. Name different storage allocation strategies used in run time environment.
9. Mention various techniques used for loop optimization.
10. List out the primary structure preserving transformations on basic block.

PART – B (5×13=65 Marks)

11. a) i) Draw a diagram for the compilation of a machine language processing system. (5)

ii) Apply the analysis phases of compiler for the following assignment statement. (8)

***position := initial + rate * 60.***

(OR)

b) i) Show the transition diagram for relational operators and unsigned numbers. (8)

ii) Outline the construction tools can be used to implement various phases of a compiler. (5)

12. a) i) Considering the alphabet Σ = {0, 1}. Construct a Non-Deterministic-Finite Automata (NFA) using the Thompson construction that is able to recognize the sentences generated by the regular expression (1 * 01 * 0 ) * 1 *. (8)

ii) Illustrate how does LEX work ? (5)

(OR)

b) Consider the regular expression below which can be used as part of a specification of the definition of exponents in floating-point numbers. Assume that the alphabet consists of numeric digits ('0' through '9') and alphanumeric characters ('a' through 'z' and 'A' through 'Z') with the addition of a selected small set of punctuation and special characters. (say in this example only the characters '+' and '–' are relevant). Also, in this representation of regular expressions the character '.' denotes concatenation.

**Exponent = (+ | – | ε). ( E | e ) . ( digit ) +**

i) Derive an NFA capable of recognizing its language using the Thompson construction.

ii) Derive the DFA for the NFA found in a) above using the subset construction.

iii) Minimize the DFA found in (ii) above using the interactive refinement algorithm described in class. (13)

13. a) Consider the Context-Free Grammar (CFG) depicted below where "begin", "end" and "x" are all terminal symbols of the grammar and stat is considered the starting symbol for this grammar. Productions are numbered in parenthesis and you can abbreviate "begin" to "b" and "end" to "e" respectively.

Stat → Block
Block → begin Block end
Block → Body
Body → x

i) Compute the set of LR(1) items for this grammar and draw the corresponding DFA. Do not forget to augment the grammar with the initial production S → Start$ as the production (0).

ii) Construct the corresponding LR parsing table. (13)

(OR)

b) i) Consider the following CFG grammar over the non-terminals {X, Y, Z} and terminals {a, c, d} with the productions below and start symbol Z. **(6)**

$X \rightarrow a$
$X \rightarrow Y$
$Z \rightarrow d$
$Z \rightarrow X\ Y\ Z$
$Y \rightarrow c$
$Y \rightarrow \varepsilon$

Compute the FIRST and FOLLOW sets of every non-terminal and the set of non-terminals that are *nullable*.

ii) Consider the following CFG grammar, **(7)**

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

where a, b, c, d, e are terminals, 'S' (start symbol), A and B are non-terminals.

a) Parse the sentence " **abbcde** " using right-most derivations.

b) Parse the sentence " **abbcde** " using left-most derivations.

c) Draw the parse tree.

14. a) i) Describe about the contents of activation record. **(6)**

ii) Create a parse trees for the following string : string **id + id – id**. Check whether the string is ambiguous or not. **(7)**

(OR)

b) i) Explain about various ways to pass a parameter in a function with example. **(6)**

ii) Construct a Syntax-Directed Translation scheme that translates arithmetic expressions from infix into postfix notation. Using semantic attributes for each of the grammar symbols and semantic rules, Evaluate the input : **3 * 4 + 5 * 2.** **(7)**

15. a) i) Determine the basic blocks of instructions, Control Flow Graph (CFG) and the CFG dominator tree for following the code. **(7)**

```
01          a = 1
02          b = 0
03      L0: a = a + 1
04          b = p + 1
05          if (a > b) goto L3
06      L1: a = 3
07          if (b > a) goto L2
08          b = b + 1
09          goto L1
10      L2: a = b
11          b = p + q
12          if (a > b) goto L0
13      L3: t1 = p * q
14          t2 = t1 + b
15          return t2
```

ii) Construct a code sequence and DAG for the following syntax directed translation of the expression : **( a + b ) – ( e – (c + d ))**. **(6)**

(OR)

b) i) Translate the following assignment statement into three address code.
D:= ( a - b) * (a - c) + ( a - c)
Apply code generation algorithm, generate a code sequence for the three address statement. (7)

ii) Summarize the issues arise during the design of code generator. (6)

PART – C (1×15=15 Marks)

16. a) Draw the symbol tables for each of the procedures in the following PASCAL code (including main) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.

```
01:  procedure main
02:  integer a, b, c;
03:  procedure f1 (a, b);
04:  integer a, b;
05:  call f2(b, a);
06:  end;
07:  procedure f2(y,z);
08:  integer y, z;
09:  procedure f3(m,n);
10:  integer m, n;
11:  end;
12:  procedure f4(m,n);
13:  integer m, n;
14:  end;
15:  call f3(c,z);
16:  call f4(c,z);
17:  end;
18:  ...
19:  call f1(a, b);
20:  end;
```

(15)

(OR)

b) Consider the following grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow ( E )$

$E \rightarrow id$

i) Find the SLR parsing table for the given grammar.

ii) Parse the sentence : **( a +b ) * c**.

(15)