TEXT BOOKS

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, ¯Compilers – Principles, Techniques and Tools‖, 2nd Edition, Pearson Education, 2007.

REFERENCES

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, ¯Compilers – Principles, Techniques and Tools‖, 2nd Edition, Pearson Education, 2007.
2. Randy Allen, Ken Kennedy, ¯Optimizing Compilers for Modern Architectures: A Dependence-based Approach‖, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, ¯Advanced Compiler Design and Implementation, ¯Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
4. Keith D Cooper and Linda Torczon, ¯Engineering a Compiler‖, Morgan Kaufmann Publishers Elsevier Science, 2004.
5. Charles N. Fischer, Richard. J. LeBlanc, ¯Crafting a Compiler with C‖, Pearson Education, 2008.

UNIT I- INRODUCTION TO COMPILERS

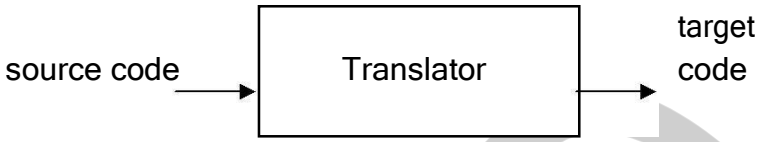
Translators-Compilation and Interpretation-Language Processors-The Phases of Compilers-Error encountered in different Phases-The Grouping of Phases-Compiler construction Tools-Programming Language Basics

PART-A

1. What is Translator and what are its types?

Translator:

It is a program that translates one language to another.



Types of Translator:

1. Interpreter
2. Compiler
3. Assembler.

2. What is an Interpreter? [Apr/May 2011]

Interpreter:

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

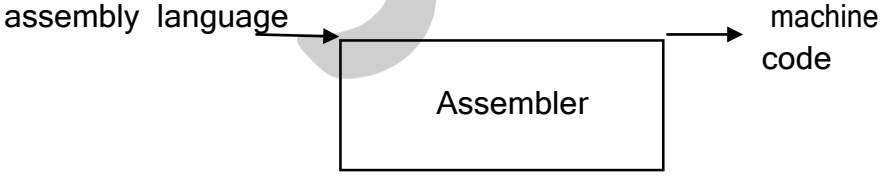
During execution, it checks line by line for errors.

Example: Basic, Lower version of Pascal.

3. What is an Assembler?

Assembler:

It translates assembly level language to machine code.



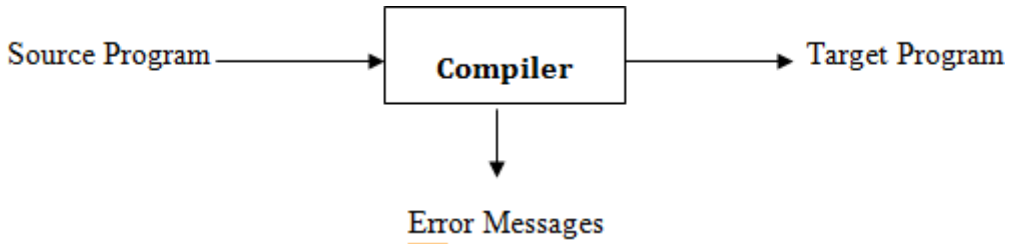
Example: Microprocessor 8085, 8086.

4. What are the classifications of compilers?

- Single Pass Compiler
- Multi Pass Compiler
- Load and go Compiler
- Debugging Compiler
- Optimizing Compiler

5. What is Compiler?

A Compiler is a program that reads a program written in one language and translates it into an equivalent program in another language – the target language.



Source Program -→ Any programming Language

Target Program → Machine Language or Any other programming Language

6. What are the two parts of compilation? [May /June 2016]

i. Analysis

ii. Synthesis

Analysis : Analysis part breaks up the source program into pieces and creates an intermediate representation of the source program.

Synthesis: Synthesis part constructs the desired target program from the intermediate representation.

7. What are the software tools used to perform analysis?

Structure Editor

1) Pretty Printer
2) Static Checker
3) Interpreters

8. What are the three phases of analysis phase of compilation?

1) Linear analysis
2) Hierarchial Analysis
3) Sementic Analysis

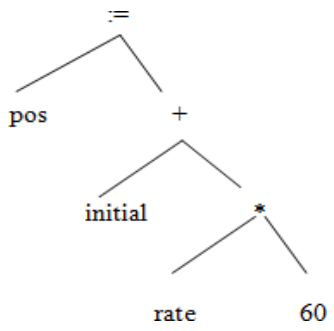
Linear Analysis : The stream of characters are read from left to right and grouped into tokens that are sequences of characters having a collective meaning.It is called as lexical analysis or scanning

Hierarchial Analysis : The characters or tokens are grouped hierarchically into nested collections with collective meaning.It is called as parsing or syntax analysis.

Semantic Analysis : In this analysis certain checks are made to ensure that the components of a program fit together meaningfully.

9. What is syntax Tree?

A syntax tree is a compressed representation of the parse tree in which the operator appears as the interior nodes and the operands are the children of the node for that operator



10. What is symbol Table? [Nov/Dec 2016]

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

11. What are the properties of intermediate form?

1) Each three address instruction has at most one operator in addition to the assignment
2) The compiler must generate a temporary name to hold the value computed by each instruction.
3) Some three address instructions have fewer than three operands.

Example:

temp1=inttoreal(60)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3

12. What are the functions of Preprocessor?

- **Macroprocessing :**

  A preprocessor may allow a user ro define macros that are shorthand for language constructs
- **File Inclusion :**

  A preprocessors include the header files into the program text
- **Rational Preprocessor**

  These processors augument older languages with more modern flow of control and data structuring facilities.
- **Language Extensions:**

  The processors attempt to add capabilities to the language by what amounts to build in macros.

  For eg) Equel is a database query language embedded in C.Statements beginning with ## are taken by the preprocessor to be data base access statements.

13. What is front end and back end of a compiler? [Nov/Dec 2013]

Front end of a compiler includes,

- Lexical Analysis
- Syntactic Analysis
- The creation of the symbol Table
- Semantic Analysis
- Intermediate code generation

These phases depend on the source languages and are independent of target machine.

Back end of a compiler includes

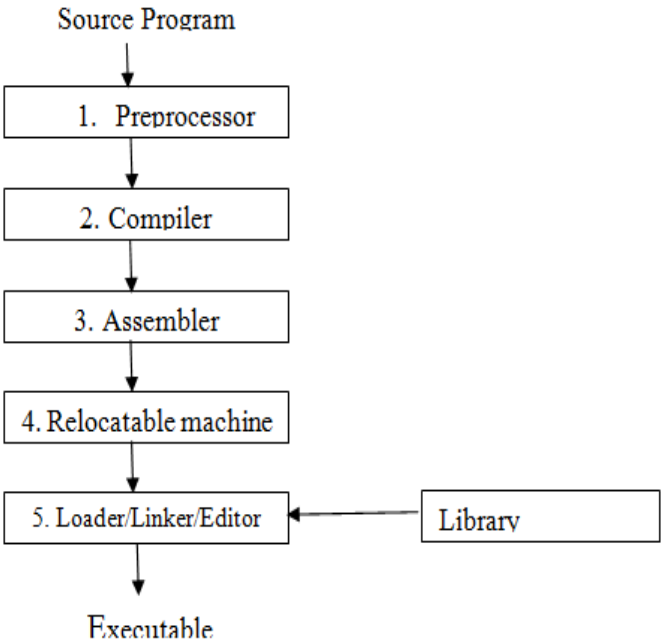
- Code Optimization
- Code Generation

These phases depend on the target machine and do not independent of the source language.

14. List the various compiler construction tools. [Nov/Dec 2016]

a. Parser Generator
b. Scanner Generator
c. Syntax directed translated engine
d. Automatic code Generator
e. Data flow engines.

15. What is a language processing system?
[May /June 2016]



16. Write a regular Definition to represent date in the following format :JAN-5th 2014.(May2015)

General RD Format for mm-dd-yyyy is: [1-12]-[1-31]-[0-9][0-9][0-9][0-9]
RD for JAN-5th-2014 is :[5]-[1]-[2][0][1][4]

PART-B

1. a.LANGUAGE PROCESSOR

❖ **Define the following terms:Compiler,Translator,Interpreter and differentiate between them.(6) [May/June 2014]**

Translator:

It is a program that translates one language to another.

Types of Translator:

1. Interpreter
2. Compiler
3. Assembler

1. Interpreter:

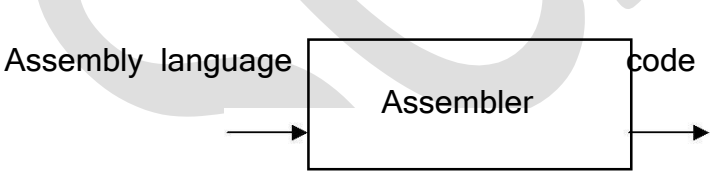
It is one of the translators that translate high level language to low level language.

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).source code

It executes the whole program and then displays the errors.

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

Difference between compiler and Interpreter.

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
It displays the errors after the whole program is executed	It checks line by line for errors.
Examples: Basic, lower version of Pascal.	Examples: C, C++, Cobol, higher version of Pascal.

1.b. SOFTWARE TOOLS USED IN COMPILATION

❖ **Describe the software tools that are used for analysis part of compilation. [Apr / May 2009]**

1) **Structure editor:**
   - Takes as input a sequence of commands to build a source program.
   - The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
   - For example, it can supply key words automatically - while …. do and begin….. end.

2) **Pretty printers :**
   - A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
   - For example, comments may appear in a special font.

3) **Static checkers :**
   - A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
   - For example, a static checker may detect that parts of the source program can never be executed.

4) **Interpreters :**
   - Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
   - An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
   - Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

2. ANALYSIS PART OF COMPILATION

❖ **Explain the analysis part of compilation with an example. [Apr / May 2009]**

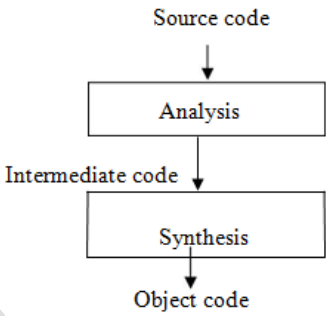
Parts of Compilation

There are 2 parts to compilation:

1. Analysis
2. Synthesis

Analysis part breaks down the source program into constituent pieces and creates an intermediate representation of the source program.

Synthesis part constructs the desired target program from the intermediate representation.



Analysis of the Source Program

Analysis consists of 3 phases:

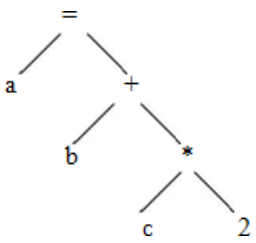
Linear/Lexical Analysis :

- 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 :

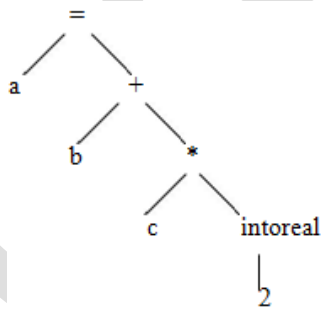
- It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- They are represented using a syntax tree as shown below:



- A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the leaf 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.
- For example if a b and c are real , then semantic analysis phase invokes the error recovery routine to convert integer 2 into real.



3. THE PHASES OF COMPILER

- **Explain the Various phases of compiler and trace the program segment pos=initial+rate * 60 for all the phases.(10)[April/May ,Nov 2011,] [May /June 2016]**
- **Expalin the different phases of compiler in detail.(12) . [Nov/Dec 2013]**
- **Explain in detail the process of compilation.Illustarte the output of each phase of the compilation for the input a=(b+c)*(b+c)*2(16) . [May/June 2014]**
- **What are the various phases of compiler?Expalin each phase in detail.(16) [May/June 2012] [Nov/Dec 2016].**

PHASES OF COMPILER

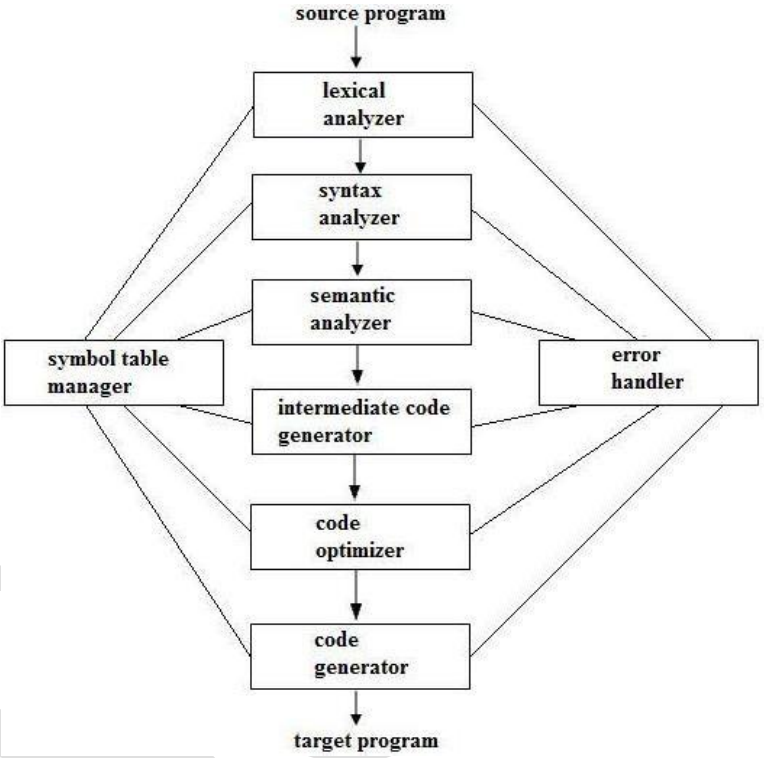
A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

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



Lexical Analysis:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

  Example: a +b =20

  Here, a,b,+,=,20 are all separate tokens.

  Group of characters forming a token is called the **Lexeme**.

The lexical analyser not only generates a token but also enters the lexeme into the

symbol table if it is not already there.

Syntax Analysis:

- It is the second phase of the compiler. It is also known as parser.

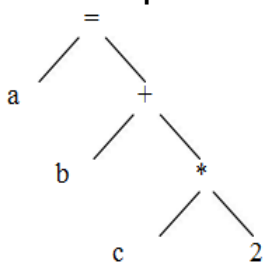
- It gets the token stream as input from the lexical analyser of the

compiler and generates syntax tree as the output.

- Syntax tree:

It is a tree in which interior nodes are operators and exterior nodes are operands.

□ **Example: For a=b+c*2, syntax tree is**



Semantic Analysis:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

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.
- Each three address instruction has at most one operator in addition to the assignment
- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some three address instructions have fewer than three operands.
- 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:

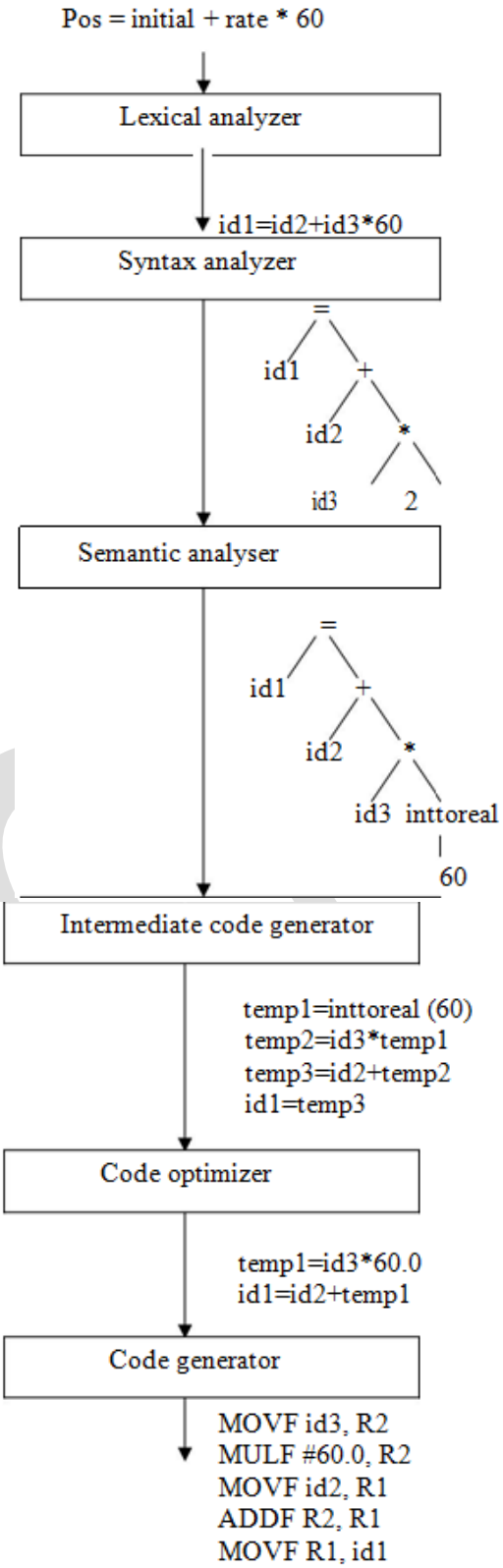
- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from 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 duringtype 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 pos=initial+rate*60. The figure shows the representation of this statement after each phase:

pos	id1
initial	id2
rate	id3



3 b) Explain the various Error Encountered in Different Phases of compiler. [May , Nov 2016]

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must some how deal with the error, so that compilation can proceed.

A program may have the following kinds of errors at various stages:

Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

Syntactical Errors

It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).

When an error is detected, it must be handled by parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

- Panic mode.
- Statement level.
- Error productions.
- Global correction.

Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

- Type mismatch.
- Undeclared variable.
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Logical errors

These errors occur due to not reachable code-infinite loop.

4.
5.
6.

- **Discuss the cousins of compiler.(4)** **[Nov/Dec 2013]**
-
- **COUSINS OF COMPILER**
    1. Preprocessor

2. Assembler
3. 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. For example, the language Equel is a database query language embedded in C.

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.

Two Pass Assembly :

Cosider the Assembly code

```
        Mov a R1
ADD #2 R1
MOV R1 b        Thus it computes b=a+2
```

The simplest form of assembler makes two passes over the input.A pass consists of reading an input file once.

In the first pass all the identifiers that denote storage locations are found and stored in a symbol table.

For eg) IDENTIFIER ADDRESS

a	0
b	4

In the second pass the assembler scans the input again.This time it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into the address given for that identifier in the symbol table.

The output of the second pass is relocatable machine code that it can be loaded starting at any location L in memory.

The following is machine code into which the assembly instructions might be translated

```
0001  01    00    00000000*
 0011   01   10    00000010
 0010   01    00    00000100*
```

The first four bits are the instruction code

0001 - Load

0010 - Store 0011 - Add

By load and store we mean moves from memory into a register and vice versa.

The next two bits designate a register and 01 refers to register 1

The two bits after that represent a ¯tag‖ represent the addressing mode.

00 → ordinary addressing mode

10 → immediate addressing mode

The last eight bits refers to a memory address.

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.

Suppose that the address space containing the data is to be loaded at starting at location L. Then L must be added to the address of the instruction.

Thus if L = 00001111 i.e 15 then a and b would be at locations 15 and 19 respectively and the instruction would appear as

```
 0001   01   00    00001111
0011   01   10    00000010
0010   01   00    00010011
```

5.a. THE GROUPING OF PHASES

❖ **Explain the need for grouping of phases of compiler.**

Nov/Dec2014,16,May/June 2014,16

GROUPING OF THE PHASES

Compiler can be grouped into front and back ends:

Front end: analysis (machine independent)

These normally include

- lexical analysis
- syntactic analysis,
- the creation of the symbol table,
- semantic analysis
- 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
- code generation phase

It also include along with the necessary error handling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

5. b. COMPILER CONSTRUCTION TOOLS.

❖ **State the compiler construction tools. Explain them.(6).[NOV/DEC 2011].**

❖ **Write short notes on compiler construction Tools.(8)[Nov/Dec 2014,2016].**

❖ **Mention any four compiler construction tools with their benefits and drawbacks.(8) [Apr/May 2015]**

❖ **Briefly explain the compiler construction Tools. (8) [May /June 2012]**

COMPILER CONSTRUCTION TOOLS

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.

UNIT II- LEXICAL ANALYSIS

Need and Role of Lexical Analyzer – Lexical Errors-Expressing Tokens by Regular Expressions. Converting Regular Expressions to DFA-Minimization of DFA-Language for specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

PART-A

1. What are the reasons for separating the analysis phase of compilation into lexical analysis and parsing? (or)
What are the issues of lexical analysis? [May /June 2016]

1. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of the phases.
2. Compiler Efficiency is improved.
    - A separate lexical analyzer allows to construct a specialized and potentially more efficient for the task.
    - Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler

3. Compiler portability is enhanced.

Input alphabet pecularities and other device specific anomalies can be restricted to the lexical analyzer.

2. Define Token, Pattern and Lexeme? [May/June 2016] [Nov/Dec 2016]

- The set of string is described by a rule called a pattern associated with the token
- The pattern is said to match each string in the set
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token
- For ex) const pi=3.1416

  pi--> lexeme for the token identifier

3.What is panic mode error recovery? [Apr/May 2015,May/Jun 2011]

Successive characters are deleted from the remaining input until the lexical analyzer can find a well defined token.

- Other possible error recovery actions are
  - Deleting an extraneous characters
  - Inserting a missing character
  - Replacing an incorrect character by a correct character
  - Transposing two adjacent characters.

4. Define the following terms.(a)Alphabet (b) String (3) Language

- The term alphabet or character class denotes any finite set of symbols
- A String over some alphabet is a finite sequence of symbols drawn from that alphabet
- The length of a string s ,|s| is the number of occurrences of symbols in s
- The language denotes any set of strings over some fixed alphabet.Abstract languages like φ the empty set or $\{\varepsilon\}$ the set containing only the empty string are languages.
- If x and y are strings then the concatenation of x and y written xy is the string formed by appending y to x

  For eg) if x=dog , y= house

  Then xy=doghouse

5.Define regular definition [Apr/May 2015]

If Σ is an alphabet of basic symbols then a regular definition is a sequence of definition of the form

d1--> r1

d2-->r2

...

dn-->rn

where each d_i is a distinct name

each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1,d_2,.....d_{i-1}\}$

6. Give the regular definition for identifier. [Nov/Dec 2013]

letter --> A|B|....|Z|a|b|c...|z

digit --> 0|1|2…|9

id --> letter(letter|digit)*

7. Give the regular definition for unsigned numbers.

digit --> 0|1|2…|9

digit -->digitdigit*

optional_fraction --> .digits|ε

optional_exponent -->(E(+|-|ε)digits)| ε

num --> digits optional_fraction optional_exponent.

8. What is Finite Automata?

There are two types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

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

M = {Qn, Σ,δ, q0, fn}

Q

n – 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

Deterministic Finite Automata

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

9. What is Regular Expression? Or List the rules that form the BASIS. [Nov/Dec 2016]

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).

10. Define Regular Set and Non regular set?

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.

Non Regular Set:

A language that cannot be described by regular expression is called a non regular set.

11. Construct Regular expression for the language

L= {w ε{a,b}/w ends in abb}

Ans: {a/b}*abb.

12. Define concrete and abstract syntax with example. (May 2009)

Abstract syntax tree is the tree in which node represents an operator and the children represents operands. Parse tree is called a concrete syntax tree, which shows how the start symbol of a grammar derives a string in the language. Abstract syntax tree, or simple syntax tree, differ from parse tree because superficial distinctions of form, unimportant for translation, do not appear in syntax tree.

13. List the operations on languages. [May /June 2016]

1. Union 2. Concotenation 3. Difference

14. Write a grammar for branching statements. [May /June 2016]

The Context free Grammar for if-else statement is,

S - > iEtS
S - >iEtSeS
S - > a
E -> b

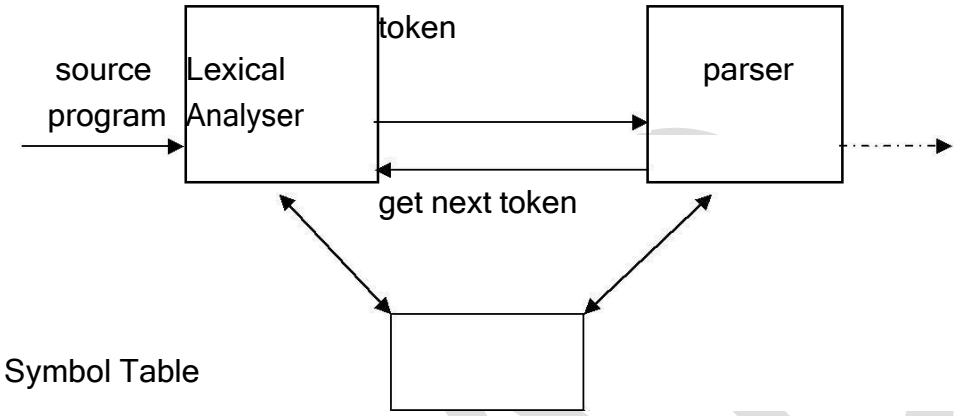
PART-B

1. NEED AND ROLE OF LEXICAL ANALYZER

- **What are the issues in the design of a lexical Analyzer.(4[May /June 2014, May /June 2012] [May /June 2016]**
- **Discuss the role of Lexical Analyzer in detail.(8).[Apr / May 2011] [Nov/Dec 2016].**
- **Describe the error recovery schemes in lexical phase of a compiler.[May 15]**

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.

ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

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. Consider this expression in the C programming language: sum=3+2;

Lexeme: . . .

A Lexeme is a sequence of characters in the source program that is matched by the token for a token . For example in the statement sum=2+3 ,

The substring sum is a lexeme for the token ¯identifier‖

Lexeme	**Token type**	**Informal Description of a Pattern**
Sum	Identifier	Letter followed by letters and digits

=	Assignment operator	<,<=,>,>=,< >,=
3	Number	Any numeric constant
+	Addition Operator	Arithmatic operators such as +,-,*,/
2	Number	Any numeric constant
	End of Statement	

Pattern:

The set of strings decribed by a rule called patterns.

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.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

Error Recovery Strategies In Lexical Analysis:

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

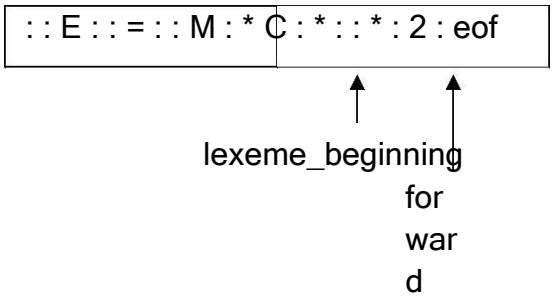
1) Deleting an extraneous character.
2) Inserting a missing character.
3) Replacing an incorrect character by a correct character
4) Transforming two adjacent characters.
5) Panic mode recovery: Deletion of successive characters from the token until error is resolved.

2. INPUT BUFFERING

- **Describe input buffering techniques in detail.[Nov /Dec 2013]**
- **Explain briefly about input buffering in reading the source program for finding the tokens.[Nov/Dec 2011].**
- **Differentiate Lexeme Pattern and Token. (6)[May/Jun 2014]**

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:

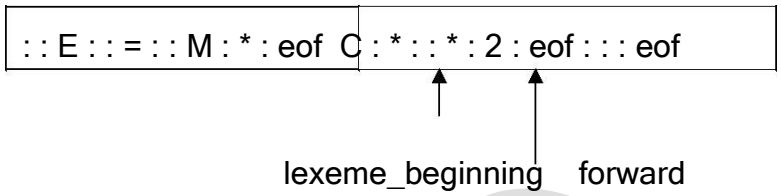
```
if forward at end of first half then
    begin reload second half;
    forward := forward + 1
end
else if forward at end of second half
    then begin reload second half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

Sentinels

- For each character read, we make two tests: one for the end of the buffer,

and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.

- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The sentinel arrangement is as shown below:



Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

Code to advance forward pointer:

```
forward : = forward +
1; if forward ↑ = eof
then begin
    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 first half;
       move  forward to beginning of
   first half end
else /* eof within a buffer signifying end of
       input */ terminate lexical analysis
end
```

3. EXPRESSING TOKEN BY REGULAR EXPRESSIONS

- **Explain – Specification and Recognition of Tokens.(8) [Nov/Dec 2014].**
- **Discuss how FA is used to represent tokens and perform lexical analysis with example.(8).[Nov/Dec 2016]**
- **Draw the transition diagram for relational operator and unsignend numbers in Pascal.(8) [Apr/May 2011]**
- **Write notes on regular expressions.[May /June 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** of string s is any string obtained by removing zero or more symbols from the end of strings.
   Forexample, ban is a prefix of banana.
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
4. **substring** of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.
5. 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.
6. 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}
   Kleene

3. closure : L^*={ ε,0,1,00….}

Positive

4. 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:

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. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r1

d2 →r2

.....

dn → rn

1. Each di is a distinct name.
2. Each ri is a regular expression over the alphabet Σ U {dl, d2,. . . , di-l}.

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 shorthands 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.

RECOGNITION OF TOKENS

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.

Transition diagrams

- As an intermediate step in the construction of a lexical Analyzer we first produce a stylized flowchart called a transition diagram.
- This Transition diagram are deterministic.
- One state is labeled as start state where control resides when we begin to recognize a token.
- On entering a state we read the next input character.
- If there is an edge from the current state whose label matches this character.
- We then go to the state pointed by the edge.Otherwise we indicate failure.
- Since keywords are sequence of letters ,they are exceptions to the rule that a sequence of letters and digits starting with a letter is an identifier.
- When the accepting state is reached we execute some code to determine if the lexeme leading to the accepting state is a keyword or an identifier.
- The return statement next to the accepting state uses
  Gettoken() → to obtain the token.
  Install-id() → to obtain the attribute values to be returned.
  - The symbol table is examined and if the lexeme is found there marked as a keyword install-id() returns 0.
  - If the lexeme is found and is a variable install-id() returns a pointer to the symbol table entry.
  - If the lexeme is not found in the symbol table it is installed as a variable and a pointer to the newly created entry is returned.

Transition diagram for relational operators

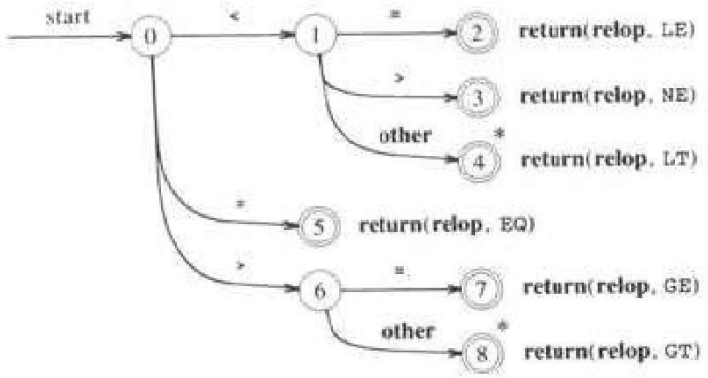
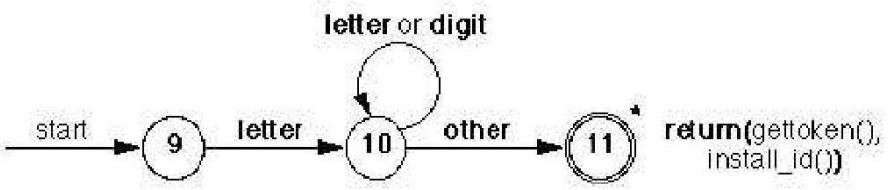


Fig. 3.12. Transition diagram for relational operators.

Transition diagram for identifiers and keywords



4. A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

❖ **Explain a language for specifying lexical Analyzer.(8).[Nov/Dec 2014]**

There is a wide range of tools for constructing lexical analyzers.

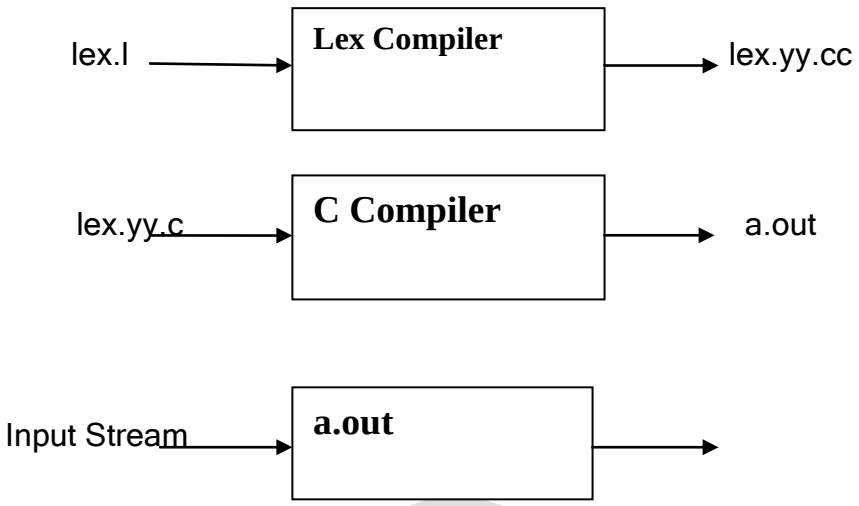
- Lex
- YACC

LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Sequence of Tokens

Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form

p1 {action1}
p2 {action2}
...
pn {actionn}

where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.

- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

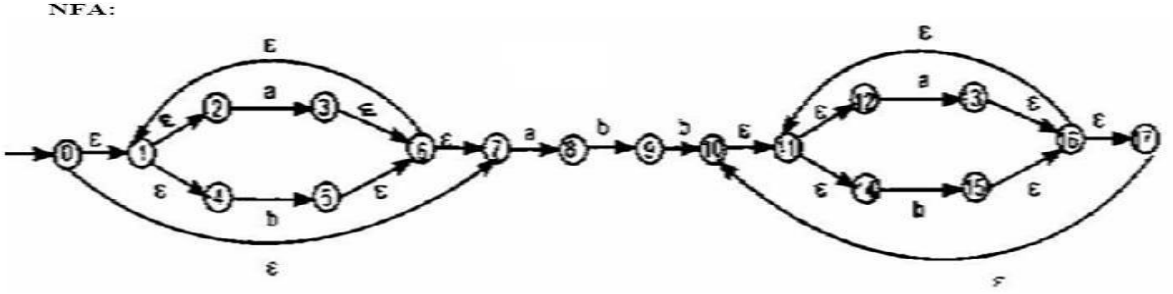
YACC- YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

5. REGULAR EXPRESSION TO DFA

Construct a NFA using Thompson‟s construction algorithm for the regular expression (a|b)*abb(a|b)* and convert it into DFA. [May /June 2016] [Nov/Dec 2016].

NFA:



The start state of DFA is E-closure (0)
E-closure (0) = {0, 1, 2, 4, 7} = A
The input symbol alphabet is {a, b}
E-closure (move(A, a)) = E-closure (move({0,1, 2, 4, 7}, a))
= E-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} = B
DTrans [A, a] = B
E-closure (move(A, b)) = E-closure (move({0,1, 2, 4, 7}, b))
= E-closure (5) = {1, 2, 4, 5, 6, 7} = C
DTrans [A, b] = C
E-closure (move(B, a)) = E-closure (move({1, 2, 3, 4, 6, 7, 8}, a))
= E-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} = B
DTrans [B, a] = B
E-closure (move(B, b)) = E-closure (move({1, 2, 3, 4, 6, 7, 8}, b))
= E-closure (5, 9) = {1, 2, 4, 5, 6, 7, 9} = C
DTrans [B, b] = D
E-closure (move(C, a)) = E-closure (move({1, 2, 4, 5, 6, 7, 8}, a))
= E-closure (3, 8) = B
DTrans [C, a] = B
E-closure (move(C, b)) = E-closure (move({1, 2, 4, 5, 6, 7, 8}, b))
= E-closure (5) = C
DTrans [C, b] = C
E-closure (move(D, a)) = E-closure (move({1, 2, 4, 5, 6, 7, 9}, a))
= E-closure (3, 8) = B
DTrans [D, a] = B
E-closure (move(D, b)) = E-closure (move({1, 2, 4, 5, 6, 7, 9}, b))
= E-closure (5, 10)
= {1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17} = E
DTrans [D, b] = E
E-closure (move(E, a)) = E-closure (move({1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17}, a))
= E-closure (3, 8, 13) = {1, 2, 3, 4, 6, 7, 8, 11,12, 13, 14, 16, 17}
= F

DTrans [E, a] = F
E-closure (move(E, b)) = E-closure (move({1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17}, b))
= E-closure (5, 15) = {1, 2, 4, 5, 6, 7, 11, 12,14, 15, 16, 17} = G
DTrans [E, b] = G
E-closure (move(F, a)) =E-closure (3, 8, 13) = F
DTrans [F, a] = F
E-closure (move(F, b)) =E-closure (5, 9, 15)
={1, 2, 4, 5, 6, 7, 9, 11, 12, 14, 15, 16, 17}= H
DTrans [F, b] = H
E-closure (move(G, a)) =E-closure (3, 8, 13) = F
DTrans [G, a] = F
E-closure (move(G, b)) =E-closure (5, 15) = G
DTrans [G, b] = G
E-closure (move(H, a)) =E-closure (3, 8, 13) = F
DTrans [H, a] = F
E-closure (move(H, b)) =E-closure (5, 10, 15)
={1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 15, 16, 17} = I
DTrans [H, b] = I
E-closure (move(I, a)) =E-closure (3, 8, 13) = F
DTrans [I, a] = F
E-closure (move(I, b)) =E-closure (5, 15) = G
DTrans [I, b] = G

Transition table for DFA:

State	Input symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E *	F	G
F *	F	H
G *	F	G
H *	F	I
I *	F	G

5 b) Write an algorithm for minimizing the number of states of a DFA. [Nov / Dec2016]

Input : A DFA M with set o states S. Inputs £ .Transitions defined for all states and inputs,initial state s0 and set of final states F.
Output : A DFA M‘ accepting the same language as M and having as few states as possible .

One important result on finite automata, both theoretically and practically, is that for any regular language there is a unique DFA having the smallest number of states that accepts it. Let M = < Q , Σ, q_0 , δ, A > be a DFA that accepts a language L. Then the following algorithm produces the DFA, denote it by M_1, that has the smallest number of states amomg the DFAs that accept L.

Minimization Algorithm for DFA
Construct a partition Π = { A, Q - A } of the set of states Q ;
Π_{new} := new_partition(Π);

while ($\Pi_{new} \neq \Pi$)

Π := Π_{new} ;
Π_{new} := new_partition(Π)
Π_{final} := Π ;

function new_partition(Π)
for each set S of Π do
partition S into subsets such that two states p and q of S are in the same subset of S
if and only if for each input symbol, p and q make a transition to (states of) the same set of Π.

The subsets thus formed are sets of the output partition in place of S.
If S is not partitioned in this process, S remains in the output partition.
end
Minimum DFA M_1 is constructed from Π_{final} as follows:

- Select one state in each set of the partition Π_{final} as the representative for the set. These representatives are states of minimum DFA M_1.
- Let p and q be representatives i.e. states of minimum DFA M_1. Let us also denote by p and q the sets of states of the original DFA M represented by p and q, respectively. Let s be a state in p and t a state in q. If a transition from s to t on symbol a exists in M, then the minimum DFA M_1 has a transition from p to q on symbol a.
- The start state of M_1 is the representative which contains the start state of M.
- The accepting states of M_1 are representatives that are in A.
  Note that the sets of Π_{final} are either a subset of A or disjoint from A.

Remove from M_1 the dead states and the states not reachable from the start state, if there are any. Any transitions to a dead state become undefined.
A state is a dead state if it is not an accepting state and has no out-going transitions except to itself.

UNIT-III
SYNTAX ANALYSIS

Need and Role of the Parser-Context free Grammars-Top Down Parsing-General Strategies-Recursive Descent Parser-Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR(0) Item-Construction of SLR Parsing Table-Introduction to LALR Parser-Error Handling and Recovery in Syntax Analyzer.-YACC-Design of a syntax Analyzer for a sample Language.

PART-A

1. What are the three general types of parsers? Differentiate them.

1. Universal parsing methods
2. Top down parsing
3. Bottom up parsing

- Universal parsing methods are too inefficient to use in production compilers and can parse any grammar
- Top down parsers build parse trees from the top (root) to the bottom (leaves).
- Bottom up parser build parse trees from the leaves and work up to the root.
- Top down and bottom up parser can parse only sub classes of Grammar such as LL Grammar and LR Grammar. These methods are efficient.

2. What are the goals of error handler?

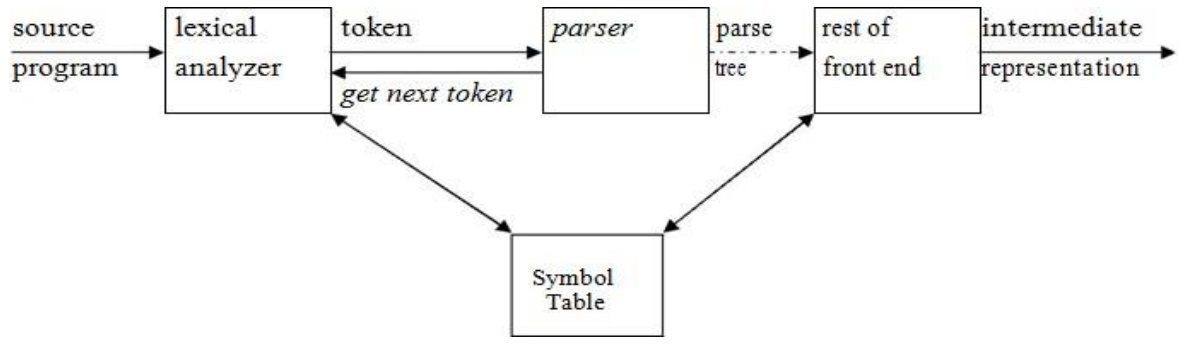
- It should report the presence of errors clearly and accurately
- It should recover from each error quickly enough to be able to detect subsequent errors
- It should not significantly slow down the processing of correct programs

3. What is the Role of the Parser?

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



The output of the parser is parse tree for the stream of tokens produced by the lexical Analyzer.

4. What are the error recovery strategies in Parsing?

- Panic mode
- Phrase level
- Error productions
- Global corrections.

5. What is meant by ambiguous grammar? [May/June 2012,2016] [Nov / Dec 2016]

A grammar that produces more than one parse tree, or more than one left most derivation or more than one right most derivation for some sentence is said to be ambiguous.

Example:

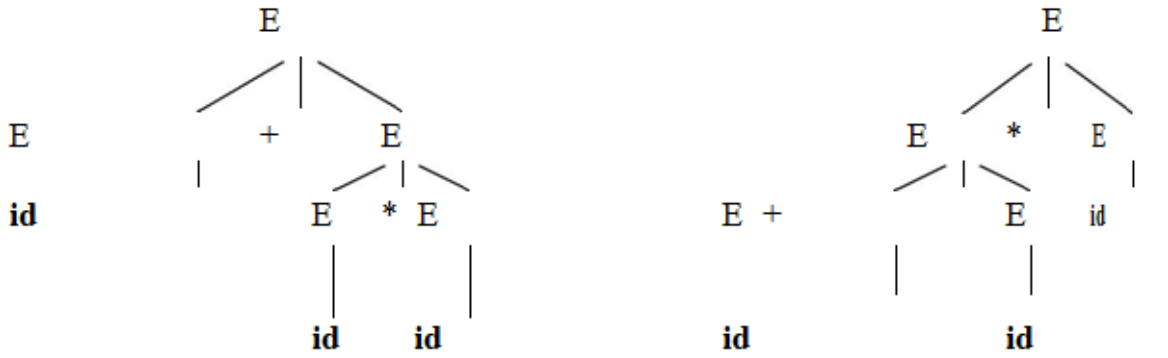
Grammar G : E → E+E | E*E | (E) | - E | id is ambiguous.

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E
E → id + E
E → id + E * E
E → id + id * E
E → id + id * id

E → E* E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

The two corresponding parse trees are :



6. What is left recursive grammar? How will you eliminate the left recursive grammar and eliminate left recursion for the following grammar? [Nov/Dec 2013]

E→ E + T | T T→ T * F | F F→ (E) | id

Left Recursive Grammar:

A Grammar is left recursive if it has a non terminal A such that there is a derivation

$A \xrightarrow{+} A\alpha$ for some string α

Rule to eliminate Left recursive Grammar:

Left recursive pair of productions

$A \rightarrow A\alpha \mid \beta$

Would be replaced by the non left recursive productions

A --> βA‟

A‟ --> α A‟| ε

without changing the set of strings derivable from A.

Non left recursive grammar:

E → TE‘ T → FT‘ F → (E) |id

E‘ →+TE‘ | ε T‘ → *FT‘| ε

7. What is left factoring? [Nov/Dec 2011, Nov/Dec 2009]

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The idea is that when it is not clear which of two alternative productions to expand a non terminal A.

If A → $\alpha\beta_1$ | $\alpha\beta_2$ are two A productions and the input begins with the non empty string derived from α ,

Left factored the original productions become

A → α A‘

A‘ → β_1 | β_2

8. Define FIRST(α) and FOLLOW(A)?What are the rules to compute FIRST(X) and FOLLOW(A)? [May /June 2016]

FIRST(α):

If α is any string of grammar symbols , let FIRST (α) be the set of terminals that begin the string derived from α . If α ⇒* ε then ε is also in FIRST(α).

FOLLOW(A):

FOLLOW(A for non terminal A to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S ⇒* αAa β for some α and β

Rules to compute FIRST(X):

1. If X is a terminal then FIRST(X) is {X}
2. If X→ε is a production then add ε to FIRST(X).
3. If X is a non terminal and X →Y_1Y_2…..Y_K is a production then place a in FIRST(X) if for some I, a is in FIRST(Y_i)

Rules to compute FOLLOW(A):

1. Place $ in FOLLOW(S) where S is the start symbol and $ is the right end marker.
2. If there is a production A → αBβ then everything in FIRST(β) except for ε is places in FOLLOW(B).
3. If there is a production A → αB or a production A → αBβ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

9. List the factors to be considered in Top down Parsing.

Top down parsing is an attempt to find a left most derivation for an input string.

Factors to be considered in Top down Parsing:

- Left recursive grammar can cause a top down parser to go into an infinite loop on writing procedure. So the Grammar should be non-left recursive for top-down parsing.
- Backtracking overhead may occur. Due to backtracking , it may reject some valid sentences. So the grammar should be left factored Grammar for top down parsing.

10. Define Handle and handle pruning? [Apr/May 2011, Nov/Dec 2011] [Nov/Dec 2016].

Handle :

A handle of a string is a sub string that matches the right side of a production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a right most derivation.

Handle Pruning:

A right most derivation in reverse can be obtained by ¯handle pruning‖.If w is a sentence of the grammar then w=γ_n , where γ_n is the right sentential form of some right most derivation.

$S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \gamma_3 \ldots\ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$

To reconstruct this derivation in reverse order we locate the handle β_n in γ_n and replace β_n by the left side of a production $A_n \rightarrow \beta_n$ to obtain the (n-1)st right sentential form γ_{n-1}.

11. Define Viable prefixes with an example.

The set of prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called viable prefixes. It is a prefix of a right sentential form that does not continue past the right end of the right most handle of that sentential form.

12. Differentiate sentence and Sentential Form.

Sentence	Sentential Form
If then the string w is called Sentence of G	If $S \overset{*}{\Rightarrow} \alpha$ then α is a sentential form of G
Sentence is a string of terminals. Sentence is a sentential form with no Non-Terminals.	Sentential form is a string of terminals and Non-Terminals.

13. Define LR(0) item or item and define the two classes of item.

LR(0) Item:

An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position of the right side Thus production A→XYZ yields four items

A→ **.** XYZ

A→X **.** YZ

A→XY **.** Z

A→XYZ **.**

Two classes of Item:

All the sets of items can be divided into two classes of items

1. Kernal items:

It includes the initial item S'→ . S and all items whose dots are not at the left end.

2. Nonkernal items:

These have their dots at the left end.

14. Define closure operation and Goto operation?

Closure

If I is a set of items for Grammar G then closure(I) is the set of items constructed from I by the two rules.

1. Initially every item in I is added to closure(I)
2. If A→α . Bβ is in closure(I) and B→γ is a production then add the item

B→ . γ to I if it is not already there. This rule is applied until no more items can be added to closure(I).

Goto:

goto(I,X) is defined to be the closure of the set of all items [A→αX . β] such that [A→α . Xβ] is in I. Where I is the set of items and X is a grammar symbol.

15. Why SLR and LALR are more economical than Canonical LR Parser.

For a comparison of parser size ,the SLR and LALR tables for a grammar always have the same number of states and this number is typically several hundred states for a language like Pascal.The canonical LR table would typically have several thousand states for the same size language.Thus it is much easier and more economical to construct SLR and LALR tables than the Canonical LR tables.

PART-B

1.a.NEED AND ROLE OF THE PARSER

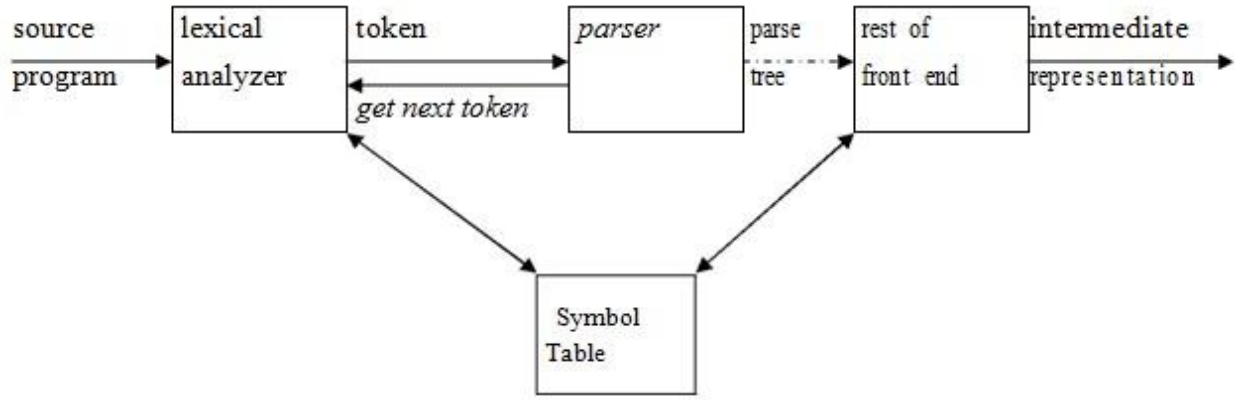
- **Discuss the Role of the Parser in detail. [Apr/May 2007, Apr/May 2015]**
- **Explain the error recovery strategies in syntax analysis.(8)[Apr/May 2011]**

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



The output of the parser is parse tree for the stream of tokens produced by the lexical Analyzer.

The Tasks of Parser

- Collecting informations about various tokens into the symbol table.
- Performing type checking.
- Semantic Analysis.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling :

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

The goals of error handler in a Parser:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.

3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

1. b. VARIOUS TERMINOLOGIES IN PARSING

❖ **Define the various Terminologies in Parsing.**

a. Parsing:

Parsing is the process of determining if a string of tokens can be generated by a Grammar. Parsing may be

- Top Down Parsing.
- Bottom up Parsing.

b. Top Down Parsing:

- Parsing method in which the construction of parse tree starts at the root and proceeds toward the leaves is called as top down parsing.
- Top Down Parsing method finds the left most derivation for a string of tokens.

c. Bottom up Parsing:

- Parsing method in which the construction of parse tree starts at the leaves and proceeds towards the root is called as Bottom up Parsing.
- Bottom Up Parsing method finds the right most derivation in reverse for a string of tokens.

d. Recursive Decent Parsing:

The general form of top down parsing called recursive descent parsing may involve backtracking that is making repeated scans of the input. It can be viewed as an attempt to construct a parse tree for the input string starting from the root and creating the nodes of the parse tree in preorder.

e. Predictive Parsing:

A special form of recursive descent Parsing in which the look ahead symbol unambiguously determines the procedures selected for each non terminal ,where no backtracking is required .It consist of stack, input buffer and finite control and tries to find the left most derivation.

f. Shift Reduce Parsing:

A general style of bottom up syntax analysis which attemts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

g. LR(k) Parsing :

The ¯L‖ is for left to right scanning of the input ,the ¯R‖ for constructing a right most derivation in reverse and the k for the number of input symbols of look ahead that are used in making parsing decisions.

h. LL(1) Grammar:

A grammar whose parsing table has multiply defined entries is said to be LL(1) grammar.

L - Left most Derivation.

L - Input scanned from left to right.

1- One input symbol used as a look ahead symbol.

2.a.CONTEXT FREE GRAMMARS

- **What is Context Free Grammar? Write short notes on derivations and ambiguous Grammar. [May /June 2016]**
- **Explain ambiguous grammar G : E->E+E|E*E|(E)|-E|id for the sentence id+id*id (8) [May /June 2014] [Nov/Dec 2016].**

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the ¯Start-symbol‖ and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

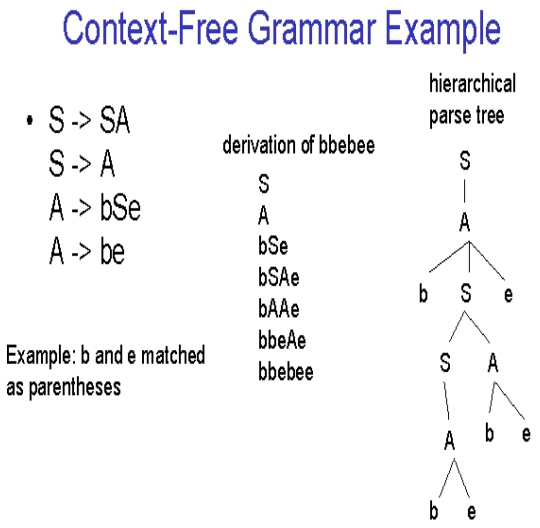
Ex: E->E+E/E*E/id

Example of context-free grammar: The following grammar defines **simple** arithmetic expressions:

expr → *expr op expr*
expr → (*expr*)
expr → - *expr*
expr → **id**
op → +|- | * |/
op →↑

In this grammar,

- **id** + - * / ↑( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.



Here is a formal CFG for { 0^n1^n | n ≥ 1}.
Terminals = {0, 1}.
Variables = {S}.
Start symbol = S.
Productions =
S -> 01
S -> 0S1

Derivations:

Two basic requirements for a grammar are:

1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

E → E+E |E*E |(E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

- E → - E
- E → - (E)
- E → - (E+E)
- E → - (id+E)
- E → - (id+id)

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E |E*E |(E) | - E |id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

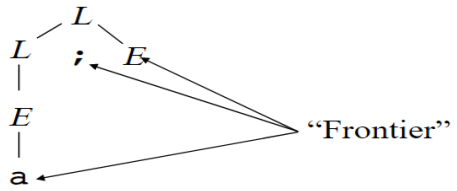
Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree .The yield is the leaf nodes which are read from left to right.

- ***Frontier*** **of tree = labels on leaves (in left-to-right order)**
- **Frontier of tree from *S* is a *sentential form***
- **Frontier of a complete tree from *S* is a *sentence***



Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be **ambiguous grammar**.

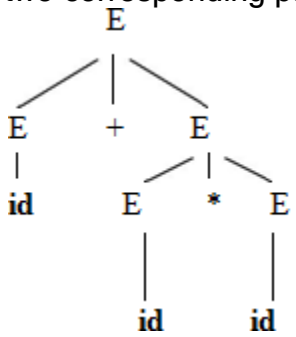
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

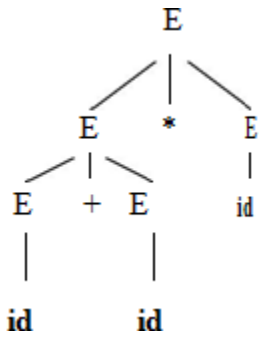
The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E
E → id + E
E → id + E * E
E → id + id * E
E → id + id * id

E → E* E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

The two corresponding parse trees are :





2.b WRITING A GRAMMAR

- **Write down the algorithm to eliminate left recursion and left factoring and apply both to the following Grammar E → E+T |T , T ->T*F| F, F→ (E) |id.(8) [Apr/May 2015]**
- **Write down the method to eliminate the ambiguity of a Grammar.**

WRITING A GRAMMAR

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

Reasons for using regular expressions to define the lexical syntax of a language.

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non-terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.

The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

To eliminate ambiguity the general rule is

‾Match each **else** with the closest previous unmatched **then**.‖

The idea is that a statement appearing between a **then** and an **else** must be matched.

i.e. It must not end with an unmatched **then** followed by any statement.

To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

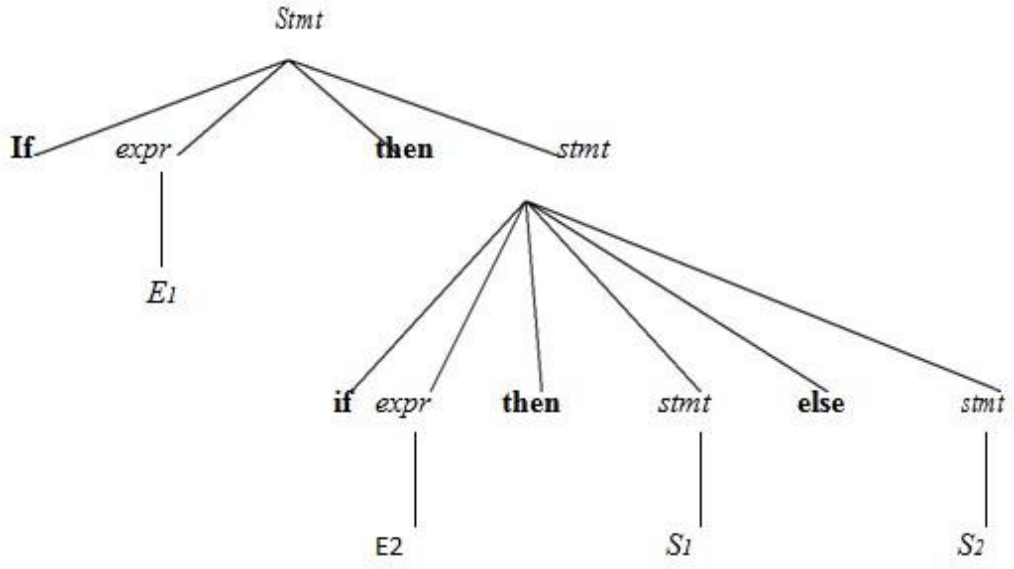
matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

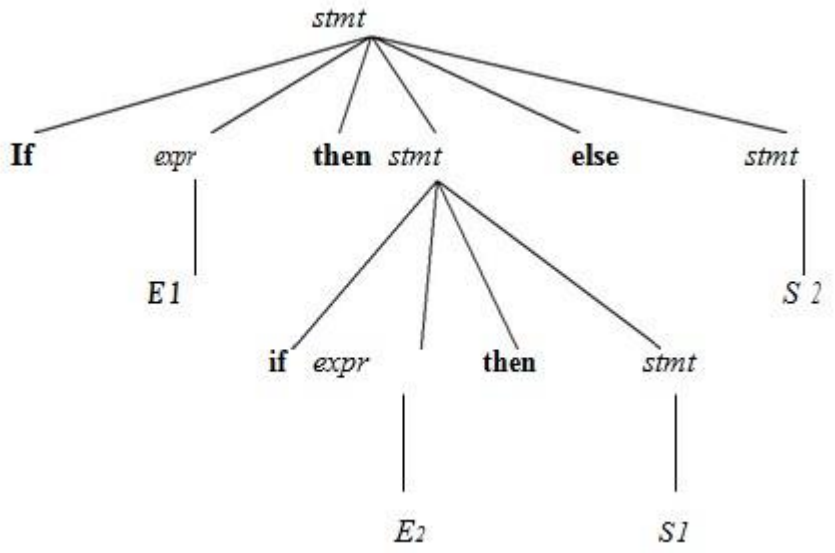
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following two parse trees for leftmost derivation:

1.



2.



Eliminating Left Recursion:

- A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α.
- Top-down parsing methods cannot handle left-recursive grammars.
- Hence, left recursion can be eliminated as follows:

If there is a production A → Aα |β it can be replaced with a sequence of two productions

A → βA‟

A‟→ αA‟ | ε

without changing the set of strings derivable from **A**

Example : Consider the following grammar for arithmetic expressions:

E → E+T |T

T → T*F

|F

F→ (E) |id

First eliminate the left recursion for E as

E → TE‘

E‘ → +TE‘ |ε

Then eliminate for

as

T → FT‘

T‘→ *FT‘ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE‘

E‘ → +TE‘ | ε

T → FT‘

T‘ → *FT‘ |

ε

F → (E) |id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order A1, A2 . . . An.
2. **for** *i* := 1 **to** *n* **do begin**

for *j* := 1 **to** *i*-1 **do begin**

replace each production of the form Ai → A j γ by the
productions Ai → δ1 γ | δ2γ | . . . | δk γ
where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;
end
eliminate the immediate left recursion among the Ai-productions
end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → αβ1 | αβ2 , it can be rewritten as
A → αA'
A'→ β1 | β2

Consider the grammar , G :
S → iEtS | iEtSeS | a
E → b
Left factored, this grammar becomes
S → iEtSS' | a
S'→ eS |ε
E → b

3.RECURSIVE DESCENT PARSER

- **Describe the Recursive Descent Parser with an example. [Apr/May 2008]**
- **Construct parse tree for the input w= cad using top down parser. S->cAd A->ab| a [Nov/Dec 2016].**

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.

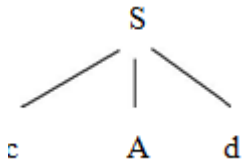
Example for backtracking :

Consider the grammar G :

S → cAd
A → ab |a
and the input string w=cad.
The parse tree can be constructed using the following top-down approach :

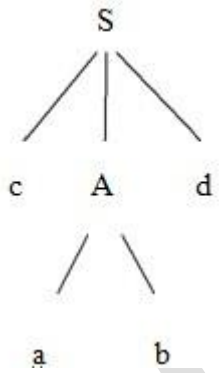
Step1:

Initially create a tree with single node labeled S. An input pointer points to _c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf _c' matches the first symbol of w, so advance the input pointer to the second symbol of w _a' and consider the next leaf _A'. Expand A using the first alternative.



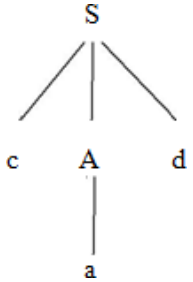
Step3:

The second symbol _a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w _d'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.

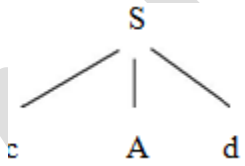


Now we can halt and announce the successful completion of parsing.

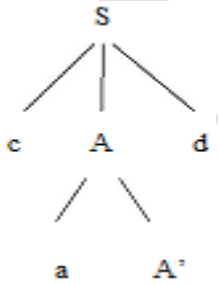
PREDICTIVE PARSER:

Eliminating left recursion from the grammar and left factoring the resulting grammar we can obtain a grammar that can be parsed by recursive descent parser

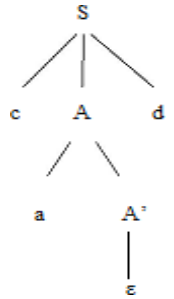
that needs no backtracking which is called as predictive parser.
Consider the grammar G :
S → cAd
A → ab |a and the input string
w=cad.
The grammar non-left recursive and left factoring the grammar, the grammar is
S → cAd
A → aA‘
A‘ → b |ε
The parse tree can be constructed using the following top-down approach :
Step1:
Initially create a tree with single node labeled S. An input pointer points to _c‘, the first symbol of w. Expand the tree with the production of S.



Step2:
The leftmost leaf _c‘ matches the first symbol of w, so advance the input pointer to the second symbol of w _a‘ and consider the next leaf _A‘. Expand A using A→aA‘



Step3:
The leftmost leaf _c‘ matches the first symbol of w, so advance the input pointer to the second symbol of w _a‘ and consider the next leaf A‘ and expand A‘ using A‘ → ε.



Now we can halt and announce the successful completion of parsing.

4.a.PREDICTIVE PARSER

- **Construct Predictive Parser with an example.[May/Jun 2012] [May /June 2016]**

PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non- terminal in case of alternatives.

The table-driven predictive parser has an

- Input buffer
- Stack
- A parsing table
- An output stream

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

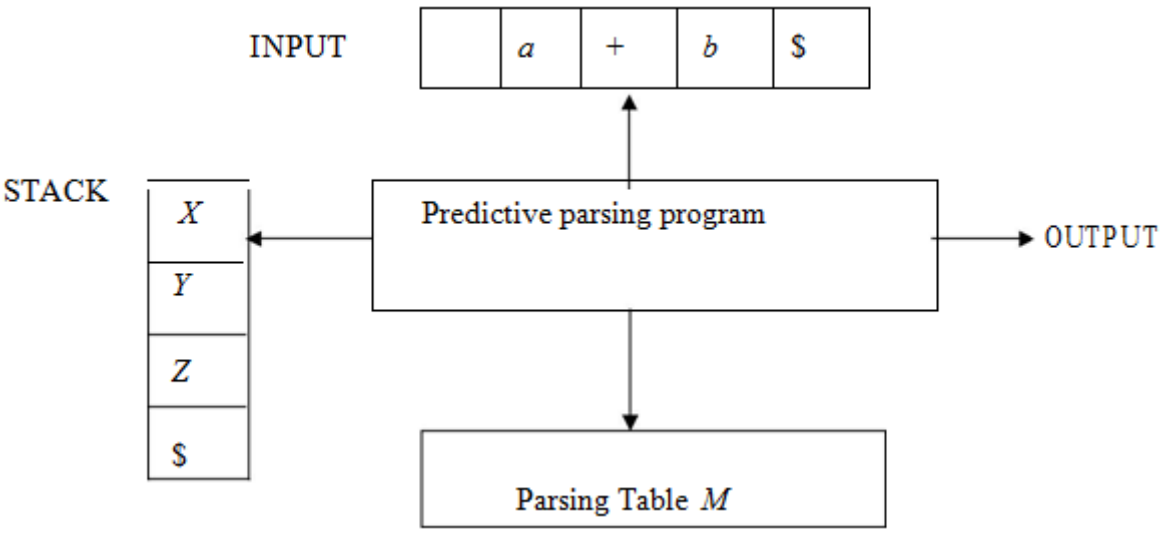
Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array *M*[*A*, *a*], where **„*A*"** is anon-terminal and **„*a*"** is aterminal.

Non-recursive predictive parser



Predictive parsing program:

The parser is controlled by a program that considers *X*, the symbol on top of stack, and *a*, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry. If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU.
4. If $M[X, a]$ = **error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing: [May /June 2016]

Input : A string w and a parsing table M for grammar G.

Output : If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method : Initially, the parser has $\$S$ on the stack with S, the start symbol of G on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

set *ip* to point to the first symbol of $w\$$;
repeat
let X be the top stack symbol and a the symbol pointed to by *ip*; **if** X is a terminal or \$ **then**
if $X = a$ **then**
pop X from the stack and advance *ip* **else** *error*()
else /* X is a non-terminal */
if $M[X, a] = X \rightarrow Y1Y2 \ldots Yk$ **then begin**
pop X from the stack;
push Yk, $Yk\text{-}1$, ... ,$Y1$ onto the stack, with $Y1$ on top;
output the production $X \rightarrow Y1\ Y2\ .\ .\ .\ Yk$
end
until X =**else** *error*()
\$ /* stack is empty */

Predictive parsing table :

Non Terminal	Id	+	*	(	)	$
E	E → TE‘			E → TE‘		
E‟		E‘ → +TE‘			E‘ → ε	E‘→ ε
T	T → FT‘			T → FT‘		
T‟		T‘→ ε	T‘→ *FT‘		T‘ → ε	T‘ → ε
F	F→ id			F→ (E)		

Parsing the input string id + id * id

STACK	INPUT	OUTPUT
$E	id+id*id $	
$E‘T	id+id*id $	E → TE‘
$E‘T‘F	id+id*id $	T → FT‘
$E‘T‘id	id+id*id $	F→ id
$E‘T‘	+id*id $	
$E‘	+id*id $	T‘ → ε
$E‘T+	+id*id $	E‘ → +TE‘
$E‘T	id*id $	
$E‘T‘F	id*id $	T → FT‘
$E‘T‘id	id*id $	F→ id
$E‘T‘	*id $	
$E‘T‘F*	*id $	T‘ → *FT‘
$E‘T‘F	id $	
$E‘T‘id	id $	F→ id
$E‘T‘	$	
$E‘	$	T‘ → ε
$	$	E‘ → ε

4 b. CONSTRUCTION OF PREDICTIVE PARSING TABLE

- **Describe the algorithm for the construction of predictive Parsing table with an example.[Apr/May 2008] [May /June 2016]**
- **Construct parsing table for the grammar and find moves made by predictive parser on input id+id*id and find FIRST and FOLLOW.**

E → E+T |T T → T*F |F F→ (E) |id [Nov/Dec 2016].

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G

1. FIRST
2. FOLLOW

FIRST:

If α is a string of grammar symbols ,FIRST(α) is the set of terminals that begin the string derivable from α.If α => ε then ε is also in FIRST(α).

Rules for computing FIRST(X):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If *X* → ε is a production, then add ε to FIRST(*X*).
3. If *X* is non- terminal and X → *a*α is a production then add *a* to FIRST(X).
4. If X is non- terminal and *X* → *Y1 Y2…Yk* is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-

1); that is, Y1,….Yi-*1* => ε.If ε is in FIRST (*Y j*) for all j=1,2,..,k, then add ε to FIRST(*X*).

FOLLOW :

FOLLOW(A) for non-terminal A is the set of terminals A that can appear immediately to the right of A in some sentential form . *

That is the set of terminals a such that there exists a derivation of the form S => αAaβ for some α and β

Rules for computing FOLLOW(A):

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production *A* → α*B*β, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method :

1. For each production *A* →α of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add *A* → α to *M*[*A*, *a*].
3. If ε is in FIRST(α), add A → α to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). Ifε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M*[*A*, $].
4. Make each undefined entry of *M* be **error**.

Example:

Consider the following grammar :

E → E+T |T

T → T*F |F

F→ (E) |id

After eliminating left-recursion the grammar is

E → TE‘

E‘ → +TE‘ |ε

T → FT‘

T‘ → *FT‘ | ε

F → (E) |id

First( ) :	**Follow()**
FIRST(E) ={ (, id}	FOLLOW(E) ={ $,) }
FIRST(E‘) ={+ , ε }	FOLLOW(E‘) ={ $,) }
FIRST(T) = { (, id}	FOLLOW(T) ={ +, $,) }
FIRST(T‘) ={*, ε }	FOLLOW(T‘) = { +, $,) }
FIRST(F) = { (, id }	FOLLOW(F) ={+, * , $,) }

Applying the Algorithm for constructing the predictive Parsing Table.

Predictive Parsing Table:

Non Terminal	Id	+	*	(	)	$
E	E → TE‘			E → TE‘		
E‟		E‘ → +TE‘			E‘ → ε	E‘→ ε
T	T → FT‘			T → FT‘		
T‟		T‘→ ε	T‘→ *FT‘		T‘ → ε	T‘ → ε
F	F→ id			F→ (E)		

4c.Explain LL(1) grammar for the sentence S->iEtS | iEtSeS | a E->b (May /June 2016)

Solution:

The production rule for S needs to be left factored.After left factoring the grammar is,

S-> iEtSS‟ | a
S‟ -> eS | €
E->b

Compute FIRST and FOLLOE for S,S‟ and E.

FIRST(S) = (I,a)
FIRST (S‟) = (e,€}
FIRST (E) = {b}
FOLLOW(S) = (e,$}
FOLLOW(S‟) = (e,$}
FOLLOW(E) = {t}

The predictive parsing table is

	a	b	e	i	t	$
S	**S->a**			**S-> iEtSS‟**		
S‟			**S‟->€ S‟->eS**			**S‟->€**
E		**E->b**				

SHIFT-REDUCE PARSING

- **Describe the conflicts that may occur during shift Reduce Parsing[May/Jun 2012]**
- **Describe the shift reduce Parser with an example.[Apr/May 2008] [May /June 2016]**

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a

parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B→ d

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)		**RIGHTMOST DERIVATION**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**d		
e	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E

E → E*E

E → (E)

E → id

And the input string id1+id2*id3

The rightmost derivation is :

E ==> **E+E**

==> E+**E*E**

==> E+E***id3**

==> E+**id2***id 3

==> **id1**+id2***id 3

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by ¯**handle pruning**‖.

(i.e:) if *w* is a sentence or string of the grammar at hand, then *w* = γ*n*, where γ*n* is the n^{th} right-sentinel form of some rightmost derivation.

Actions in shift -reduce parser:

□ Shift – The next input symbol is shifted onto the top of the stack.

- Reduce – The parser replaces the handle within a stack with a non-terminal.
- Accept – The parser announces successful completion of parsing.
- Error – The parser discovers that a syntax error has occurred and calls an error recovery routine

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	id1+id2*id3 $	Shift
$ id1	+id2*id3 $	reduce by E→id
$ E	+id2*id3 $	Shift
$ E+	id2*id3 $	Shift
$ E+id2	*id3 $	reduce by E→id
$ E+E	*id3 $	Shift
$ E+E*	id3 $	Shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

□ **Shift-reduce conflict**: The parser cannot decide whether to shift or to reduce.

□ **Reduce-reduce conflict**: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E |E*E |id and input id+id*id

Stack	Input	Action	Stack	Input	Action

$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

Reduce-reduce conflict:

Consider the grammar:

M → R+R |R+c |R

R → c

and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

LR PARSER

- **Explain LR Parser with an example.[Apr/May 2009]**

LR PARSER

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The _L' is for left-to-right scanning of the input,

the ‗R' for constructing a rightmost derivation in reverse, and the ‗*k*' for the number of input symbols. When ‗*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- □ It recognizes virtually all programming language constructs for which CFG can be written.
- □ It is an efficient non-backtracking shift-reduce parsing method.
- □ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- □ It detects asyntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

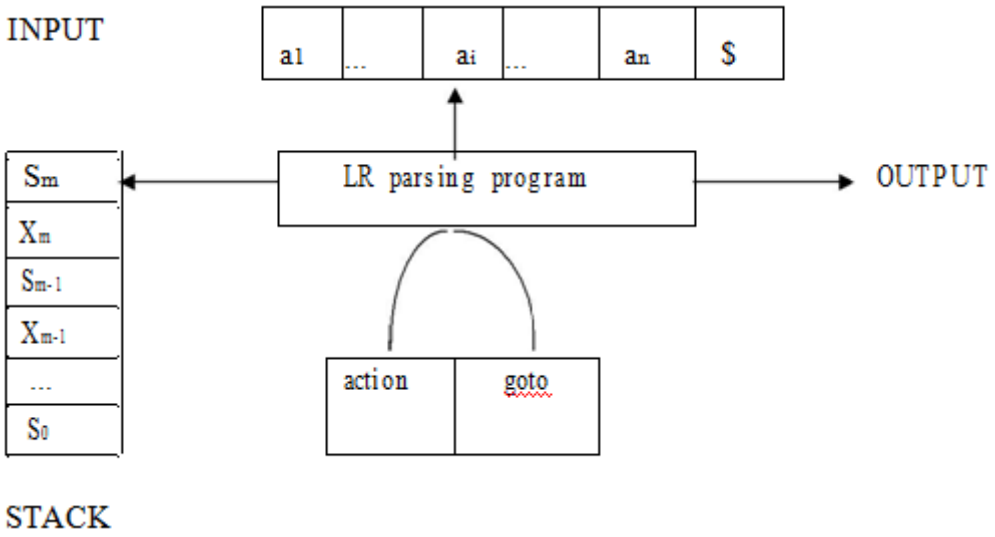
1. SLR- Simple LR
   - □ Easiest to implement, least powerful.
2. CLR- Canonical LR
   - □ Most powerful, most expensive.
3. LALR- Look -Ahead LR
   - □ Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

It consists of :

- An input
- An output
- A stack
- A driver program
- A parsing table that has two parts (*action* and *goto*).

The schematic form of an LR parser is as follows:



- □ The driver program is the same for all LR parser.

- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults *action*[sm,ai] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

A Configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input.

$$(s0X1s1X2s2…Xmsm\ ,\ a_ia_{i+1}.....a_n\$)$$

This configuration represents the right – sentential form.

$$X_1X_2.....X_m\ a_ia_{i+1}\ .....a_n.$$

The next move of the parser is determined by reading a_i , the current input symbol and sm, the state on the top of the stack and then consulting the parsing action table entry action[sm, a_i].

The configurations resulting after each of the four types of move are as follows.

1. If action[sm, a_i] = shift s, the parser executes a shift move entering the configuration

$(s_0X_1s_1X_2....X_ms_ma_is,a_{i+1}...\ a_n\$)$.

Here the parser has shifted both the current input symbol a_i and the next state s which is given in action[sm, a_i] onto the stack a_{i+1} becomes the current input symbol.

2. If action[sm, a_i]= reduce A-> β then the parser executes a reduce move entering the configuration

$$(s_0X_1s_1X_2....X_{m-r}s_{m-r}As,\ a_ia_{i+1}\ \ a_n\$)$$

Where s = goto[s_{m-r},A] and r is the length of β,the right side of production.Here the parser popped 2r symbols off the stack (r state symbols and r Grammar symbols),exposing state $s_{m-r.}$ The parser then pushed both A ,the left side of the production and s ,the entry for goto[$s_{m-r,}$A] onto the stack.

3. If action[sm, a_i]=accept, then Parsing is completed.
4. If action[sm, a_i]=error , the Parser has discovered an error and calls an error recovery routine.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.
Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :
set *ip* to point to the first input symbol of *w*$;
repeat forever begin
let *s* be the state on top of the stack and *a* the symbol pointed to by *ip*;
if *action*[*s*, *a*] =shift *s*' **then begin** push *a* then *s*' on top of the stack; advance *ip* to the next input symbol
end
else if *action*[*s*, *a*]=reduce A→β **then begin** pop 2* |β |symbols off the stack; let *s*' be the state now on top of the stack; push A then *goto*[*s*', A] on top of the stack; output the production A→ β
end
else if *action*[*s*, *a*]=accept **then return**
else *error*( )
end

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I5 , +) = r6 ; **reduce** by F→id
0 F 3	+id * id $	GOTO (I0 , F) = 3 GOTO (I3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I0 , T) = 2 GOTO (I2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I0 , E) = 1 GOTO (I1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I5 , *) = r6 ; **reduce** by F→ id
0 E 1 + 6 F 3	* id $	GOTO (I6 , F) = 3 GOTO (I3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I6 , T) = 9 GOTO (I9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I7 , id) = s5 ; **shift**
0 E 1+ 6 T 9 * 7 id 5	$	GOTO (I5 , $) = r6 ; **reduce** by F→ id
0 E 1+ 6 T 9 * 7 F 10	$	GOTO (I7 , F) = 10 GOTO (I10 , $) = r3 ; **reduce** by T → T * F

0 E 1+ 6 T 9	$	GOTO (I6 , T)=9 GOTO(I9,$)=r1 ; reduce by E-→E+T

6.a.CONSTRUCTION OF SLR PARSING TABLE

❖ **Construct SLR Parsing Table for the following Grammar.**
E → E + T | T T → T * F | F F → (E) | id and show the actionsfor the input string id1*id2+id3 [Nov/Dec 2011,2016,Apr/May 2011]

CONSTRUCTING SLR(1) PARSING TABLE:
To perform SLR parsing, take grammar as input and do the following:
1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:
An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :
A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

Closure operation:
If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:
\{ Initially, every item in I is added to closure(I).
\{ If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:
Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.
Steps to construct SLR parsing table for grammar G are:
1. Augment G and produce G‘
2. Construct the canonical collection of set of items C for G‘
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

The Sets of Items Construction:
procedure items(G‘);
begin
C= {closure({[S‘→ **. S]})};**
repeat

for each set of items I in C and each Grammar symbol X
Such that goto(I,X) is not empty and not in C do
Add goto(I,X) to C
until no more sets of items can be added to C
end

Algorithmfor construction of SLR parsing table:
Input : An augmented grammar G‘
Output : The SLR parsing table functions *action* and *goto* for G‘
Method :
1. Construct C ={I0, I1, …. In}, the collection of sets of LR(0) items for G‘.
2. State *i* is constructed from I*i*.. The parsing functions for state *i* are determinedas follows:
   (a) If [A→α·*a*β] is in Ii and goto(Ii,*a*) = Ij, then set *action*[*i*,*a*] to ¯shift j‖. Here *a* must be terminal.
   (b) If[A→α·] is in Ii , then set *action*[*i*,*a*] to ¯reduce A→α‖ for all *a* in FOLLOW(A).
   (c) If [S‘→S.] is in Ii, then set *action*[*i*,$] to ¯accept‖.

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(Ii,A)= Ij, then *goto*[i,A] = *j*.

Σ All entries not defined by rules (2) and (3) are made ¯error‖
Σ The initial state of the parser is the one constructed from the set of items containing [S‘→.S].

Example for SLR parsing:
Find the SLR parsing table for the given grammar E->E+E | E*E | (E) | id.
Given grammar:
1. E -> E+E
2. E -> E*E
3. E-> (E)
4. E -> id

Augmented grammar:
E.->E
E->E+E
E->E*E
E->(E)
E->id
I0: E.->.E
E->.E+E
E->.E*E
E->.(E)

E->.id
I1: goto(I0, E)
E.->E.
E->E.+E
E->E.*E
I2: goto(I0, ()
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id
I3: goto(I0, id)
E->id.
I4: goto(I1, +)
E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id
I5: goto(I1, *)
E->E*.E
E->.E+E
E->.E*E
E->.(E)
E->.id
I6: goto(I2, E)
E->(E.)
E->E.+E
E->E.*E
I7: goto(I4, E)
E->E+E.
E->E.+E
E->E.*E
I8: goto(I5, E)
E->E*E.
E->E.+E
E->E.*E
goto(I2, ()=I2
goto(I2, id)=I3
goto(I4, ()=I2
goto(I4, id)=I3
goto(I5, ()=I2
goto(I5, id)=I3

I9: goto(I6,))
E->(E).
goto(I6, +)=I4
goto(I6, *)=I5
goto(I7, +)=I4
goto(I7, *)=I5
goto(I8, +)=I4
goto(I8, *)=I5

First(E) = {(, id}

Follow(E)={+, *,), $}

SLR parsing table:

States	Action						Goto
	+	*	(	)	id	$	E
0			S2		S3		1
1	S4	S5				Acc	
2			S2		S3		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	S4, r1	S5, r1		r1		r1	
8	S4, r2	S5, r2		r2		r2	
9	r3	r3		r3		r3	

Blank entries are error entries.

6. b. CONSTRUCTION OF LALR PARSING TABLE

- **Design an LALR Parser for the following Grammar and parse the input string w=*id**

S→ L=R S→R L→*R L→id R→**L** **(16)** **[Nov/Dec 2013]**

- **Find the LALR Parser for the following Grammar**

S→ L=R S→R L→*R L→id R→**L** **(16)** **[Nov/Dec 2014]**

LALR PARSER:

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

\{ Initially, every item in I is added to closure(I).

\{ If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such

that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

4. Augment G and produce G‘
5. Construct the canonical collection of set of items C for G‘
6. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

The Sets of Items Construction:

procedure items(G‘);
begin
C= {closure({[S‘→ **. S,$]})};**
repeat
for each set of items I in C and each Grammar symbol X
Such that goto(I,X) is not empty and not in C do
Add goto(I,X) to C
until no more sets of items can be added to C
end

Algorithm for construction of LALR parsing table:

Input : An augmented grammar G‘

Output : The LALR parsing table functions *action* and *goto* for G‘

Method :

1. Construct C ={I0, I1, …. In}, the collection of sets of LR(0) items for G‘.
2. For each core present among the set of LR(1) items, find all sets having that core , and replace these sets by their union.
3. Let C‘={J_0,J_1,....J_m} be the resulting sets of LR(1) items. State *i* is constructed from J*i*.. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in Ji and goto(Ji,*a*) = Jj, then set *action*[*i*,*a*] to ¯shift j‖. Here *a* must be terminal.
   (b) If[A→α·] is in Ji , then set *action*[*i*,*a*] to ¯reduce A→α‖ for all *a* in FOLLOW(A).
   (c) If [S‘→S.] is in Ji, then set *action*[*i*,$] to ¯accept‖.

If any conflicting actions are generated by the above rules, we say grammar is not LALR(1).

4. The goto table is constructed as follows. If J is the union of one or more sets of LR(1) items , that is ,J=I_1UI_2U.....UI_k, then the cores of goto(I_1,X), goto(I_2,X) , ... GOTO(I_k,X) are the same, since I_1, I_2,....I_k all have the same core. Let K be the union of all sets of items having the same core as goto(I_1,X). Then goto(J,X)=K.

Example :

S→ L=R S→R L→*R L→id R→**L**

Augumented Grammar(I)

S‘→S

S→ L=R
S→R
L→*R
L→id
R→**L**
Closure(I) :
I_0 : S'→**.** S , $
S→ **.** L=R , $
S→**.** R , $
L→**.** *R , $
L→**.** id , $
R→**.** **L** , $

I_1: goto(I_0 ,S)
S'→ S **.** , $

I_2: goto(I_0 ,L)

S→ L **.** =R , $
R→**L .** , $

I3: goto(I_0 ,R)
S→R **.** , $

I4: goto(I_0 ,*) , goto(I4,*) ,goto(I6,*)
L→ * **.** R , $
R→**.** **L** , $
L→**.** *R , $
L→**.** id , $
I5: goto(I_0 ,id) , goto(I4,id) , goto(I6,id)
L→ id **.** , $
I6: goto(I2, =)
S→ L = **.** R , $
R→**.** **L** , $
L→**.** *R , $
L→**.** id , $
I7: goto(I4,R)
L→ * R **.** , $
I8: goto(I4,L) ,goto (I6,L)
R→**L .** , $

I9: goto(I6,R)
S→ L = R **.** , $

Applying the Algorithm :

State	Action				Goto		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

Parsing the input String w= *id :

Stack	Input	Action
0	*id$	Shift 4
0 * 4	id$	Shift 5
0 * 4 id 5	$	Reduce by L→id
0 * 4 L	$	Goto(4,L)=8
0 * 4 L 8	$	Reduce by L→R
0 * 4 R	$	Goto(4,R)=7
0 * 4 R 7	$	Reduce by L→*R
0 L	$	Goto(0,L)=2
0 L 2	$	Reduce by R→L
0 R	$	Goto(0,R)=3
0 R 3	$	Reduce by S→R
0 S	$	Goto(0,S)=1
0 S 1	$	Accept

7.YACC-DESIGN OF A SYNTAX ANALYZER FOR A SAMPLE LANGUAGE

❖ **Discuss about YACC for constructing a parser for a sample Language.**

YACC:

- YACC Stands for Yet Another Compiler Compiler.
- YACC is a tool available for converting parse trees from source programs. The input to YACC is a CFG and is having a specific structure. For each production there is an action implemented.

- YACC parses a stream of tokens generated by LEX according to a user specified Grammar.

Structure of a YACC file :

... definitions . . .
%%
. . . rules . . .
%%
. . . code . . .

Definitions:

There are three points to be considered.

i. C Code:

All code between %{ and % } is copied to the beginning of the resulting C File.TThis is used to define Variables Prototypes and routines etc..

ii. Definition:

The definition of a YACC file is concerned with tokens. These token definitions are written to a .h file when YACC compiles this file.

iii. Associativity Rules:

These handle associativity and priority of operators.

Rules:

A number of combinations of patterns and action is more than a single command then it needs to be in braces.

Code:

This can be very elaborate. It includes a call to yylex.

LEX – YACC Interaction :

The LEX Code parses a file of characters and outputs a stream of tokens. YACC accepts a stream of tokens. YACC accepts a stream of tokens and parses it, performing appropriate actions.

If the LEX program is supplying tokenizer YACC program will repeatedly call the yylex routine. The rules will repeatedly call the yylex routine. The rules will probably function by calling return every time LEX have passed a token.

If LEX returns tokens then YACC will process them and they have to agree on what tokens there are. This is done as follows.

- The YACC file will have token definitions in the definition section.

% token NUMBER

- When the YACC file is translated with YACC -d , a header file y.tab.h is created.

#define NUMBER 258

This file can be included in both LEX and YACC Program.

- The Lex file can then call return NUMBER and the YACC program can match on this token.

Return Values:

The LEX Parse can return a symbol that is put on top of the stack so that YACC can access it. This symbol is returned in the variable yylval.

By default this is defined as an int

```
Extern int llval;
%%
[0-9]+ {llval=atoi(yytext); return NUMBER ;}
```

Rules Section:

The rules section contains the grammar of the language we want to parse. This looks like

name1 : THING something OTHERTHING {action}|othersomething THING {other action }
name 2 : . . . . .

This is the general form of CFG with a set of actions associated with each matching right hand side. It is a good convention to keep non terminals in lower case and terminals in uppercase.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically define coming from % token definitions in the YACC Program.

User Code Section :

The minimal main program is

```
int main()
{
  yyparse();
  return 0;
        }
```

In addition to the main program the code section will also contain subroutines to be used either in the YACC or the LEX Program.

UNIT – IV

SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

Syntax Directed Definitions-Construction of syntax Tree-Bottom up Evaluation of S-Attribute Definitions-Design of predictive Translator-Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions. RUN-TIME ENVIRONMENT : Source Language Issues–Storage organization-Storage Allocation-Parameter Passing-Symbol Tables. Dynamic Storage Allocation-Storage Allocation in FORTRAN.

PART-A

1. Define syntax directed definition

A Syntax Directed Definition(SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labled X.

2. List the various methods to evaluate semantic rules.

- Parse Tree Methods
- Rule Based Methods
- Oblivious Method.

3. What is Synthesized Attribute and Inherited Attribute??

A **synthesized attribute** for a non terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.

- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

An **inherited attribute** for a non terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

- An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.
-

4. What is an attribute Grammar?

An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

5. What is Dependency Graph?

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can

be computed.

6. What is S – Attribute Definition and L-Attribute Definition?

S – Attribute Definition

An SDD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SDD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal

L-Attribute Definition

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left(hence "L-attributed"). Each attribute must be either

a. Synthesized, or
b. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X1\ X2 \bullet \bullet \bullet Xn$, and that there is an inherited attribute $Xi.a$ computed by a rule associated with this production. Then the rule may use only:
   (a) Inherited attributes associated with the head *A.*
   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X1\ X2 \bullet \bullet \bullet Xi\text{-}1$ located to the left of Xi
   (c) Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi.

7. What are the two approaches used for de-allocation of blocks?

a. Reference Count:

We keep track of the number of blocks that point directly to the present block. If this count drops to 0 then the block can be de-allocated.

b. Marking Techniques:

This is used to suspend temporarily execution of the program and use the frozen pointers to determine which blocks are in use. We pour paint through these pointers. Any block that is reached by the paint is in use and the rest can be de-allocated.

8. What is Type Systems?

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

9. What are the types of type checking? Write various static check methods.

Static checking: A compiler check the source program follows both the syntactic and semantic conventions of the source language at compile time

Dynamic checking: A compiler check the source program during execution of the target program

Examples of static checks:

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

10. List the different storage allocation strategies and write down the limitations of static allocation.

The strategies are:

- Static allocation
- Stack allocation
- Heap allocation

Limitations of static allocation:

- The size of a data object and constraints on its position in memory must be known at compile time
- Recursive procedure are restricted, because all activations of a procedure use the same binding for local name
- Data structures cannot be created dynamically since there is no mechanism for storage allocation at run time

11. Define activation tree and what are the contents of activation record?

Activation tree

- Control flows sequentially; the execution of a program consists of a sequence of steps, with control being at some specific point in the program at each step.
- Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called. This means the flow of control between procedures can be depicted using trees.

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

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

12. Write down syntax directed definition of a simple desk calculator. [Nov/Dec 2016]

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Fig 5.1: Syntax Directed Definition of simple desk calculator

13. List Dynamic Storage allocation techniques. [Nov/Dec 2016]

- Explicit Allocation of fixed sized blocks.
- Explicit Allocation of variable sized blocks.
- Implicit Deallocation.

14. What is coercion?

Conversion from one type to another is said to be implicit, if it is done automatically by the
compiler. Implicit type conversions also called coercions are limited in many languages. Eg.
Integer can be converted to real but not vice-versa.

15. What is dangling reference and when does it occurs?[May/June2016]

Whenever storage can be deallocated, the problem of dangling references arises.
A dangling reference occurs when there is a reference to storage that has been deallocated

PART-B

SOURCE LANGUAGE ISSUES

- **Describe the source language issues in detail.[Apr/May 2008]**

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition is a declaration that associates an identifier with a* statement. The identifier is the *procedure name*, and the statement is the *procedure body*.
For example, the following is the definition of procedure named *readarray* :

procedure *readarray*; var i : integer;

begin
for i : = 1 to 9 do read(a[i]) end;

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.

4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

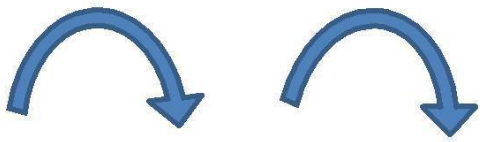
- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.
Declarations may be explicit, such as:
var i : integer ;
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.
The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. ¯Data object‖ corresponds to a storage location that holds values.
The term *environment* refers to a function that maps a name to a storage location.
The term *state* refers to a function that maps a storage location to the value held there.
environment state



name storage value

When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANIZATION

- **Mention in detail any 4 issues in storage organization.[Apr/May 2015]**

STORAGE ORGANISATION

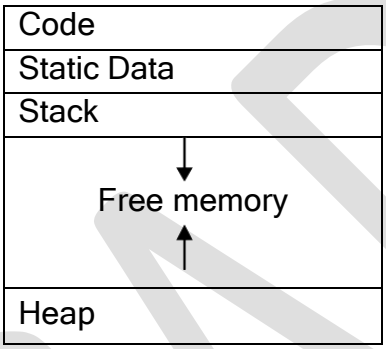
- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating

system maps the logical address into physical addresses, which are usually spread throughout memory.

Typical subdivision of run-time memory:

Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.

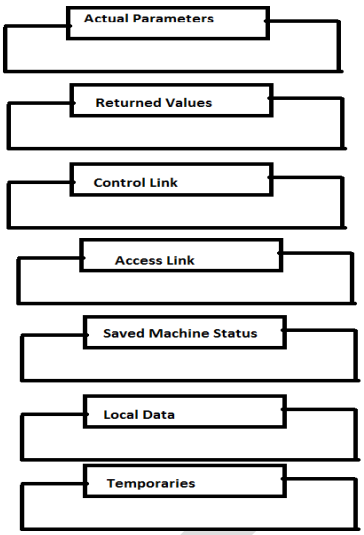
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.



- □ This unused space due to alignment considerations is referred to as padding.
- □ The size of some program objects may be known at run time and may be placed in an area called static.
- □ The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- □ Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- □ Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- □ The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.
- □ Temporary values such as those arising from the evaluation of expressions.



- □ Local data belonging to the procedure whose activation record this is.
- □ A saved machine status, with information about the state of the machine just before the call to procedures.
- □ An access link may be needed to locate data needed by the called procedure but found elsewhere.
- □ A control link pointing to the activation record of the caller.
- □ Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- □ The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

2.STORAGE ALLOCATION

- ❖ **Discuss in detail about Storage Allocation Strategies.[Nov/Dec 2013] [May /June 2016]**
- ❖ **Discuss Run Time Storage Management in detail.[Nov/Dec 2013][Apr/May 2011].**
- ❖ **What are the different Storage Allocation Strategies?Explain.[Nov/Dec 2011]**

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run•time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

Static Allocation

- ➢ In static allocation, names are bound to storage as the program is compiled, so there is no need for a run•time support package.

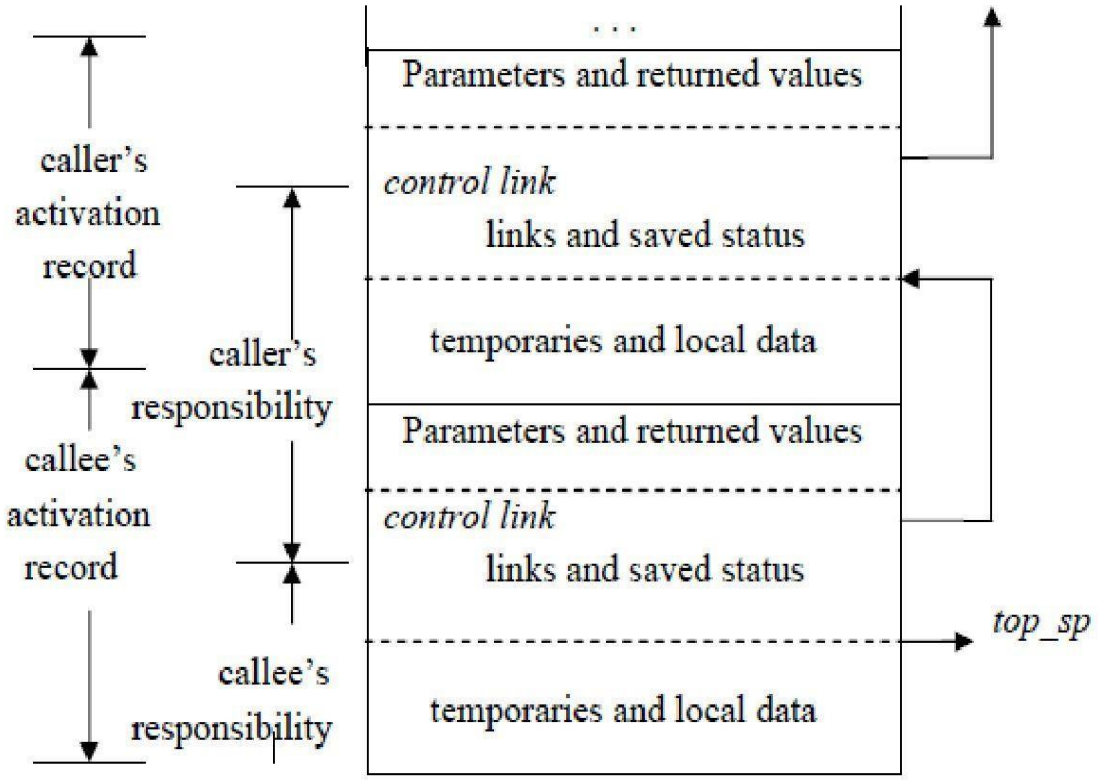
- Since the bindings do not change at run•time, everytime a procedure is activated, its
- names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

Stack Allocation of Space

- All compilers for languages that use procedures, functions or methods as units of user defined actions manage at least part of their run•time memory as a stack. Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack. **Calling sequences:**
- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).

When designing calling sequences and the layout of activation records, the following principles are helpful:

- Values communicated between caller and callee are generally placed at the beginning of the callee‘s activation record, so they are as close as possible to the caller‘s activation record.
- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the
- value of one of the callee‘s parameters determines the length of the array



We must locate the top•of•stack pointer judiciously. A common approach is to have it point to the end of fixed•length fields in the activation record. Fixed•length data can then be accessed by fixed offsets, known to the intermediate•code generator, relative to the top•of•stack pointer.

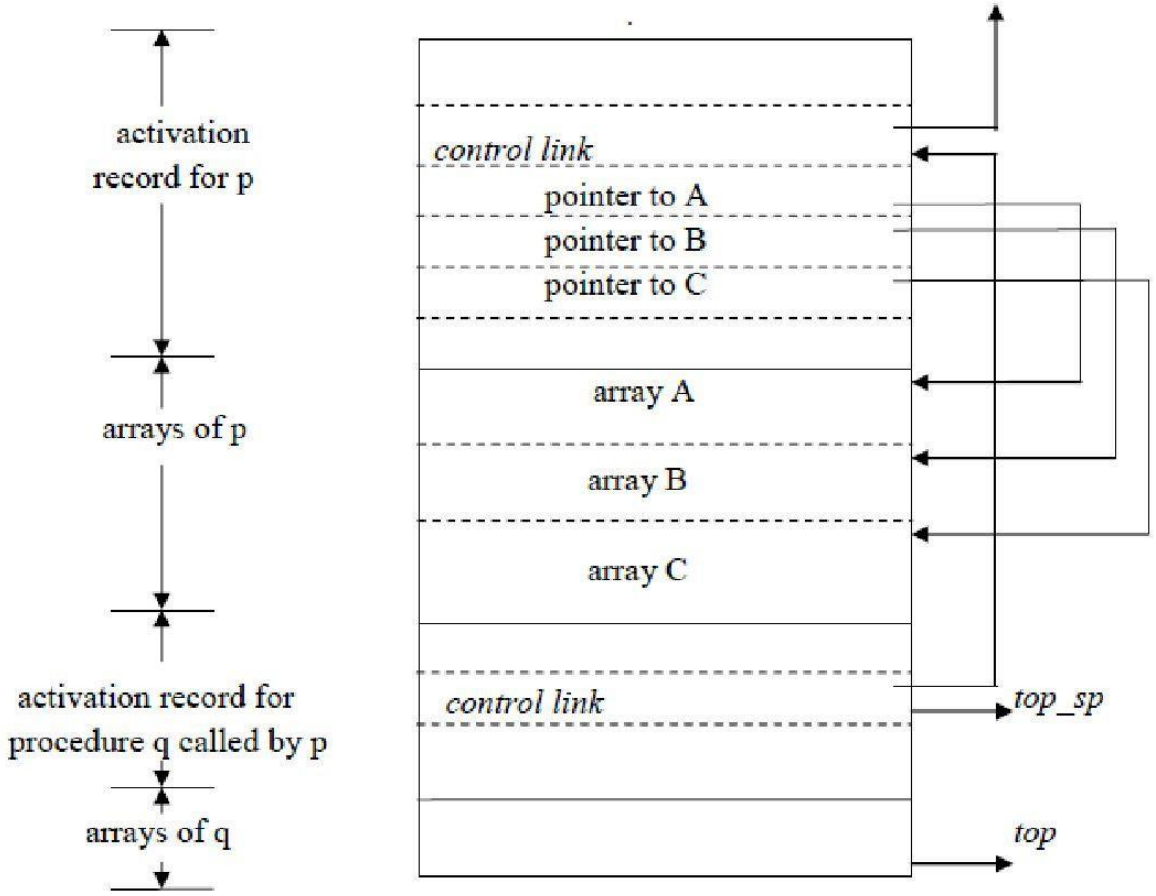
- The calling sequence and its division between caller and callee are as follows. The caller evaluates the actual parameters.
- The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
- The callee saves the register values and other status information. The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

- The callee places the return value next to the parameters.
- 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.

Variable length data on stack:

- The run•time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.

- Access to the data is through two pointers, *top* and *top•sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top•sp* is used to find local, fixed•length fields of the top activation record.
- The code to reposition *top* and *top•sp* can be generated at compile time, in terms of sizes that will become known at run time.

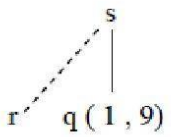
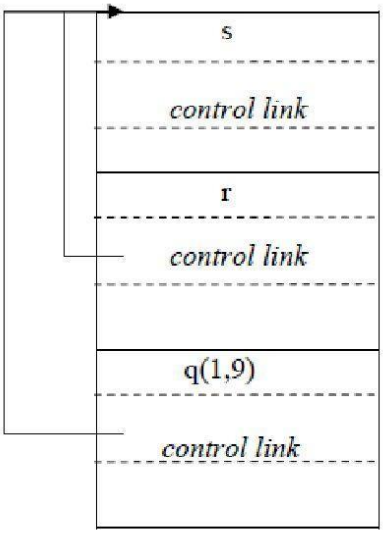
Heap Allocation

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

1. The values of local names must be retained when 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 heapbetween the activation records for s and q.

TYPE SYSTEMS

- **Illustrate type checking with necessary diagram . [Nov/Dec 2016].**

TYPE CHECKING

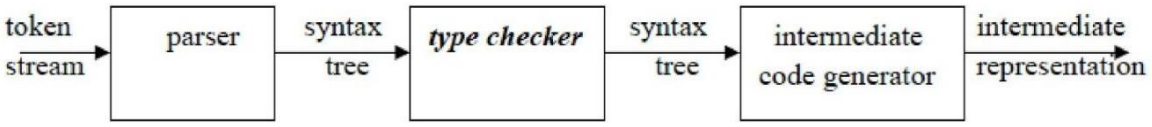
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.

This checking, called *static checking,* detects and reports programming errors. Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. **Flow•of•control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
Type information gathered by a type checker may be needed when code is generated.

SPECIFICATION OF A SIMPLE TYPE CHECKER
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 subexpressions. 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 type saved in 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* **then** S1.*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* }

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : ¯ if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer ‖

Type Expressions

The type of a language construct will be denoted by a ¯type expression.‖

A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.

The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

A special basic type, *type_error* , will signal an error during type checking; *void* denoting ¯the absence of a value‖ allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression. Constructors include:

Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

type row = record address: integer;
lexeme: array[1..15] of char end;
var table: array[1...101] of row;

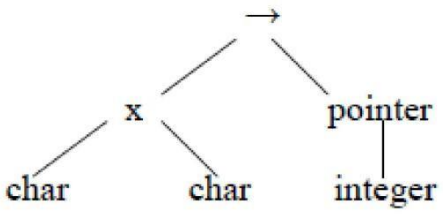
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type ¯pointer to an object of type T‖.

For example, ***var p:*** ↑ ***row*** declares variable p to have type *pointer*(row). ***Functions :*** A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → *pointer* (integer)



Type systems

A *type system* is a collection of rules for assigning type expressions to the various parts of a program.

A type checker implements a type system. It is specified in a syntax•directed manner.

Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.

Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.

Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

- **Explain the specification of a simple type checker.[Nov/Dec 2014] [May /June 2016]**

SPECIFICATION OF A SIMPLE TYPE CHECKER

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 subexpressions. 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* forE 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 E 2 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* **then** S1.*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* }

4. SYNTAX DIRECTED DEFINITION

❖ **For the input expression 3*5+4n, construct an annotated parse tree using syntax directed definition. [May /June 2016]**

Syntax-directed translation (SDT)

- SDT refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.
- SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.
- We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production.
- The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.
- A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

1. Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.
- An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

Example: PRODUCTION SEMANTIC RULE

E.code = E1.code || T.code || '+'

E → E1 + T

- SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

2. Syntax-Directed Translation Schemes (SDT)

- SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.
- Example: In the rule E → E1 + T { print '+' }, the action is positioned after the body of the production.

Inherited and Synthesized Attributes

- Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.
- A non terminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

- A **synthesized attribute** for a non terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.
  - A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- An **inherited attribute** for a non terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Production	Semantic Rules
1) $L \rightarrow E$ **n**	*L.val* = *E.val*
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	*E.val* = *T.val*
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	*T.val* = *F.val*
6) $F \rightarrow (E)$	*F.val* = *E.val*
7) $F \rightarrow$ **digit**	*F.val* = **digit**.*lexval*

Fig 5.1: Syntax Directed Definition of simple desk calculator

Example : The SDD in Fig. 5.1 is based on grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker n.

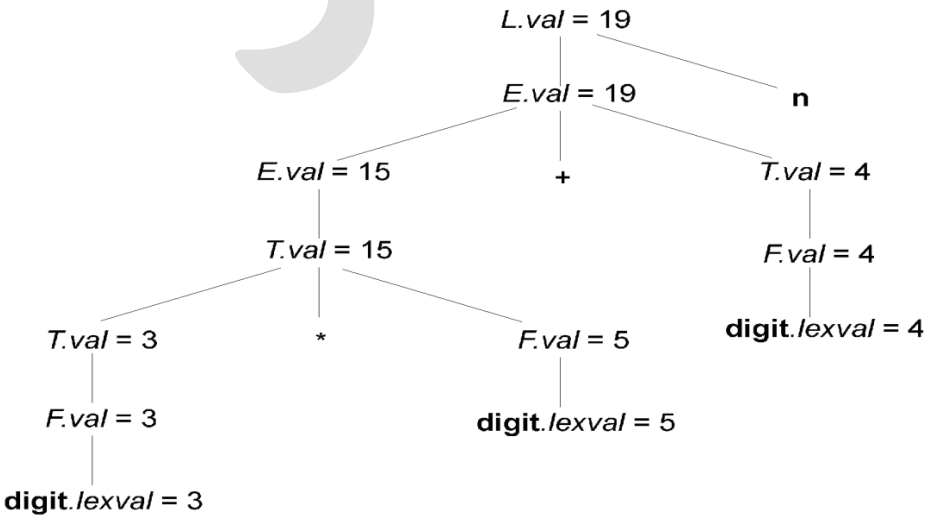
- In the SDD,each of the nonterminals has a single synthesized attribute, called val. We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.
- An SDD that involves only synthesized attributes is called **S-attributed**; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

Attribute Grammar: An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Evaluating an SDD at the Nodes of a Parse Tree

- Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**
- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

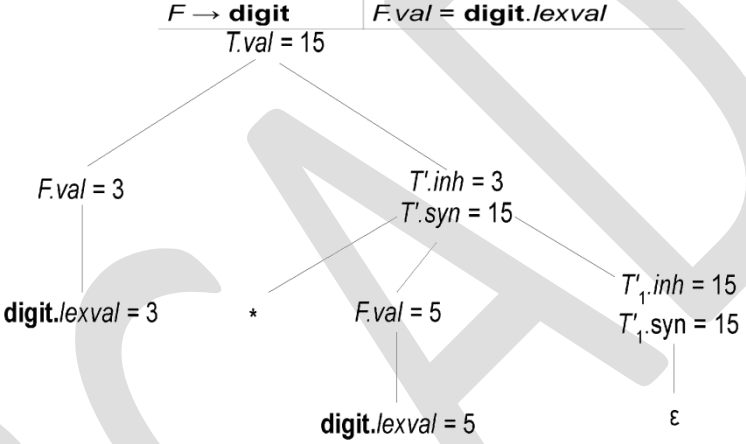
Example: Annotated Parse Tree for 3*5+4n



- Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.
- In the SDD below, the nonterminal T has an inherited attribute *inh* as well as a synthesized attribute val. T inherits F.val from its left sibling F in the production $T \rightarrow F\ T'$.

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

Annotated Parse Tree for 3*5 using the above SDD is as below.



An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all. For example, consider nonterminals A and B, with synthesized and inherited attributes A.s and B.i, respectively, along with the production and rules as in Fig.5.2. These rules are circular; it is impossible to evaluate either A.s at a node N or B.i at the child of N without first evaluating the other. The circular dependency of A.s and B.i at some pair of nodes in a parse tree is suggested by Fig.

e.g.

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

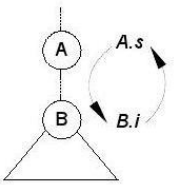


Fig . The circular dependency of *A.s* and *B.i* on one another .

Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

- Each attribute is associated to a node
- If a semantic rule associated with a production *p defines the* value of synthesized attribute *A.b in terms of the value of X.c, then graph has an edge from X.c to A.b*
- If a semantic rule associated with a production *p defines the* value of inherited attribute *B.c in terms of value of X.a, then* graph has an edge from *X.a to B.c*

4 b) Construct a syntax directed definition scheme that takes strings of a"s,b"s and c"s as input and produces as output the number of substrings in the input string that correspond to the pattern a(a/b * c +(a/b)*b. For example the translation of the input string "abbcabcababc" is "3".

1. **Write a context free grammar that generate all strings of a"s,b"s and c"s**
2. **Give the semantic attributes for the grammar symbols.**
3. **For each productions of the grammar a set of rules for evaluation o the semantic attributes.**

Nov/Dec :2016

Solution:

1. The CFG
   G = {{a,b,c},{S},S,P} for all strings over the alphabet {a,b,c} with Pas the set of productions given below .S->SAS->SbS->ScS->aS->bS->c.
2. Given the grammar above

Define three synthesized attributes for the non terminals symbol S,namely nA1,nA2, and total.The idea of these attributes is that in the first attribute will capture the number of a‘s to left of a given c character, the second attribute, nA@, the number of a‘s to the right of that character.so that we can then add the value of a‘s to the right of c ,so that when find a new c,we copy the value of a‘s that were to the right of the first c and which are now to the left of the second c.

3. As suc ha set of rules is as follows, jere written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.
   S1-> S2 a {S1.nA1=S2.nA1 +1;S1.nA2=S2.total = S2.total;}
   S1-> S2 b {S1.nA1=S2.nA1;S2.nA2=S2.nA2;S1.total=S2.total+S2.nA2;}
   S1-> S2 c {S1.nA1=0;S1.nA2=S2.nA1;S1.total = S2.total;}
   S1->a{S1.nA1=1;S1.nA2=0;S1.total=0;}
   S1->b{S1.nA1=0;S1.nA2=0;S1.total=0;}
   S1->c{S1.nA1=1;S1.nA2=0;S1.total=0;}

5.CONSTRUCTION OF SYNTAX TREE

❖ **Describe the procedure to construct the syntax tree using syntax directed translation scheme.**

CONSTRUCTION OF SYNTAX TREE

Construction of Syntax Trees

- SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.
- Syntax trees are useful for representing programming language constructs like expressions and statements.

- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  - e.g. a syntax-tree node representing an expression *E1 + E2 has* label + and two children representing the sub expressions *E1 and E2*
- Each node is implemented by objects with suitable number of fields; each object will have an *op field that is the label of the node with* additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function **Leaf(op, val)**
- If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function **Node(op, c1, c2,...,ck)** .

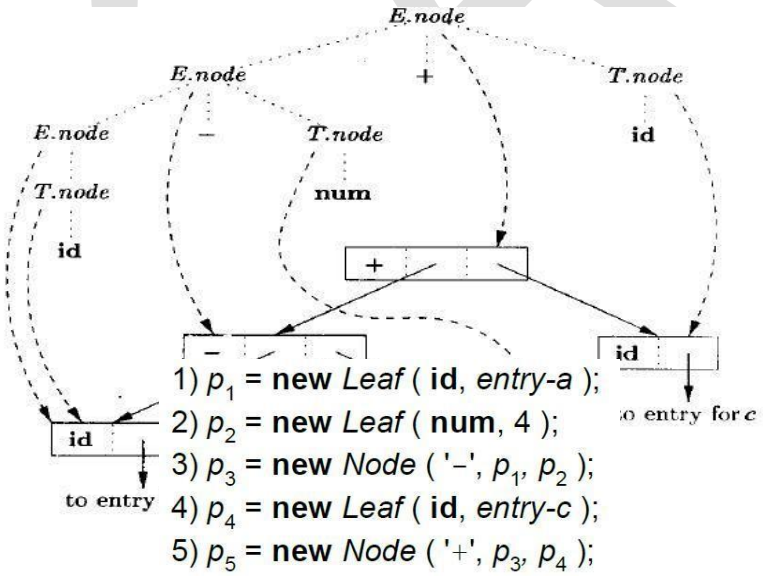
Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node,* which represents a node of the syntax tree.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('−', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, *id.entry*)
6) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, *num.val*)

Steps in the construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

Syntax tree for a-4+c using the above SDD is shown below.



Constructing Syntax Trees during Top-Down Parsing

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

6. PARAMETER PASSING

❖ **Explain the various parameter Passing Methods. [Regulation 13-New]**

PARAMETER PASSING

When one procedure calls another the usual method of communication between them is through non local names and through parameter of the called procedure.

Various parameter passing methods:

- call by value: passing r-values.
- call by reference: passing l-values.
- call by copy-restore: hybrid between call by-value and call by-reference.
- call by name: passing via name substitution.

Call by Value

• Each actual argument is evaluated before call. On entry, the resulting value is copied and bound to the formal parameter; which behaves just like a local variable.

• Advantages: –

Simple; easy to understand

• Formal parameters can be used as local variables, Updating them doesn‘t affect actuals in calling procedure:

double hyp(double a, double b)

{ a = a * a; b = b * b;

return sqrt(a + b); // use built-in sqrt() } //end

Call by Reference

Implemented by passing address of actual parameter

• On entry, the formal is bound to the address, providing a reference to actual parameter from within the subroutine

• If actual argument doesn‘t have an l-value (e.g., ¯2 + 3‖), then either: – Forbid in language, i.e. treat as an error; compiler catches this – evaluate it into a temporary location and pass its address

• Advantages

No more large copying

Actual parameter can be updated

Call by Copy-Restore

• Each actual argument is evaluated to a value before call

• On entry, value is bound to formal parameter just like a local

• Updating formal parameters doesn‘t affect actuals in calling procedure during execution

• Upon exit, the final contents of formals are copied into the actual

• Thus, behaves like call by reference in most ¯normal‖ situations, but may give different results when concurrency or aliasing are involved:

type t is record a, b: integer; end record; r : t;

procedure foo(s : in out t)

begin r.a := 2; s.a := s.a + 3;

end foo;

r.a := 1; foo(r);

print(r.a);

7. DESIGN OF A PREDICTIVE TRANSLATOR

❖ **Explain Design of a Predictive Translator. (Regulation 2013)**

DESIGN OF A PREDICTIVE TRANSLATOR

The construction of predictive parsers to implement a translation scheme based on a grammar suitable for top-down parsing.

Algorithm: Construction of a predictive syntax-directed translator.

Input. A syntax-directed translation scheme with an underlying grammar suitable for predictive parsing.

Output. Code for a syntax-directed translator.

Method.

1. For each nonterminal A, construct a function that has a formal parameter for each inherited attribute of A and that returns the values of the synthesized attributes of A.

2. The code for nonterminal A decides what production to use based on the current input symbol.

3. The code associated with each production does the following:

We consider the tokens, nonterminals, and actions on the right side of the production from left to right.

i) For token X with synthesized attribute x, save the value of x in the variable declared for X**.** x. then generate a call to match token X and advance the input.

ii)For nonterminal B, generate an assignment c:=B(b_1,b_2,…,b_k)with a function call on the right side, where b_1,b_2,…,b_k are the variables for the inherited attributes of B and c is the variable for the synthesized attribute of B.

iii)For an action , copy the code into the parser, replacing each reference to an attribute by the variable for that attribute.

Example: Translation Scheme for constructing Syntax Trees

E → T {R.i := T.nptr}
R {E.nptr := R.s}
R → +
T {R_1.i := mknode (_+‘, R.i, T.nptr)}
R_1 {R.s := R_1.s}
R → -
T {R_1.i := mknode (_-‘, R.i, T.nptr)}
R_1 {R.s := R1.s}
R → ε {R.s := R.i}
T → (E) {T.nptr := E.nptr}
T → id {T.nptr := mkleaf (id, id.entry)}
T → num {T.nptr := mkleaf (num, num.value)}

The above grammar is LL(1), and hence suitable for top-down parsing. From the attributes of the nonterminals in the grammar, we obtain the following types for the arguments and results of the functions for E, R, and T. since E and T do not have inherited attributes, they have no arguments.

function E :↑syntax_tree_node;

function R(i:↑syntax_tree_node):↑syntax_tree_node;

function T:↑syntax_tree_node;

we combine two of the r-productions in fig 4.1 to make the translator smaller.

The new productions use token **addop** to represent + and -:

Parsing procedure for the productions R → addop T R | ε:

R → **addop**

T {R_1.i := mknode(**addop**.lexeme, R.i, T.nptr)}

R_1 {R.s := R_1.s}

R → ε {R.s := R.i}

The code for R is based on the parsing procedure is

Procedure R;

begin

if lookahead=**addop** then begin

match(**addop**);T;R

end

else begin /*do nothing*/
end

end;

If the lookahead symbol is addop, then the production **R → addop T R** is applied by using the procedure match to read the next input token after addop, and then calling the procedures for T and R. otherwise, the procedure does nothing, to mimic the procedure R → ε.

The procedure for R in
Recursive-descent construction of syntax trees:

```
function R(i:↑syntax_tree_node):↑syntax_tree_node;
        var nptr, i1, s1, s: ↑syntax_tree_node;
        addoplexeme : char;
begin
        if lookahead = addop then begin
        /* production R → addop T R */
        match(addop);
        nptr := T;
        i1 := mknode(addoplexeme, i, nptr);
        s1 := R(il)
        s := s1;
        end
```

else s := i; /* production R → ε */

return s

end;

Contains code for evaluating attributes. The lexical value lexval of the token addop is saved in addoplexeme, addop is matched, T is called, and its result is saved using nptr. Variable i1 corresponds to the inherited attribute R_1.i, and s1 to the synthesized attribute R_1.s. the **return** statement returns the value of s just before control leaves the function. The functions for E and T are constructed similarly.

UNIT – 5
CODE OPTIMIZATION AND CODE GENERATION

Principal Sources of Optimization-DAG-Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator-A simple Code Generator Algorithm.

PART – A

1. **What are the properties of optimizing compilers? [May /June 2016]**
   - 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.
     - Common subexpression elimination
     - Dead code elimination
     - Code movement
     - Strength reduction
2. **List the terminologies used in basic blocks.**

   Define and use – the three address statement a:=b+c is said to define a and to use b and c.

   Live and dead – the name in the basic block is said to be live at a given point if its value is used after that point in the program. And the name in the basic block is said to be dead at a given point if its value is never used after that point in the program.
3. **What is a DAG? Mention its applications**. **[May /June 2016]**

   Directed acyclic graph(DAG) is a useful data structure for implementing transformations on basic blocks. DAG is used in
   - Determining the common sub-expressions.

- Determining which names are used inside the block and computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- Simplifying the list of quadruples by eliminating the common su-expressions and not performing the assignment of the form x := y unless and until it is a must.

4. What is code motion and dead code elimination?

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.

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.

5. What are basic blocks and flow graphs?

A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Eg. t1:=a*5
t2:=t1+7
t3:=t2-5
t4:=t1+t3
t5:=t2+b

Flow graphs: A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by basic blocks.
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B1 to block B2 if B2 immediately follows B1 in the given sequence. We can say that B1 is a predecessor of B2.

6. What are machine idioms and constant folding?

Machine idioms:

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.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Ex: int i=5; k=i+j; after, int i=5; k 5+j;

7. Define peephole optimization. List the characteristics of peephole optimization.

Definition: Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence.

Characteristics of peephole optimization: [Nov/Dec 2016]

- Redundant instruction elimination
- Flow of control optimization
- Algebraic simplification
- Use of machine idioms

8. How do you calculate the cost of an instruction?

The cost of an instruction can be computed as one plus cost associated with the source and destination addressing modes given by added cost.

MOV R0,R1	1
MOV R1,M	2
SUB 5(R0),*10(R1)	3

9.Identify the constructs for optimization in basic blocks. [Nov/Dec 2016]

Leaders : The first statement of basic blocks.

- The first statement is a leader
- Any statement that is the target of a conditional or unconditional goto is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

9. Generate code for the following C Statement assuming three registers are available :

x=a/(b+c)-d*(e+f). **[April/May 2015]**

The three address code will be:

```
t1 := b+c
t2 := a1/t1
t3 := t2-d
t4 := e+f
t5 := t3 * t4
x := t5
```

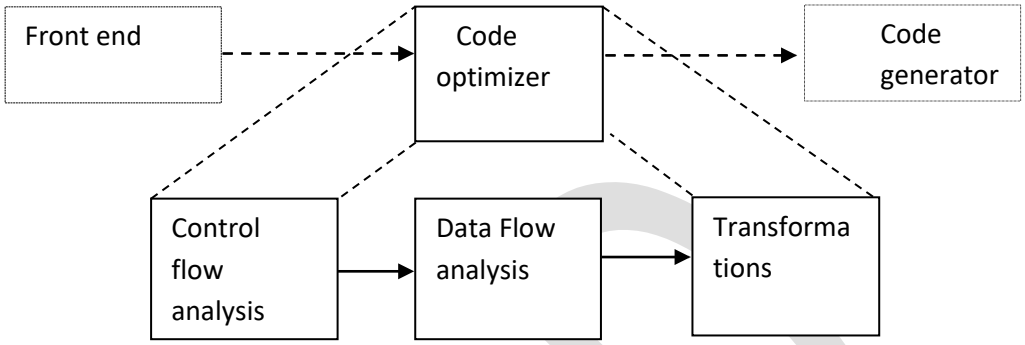
Code will be :

```
MOV b, R0
ADD c, R0
MOV a1, R1
DIV   Ro , R1
SUB d , R1
MOv f , R2
ADD e , R2
MUL   R1 , R2
MOV R2 , x
```

11. What is register descriptor and address descriptor?

- For each available register, a register descriptor keeps track of the variable names whose current value is in that register.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found.

12. Give the block diagram of organization of code optimizer.



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. 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.

15. What are the different data flow properties?

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

16.Write three address code sequence for the assignment statement. [May /June 2016]

d := (a-b) + (a-c) +(a-c)

The three address code is,

t1 := a-b
t2 := a-c
t3 := t1 + t2
t4 := t3 + t3

d := t4

PART-B

1.PRINCIPAL SOURCES OF OPTIMIZATION

- **Write in detail about the Function Preserving Transformation.[Apr/May 2011]**
- **Explain loop optimization in detail and apply to an example. [Apr/May 2015]**
- **Explain Principal sources of optimization. [Nov/Dec 2014]**
- **Write in detail about loop optimization. [Nov/Dec 2013]**
- **Describe in detail the principal sources optimization. [Nov/Dec 2011]**
- **Explain the principal sources of optimization with examples. [May/Jun 2016]**

PRINCIPAL SOURCES OF OPTIMIZATION

- A transformation of a program is called local if it can be performed by looking only at the statements 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
t 6: =b [t 4] +t 5

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 t 4: =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 byeliminating 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:

- We now give a brief introduction to 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:

1.code motion, which moves code outside a loop;

2. Induction -variable elimination, which we apply to replace variables from inner loop.

3. Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

1. 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 */
```

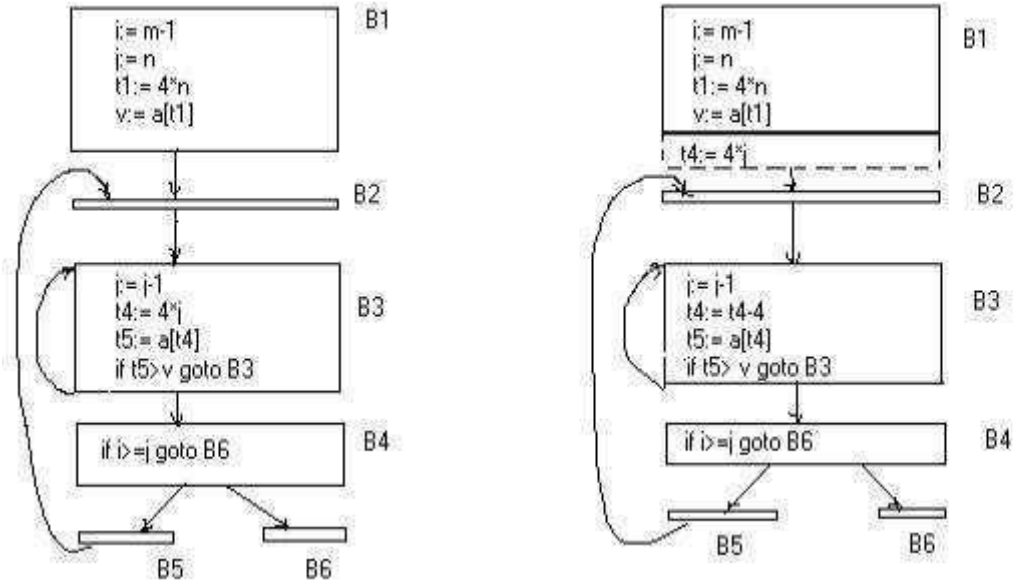
2. Induction Variables :

- 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 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 t 4:=4*j surely holds after such an assignment to t 4 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 t 4:= 4*j by t4:= t4-4. The only problem is that t 4 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 initialized, shown by the dashed addition to block B1 in second Fig.



before **after**

- 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.

3. 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.

For Example

In block B3

j=j-1

t4=4*j

Can be replaced by

j=j-1
t4=t4-4

2.DAG

- **What are the advantages of DAG Representation?Give example. [Apr/May 2015]**
- **Describe the algorithm for constructing DAG with an example. [Apr/May 2015]**
- **Generate DAG Representation with an example and list out the applications of DAG Representation. [Nov/Dec 2014]**
- **Construct DAG and three address code for the following C Code [Nov/Dec 2013]**

```
prod=0
i=1
while(i<=20)
{ prod=prod+a[i]*b[i]
i=i+1
}
```

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  - Leaves are labeled by unique identifiers, either variable names or constants.
  - Interior nodes are labeled by an operator symbol.
  - 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

Application of DAGs:

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.

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) . (Checkingfor 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).

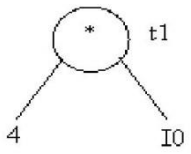
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the noden found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

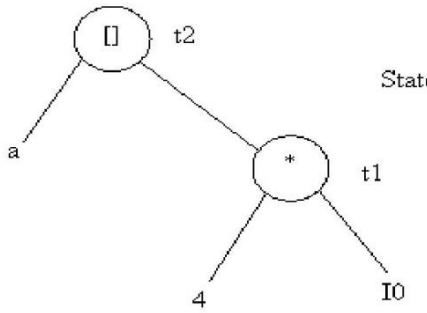
Stages in DAG Construction

(a)



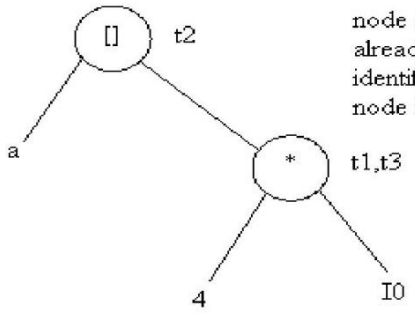
Statement (1)

(b)



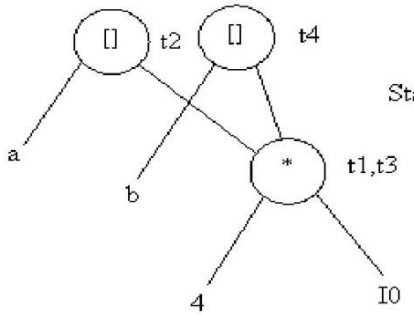
Statement (2)

(c)



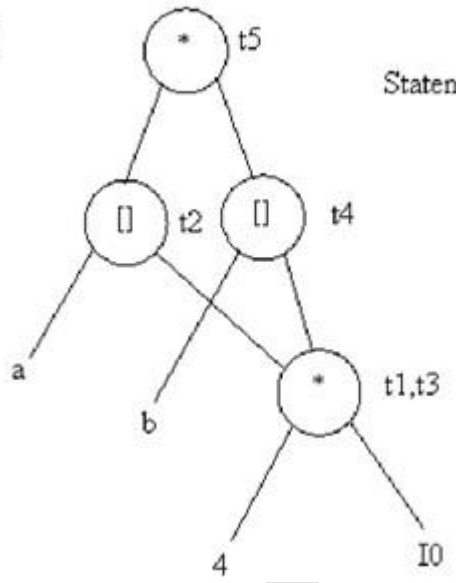
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)

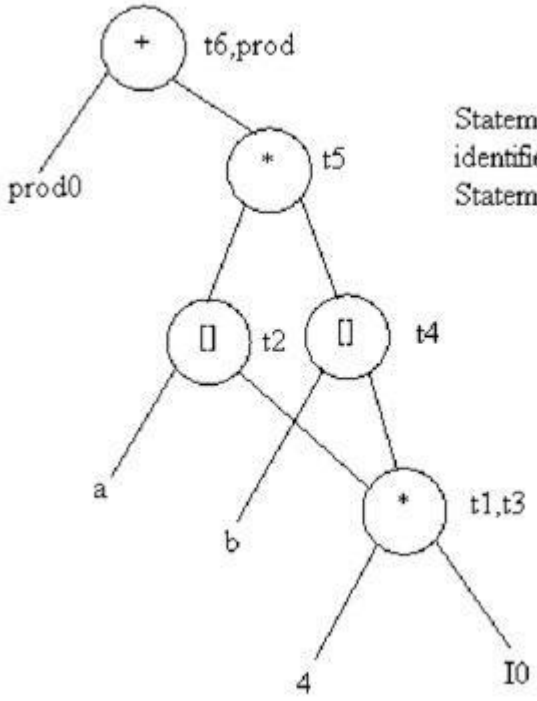


Statement (4)

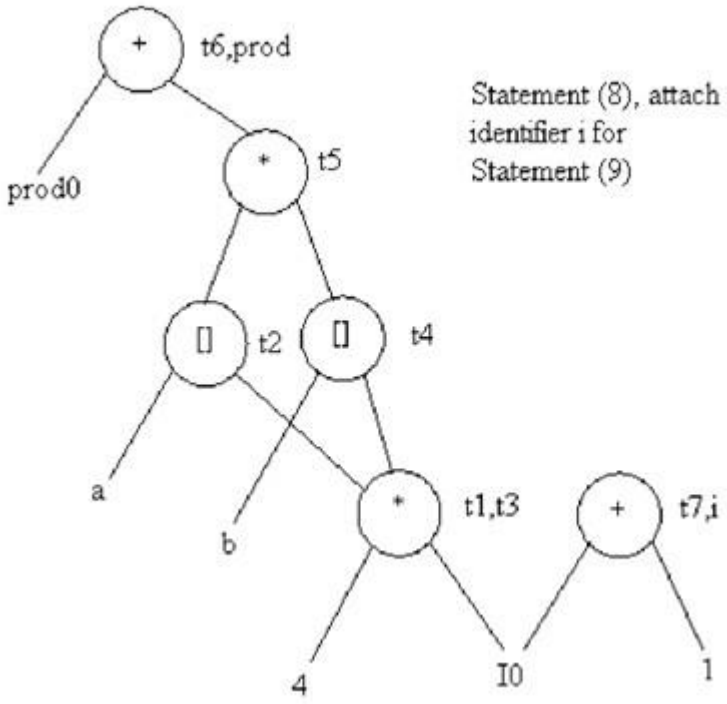
(e)


*
t5
Statement (5)
[]
t2
[]
t4
a
b
*
t1,t3
4
I0

(f)

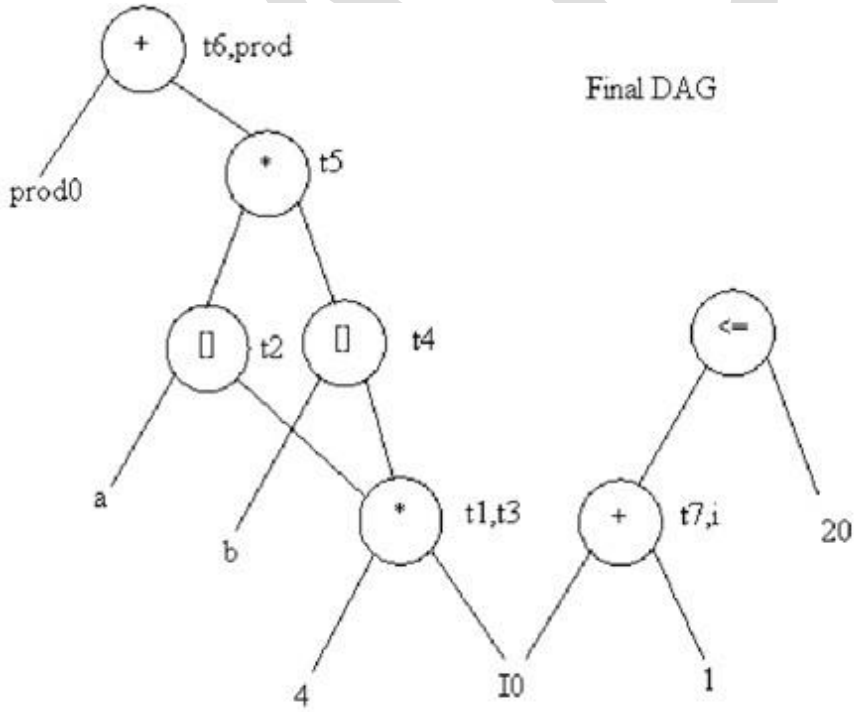

+
t6,prod
prod0
*
t5
Statement (6), attach identifier prod for Statement (7)
[]
t2
[]
t4
a
b
*
t1,t3
4
I0

(g)



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

(h)



Final DAG

3. OPTIMIZATION OF BASIC BLOCKS

❖ **Illustrate the optimization of basic blocks with an example. [Nov/Dec 2014]**

❖ **Discuss in detail the process of optimization of basic blocks. Give an example. [May/Jun 2014]**

❖ **Explain in detail the optimization of basic blocks. [Nov/Dec 2011]**

OPTIMIZATION OF BASIC BLOCKS

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

1. Structure -Preserving Transformations
2. Algebraic Transformations

1. 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
d=a-d
c=d+c
```

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.

2. 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. GLOBAL DATA FLOW ANALYSIS

- **Explain Global Data Flow Analysis in detail. [Nov/Dec 2013,2016]**
- **What is Data Flow Analysis? Explain Data Flow Analysis with amples. [May/Jun 2014]**
- **Write about Data Flow Analysis of Structured Programs. [Nov/Dec 2011]**
- **How to trace Data Flow Analysis of Structured Programs. [May/Jun 2012]**

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- 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 :

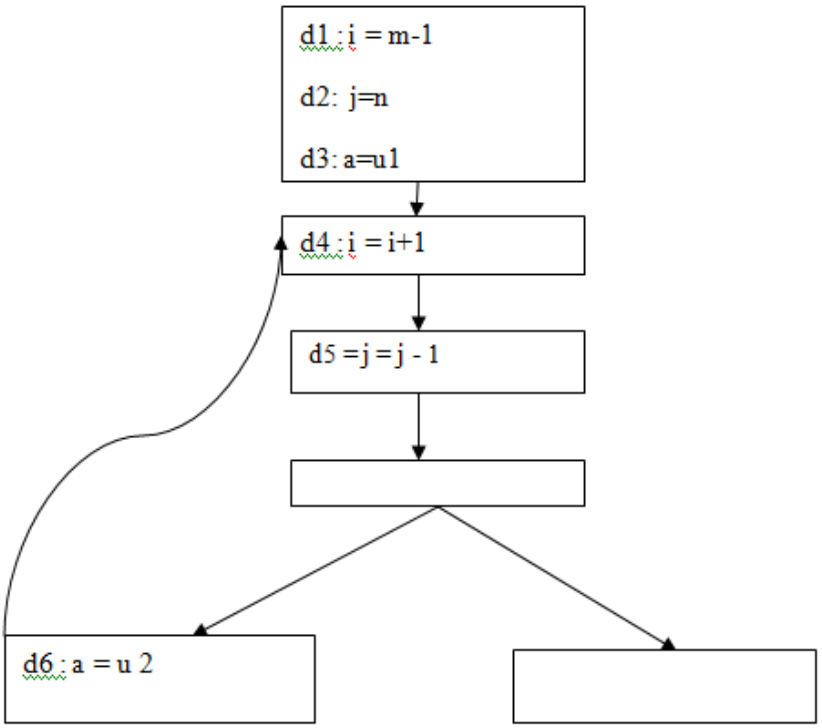
out [S] = gen [S] U (in [S] – kill [S])

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:

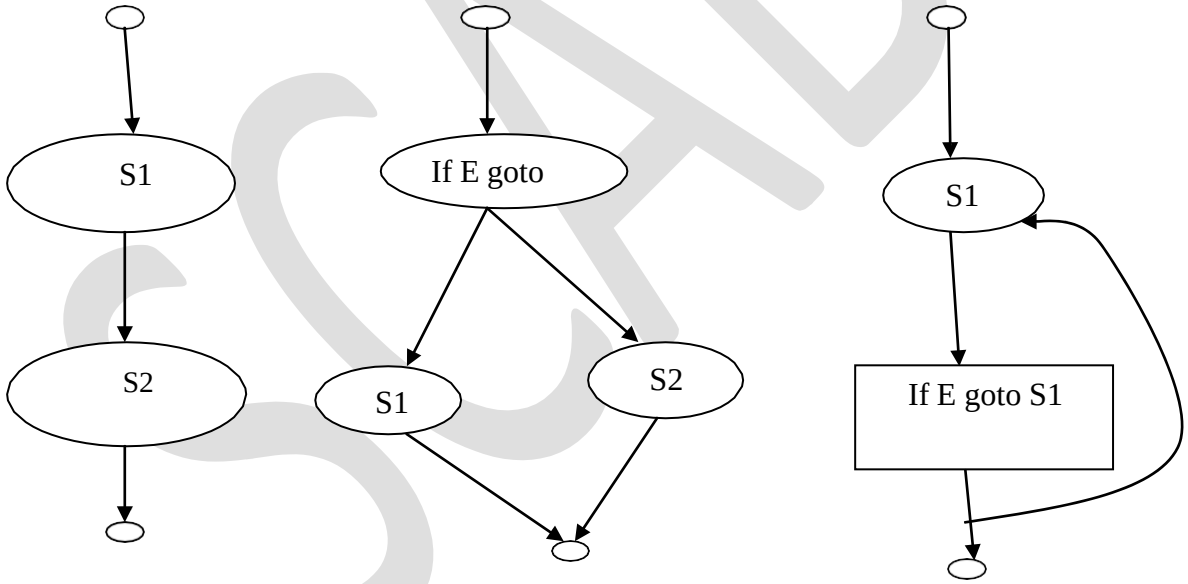
- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.
- These statements certainly define a value for x, and they are referred to as **unambiguous** 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.

Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

- S ⟶ id: = E| S; S | if E then S else S | do S while E E ⟶ id + id| id
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2 if E then S1 else S2 do S1 while E

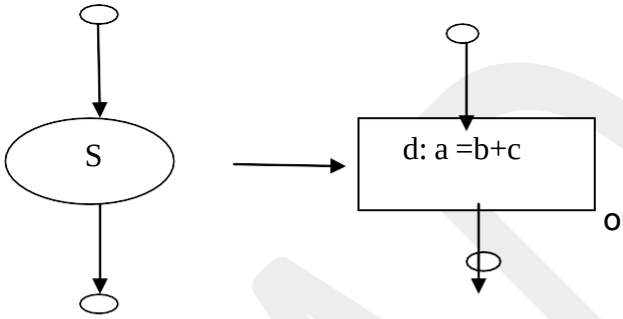
Structured Control Constructs

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block

when it leaves the region.

- We say that the beginning points of the dummy blocks at the entry and exit of a statement‘s region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions “generated” by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



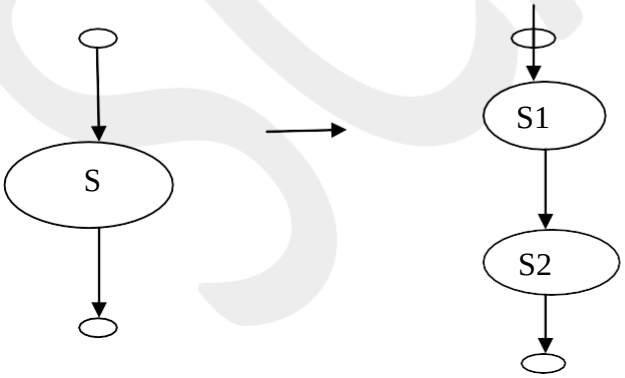
gen[S]={d}

l[[S]=Da-{d}

out [S]=gen[S]U(in[S]-kill[S]

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
- Gen[S]={d}
- On the other hand, d ¯kills‖ all other definitions of a, so we write Kill[S] = Da – {d}
- Where, Da is the set of all definitions in the program for variable a.

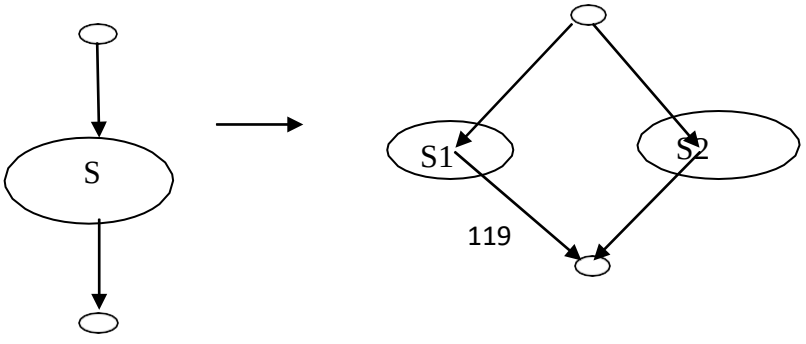
ii)



gen[S]=gen[S2]U(gen[S1]-kill[S2]) Kill[S] = kill[S2] U (kill[S1] – gen[S2])

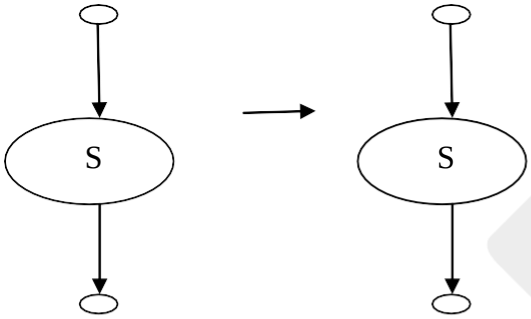
in [S1] = in [S] in [S2]= out [S1] out [S] = out[S2]

iii)



gen[S]=gen[S1]Ugen[S2]
kill[S]=kill[S1]∩kill[S2]
in [S1] = in [S]
in [S2]= in [S]
out [S] = out[S1]

iv)



gen[S]=gen[S1]
kill[S]=kill[S1]
in[S1]=in[S]Ugen[S1]
out[S]=out[S1]

- Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write
- gen[S]=gen[S2] U (gen[S1]-kill[S2])
- Similar reasoning applies to the killing of a definition, so
  we have Kill[S] = kill[S2] U (kill[S1] – gen[S2])

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are ¯uninterpreted‖; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the ¯true‖ gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer

lies in the use intended for these data.

- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached.

Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.
- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know in[S] we compute out by equation, that is
- Out[S] = gen[S] U (in[S] - kill[S])
- Considering cascade of two statements S1; S2, as in the second case. We start by observingin[S 1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S.
- In[S1] = in[S2] = in[S]
- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

- Out[S]=out[S 1] U out[S2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming
- languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as‖ use-definition chains‖ or ¯ud-chains‖, which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions ofa.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after

all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

5. ISSUES IN THE DESIGN OF A CODE GENERATOR

- **Write in detail about the issues in the design of a code Generator.(10) [Apr/May 2011 ,May/Jun 2012] [Nov/Dec 2016].**
- **Discuss the various issues in the code Generation.[May/Jun 2014,Nov/Dec 2011]**
- **Explain various issues in the design of code generator. [May 2016]**

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.

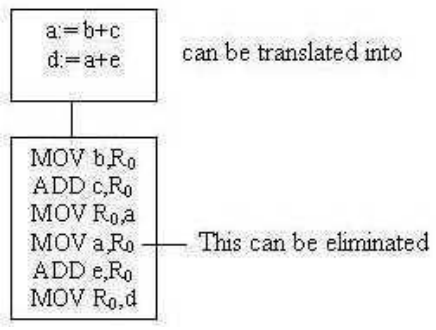
4. Instruction selection:

□ 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.

6.a. SIMPLE CODE GENERATOR ALGORITHM

- ❖ **Explain code generation phase with simple code generation Algorithm.[Nov/Dec 2014]**
- ❖ **Explain in detail about the simple code generator. [Nov/Dec 2011]**
- ❖ **Explain code Generation Algorithm in detail.[May/Jun 2012]**
- ❖ **Write note on simple code generator. [May /June 2016]**

A SIMPLE CODE GENERATOR

- □ 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.

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.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v : =t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements **a : = b [i]** and **a [i] : = b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments **a : = *p** and ***p : = a**

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x <0 goto z	MOV y, R0 ADD z, R0 MOV R0,x CJ< z

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p : = a	MOV a, *Rp	2

7. PEEPHOLE OPTIMIZATION

- **Explain peephole optimization in detail.**
- **Write an algorithm for constructing natural loop of a back edge. Nov : 16**

PEEPHOLE OPTIMIZATION

Many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is *peephole optimization,* which is done by examining a sliding window of target instructions (called the *peephole)* and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation. The peephole is a small, sliding window on a program. The characteristics of Peephole Optimization:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

If we see the instruction sequence

LD **a,** R_0

ST R_O, **a**

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of **a** has already been loaded into register R0. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

Eliminating Unreachable Code

Another opportunity for peephole optimization 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 code fragments that are executed only if a variable debug is equal to 1. In the intermediate representation, this code may look like

if debug == 1 goto LI
goto L2
LI: print debugging information
L2:

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code sequence above can be replaced by

if **debug != 1 goto L2**
print debugging information
L2:

If **debug** is set to 0 at the beginning of the program, constant propagation would transform this sequence into

if **0 != 1 goto L2**
print debugging information
L2:

Now the argument of the first statement always evaluates to ***true,*** so the statement can be replaced by **goto L2.** Then all statements that print debugging information are unreachable and can be eliminated one at a time.

Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These 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 sequence

goto Li
LI: goto L2

by the sequence

goto L2
LI: goto L2

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

if **a < b goto LI**
LI: goto L2

Can be replaced by the sequence

if **a < b goto L2**
LI: goto L2

Finally, suppose there is only one jump to **LI** and **LI** is preceded by an unconditional goto. Then the sequence

goto LI
LI: if a < b goto L2
L3:

may be replaced by the sequence

if a < b goto L2
goto L3
L3:

While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

x = x + 0
or
x = x * 1

in the peephole. Similarly, reduction-in-strength transformations can be applied in the peephole to replace 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, *x2* 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 approximated as multiplication by a constant, which may be cheaper.

Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. 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 the 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 x = x + 1.

Industrial / Practical Connectivity of the subject

Industry Connectivity and Latest Developments

Industry Connectivity

We have collaborated with Infosys Pvt Limited, called Infosys campus connect. Based on the Industry requirement Problem Solving Techniques (PST) such as NFA, DFA design for tokens are practiced among the students.

Latest Development Tools

1. Selenium IDE
2. JFLAP 7.0

B.E/B.Tech DEGREE EXAMINATION, MAY/JUNE 2016
Sixth semester
Computer science and Engineering

CS6660 - COMPILER DESIGN
(Regulation 2013)
Answer ALL questions
PART- A (10 x 2 =20)

1. What are the two parts of compilation? Pg.No : 5
2. Illustrate diagrammatically how a language is processed. Pg .No : 7
3. Write a grammar for branching statements. Pg. NO : 23
4. List the operations on languages. Pg.NO : 23
5. Write an algorithm for FIRST and FOLLOW in parser. Pg.Nn : 39
6. Define ambiguous grammar. Pg.No : 38
7. What is DAG ? Pg .N o :101
8. when does dangling reference occurs? Pg. No : 76
9. What are the properties of optimizing compilers? Pg. No : 101
10. Write three address code sequence for the assignment statement. Pg. No : 104

PART – B (5x16=80 Marks)

11 . a) Describe the various phases of compiler and trace it with the program segment (position =initial+rate * 60). (16) Pg.No : 11

OR

b) i) Explain Language processing system wit hneat diagram. (8) Pg.No:7

ii) Explain the need for grouping of phases. (4) Pg.No : 19

iii) Explain Various Error Encounered in different phases of compiler. (4)

Pg.No : 16

12.a) i) Differentiate between Token, Pattern and Lexeme.(6) Pg.NO : 21

ii) What are the issues of lexical analysis? (4) Pg.NO : 23

iii) Write notes on Regular Expressions. (6) Pg.NO : 34

OR

b) i) Write notes on Regular Expressions to NFA. Construct Regular Expressions to NFA for the sentence (a/b)*a. (10) Pg.No : 34

ii) Construct DFA to recognize the language (a/b)*ab. (6) Pg.No : 34

13. a) i) Consider Stack implementation of shift reduce parsing for the grammar:

E → E+E

E → E*E

E → (E)

E → id and the input string id1+id2*id3 (8) Pg.No :57

ii) Explain LL(1) grammar for the sentence S->iEtS | iEtSeS | a E->b(8) Pg .No:57

OR

b) i)Write an algorithm for Non recursive predictive parsing. (6)Pg.No : 54

ii) Explain context free grammars with examples. (10) Pg.No :44

14.a) i) Construct a syntax directed definition for constructing a syntax tree for assignment statements. (8) Pg.No:89

S-> id:=E

E-> E1 +E2

E-> E1*E2

E->-E1

E-> (E1)

E->id

ii) Discuss specification of a simple type checker. (8) Pg.No : 87

OR

b) Discuss different storage allocation strategies. (16) Pg.NO:79

15. a) Explain principal sources of optimization with examples. (16) Pg.No : 104

Or

b) i) Explain various issues in the design of code generator.(8) Pg.No :123

ii) Write note on simple code generator. (8) Pg.No :125

B.E/B.Tech DEGREE EXAMINATION, NOV / DEC 2016
Sixth semester
Computer science and Engineering
CS6660 - COMPILER DESIGN
(Regulation 2013)
Answer ALL questions
PART- A (10 x 2 =20)

1. What is symbol Table? [Pg.No :6]
2. List the various compiler construction tools. [Pg.No :7]
3. List the rules that form the BASIS. [Pg.No :22]
4. Differentiate Token, Pattern and Lexeme. [Pg.No :21]
5. Construct the parse tree for -(id+id) [Pg.No :38]
6. What is meant by handle pruning? [Pg.No :40]
7. Write down syntax directed definition of a simple desk calculator. [Pg.No :75]
8. List Dynamic Storage allocation techniques. [Pg.No :76]
9. Identify the constructs for optimization in basic blocks. [Pg.No :103]
10. What are the Characteristics of peephole optimization? [Pg.No :102]

PART – B (5x16=80)

11.(a) (i)Explain the phases of compiler wit ha neat diagram.(10) [Pg.No :11]

(ii) Write notes on compiler construction tools. (6) [Pg.No :19]

OR

(b) (i) Explain the need for grouping of phases. (8) [Pg.No :19]

(ii) Explain the various errors encountered in different phases of compiler.(8) [Pg.No :16]

12.(a) (i) Discuss the role of lexical analyzer in detail with necessary examples. (8) [Pg.No :23]

(ii) Disuss how FA is used to represent tokens and perform lexical analysis with examples. (8) [Pg.No :27]

OR

(b) (i) Conversion of regular expression (a/b)* abb to NFA. (8) [Pg.No :34]

(ii) Write an algorithm for minimizing the number of states of a DFA. (8) [Pg.No :35]

13.(a) (i) Construct parse tree for the input w= cad using top down parser.(6)

S->cAd [Pg.No :50]
A->ab| a

(ii) Construct parsing table for the grammar and find moves made by predictive parser on input id+id*id and find FIRST and FOLLOW. (10) [Pg.No :55]

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

OR

(b) (i) Explain ambiguous grammar G : E->E+E|E*E|(E)|-E|id for the sentence id+id*id. (6) [Pg.No :44]

(ii) Construct SLR Parsing table for the following grammar:

G : E->E+T|T T-> T*F|F F->(E)| id. (10) [Pg.No :64]

14 . (a) (i) Construct a syntax directed definition scheme that takes strings of a's,b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern a(a/b * c +(a/b)*b. For example the translation of the input string ¯abbcabcababc‖ is ¯3‖.

4. Write a context free grammar that generate all strings of a's,b's and c's
5. Give the semantic attributes for the grammar symbols.

For each productions of the grammar a set of rules for evaluation o the semantic attributes. (8) [Pg.No :94]

(ii) Illustrate type checking with necessary diagram . (8) [Pg.No :83]

OR

(b) Explain the following with reapect to code generation phase. (16)

I. Input to code generator. [Pg.No :123]
II. Target program.
III. Memory management.
IV. Instruction Selection.
V. Registr allocation.
VI. Evaluation order.

15. (a) (i) Write an algorithm for constructing natural loop of a back edge. (8)

[Pg.No :128]

(ii) Explain any four issues that crop up when designing a code generator. (8)

[Pg.No :123]

OR

(b) Expalin global data flow analysis with necessary equations. (16)
[Pg.No :116]

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

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.

PART B — (5 × 16 = 80 marks)

11. (a) Prove that the following two regular expressions are equivalent by showing that the minimum state DFA's are same.

(i) (a / b)* (8)

(ii) (a* / b*)* (8)

Or

(b) (i) Describe the error recovery schemes in the lexical phase of a compiler (8)

(ii) Mention any four compiler construction tools with their benefits and drawbacks. (8)

12. (a) (i) Generate SLR Parsing Table for the following grammar (12)

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

And parse the sentence "bdc" and "dd".

(ii) Mention in detail any 4 issues in storage organization. (4)

Or

(b) (i) Write down the algorithm to eliminate left-recursion and left-factoring and apply both to the following grammar (8)

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

$T \rightarrow a \mid b \mid (E)$

(ii) Give a syntax-directed definition to differentiate expressions formed by applying the arithmetic operators + and * to the variable x and constants ; expression : $x * (3 * x + x * x)$. (8)

13. (a) For the given program fragment $A[i, j] = B[i, k]$ do the following :

(i) Draw the annotated parse tree with the translation scheme to convert to three address code (6)

(ii) Write the 3-address code (6)

(iii) Determine the address of $A[3,5]$ where, all are integer arrays with size of A as 10 × 10 and B as 10 × 10 with k =2 and the start index position of all arrays is at 1. (Assume the base addresses) (4)

Or

(b) (i) Apply Back-patching to generate intermediate code for the following input.

```
x : 2 + y ;
if x < y then x : = x + y ;
repeat
        y : = y * 2 ;
        while x > 10 do x : = x/2
until x < y
```

Write the semantic rule and derive the Parse tree for the given code. (12)

(ii) What is an Activation Record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations. (4)

14. (a) (i) Write the Code Generation Algorithm using Dynamic Programming and generate code for the statement x = a / (b – c) — s*(e + f). [Assume all instructions to be unit cost] (12)

(ii) What are the advantages of DAG representation? Give example. (4)

Or

(b) (i) Write the procedure to perform Register Allocation and Assignment with Graph Coloring. (8)

(ii) Construct DAG and optimal target code for the expression $x = ((a+b)/(b-c)) - (a+b)*(b-c)+f$. (8)

15. (a) Perform analysis of available expressions on the following code by converting into basic blocks and compute global common sub expression elimination

(i) i : = 0
(ii) a : = n _ 3
(iii) IF i < a THEN loop ELSE end
(iv) LABEL loop
(v) b : = i _ 4
(vi) c : = p + b
(vii) d : = M[c]
(viii) e : = d _ 2
(ix) f : = i _4
(x) g := p + f
(xi) M[g] : = e
(xii) i : = i +1
(xiii) a : = n _ 3
(xiv) IF i < a THEN loop ELSE end
(xv) LABEL end. (16)

Or

(b) (i) Explain Loop optimization in detail and apply it to the code in 15 (a). (10)

(ii) What are the optimization techniques applied on procedures calls? Explain with example (6)

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

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)

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.

———————

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.
