Unit-I Introduction To Compilers

Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools -Programming Language basics.

Part-A

1. Define Compiler (or) What does translator mean? (May/June,2006)

Compiler is a program that reads a program written in one language –the source language- and translates it into an equivalent program in another language- the target language. In this translation process, the complier reports to its user the presence of the errors in the source program.

2. What are the classifications of a compiler?

The classifications of compiler are: Single-pass compiler, Multi-pass compiler, Load and go compiler, Debugging compiler, Optimizing compiler.

3. Define linker.

Linker is a program that combines (multiple) objects files to make an executable. It converts names of variables and functions to numbers (machine addresses).

4. Define Loader.

Loader is a program that performs the functions of loading and linkage editing. The process of loading consists of taking re-locatable machine code, altering the re-locatable address and placing the altered instruction and data in memory at the proper location.

5. Define interpreter. Interpreter is a language processor program that translates and executes source code directly, without compiling it to machine code.

6. Define editor.

Editors may operate on plain text, or they may be wired into the rest of the complier, highlighting syntax errors as you go, or allowing you to insert or delete entire syntax constructed at a time.

7.What is meant by semantic analysis?

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.

8.What are the phases of a compiler? or How will you group the phases of compiler ?(Nov/Dec ,2013)

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Code optimization
- Code generation

9.Mention few cousins of the compiler?(May/June,2012)

- Pre-processors.
- Assemblers.
- Two pass assembly.
- Loader and link editors.

10.What is front-end and back-end of the compiler?

Often the phases of a compiler are collected into a front-end and back-end.

Front-end consists of those phases that depend primarily on the source language and largely independent of the target machine. **Back-end** consists of those phases that depend on the target machine language and generally those portions do not depend on the source language, just the intermediate language. In back end we use aspects of code optimization, code generation, along with error handling and symbol table operations.

11.List of some compiler construction tools.

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data flow engines.

12.Define Pre-processor. What are its functions?

Pre-processors are the programs that allow user to use macros in the sources program. Macro means some set of instructions which can be repeatedly used in the sources program . The output of pre-processor may be given as input to complier.

The functions of a pre-processor are:

- Macro processing.
- File inclusion.
- Rational pre-processors
- Language extensions

13.What is linear analysis?

The stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequence of characters having a collective meaning.

14.What is cross complier? (May/June, 2014)

There may be a complier which run on one machine and produces the target code for another machine. Such a complier is called cross complier. Thus by using cross complier technique platform independency can be achieved.

15.What is Symbol Table or Write the purpose of symbol table.(May/June, 2014)

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

16.Define Passes.

In an implementation of a compiler, portion of one or more phases are combined into a module called pass. A pass reads the source program or the output of the previous pass, makes the transformation specified by its phases and writes output into an intermediate file, which is read by subsequent pass.

17. Difference Between Assembler, Compiler and Interpreter.

Assembler: It is the Computer Program which takes the Computer Instructions and converts them in to the bits that the Computer can understand and performs by certain Operations.

Compiler:

- It Converts High Level Language to Low Level Language.
- It considers the entire code and converts it in to the Executable Code and runs the code.
- C, C++ are Compiler based Languages.

Interpreter:

- It Converts the higher Level language to low level language or Assembly level language. It converts to the language of 0' and 1's.
- It considers single line of code and converts it to the binary language and runs the code on the machine.
- It it founds the error, the programs need to run from the beginning.
- BASIC is the Interpreter based Language.

18.State some software tools that manipulate source program?(May/June ,2006)

i. Structure editors
ii. Pretty printers
iii. Static checkers
iv. Interpreters.

19. Define compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages , and they are suitable for generating compilers of languages similar model.

20.Draw the diagram of the language processing system.(Nov/Dec,2006)

21.Difference between phase and pass.

Phase:

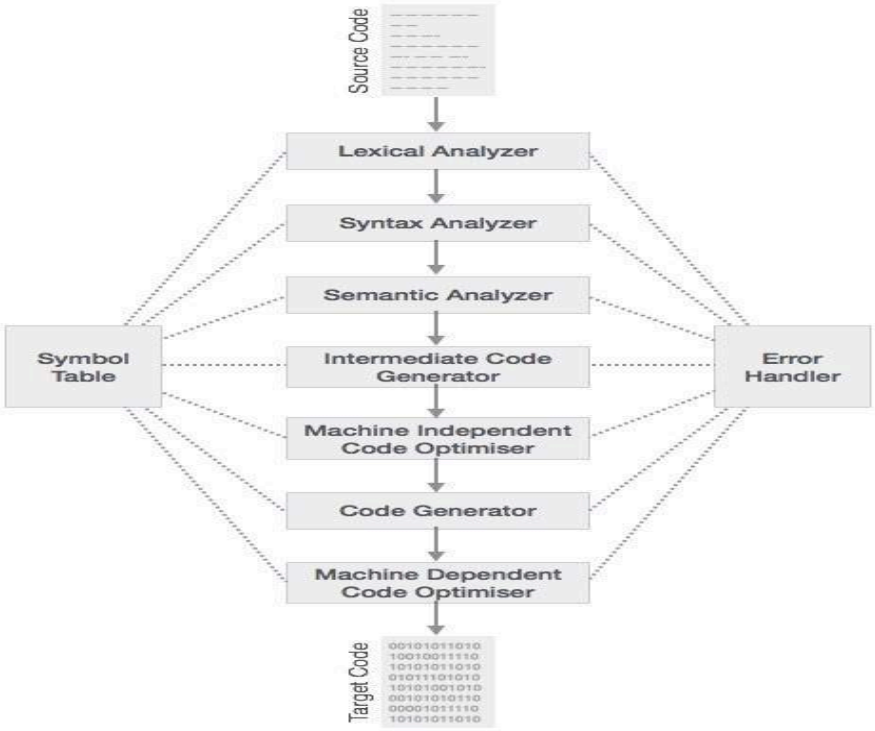
The compilation process into a series of sub processes are called as phase

Pass:

The collection of phase are called as pass.

PART-B

1.What is a compiler? State various phases of a compiler and explain them in detail.(16)



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or

not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

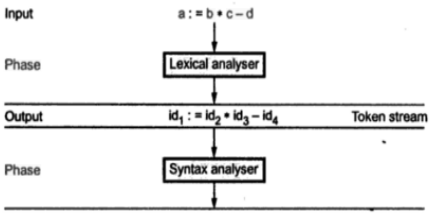
Code Generation

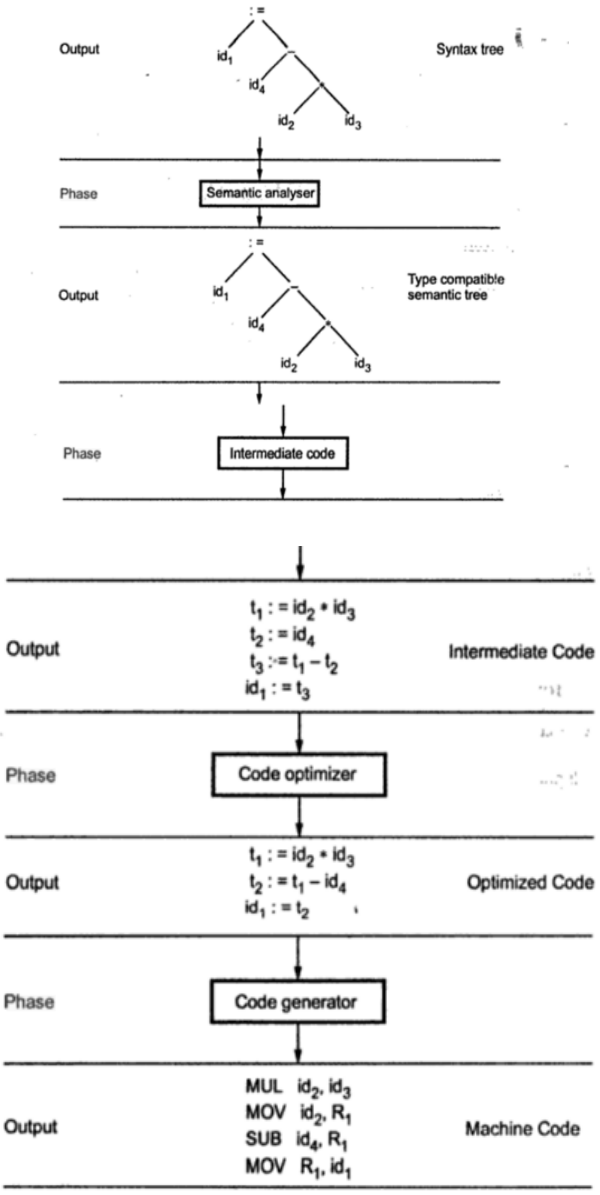
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

2.Explain the various phases of a compiler in detail. Also write down the output for the following expression after each phase a:= b*c-d.(16)





3.Write in detail about the analysis cousins of the compiler.(16) (May/June-2013)

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. The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

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 (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure. The mapping processes that instantiates (transforms) a macro into a specific output sequence is known as *macro expansion*. macro definitions (#define, #undef)To define preprocessor macros we can use #define. Its format is:

#define identifier replacement
int table2[100];

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. There are two ways to specify a file to be included:

#include "file"
#include <file>

3."Rational Preprocessors:
These processors augment older languages with more modern flow of control and data structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements,where none exist in the programming language itself.

4.Language extension :
These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language equal is a database query language embedded in C. Statements begging with ## are taken by the preprocessor to be database access statements unrelated to C and are translated into procedure calls on routines that perform the database access.

ASSEMBLER
Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution-e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines, or even generate entire programs or program suites.
There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- **One-pass assemblers** go through the source code once and assumes 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, then use the table in a second pass to generate code. The assembler must at least be able to determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.

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

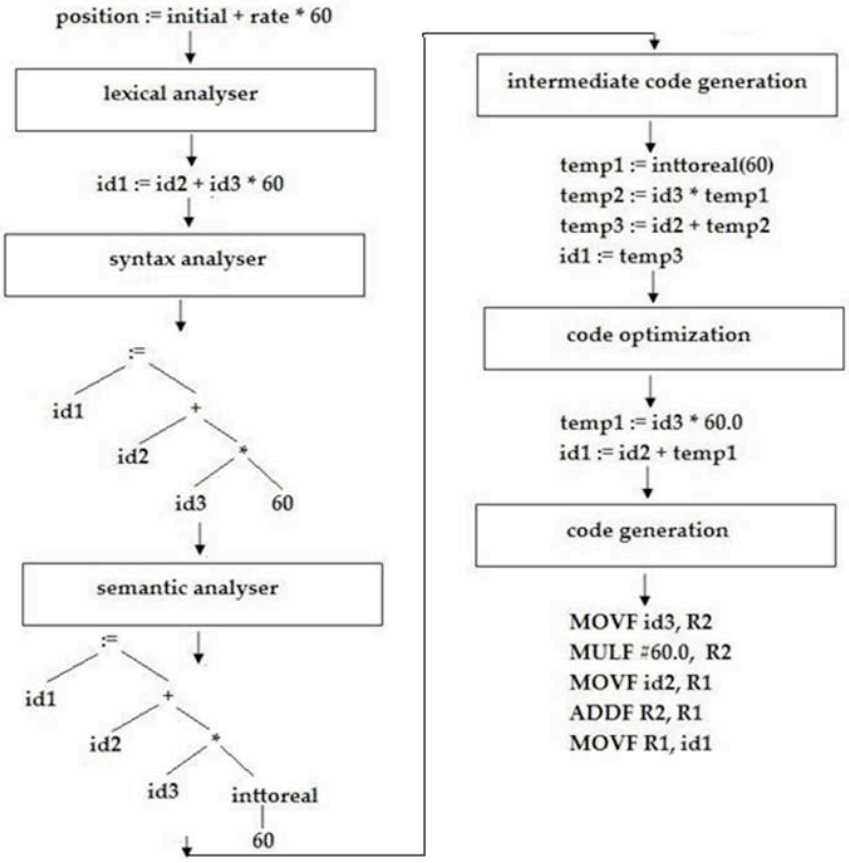
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files Loader

A **loader** is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program. Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

4.Describe how various phases could be combined as a pass in a compiler? April/May 2008
Also briefly explain Compiler construction tools . (April/May 2015)

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- Code Optimizer
- Code Generator

5.For the following expression
Position:=initial+ rate*60
Write down the output after each phase



6. Describe the following software tools i. Structure Editors ii. Pretty printers iii. Interpreters (8)

1. ***Structure Editors***: takes input a sequence of commands to build a source program.
   - Performs text creation and modification, analyzes the program text – hierarchical structure (check the i/p is correctly formed).

2 . ***Pretty Printers***: analyzes the program and prints the structure of the program becomes clearly visible.

3. ***Interpreters***: Instead of producing a target program as a translation, it performs the operations implied by the source program.

7.Elaborate on grouping of phases in a compiler.

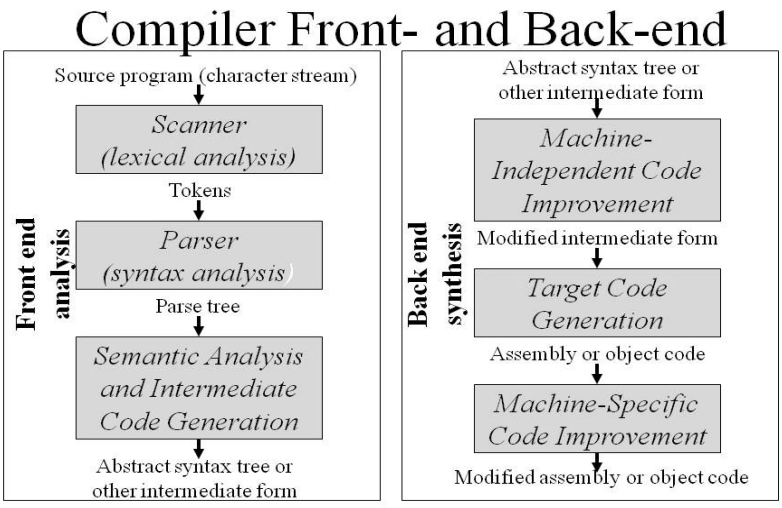
The Grouping of Phases:

- Phases deals with logical organisation of compiler.
- In an implementation, activities from more than one phase are often grouped together.
- Front and Back Ends:
  - The phases are collected into a front and a back end.
  - The front end consists of phases or part of phases that depends primarily on source language and is largely independent of the target machine.
    - These normally include lexical and syntactic analysis, the creation of symbol table, semantic analysis and intermediate code generation.
  - The back end includes portions of the compiler that depend on the target machine.
    - This includes part of code optimization and code generation.

8.What is difference between a phase and pass of a compiler? Explain machine dependent and machine independent phase of compiler.

phase and pass of a compiler

Phase and Pass are two terms used in the area of compilers. A pass is a single time the compiler passes over (goes through) the sources code or some other representation of it. Typically, most compilers have at least two phases called front end and back end, while they could be either one-pass or multi-pass. Phase is used to classify compilers according to the construction, while pass is used to classify compilers according to how they operate.



Unit-II lexical analysis

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

PART-A

1. What is Lexical Analysis?

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

2.What is a lexeme? Define a regular set. (Nov/Dec 2006)

- A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A language denoted by a regular expression is said to be a regular set

2.What is a sentinel?What is its usage? (April/May 2004)

A Sentinel is a special character that cannot be part of the source program. Normally we use 'eof' as the sentinel. This is used for speeding-up the lexical analyzer.

3.Mention the issues in a lexical analyzer.(May/June-2013)

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing.1.Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases. 2. Compiler efficiency is improved. 3. Compiler portability is enhanced.

4.What is the role of lexical analysis phase?

The lexical analyzer breaks a sentence into a sequence of words or tokens and ignores white spaces and comments. It generates a stream of tokens from the input. This is modeled through regular expressions and the structure is recognized through finite state automata.

5. What are the possible error recovery actions in a lexical analyzer?(May/june-2012),(April/May 2015)

i. Panic mode recovery.

Delete successive characters from the remaining input until the lexical analyzer can find a Well-formed token. This technique may confuse the parser.

ii. Other possible error recovery actions:

- Deleting an extraneous characters
- Inserting missing characters.
- Replacing an incorrect character by a correct character.
- Transposing two adjacent characters

6.Define Tokens, Patterns and Lexemes (or) Define Lexeme.(May/June 2013&2014)

Tokens: A token is a syntactic category. Sentences consist of a string of tokens. For example number, identifier, keyword, string etc are tokens

Lexemes: Lexeme: Sequence of characters in a token is a lexeme. For example 100.01, counter, const, "How are you?" etc are lexemes.

Patterns: Rule of description is a pattern. For example letter (letter | digit)* is a pattern to symbolize a set of strings which consist of a letter followed by a letter or digit.

7. Write short note on input buffering. (or).Why is buffering used in lexical analysis ? What are the commonly used buffering methods?(May/June 2014)

Lexical analyzer scan the sources program line by line. For storing the input string, which is to be read, the lexical analyzer makes use of input buffer. The lexical analyzer maintains two pointer forward and backward pointer for scanning the input string. There are two types of input buffering schemes- one buffer scheme and two buffer scheme.

8.What are the drawbacks of using buffer pairs?

i. This buffering scheme works quite well most of the time but with it amount of look ahead is limited.

ii. Limited look ahead makes it impossible to recognize tokens in situations Where the distance, forward pointer must travel is more than the length of buffer.

9.Define regular expressions.(or) Write a regular expression for an identifier.

Regular expression is used to define precisely the statements and expressions in the source language. For e.g. in Pascal the identifiers is denotes in the form of regular expression as letter **letter(letter|digit)*.**

10. What are algebraic properties of regular expressions?

The algebraic law obeyed by regular expressions are called algebraic properties of regular expression. The algebraic properties are used to check equivalence of two regular expressions.

S.No	Properties	Meaning
1	r1|r2=r2|r1	| is commutative
2	r1|(r1|r3)=(r1|r2)|r3	| is associative
3	(r1 r2)r3=r1(r2 r3)	Concatenation is associative
4	r1(r2|r3)=r1 r2|r1 r3 (r2|r3) r1=r2 r1|r3 r1	Concatenation is distributive over |
5	Є r = r є = r	Є is identity
6	r*=(r|є)*	Relation between є and *
7	r**=r*	*is idempotent

11. What is meant by recognizer?

It is a part of LEX analyzer that identifies the presence of a token on the input is a recognizer for the language defining that token. It is the program, which automatically recognizes the tokens. A recognizer for a language L is a program that takes an input string "x" and response "yes" if "x" is sentence of L and "no" otherwise.

12. List the operations on languages.

- **Union -** L U M ={s | s is in L or s is in M}
- **Concatenation** – LM ={st | s is in L and t is in M}
- **Kleene Closure –** L* (zero or more concatenations of L)
- **Positive Closure –** L+ (one or more concatenations of L)

13. Mention the various notational shorthands for representing regular expressions.

- One or more instances (+)
- Zero or one instance (?)
- Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a | b | c.)
- Non regular sets

14. List the rules for constructing regular expressions?

The rules are divided into two major classifications(i) Basic rules (ii) Induction rules

Basic rules:

i) **ε** is a regular expression that denotes {**ε**} (i.e.) the language contains only an empty string.

ii) For each a in ∑, then a is a regular expression denoting {a} , the language with only one string , which consists of a single symbol a.

Induction rules:

i) If R and S are regular expressions denoting languages LR and LS respectively then,

ii)(R)/(S) is a regular expression denoting LR U LS

iii)(R).(S) is a regular expression denoting LR . LS

iv)(R)* is a regular expression denoting LR*.

15. Define DFA.

A DFA is an acceptor for which any state and input character has utmost one transition state that the acceptor changes to. If no transition state is specified the input string is rejected.

16 . What is DFA ?

(i) It has no e-transitions.

(ii)For each state "S" and input symbol 'a', there is at most one edge labeled 'a' leaving from "S"

(iii) DFA is easy to simulate.

17. What are the conditions to satisfy for NFA?

(i) NFA should have one start state.

(ii) NFA may have one or more accepting states.

(iii) ε-Transitions may present on NFA.

(iv) There can be more than transitions, on same input symbol from any one state.

(v) In NFA, from any state "S"

A. There can be at most 2 outgoing e-transitions.

B.There can be only one transition on real input symbol.

C.On real input transition it should reach the new state only.

18. Write a short note on LEX.

A LEX source program is a specification of lexical analyzer consisting of set of regular expressions together with an action for each regular expression. The action is a piece of code, which is to be executed whenever a token specified by the corresponding regular expression is recognized. The output of a LEX is a lexical analyzer program constructed from the LEX source specification.

19. Define ϵ -closure?

ϵ -Closure is a set of states of an NFA with e-transitions, such that e -closure for some state includes all states attainable from that state by making state transitions with e -transitions.

This is denoted by e-closure (state).

20. How to recognize tokens

Consider the following grammar and try to construct an analyzer that will return <token, attribute> pairs.

relop < | = | = | <> | = | >

id letter (letter | digit)*

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

delim blank | tab | newline

ws delim+

21. What is finite automate ?

A finite automate is a mathematical model that consists of-

i)Input set Σ

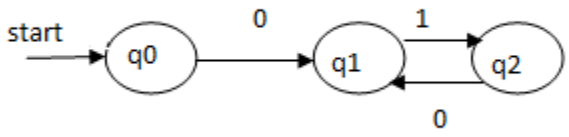
ii)A finite set of states S

iii)A transition function to move from one state to other.

iv)A special state called start state.

v)A special state called accept state.

The finite automate can be diagrammatically represented by a directed graph called transition graph.



22. **What is the need for separating the analysis phase into lexical analysis and parsing?**

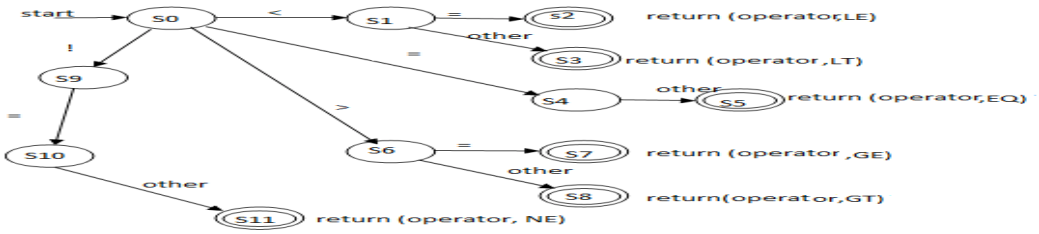
i)Separation of lexical analysis from syntax allows the simplified design.

ii)Efficiency of complier gets increased. Reading of input source file is a time consuming process and if it is been done once in lexical analyzer then efficiency get increased . Lexical analyzer uses buffering techniques to improve the performances.

iii)Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer.

23.Draw a transition diagram to represent relational operators.

The relational operators are:<,<=,>,>=,=,!=



24. State any reasons as to why phases of compiler should be grouped.(May/June 2014)

1.By keeping the same front end and attaching different back ends one can produce a compiler for dame source language on different machine.

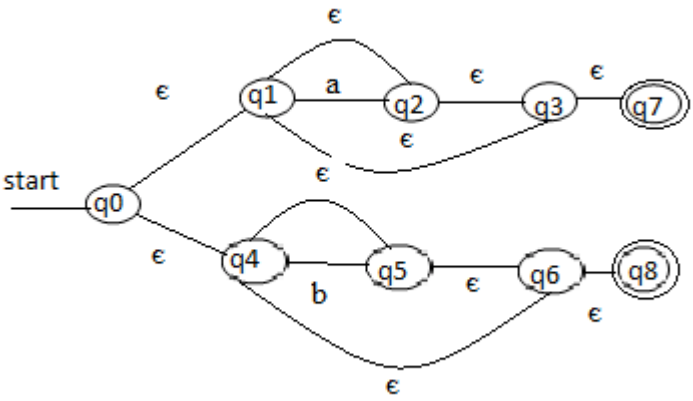
2.By keeping different fronts ends and same back end one can compile several different language on the same machine.

25. Define a context free grammar.

A context free grammar G is a collection of the following

- V is a set of non terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules
- G can be represented as G = (V,T,S,P)
- Production rules are given in the following form
- Non terminal → (V U T)*

26.Draw a NFA for a*|b*.(April/May-2004,Nov/Dec-2005)



27.Difference between Deterministic FA and Non Deterministic FA.

S.No	NFA	DFA
1.	E-Transition is present	No E-Transition
2.	More than one transition for real variable	Only one transition for real symbol
3.	It's not easy implemented	It's easily implemented
4.	Compare to DFA not fast	Can load to Faster recognizer.

28.What is the time and space complexity of NFA and DFA.

Automation	Space	Time
NFA	O(|r|)	O(|r|*|r|)
DFA	O(2|r|)	O(|x|)

29.Write a regular definition to represent date and time in the following format: MONTH DAY YEAR.(April/May-2015)

MONTH--> [Jan-Dec]
DAY-->[1-31]
YEAR-->[1900-2025]

PART-B

1. Explain in detail about lexical analyzer generator.

Hints:

Definition

- Lexical analyzers tokenize input streams
- Tokens are the terminals of a language
  - English
    - words, punctuation marks, …
  - Programming language
    - Identifiers, operators, keywords, …
- Regular expressions define terminals/tokens

Typical program style

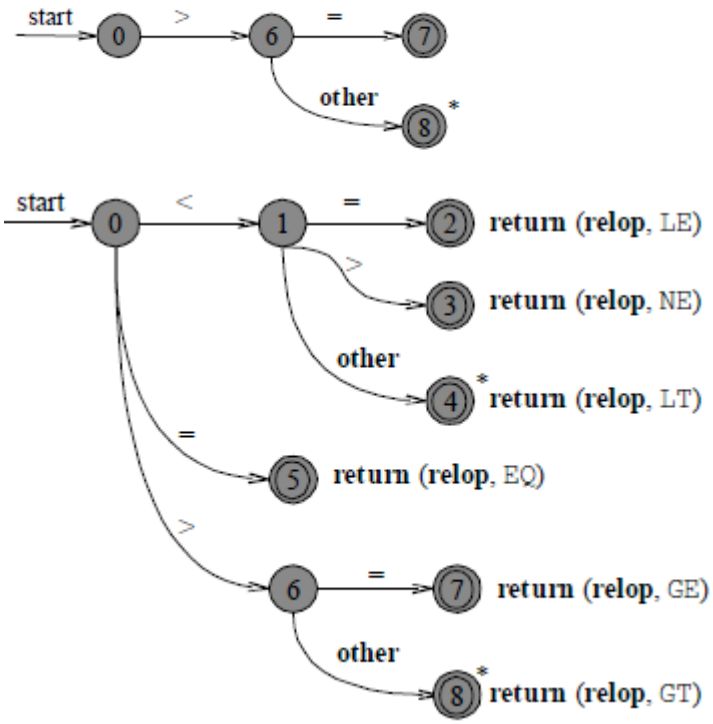
- A program consists of one or more functions; it may also contain global variables.
- At the top of a source file are typically a few boilerplate lines such as `#include <stdio.h>`, followed by the definitions (i.e. code) for the functions.
- Each function is further composed of *declarations* and *statements*, in that order.
- When a sequence of statements should act as one they can be enclosed in braces.
- The simplest kind of statement is an *expression statement*, which is an expression followed by a semicolon. Expressions are further composed of *operators*, *objects* (variables), and *constants*.
- C source code consists of several *lexical elements*. Some are words, such as `for`, `return`, `main`, and `i`, which are either *keywords* of the language (`for`, `return`) or *identifiers* (names) we've chosen for our own functions and variables (`main`, `i`).
- There are *constants* such as `1` and `10` which introduce new values into the program. There are *operators* such as `=`, `+`, and `>`, which manipulate variables and values.
- There are other punctuation characters (often called *delimiters*), such as parentheses and squiggly braces `{}`, which indicate how the other elements of the program are grouped.
- Finally, all of the preceding elements can be separated by *whitespace*: spaces, tabs, and the ``carriage returns'' between lines.

Transition diagram.

- A useful intermediate step between r.e.'s and lexer
- Shows actions taken by lexer
- Move from position to position as characters

are read, advancing input (lookahead) pointer
• A graph in which nodes are states of lexer,
edges are input characters
• Special states:
– Start : no input
– Accept : done / success

Example: scanning >= or >



2. .Explain about input buffering.(8)

Input Buffering:

- Some efficiency issues concerned with the buffering of input.
- A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.
- Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- There are three general approaches to the implementation of a lexical analyser:
  1. Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
  2. Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input.
  3. Write the lexical analyser in assembly language and explicitly manage the reading of input.

***Buffer pairs*:**

- Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. The scheme to be discussed:

- Consists a buffer divided into two N-character halves.

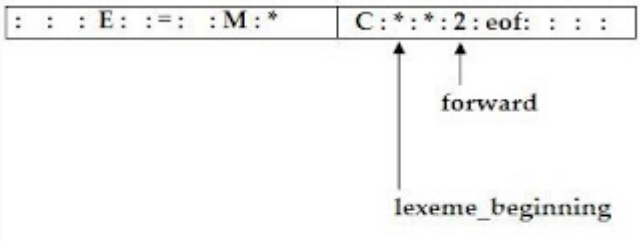


Fig 1.8 An input buffer in two halves

- N – Number of characters on one disk block, e.g., 1024 or 4096.
  - Read N characters into each half of the buffer with one system read command.
  - If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.
  - Two pointers to the input buffer are maintained.
  - The string of characters between two pointers is the current lexeme.
  - Initially both pointers point to the first character of the next lexeme to be found.
  - Forward pointer, scans ahead until a match for a pattern is found.
  - Once the next lexeme is determined, the forward pointer is set to the character at its right end.
  - If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.
  - If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

***Disadvantage of this scheme*:**

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
- For example: DECLARE (ARG1, ARG2, … , ARGn) in PL/1 program;
- Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

***Sentinels*:**

- In the previous scheme, must check each time the move forward pointer that have not moved off one half of the buffer. If it is done, then must reload the other half.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
- This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).

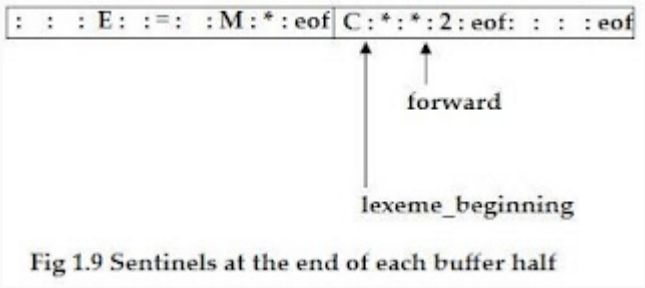
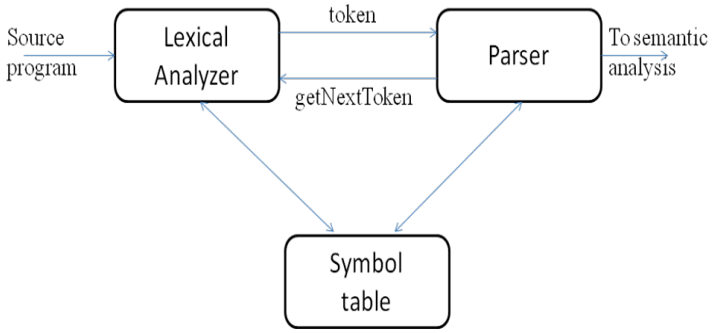


Fig 1.9 Sentinels at the end of each buffer half

- In this, most of the time it performs only one test to see whether forward points to an eof.
- Only when it reach the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eof's, the average number of tests per input character is very close to 1.

3.Explain in detail about the role of Lexical analyzer with the possible error recovery actions. (May/June-2013) (16)



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

Its secondary tasks are,

- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

1) Scanning
2) lexical analysis.

The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

4.Describe the specification of tokens and how to recognize the tokens (16) (May/June-2013)

Specification of Tokens

An alphabet or a character class is a finite set of symbols. Typical examples of symbols are letters and characters.

The set {0, 1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.

Strings

A string over some alphabet is a finite sequence of symbol taken from that alphabet.

For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII

computer alphabet. The empty string denoted by ∈, is a special string with zero symbols (i.e., string length is 0).
If x and y are two strings, then the concatenation of x and y, written xy, is the string formed by appending y to x.
For example, If x = dog and y = house, then xy = doghouse. For empty string, ∈, we have S∈ = ∈ S = S.
String exponentiation concatenates a string with itself a given number of times:
S^2 = SS or S.S
S^3 = SSS or S.S.S
S^4 = SSSS or S.S.S.S and so on
By definition S^0 is an empty string, ∈, and S` = S. For example, if x =ba and na then xy^2 = banana.

Languages

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.
Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then

- Union: LUM = {dog, ba, na, house}
- Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}
- Expontentiation: L2 = LL
- By definition: L^0 ={∈} and L` = L

The kleene closure of language L, denoted by L*, is "zero or more Concatenation of" L.
$L^* = L^0 \cup L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$
For example, If L = {a, b}, then
L* = {∈, a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }
The positive closure of Language L, denoted by L^+, is "one or more Concatenation of" L.
$L^+ = L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$
For example, If L = {a, b}, then
L^+ = {a, b, aa, ba, bb, aaa, aba, . . . }

Recognize the tokens

Finite Automata

- A recognizer for a language is a program that takes a string x as an input and answers "yes" if x is a sentence of the language and "no" otherwise.
- One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.
- A finite automation can be deterministic means that more than one transition out of a state may be possible on a same input symbol.
- Both automata are capable of recognizing what regular expression can denote.

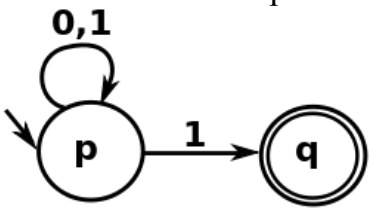
Nondeterministic Finite Automata (NFA)

A nondeterministic finite automaton is a mathematical model consists of

1. a set of states S;
2. a set of input symbol, ∑, called the input symbols alphabet.
3. a transition function move that maps state-symbol pairs to sets of states.
4. a state so called the initial or the start state.

5. a set of states F called the accepting or final state.

- An NFA can be described by a transition graph (labeled graph) where the nodes are states and the edges shows the transition function.
- The labeled on each edge is either a symbol in the set of alphabet, ∑, or denoting empty string.
- This automation is nondeterministic because when it is in state-0 and the input symbol is a, it can either go to state-1 or stay in state-0.
- The advantage of transition table is that it provides fast access to the transitions of states and the disadvantage is that it can take up a lot of soace.
- In general, more than one sequence of moves can lead to an accepting state. If at least one such move ended up in a final state. For instance
- The language defined by an NFA is the set of input strings that particular NFA accepts.

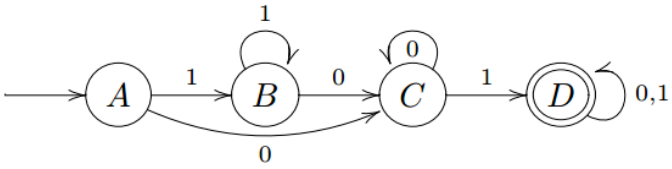


Deterministic Finite Automata (DFA)

A deterministic finite automation is a special case of a non-deterministic finite automation (NFA) in which

1. no state has an ∈-transition
2. for each state s and input symbol a, there is at most one edge labeled a leaving s.

- A DFA has st most one transition from each state on any input. It means that each entry on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA).
- Because of single transition attached to each state, it is vary to determine whether a DFA accepts a given input string.



5.Describe the language for specifying lexical Analyzer.(16)

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

Lex

- The main job of a lexical analyzer (scanner) is to break up an input stream into more usable elements (tokens)
  a = b + c * d;
  ID ASSIGN ID PLUS ID MULT ID SEMI
- Lex is an utility to help you rapidly generate your scanners
- Lexical analyzers tokenize input streams
- Tokens are the terminals of a language
  - English
    - words, punctuation marks, …
  - Programming language

- Identifiers, operators, keywords, …

- Regular expressions define terminals/tokens
- Lex source is a table of
  - regular expressions and
  - corresponding program fragments

```
digit    [0-9]
letter   [a-zA-Z]
%%
{letter}({letter}|{digit})*      printf("id: %s\n", yytext);
\n                               printf("new line\n");
%%
main() {
        yylex();
}
```

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is
- The absolute minimum Lex program is thus

```
{definitions}
%%
{transition rules}
%%
{user subroutines}
```

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is stdin
- yyout -- the output stream pointer
  - the default output of default main() is stdout.
- E.g.

```
[a-z]+          printf("%s", yytext);
[a-z]+          ECHO;
[a-zA-Z]+       {words++; chars += yyleng;}
```

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

6.prove that the following two regular expressions are equivalent by showing that minimum state DFA's are same.
(i)(a|b)*#
(ii)(a*|b*)*#

7.Describe the error recovery schemes in the lexical phase of a compiler.(8) (April/May 2015)

Panic mode

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

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

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

Global correction

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

8.Write short note on token specification.(10) Nov/Dec 2005.

Regular expressions are notation for specifying patterns.

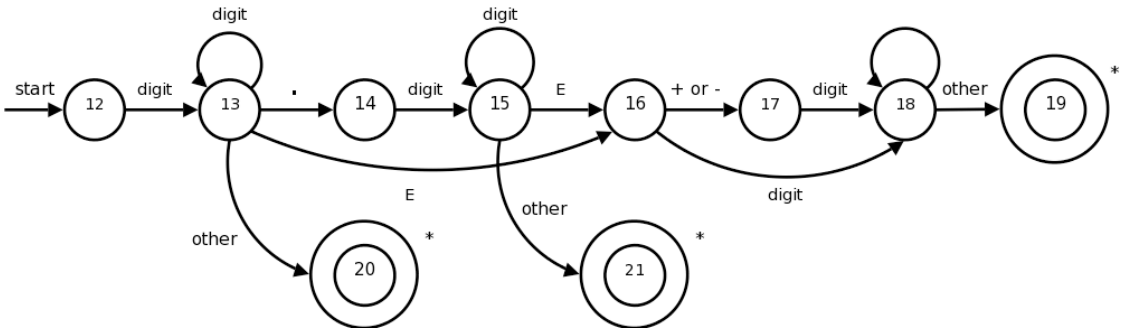
- Each *pattern* matches a set of strings.
- *Regular expressions* will serve as names for sets of strings.
- *Strings and Languages*:
- The term *alphabet or character class* denotes any finite set of symbols.
- e.g., set {0,1} is the binary alphabet.
- The term *sentence and word* are often used as synonyms for the term string.
- The *length of a string s* is written as |s| - is the number of occurrences of symbols in *s*.
- e.g., string “banana” is of length six.
- The *empty string* denoted by ε – length of empty string is zero.
- The term *language* denotes any set of strings over some fixed alphabet.
- e.g., {ε} – set containing only empty string is language under φ.
- If *x* and *y* are strings, then the *concatenation* of *x* and *y* (written as *xy*) is the string formed by appending *y* to *x*. *x* = *dog* and *y* = *house*; then *xy* is *doghouse*.
- $s\varepsilon = \varepsilon s = s$.

- $s^0 = \varepsilon$, $s^1 = s$, $s^2 = ss$, $s^3 = sss$, … so on.

***Regular Expressions*:**

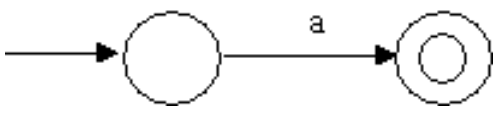
- It allows defining the sets to form tokens precisely.
- e.g., *letter (letter | digit) **
- Defines a Pascal identifier – which says that the identifier is formed by a letter followed by zero or more letters or digits.
- A regular expression is built up out of simpler regular expressions using a set of defining rules.
- Each regular expression r denotes a language L(r).
  - The rules that define the regular expressions over alphabet ∑.
- (Associated with each rule is a specification of the language denoted by the regular expression being defined)
  - 1. *ε* is a regular expression that denotes {ε}, i.e. the set containing the empty string.
  - 2. If *a* is a symbol in ∑, then *a* is a regular expression that denotes {a}, i.e. the set containing the string *a*.
  - 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 languages L(r) U L(s).
- b) (r)(s) is a regular expression denoting the languages L(r)L(s).
- c) (r)* is a regular expression denoting the languages (L(r))*.
- d) (r) is a regular expression denoting the languages L(r).
  - A language denoted by a regular expression is said to be a regular set.
  - The specification of a regular expression is an example of a recursive definition.
  - *Rule (1) and (2)* form the basis of the definition.
  - *Rule (3)* provides the inductive step.

9.Draw the transition diagram for unsigned numbers.(6) Nov/Dec,2006,2007

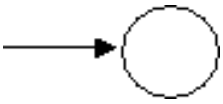


10.construct the NFA from the (a/b)*a(a/b) using Thompson's construction algorithm.(10) May/June 2007

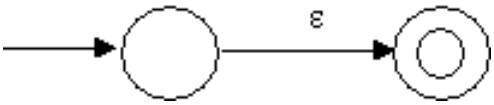
(1) Any letter a of the alphabet is recognized by:



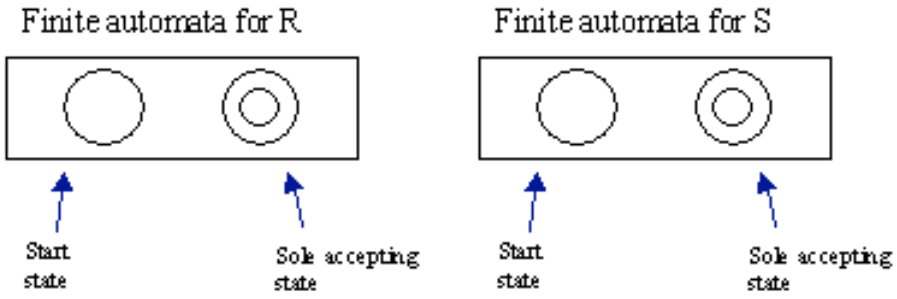
(2) The empty set Æ is recognized by:



(3) The empty string e is recognized by:



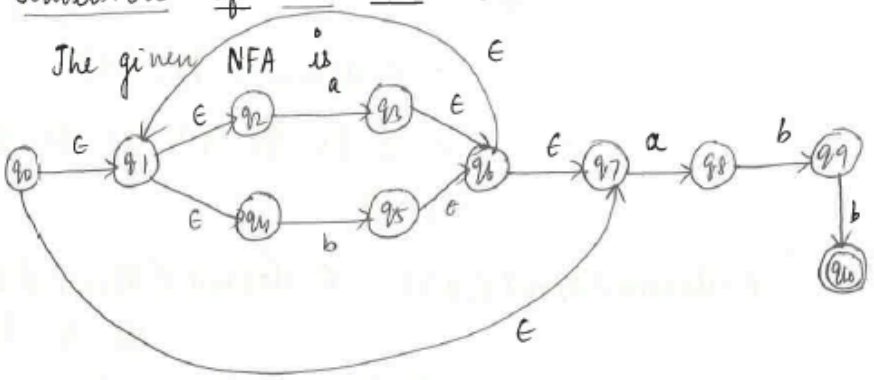
(4) Given a regular expression for R and S, assume these boxes represent the finite automata for R and S:



11.Give the minimized DFA for the following expression.(10) Nov/Dec,2006,2007
(a/b)*abb

Conversion of NFA into DFA :-

The given NFA is



ε - closure (Initial state) = ε- closure (q_0) =

$= \{ q_0, q_1, q_2, q_4, q_7 \}$

$= A$

ε - closure(Mov(A, a)) = ε-closure (Mov($\{q_0, q_1, q_2, q_4, q_7\}$, a)

$= ε$-closure (q_3, q_8)

$= \{ q_3, q_6, q_7, q_1, q_2, q_4, q_8 \}$

$= B$

ε - closure(Mov (A, b)) = ε-closure(Mov ($\{q_0, q_1, q_2, q_4, q_7\}$, b))

$= ε$-closure (q_5)

$= \{ q_5, q_6, q_7, q_1, q_2, q_4 \}$

$= \{ q_1, q_2, q_4, q_5, q_6, q_7 \}$

$= C$

ε-closure(Mov (B, a)) = ε-closure (Mov($\{q_1, q_2, q_3, q_4, q_6, q_7, q_8\}$, a))

$= ε$-closure (q_3, q_8)

$= B$

ϵ-closure(Mov(B, b)) = ϵ-closure(Mov({$q_1, q_2, q_3, q_4, q_6, q_7, q_8$}, b))

= ϵ-closure(q_5, q_9)

= {$q_5, q_6, q_7, q_1, q_2, q_4, q_9$}

= D

ϵ-closure(Mov(C, a)) = ϵ-closure(Mov({$q_1, q_2, q_4, q_5, q_6, q_7$}, a))

= ϵ-closure(q_3, q_8)

= B

ϵ-closure(Mov(C, b)) = ϵ-closure(Mov({$q_1, q_2, q_4, q_5, q_6, q_7$}, b))

= ϵ-closure(q_5)

= C

ϵ-closure(Mov(D, a)) = ϵ-closure(Mov({$q_1, q_2, q_4, q_5, q_6, q_7, q_9$}, a))

= ϵ-closure(q_3, q_8)

= B

ϵ-closure(Mov(D, b)) = ϵ-closure(Mov({$q_1, q_2, q_4, q_5, q_6, q_7, q_9$}, b))

= ϵ-closure(q_5, q_{10})

= {$q_5, q_6, q_7, q_1, q_2, q_4, q_{10}$}

= E

ϵ-closure(Mov(E, a)) = ϵ-closure(Mov({$q_1, q_2, q_4, q_5, q_6, q_7, q_{10}$}, a)

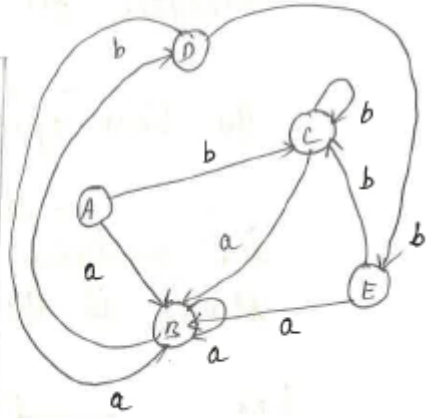
ε-closure(Mov(E, b)) = ε-closure(95)

= C

∴ The final DFA for the NFA is

Transistion table:

	I/p Symbols	
States	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C



Minimization of DFA:- [for the above Problem]:-

[A B C D E]

Step 1:- Divide the States into 2 groups.
Accepting state and non-accepting state

[A B C D] [E]

[A B C] [D] [E]

[A C] [B] [D] [E]

∴ A is replaced by 'C' / 'C' is replaced by 'A'

∴ After the minimization, the final DFA is

	I/p symbols	
states	a	b
A	B	A
B	B	D

12.Write LEX specifications and necessary C code that reads English words from a text file and response every occurrence of the sub string 'abc' with 'ABC'. The program should also compute number of characters, words and lines read. It should not consider and count any lines(s) that begin with a symbol '#'

```
%{
#include <stdio.h>
```

```
#include <stdlib.h>
int cno = 0, wno = 0, lno = 0; /*counts of characters, words and lines */
%}
character [a-z]
digit [0-9]
word ({character}|{digit})+[^({character}|{digit})]
line \n
%%
{line} { lno++; REJECT; }
{word} { wno++; REJECT; }
{character} { cno++; }
%%
void main()
{ yylex();
  fprintf(stderr, "Number of characters: %d; Number of words: %d; Number of lines: %d\n", cno, wno, lno);
  return;
}
```

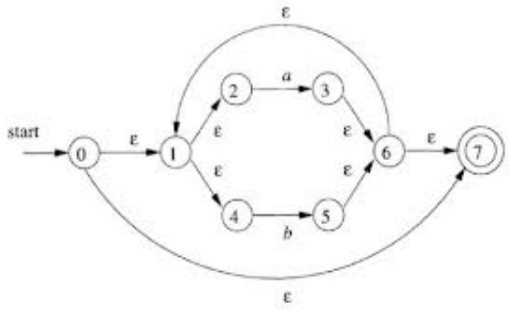
13.Write a algorithm to construct an NFA into a regular expression.(8) Nov/Dec 2010

We will use the rules which defined a regular expression as a basis for the construction:

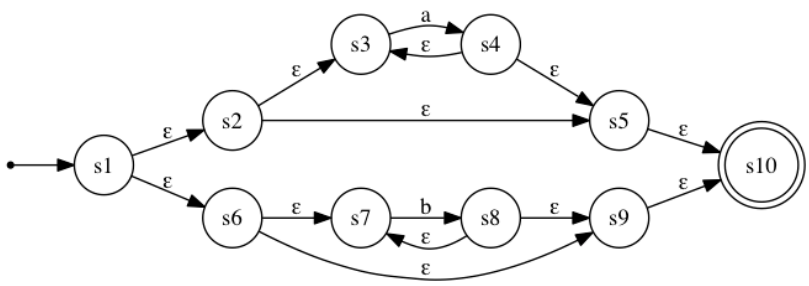
- The NFA representing the empty string
- If the regular expression is just a character, eg. `a`, then the corresponding NFA
- The union operator is represented by a choice of transitions from a node
- Concatenation simply involves connecting one NFA to the other
- The Kleene closure must allow for taking zero or more instances of the letter from the input

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

(i) (a/b)*



(ii)(a*/b*) (April/May 2015)



Input: DFA
Output: Minimized DFA
Step 1

- Draw a table for all pairs of states (Qi, Qj) not necessarily connected directly [All are unmarked initially]

Step 2

- Consider every state pair (Qi, Qj) in the DFA where Qi ∈ F and Qj ∉ F or vice versa and mark them. [Here F is the set of final states].

Step 3

- Repeat this step until we cannot mark anymore states −
- If there is an unmarked pair (Qi, Qj), mark it if the pair {δ(Qi, A), δ (Qi, A)} is marked for some input alphabet.

Step 4

- Combine all the unmarked pair (Qi, Qj) and make them a single state in the reduced DFA.

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.Define parser.(or) What is the role of passer ?(April/May 2015)

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

2.Mention the basic issues in parsing.

There are two important issues in parsing:

1) Specification of syntax.
2) Representation of input after parsing.

3. Why lexical and syntax analyzers are separated out?

The reason for separating the lexical and syntax analyzer.

- Simpler design.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

4. Define a context free grammar.

A context free grammar G is a collection of the following

- V is a set of non terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules
- G can be represented as G = (V,T,S,P)

5.Briefly explain the concept of derivation.

Derivation from S means generation of string w from S. For constructing derivation two things are important.

i. Choice of non-terminal from several others.
ii. Choice of rule from production rules for corresponding non terminal

Instead of choosing the arbitrary rules for corresponding non terminal

i. Either leftmost derivation-leftmost non terminal in a sentinel form
ii. Or rightmost derivation -rightmost non terminal in a sentinel form

6. Define ambiguous grammar.

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

7.What is operator precedence parser?

A grammar is said to be operator precedence if ti posses the following properties:

i. No production on the right side is ϵ
ii. There should not be any production rule possessing two adjacent terminals at the right hand side.

8. Define the rule to develop an unambiguous grammar from ambiguous grammar.

If S->αSβSγ | α1 ǀ α2 ǀ α3 ǀ α4 ǀ. ……ǀ αn is an ambiguous grammar than Unambiguous grammar becomes

S-> αSβS'γ | S'
S' ->α1 ǀ α2 ǀ α3 ǀ α4 ǀ. ……ǀ αn

9.What is meant by left recursion?

A grammar is left recursive if it has a non terminal A such that there is derivation $A =^{+}> A\alpha$ for some string α. Top down parsing methods cannot handle left-recursion grammars, so a transformation that eliminates left recursion in needed.

Ex:-

E → E +T | T
T → T * F | F
F → (E) | id

10.Write the algorithm to eliminate left recursion from a grammar?

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 →AjY by
the productions
$A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots\ldots | \delta_k\gamma$,
where $A_j \rightarrow \delta_1 | \delta_2 | \ldots\ldots | \delta_k$
are all the current A_j –productions;
end
eliminate the immediate left recursion among the Ai-- productions
end.

11.What is meant by left factoring?

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non terminal A, we may be able to rewrite the A production to defer the decision until we have seen enough of the input to make the right choice.

12.What is LR Parsers?

LR(k) parsers scan the input from (L) left to right and construct a (R) rightmost derivation in reverse. LR parsers consist of a driver routine and a parsing table. The k is for the number of input symbols of look ahead that are used in making parsing decisions. When k is omitted , k is assumed to be 1.

13. Mention the properties of parse tree?

- The root is labeled by the start symbol.

- Each leaf is labeled by a token.
- Each interior node is labeled by a non-terminal .
- If A is the Non terminal, labeling some interior node and x1, x2, x3 … .xn are the labels of the children

14. Define Handles.

A handle of a right-sentential form ϒ is a production A → β and a position of ϒ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of ϒ

15.What are the problems in top down parsing?

a) Left recursion.
b) Backtracking.
c) The order in which alternates are tried can affect the language accepted.

16.Define recursive-descent parser.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser. The recursive procedures can be quite easy to write.

17.Define predictive parsers.

A predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly. The predictive parser has an input, a stack , a parsing table and an output.

18.What is non recursive predictive parsing?

Non recursive predictive parser can be maintained by a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that determining the production to e applied for a non-terminal. The non recursive parser looks up the production to be applied in a parsing table.

19.Write the algorithm for FIRST? (Or) Define FIRST in predictive parsing.

1. If X is terminal, then FIRST(X) is {X}.
2. If X →Є is a production, then add Є to FIRST(X).
3. 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(Y_i) , and Є is in all of FIRST(Y_1),…FIRST(Y_i-1);

20.Write the algorithm for FOLLOW? (Or) Define FOLLOW in predictive parsing.

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production A → aBß, then everything in FIRST(ß) except for Є is placed in FOLLOW(B).
3. If there is a production A → aB, or a production A→aBß where FIRST(ß) contains Є, then everything in FOLLOW(A) is in FOLLOW(B).

21.Write the algorithm for the construction of a predictive parsing table?

Input : Grammar G
Output : Parsing table M
Method :

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

22.What is LL(1) grammar?

A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

23.Define handle pruning.

A technique to obtain the rightmost derivation in reverse (called canonical reduction sequence) is known as handle pruning (i.e.) starting with a string of terminals w to be parsed. If w is the sentence of the grammar then w=ϒn, where ϒn is the nth right sentential form of unknown right most derivation.

24.What is shift reduce parsing?

The bottom up style of parsing is called shift reduce parsing. This parsing method is bottom up because it attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

25.What are the four possible action of a shift reduce parser?

a) Shift action – the next input symbol is shifted to the top of the stack.
b) Reduce action – replace handle.
c) Accept action – successful completion of parsing.
d) Error action- find syntax error.

26.Define viable prefixes.

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

27.List down the conflicts during shift-reduce parsing.

- Shift/reduce conflict.-Parser cannot decide whether to shift or reduce.
- Reduce/reduce conflict-Parser cannot decide which of the several reductions to make.

28.Define LR (0) item.

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.
Eg: A→ .XYZ A→X.YZ A→XY.Z A→XYZ.

29.What is augmented grammar? (or) How will you change the given grammar to an augmented grammar?

If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production S'→ S. It is to indicate the parser when it should stop and announce acceptance of the input.

30.Left factor the following grammar:

S → iEtS | iEtSeS |a
E → b.

Ans:

The left factored grammar is,
S → iEtSS′ | a
S′ → eS | ε
E → b

31.Explain the need of augmentation.

To indicate to the parser that, when it should stop parsing and when to announce acceptance of the input. Acceptance occurs when the parser is about to reduce by S'→S.

32.What are the rules for "Closure operation" in SLR parsing?

If 'I' is a set of items for grammar G then Closure (I) is the set of items constructed from I by the following 2 rules.

(i) Initially every item in I is added to Closure (I)
(ii) If A□α .Bβ is in Closure(I) and B □ α to I , then add the item B □ .α to I, if it is not already there. Apply this until no more new items can be added to Closure (I).

33.How to make an ACTION and GOTO entry in SLR parsing table?

i. If [A→α.aβ] is in Ii and goto (I_i, a)=Ij then set ACTION[I, a] to =shift j. Here _a' must be a terminal.
ii. If [A→a.] is in I_i then set ACTION [I, a] to "reduce AA→a" for all a in FOLLOW (A), here A should not be S'.
iii. If S'→S is in Ii then set ACTION[I, $] to "accept" .

34. Explain the rules for GOTO operation in LR parsing.

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

35. What are the rules to apply Closure function in CLR parser?

Let 'L' is a set of LR (1) items for grammar then 2 steps can compute Closure of I.

i.Initially every item in I is added to Closure (I).
ii. Consider,
A □ X. BY, a
B □ Z
are 2 productions and X, Y,Z are grammar symbols. Then add B □ .Z ,FIRST(Ya) as the new

LR(1) item ,if it is not already there. This rule has to be applied till no new items can be added to Closure (I).

36. Find the item I_0 for the following grammar using CLR parsing method.

G: S ☐ AS
S ☐ b
A ☐ SA
A ☐ a

I_0:

S ′ ☐. S, $
S ☐ .AS , $
S ☐ .b , $
A ☐ .SA , alb
A ☐ .a , alb
S ☐ .AS , alb
S ☐ .b , alb

37. Specify the advantages of LALR.

Merging of states with common cores can never produce a shift/reduce conflict that was not present in any one of the original states. Because shift actions depends only one core, not the look ahead.

38. Mention the demerits of LALR parser.

- Merger will produce reduce / reduce conflict.
- On erroneous input, LALR parser may proceed to do some reductions after the LR parser has declared an error, but LALR parser never shift a symbol after the LR parser declares an error.

39.What is the syntax for YACC source specification program?

Declarations
%%
Translation rules
%%
Supporting C-routines

40.Define terminal.

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

41. Define Nonterminal

Nonterminal are syntactic variables that denote set of strings.

42.What are the possible actions of a shift reduce parser?

- Shift.
- Reduce.
- Accept.
- Error.

43.List down the conflicts during shift-reduce parsing.

Shift/reduce conflict.

- Parser cannot decide whether to shift or reduce.

Reduce/reduce conflict.

- Parser cannot decide which of the several reductions to make.

44. Differentiate Kernel and non-Kernel items.

Kernel items, which include the initial item, S' -> **.**S and all items whose dots are not at the left end. Whereas the nonkernel items have their dots at the left end.

45. What are the components of LR parser?

- An input.
- An output.
- A stack.
- A driver program.
- A parsing table.

46. List the different techniques to construct an LR parsing table?

- Simple LR(SLR).
- Canonical LR.
- Lookahead LR (LALR).

47. Elininate left recursion from the following grammar A→ Ac/Aad/bd/ϵ . (May/June 2013)

A→bdA'
A'→ aAcA' / adA' /ϵ

48.Write the regular expression for identifier and whitespace.(Nov/Dec 2013)

An Identifier (or Name)

/[a-zA-Z_][0-9a-zA-Z_]*/

1. Begin with one letters or underscore, followed by zero or more digits, letters and underscore.
2. You can use meta character \w (word character) for [a-zA-Z0-9_]; \d (digit) for [0-9]. Hence, it can be written as /[a-zA-Z_]\w*/.
3. To include dash (-) in the identifier, use /[a-zA-Z_][\w-]*/. Nonetheless, dash conflicts with subtraction and is often excluded from identifier.

An Whitespace

/\s\s/ # Matches two whitespaces
/\S\S\s/ # Two non-whitespaces followed by a whitespace
/\s+/ # one or more whitespaces

/\S+\s\S+/ # two words (non-whitespaces) separated by a whitespace

49. Eliminate the left recursion for the grammar (Nov/Dec 2013)

S→Aa|b
A→Ac|Sd|ϵ

Sol:

Let's use the ordering S, A (S = A1, A = A2).
• When i = 1, we skip the "for j" loop and remove immediate left recursion from the S productions (there is none).
• When i = 2 and j = 1, we substitute the S-productions in A ☐ Sd to obtain the A-productions A ☐ Ac | Aad | bd | ε
• Eliminating immediate left recursion from the A productions yields the grammar:

S ☐ Aa | b
A ☐ bdA' | A'
A' ☐ cA' | adA' | ε

50. Compare the feature of DFA and NFA.(May/June 2014)

1.Both are transition functions of automata. In DFA the next possible state is distinctly set while in NFA each pair of state and input symbol can have many possible next states.
2.NFA can use empty string transition while DFA cannot use empty string transition.
3.NFA is easier to construct while it is more difficult to construct DFA.
4.Backtracking is allowed in DFA while in NFA it may or may not be allowed.
5.DFA requires more space while NFA requires less space.

51. Eliminate left recursion for the grammar.

E→E+T/T, T→T*F/F,F→(E)/id (April/May-08,Marks 2)

Answer

A→Aα/β then convert it to
A→βA'
A'→αA'
A'→ϵ

E→TE'
E'→+TE'/ϵ
T→FT'
T'→*FT'/ϵ
F→(E)/id

PART-B

1.(i) Construct Predictive Parser for the following grammar:(May/June-2012&13)(10)

S->(L)/a
L->L,S/S

After left recursion elimination, it becomes

1) S -> (L)
2) S -> a
3) L -> S L'
4) L'-> , S L'

5) L' -> ϵ

The first/follow tables are:

	FIRST	FOLLOW
G	(a	
S	(a	,) $
L	(a	)
L'	,	)

which are used to create the parsing table:

	(	)	a	,	$
G	0		0		
S	1		2		
L	3		3		
L'		5		4	

(ii) Describe the conflicts that may occur during shift reduce parsing. (may/june-2012).(6)

Hints:

- Shift/Reduce conflict: The entire stack contents and the next input symbol cannot decide whether to shift or reduce.
- Reduce/Reduce conflict: The entire stack contents and the next input symbol cannot decide which of several reductions to make.

2.What is FIRST and FOLLOW? Explain in detail with an example. Write down the necessary algorithm.(16)

FIRST

If X is a terminal then First(X) is just X!
If there is a Production X → ε then add ε to first(X)
If there is a Production X → Y1Y2..Yk then add first(Y1Y2..Yk) to first(X)
First(Y1Y2..Yk) is either
First(Y1) (if First(Y1) doesn't contain ε)
OR (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
If First(Y1) First(Y2)..First(Yk) all contain ε then add ε to First(Y1Y2..Yk) as well.

The CFG is

E → TE'

E' → +TE'
E' → ε
T → FT'
T' → *FT'
T' → ε
F → (E)
F → id

Example

FIRST(E) = {'(',id}
FIRST(E') = {+,ε}

FIRST(T) = {'(',id}
FIRST(T') = {*,ε}
FIRST(F) = {'(',id}

FOLLOW

First put $ (the end of input marker) in Follow(S) (S is the start symbol)
If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).
If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Example

FOLLOW(E) = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T) = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F) = {*,+,$,)}

3. Consider the grammar given below:

E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

Construct an LR parsing side for the above grammar. Give the moves of LR parser on id*id+id

The CFG is

After Elimination of Left recursion the grammar looks like the following

E → TE'
E' → +TE'
E' → ε
T → FT'
T' → *FT'
T' → ε
F → (E)
F → id

FIND FIRST

FIRST(E) = {'(',id}
FIRST(E') = {+,ε}
FIRST(T) = {'(',id}
FIRST(T') = {*,ε}
FIRST(F) = {'(',id}

FIND FOLLOW

FOLLOW(E) = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T) = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F) = {*,+,$,)}

Constructing parsing table

	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)		

Stack Implementation of w = id+id*id

Stack	**Production Rule**
$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. Consider the following grammar(16). (Nov/Dec-2012)

E→E+T | T
T→T*F | F
F→(E) |id

construct the SLR parsing table for this grammar. Also parse the input id*id+id (i.e a*b+a) .

Constructing an SLR parse table

S ->simple

L ->left-to-right scan of input

R-> rightmost derivation in reverse

Part 1: Create the set of LR(0) states for the parse table

For the rules in an augmented grammar, G', begin at rule zero and follow the steps below:

State creation steps (big picture algorithm)

1. Apply the start operation and

2. complete the state:

   a. Use one read operation on each item C (non-terminal or terminal) in the current state to create more states.

   b. Apply the complete operation on the new states.

   c. Repeat steps a and b until no more new states can be formed.

Operations defined

A, S, X: non-terminals

w,x,y,z: string of terminals and/or non-terminals

C: one terminal or one non-terminal

start: if S is a symbol with [S -> w] as a production rule, then [S -> .w} is the item associated with the start state.

read: if [A --> x.Cz] is an item in some state, then [A --> xC.z] is associated with some other state.When performing a read, all the items with the dot before the same C are associated with the same state.

complete: if [A --> x.Xy] is an item, then every rule of the grammar with the form [X --> .z] must be included within this state. Repeat adding items until no new items can be added.

CFG:

E -> E + T | T

T -> T * F | F

F -> (E) | id

State	Item	Goto	State	Item	Goto
0:	**E' → . E** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	1 1 2 2 3 4 5	1:	**E' → E .** **E → E . + T**	 6
2:	**E → T .** **T → T . * F**	 7	3:	**T → F .**	
4:	**F → (. E)** E → . E + T E → . T T → . T * F T → . F F → . (E) F → . id	8 8 2 2 3 4 5	5:	**F → id .**	
6:	**E → E + . T** T → . T * F T → . F F → . (E) F → . id	9 9 3 4 5	7:	**T → T * . F** F → . (E) F → . id	10 4 5
8:	**F → (E .)** **E → E . + T**	11 6	9:	**E → E + T .** **T → T . * F**	 7
10:	**T → T * F .**		11:	**F → (E) .**	

The action/goto tables are extracted directly from the above information in the following manner. First, each state in the above table will be a row in the action/goto tables.

Filling in the entries for row I is done as follows:

- Action[I,c] = shift to state goto(items in state I, c) for terminal c
- Goto[I,c] = goto(items in state I, c) for non-terminal c
- For production number n: A Æ α in state I where . occurs at the end, Action[I,a] = reduce production n, for all elements a in Follow(A)
- For the added production(augmented production) with the . at the end in state I, Action[I,$] = accept.

Consider state 6. Action[6,(] = shift 4, Action[6,id] = shift 5, Goto[6,T] = 9, and Goto[6,F] = 3. For state 5, Action[5,*] = Action[5,)] = Action[5,+] = Action[5,$] =reduce 6.

SLR Parsing Table:

State	+	*	(	)	id	$		E	T	F
0			s4		s5			1	2	3
1	s6					accept				
2	r2	s7		r2		r2				
3	r4	r4		r4		r4				
4			s4		s5			8	2	3
5	r6	r6		r6		r6				
6			s4		s5				9	3
7			s4		s5					10
8	s6			s11						
9	r1	s7		r1	r1					
10	r3	r3		r3	r3					
11	r5	r5		r5	r5					

5. (i)Construct SLR parsing table for the following grammar (10). (Nov/Dec-2012,April/May-04,Marks 8)
S→ L=R|R
L→*R| id
R→L Write down the rules for FIRST and FOLLOW (6). (Nov/Dec-2012)

1. Construct F = {I0, I1, ... In}, the collection of LR(0) configurating sets for G'.

2. State i is determined from Ii. The parsing actions for the state are determined as follows:

   a) If A –> u• is in Ii then set Action[i,a] to reduce A –> u for all a in Follow(A) (A is not S').

   b) If S' –> S• is in Ii then set Action[i,$] to accept.

   c) If A –> u•av is in Ii and successor(Ii, a) = Ij, then set Action[i,a] to shift j (a must be a terminal).

3. The goto transitions for state i are constructed for all non-terminals A using the rule:

   If successor(Ii, A) = Ij, then Goto [i, A] = j.

4. All entries not defined by rules 2 and 3 are errors.

5. The initial state is the one constructed from the configurating set containing

   S' –> •S

6. . Check whether the following grammar is a LL(1) grammar
S → iEtS | iEtSeS | a
E → b
Also define the FIRST and FOLLOW procedures. (16)

Steps:

Step 1: Eliminate left recursion and left factoring
Step 2: Find out the FIRST

Step 3: Find out FOLLOW
Step 4: Constructing a parsing table
Step 5: Stack Implementation

7. Construct non recursion predictive parsing table for the following grammar.
E→E or E /E and E/ not E / (E) / 0 /1. (Dec-12,Marks 16).

Steps:

Step 1: Eliminate left recursion and left factoring
Step 2: Find out the FIRST
Step 3: Find out FOLLOW
Step 4: Constructing a parsing table
Step 5: Stack Implementation

8. . i. Find the language from (4)
S → 0S1 | 0A1 A→1A0 | 10
ii.Define Parse tree , Regular Expression , Left most derivation , Right most (4) derivation , and write example for each.
iii.Write algorithm to convert NFA from Regular expression. (4)

i.Find the language from (4)
S→0S1 | 0A | 0 |1B | 1
A→0A | 0
B→1B|1

Answer:

The minimum string is S-> 0 | 1

S->0S1=>001

S->0S1=>011

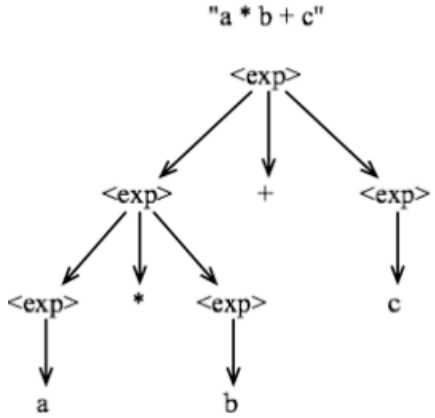
S->0S1=>00S11=>000S111=>0000A111=>00000111

Thus L={ 0^n 1^m | m not equal to n, and n,m >=1}

ii.Definition of parse tree, an regular expression and LMD, RMDwith examples

parse tree

A parse tree or parsing tree or derivation tree or (concrete) syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.



regular expression

Regular expressions are mathematical symbolism which describes the set of strings of specific language. It provides convenient and useful notation for representing tokens. Here are some rules that describe definition of the regular expressions over the input set denoted by ∑

Left most derivation

A top-down parse we always choose the leftmost non-terminal in a sentential form to apply a production rule to - this is called a leftmost derivation.

E → E + E → a + E → a + E + E
→ a + a + E → a + a + a

Rightmost Derivation

A bottom-up parse then the situation would be reversed, and we would want to do apply the production rules in reverse to the leftmost symbols; thus we are performing a rightmost derivation in reverse.

E → E + E → E + E + E → a + E + E
→ a + a + E → a + a + a

iii.Write algorithm to convert NFA from Regular expression. (4)

We will use the rules which defined a regular expression as a basis for the construction:

- The NFA representing the empty string
- If the regular expression is just a character, eg. a, then the corresponding NFA
- The union operator is represented by a choice of transitions from a node
- Concatenation simply involves connecting one NFA to the other
- The Kleene closure must allow for taking zero or more instances of the letter from

the input
9.i.Prove the grammar is ambiguous. (4) ***E→E+E | E*E | (E) | id*** ***ii. Specify the demerits of ambiguous grammar. (2)*** ***iii. What are the rules to convert an unambiguous grammar from ambiguous grammar .Write necessary steps for the above ambiguous grammar.*** ***iv. Using unambiguous grammar, write Leftmost derivation ,draw parse tree for the string id*id*id+id*id*** **Hints:** • Draw more than parse tree for any expression. • It is difficult to find which parse tree is correct for evaluation • Refer rules. • Apply rules for conversion.
10. Write in detail about ***i.Recursive descent parsing with algorithm.*** ***ii.Top down parsing with algorithm.*** **Hints:** • Definition of recursive descent and algorithm. • Definition of top down parser , and algorithm for top down parsing table , parsing method of top down parser.
11. Construct predictive parsing table and parse the string NOT(true OR false) ***bexpr→bexpr OR bterm | bterm*** ***bterm→bterm AND bfactor | bfactor*** ***bfactor→NOT bfactor | (bexpr) | true | false*** **Hints:** • Find FIRST and FOLLOW and • construct table and • parse the string.
12. . Construct CLR parsing table to parse the sentence id=id*id for the following grammar. ***S→ L=R | R*** ***L→*R | id*** ***R→L*** **Hints:** • Find LR(1) items; • Construct CLR table. • Parse the string.
13. Parse the string (a,a) using SLR parsing table. ***S→ (L) | a*** ***L→L , S | S*** **Hints:** • Find LR(0) items.

- Construct SLR parsing table.
- Parse the string.

14. Construct LALR parsing table for the grammar.
E→ E+T | T
T→ T*F | F
F→ (E) | id

Hints:

- Find LR(1) items.
- Find same core items different second component .them merge it
- After merging construct LALR parsing table.

15. Write algorithms for SLR and CLR string parsing algorithm.
Hints:

- Brief introduction about SLR and CLR parsers.
- Algorithm

16.(i) Generate SLR Parsing table for the following grammar. (12)
S-->Aa|bAc|Bc|bBa
A-->d
B-->d
And parse the sentences "bdc"and "dd".(April/May 2015)

1. Construct F = {I0, I1, ... In}, the collection of LR(0) configurating sets for G'.

2. State i is determined from Ii. The parsing actions for the state are determined as follows:

   a) If A –> u• is in Ii then set Action[i,a] to reduce A –> u for all a in Follow(A) (A is not S').

   b) If S' –> S• is in Ii then set Action[i,$] to accept.

   c) If A –> u•av is in Ii and successor(Ii, a) = Ij, then set Action[i,a] to shift j (a must be a terminal).

3. The goto transitions for state i are constructed for all non-terminals A using the rule:

   If successor(Ii, A) = Ij, then Goto [i, A] = j.

4. All entries not defined by rules 2 and 3 are errors.

5. The initial state is the one constructed from the configurating set containing

   S' –> •S

17.(i) Write the algorithm to eliminate left-recursion and left-factoring and apply both to the following grammar. (8)
E-->E+T|E-T|T
T-->a|b|(E) ***(April/May 2015)***

Steps:

Step 1: Eliminate left recursion and left factoring
Step 2: Find out the FIRST

Step 3: Find out FOLLOW
Step 4: Constructing a parsing table
Step 5: Stack Implementation

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 FORTAN.

PART-A

1.Define syntax directed definition.

Syntax directed definition is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is a associated with it a set of semantic rules of the form a:=f(b1,b2,...bk) , where a is an attribute obtained from the function f.

2.What is mean by syntax directed definition.

It is a generalization of a CFG in which each grammar symbol has an associated set of attributes like, synthesized attribute and inherited attribute

3.How the value of synthesized attribute is computed?

It was computed from the values of attributes at the children of that node in the parse tree.

4.How the value of inherited attribute is computed?

It was computed from the value of attributes at the siblings and parent of that node.

5.What is mean by construction of syntax tree for expression.

Construction syntax tree for an expression means translation of expression into postfix form. The nodes for each operator and operand is created. Each node can implemented as a record with multiple fields.Following are the function used in syntax tree for expression.

1. Mknode(op,left,right)
2. Mknode(id,entry)
3. Mkleaf(num,val)

6.What are the function of construction of syntax tree for expression? Explain.

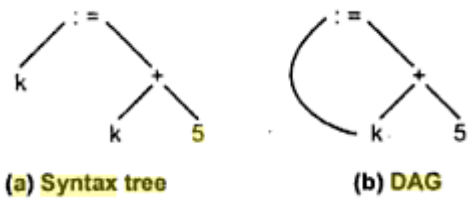
1. **Mknode(op,left,right)**
   This function creates a node with field operator having operator as label, and the two pointer to left and right.
2. **Mknode(id,entry)**
   This function creates identifier node with label id and a pointer to symbol table is given by 'entry'.
3. **Mkleaf(num,val)**
   This function creates node for number with label num and val is for value of that number.

7.What do you mean by DAG?

It is Directed Acyclic Graph. In this common sub expressions are eliminated. So it is a compact way of representation. Like syntax tree DAG has nodes representing the subexpressions in the expression. These nodes have operator, operand1 and operand2 where operands are the children of that node.

The Difference between DAG and syntax tree is that common subexpressions has more than one parent and in syntax tree the common subexpression would be represented as duplicated subtree.

8.Construct a syntax tree and DAG for k:=k+5.



(a) Syntax tree (b) DAG

9.What is S-attributed definition?

S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, but only synthesized attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created after creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

10.What is L-attributed definition?

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing. Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are comparable with S-attributed grammars. Used for code synthesis.

The Class of L-attributes can be evaluated in depth first order.

11.Define a translation scheme.

A translation scheme is a context-free grammar in which semantic rules are embedded within the right sides of the productions. So a translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly shown. The position at which an action is to be executed.

12.What are the advantage of SDT?

Syntax directed translation is a scheme that indicate the order in which semantic rules are to be evaluated. The main advantage of SDT is that it helps in deciding evaluation order. The evaluation of semantic actions associated with SDT may generate code, save information in symbol table, or may issue error messages.

13.What is type checking?

Type checker verifies that the type of a construct (constant,variable,array,list,object) matches what is expected in its usage context.

14.What are static and dynamic errors?

Static error: It can be detected at compile time. Eg: Undeclared identifiers.
Dynamic errors: It can be detected at run time. Eg: Type checking

15.What are the advantages of compile time checking?

(i) It can catch many common errors.
(ii) Static checking is desired when speed is important, since it can result faster code that does not perform any type checking during execution.

16.What are the advantages of the dynamic checking?

- It usually permits the programmer to be less concerned with types. Thus, if frees the programmer.
- It may be required in some cases like array bounds check, which can be performed only during execution.
- It can give in clearer code.
- It may rise to in more robust code by ensuring thorough checking of values for the program identifiers during execution.

17.Define type systems.

Type system of a language is a collection of rues depicting the type expression assignments to program objects. An implementation of a type systems is called a type checker.

18.Write Static vs. Dynamic Type Checking

Static: Done at compile time (e.g., Java)

Dynamic: Done at run time (e.g., Scheme)

Sound type system is one where any program that passes the static type checker cannot contain run-time type errors. Such languages are said to be strongly typed.

19.Define procedure definition.

A procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the procedure name, and the statement body. Some of the identifiers appearing in a 'procedure definition' are special and are called 'formal parameters ' of the procedure. Arguments,

known as 'actual parameters' may be passed to a called 'procedure'; they are substituted for the formal in the body.

20.Define activation trees.

A recursive procedure p need not call itself directly; p may call another procedure q, which may then call p through some sequence of procedure calls. We can use a tree called an activation tree, to depict the way control enters and leaves activation. In an activation tree

a) Each node represents an activation of a procedure,

b) The root represents the activation of the main program

c) The node for a is the parent of the node for b if an only if control flows from activation a to b, and

d) The node for a is to the left of the node for b if an only if the lifetime of a occurs before the lifetime of b.

21.Write notes on control stack?

A control stack is to keep track of live procedure activations. The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.

22.Write the scope of a declaration?

A portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to eh procedure if it is in the cope of a declaration within the procedure; otherwise, the occurrence is said to be nonlocal.

23.Define binding of names.

When an environment associates storage location s with a name x, we say that x is bound to s; the association itself is referred to as a binding of x. A binding is the dynamic counterpart of a declaring.

24.What is the use of run time storage?

The run time storage might be subdivided to hold: **a)** The generated target code **b)** Data objects, and **c)** A counterpart of the control stack to keep track of procedure activation.

25.What is an activation record? (or) What is frame?

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of the collection of fields such as a) Return value b) Actual parameters c) Optional control link d) Optional access link e) Saved machine status f) Local data g) Temporaries.

26.What does the runtime storage hold?

Runtime storage holds

1) The generated target code
2) Data objects
3) A counter part of the control stack to keep track of procedure activations.

27.What are the various ways to pass a parameter in a function?

- call-by-value
- call-by-reference
- call-by-value-result(copy-restore) :this method is a hybrid between call by value and call by references.
- call-by-name

28.What are the limitations of static allocation? (Nov/dec 2012)

a) The size of a data object and constraints on its position in memory must be known at compile time.

b) Recursive procedure is restricted.

c) Data structures cannot be created dynamically.

29What is stack allocation?

Stack allocation is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end respectively.

30.List the fields in activation record .(Nov/Dec 2014)

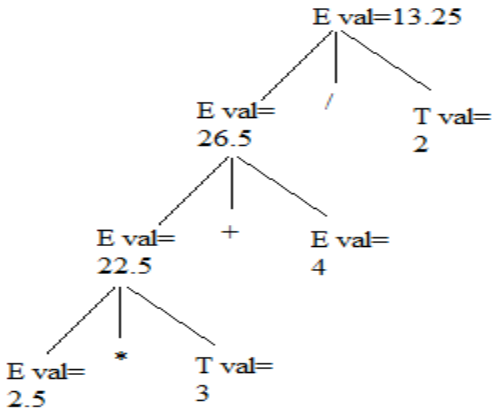
- Actual parameters
- Returned Values
- Control link

- Access link
- Saved machine status
- Local data
- Temporaries

31.What is dangling references?

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

32.Constructed a decorated parse tree according to the syntax directed definition, for the following input statement (4+7.5*3)/2. (April/May 2015)



33.Write a 3-address code for; x=*y ; a=&x. (April/May 2015)

t1:=*y
x:=t1

t1:=&x
a:=t1

34.Place the above code in Triplets and indirect Triplets.(April/May 2015)

Triple:

	Op	Arg1	Arg2
(0)	*	Y	
(1)	=	X	(0)

Indirect:

	Op	Arg1	Arg2
11	*	Y	
12	=		(11)

	statement
(0)	11
(1)	12

Triple:

	Op	Arg1	Arg2
(0)	&	x	
(1)	=	a	(0)

Indirect:

	Op	Arg1	Arg2
11	&	x	
12	=	a	(11)

	statement
(0)	11
(1)	12

PART –B

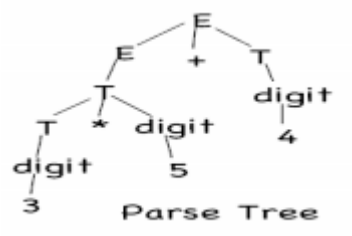
1.Explain the concept of syntax directed definition.

- A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions.
- The semantic rules define values for attributes associated with the symbols of the productions.
- These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass.
- SDDs are useful for specifying translations.

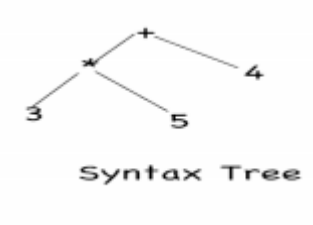
  CFG + semantic rules = Syntax Directed Definitions

2.Construct parse tree, syntax tree and annotated parse tree for the input string is 3*5+4;

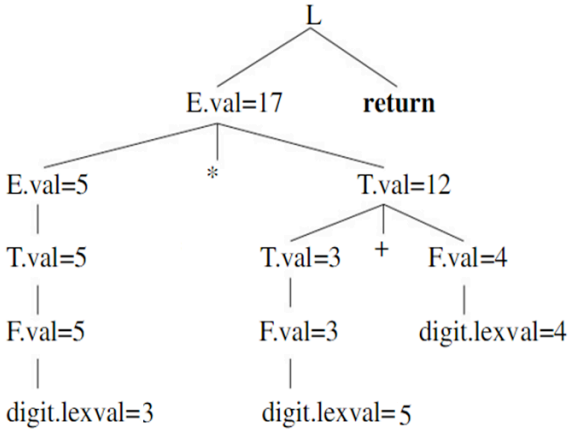
parse tree



syntax tree



annotated parse tree



3.Explain 1)Synthesized attribute 2)inherited attribute with suitable examples.

Synthesized attributes

These attributes get values from the attribute values of their child nodes.

- Example

  1. S → ABC

  If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

2. E → E + T

The parent node E gets its value from its child node E and T.

Example

E → E1 + T { E.val = E1.val + T.val; }
E → T { E.val = T.val; }
T → (E) { T.val = E.val; }
T → digit { T.val = digit.lexval; }

Inherited attributes

- inherited attributes can take values from parent and/or siblings.
- Example

  S → ABC

  A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Example

E → T A { E.node = A.s;
A.i = T.node; }
A → + T A1 { A1.i = Node('+', A.i, T.node);
A.s = A1.s; }
A → e { A.s = A.i; }

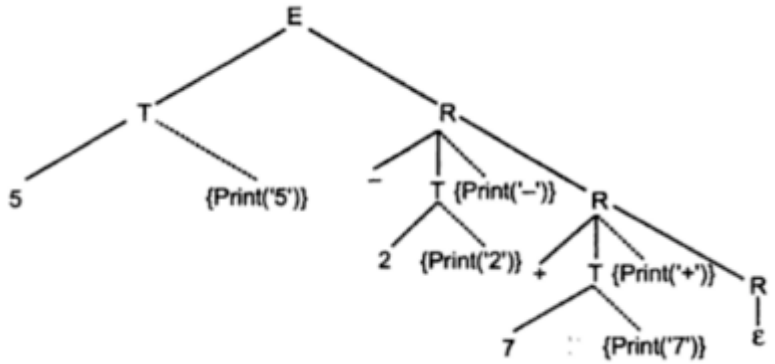
T → (E) { T.node = E.node; }

T → id { T.node = Leaf(id, id.entry); }

4.Write a syntax directed definition and evaluate 9*3+2 with parser stack using LR parsing method.

- 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.

5.Consider the following CFG,
E→TR
R→+TR
R→-TR
R→ϵ
T→num
With translation scheme to generate to generate postfix expression equivalent to the given infix expression which is recognized by above grammar. All actions in the translation should be at the end of each production



6.Explain the Implementing L-Attributed SDD's

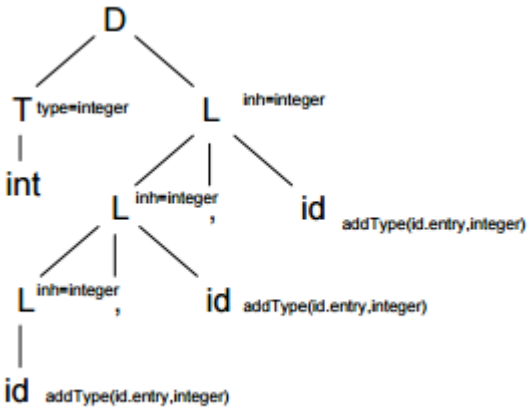
- This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
- Example

$$S \rightarrow ABC$$

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

7.(i)Given the Syntax-Directed Definition below construct the annotated parse tree for the input expression: "int a, b, c".

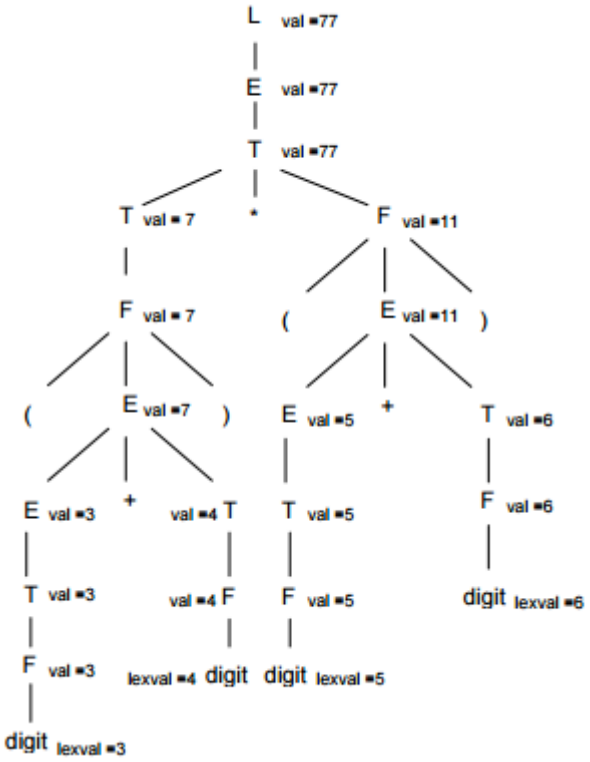
D → T L L.inh = T.type
T → int T.type = integer
T → float T.type = float
L → L1, id L1.inh = L.inh addType(id.entry,L.inh)
L → id addType(id.entry,L.inh)



(ii) Given the Syntax-Directed Definition below with the synthesized attribute val, draw the annotated parse tree for the expression (3+4) * (5+6).

L → E L.val = E.val
E → T E.val = T.val
E → E1 + T E.val = E1.val + T.val
T → F T.val = F.val

T → T1 * F T.val = T1.val * F.val
F → (E) F.val = E.val
F → digit F.val = digit.lexval



8.Explain the various structures that are used for the symbol table constructions.(April/may 2012,2014)

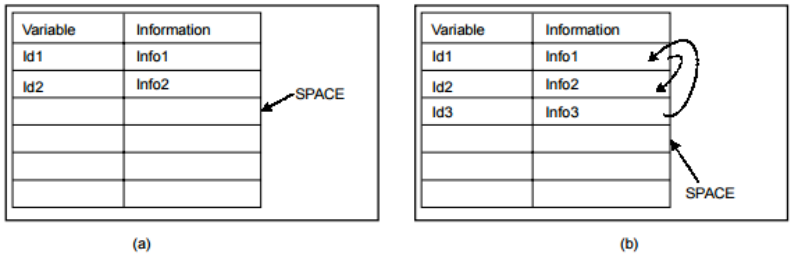
- A separate array 'arr_lexemes' holds the character string forming an identifier. The string is terminated by an end-of-string character, denoted by EOS, that may not appear in identifiers.
- Each entry in symbol-table array 'arr_symbol_table' is a record consisting of two fields, as "lexeme_pointer", pointing to the beginning of a lexeme, and token.
- Additional fields can hold attribute values. 0th entry is left empty, because lookup return 0 to indicate that there is no entry for a string.
- The 1st, 2nd, 3rd, 4th, 5th, 6th, and 7th entries are for the 'a', 'plus' 'b' 'and', 'c', 'minus', and 'd' where 2nd, 4th and 6th entries are for reserve keyword.

List

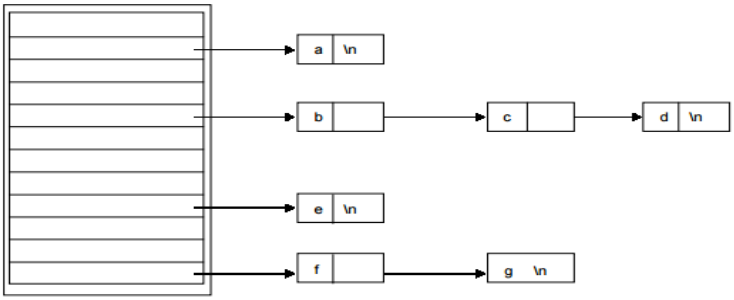
Variable	Information(type	Space (byte)
a	Integer	2
b	Float	4
c	Character	1
d	Long	4

SPACE

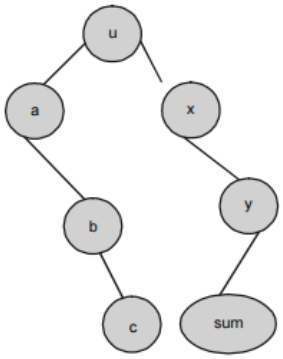
Self Organizing List



Hash Table



Search Tree



9.(i)What are different storage allocation strategies? Explain.(8)(may/june-2013)

There are three different storage allocation strategies based on this division of run-time storage. The strategies are-

1. **Static allocation**- The static allocation is for all the data objects at compile time.
2. **Stack allocation**- In the stack allocation a stack is used to manage the run time storage.
3. **Heap allocation**- In heap allocation the heap is used to manage the dynamic memory allocation.

Static Allocation

The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.

The binding of name with the amount of storage allocated do not change at run-time. Hence the name of this allocation is static allocation.

In static allocation the compiler can determine the amount of storage required by each data object. And therefore it becomes easy for a compiler to find the addresses of these data in the activation records.

At compile time compiler can fill the addresses at which the target code find the data it operates on.

FORTRAN uses the static allocation strategy.

Stack Allocation

Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.

As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.

The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.
The data structures can be created dynamically for stack allocation.

Heap Allocation

If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last In First Out nature. For retaining of such local variables heap allocation strategy is used.
The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object. This allocated memory can be deallocated when activation ends. This deallocated space can be further reused by heap manager.
The efficient heap management can be done by,
i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.
ii) Allocate the most suitable block of memory from the linked list. i.e., use best fit technique for allocation of block.

(ii)Specify a type checker which can handle expressions, statements and functions.(8)

The validity of statements

Expressions have types, but statements do not.
However, also statements are checked in type checking.
We need a new judgement form, saying that a statement S is valid:

```
F,G => S valid
```

Example: typing rule for an assignment

```
   F,G => e : T
   ---------- x : T is in G
   F,G => x = e ; valid
```

Example: typing rule for while loops

```
   F,G => e : bool   F,G => S valid
   --------------------------------
   F,G => while (e) S valid
```

Expression

We prove that int x ; x = x + 5 ; is valid in the empty context ().

```
   x : int => x : int     x : int => 5 : int
   ------------------------------------------
        x : int => x + 5 : int
      -------------------------
      x : int => x = x + 5 ; valid
    -------------------------------
    () => int x ; x = x + 5 ; valid
```

The signature is omitted for simplicity.

Function types

No expression in the language has **function types**, because functions are never returned as values or used as arguments.
However, the compiler needs internally a data structure for function types, to hold the types of the parameters and the return type. E.g. for a function

```
   bool between (int x, double a, double b) {...}
```

we write

```
   between : (int, double, double) -> bool
```

to express this internal representation in typing rules.

10.Explain the organization of runtime storage in detail.

STATIC STORAGE ALLOCATION

In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:

1. The size of each object must be known at compile time.

2. Only one occurrence of each object is allowable at a given moment during program execution.

A static storage-allocation strategy is very simple to implement.

An object address can be either an absolute or a relative address.

DYNAMIC STORAGE ALLOCATION

In a dynamic storage-allocation strategy, the data area requirements for a program are not known entirely at compilation time. In particular, the two criteria that were given in the previous section as necessary for static storage allocation do not apply for a dynamic storage-allocation scheme. The size and number of each object need not be known at compile time; however, they must be known at run time when a block is entered. Similarly more than one occurrence of a data object is allowed, provided that each new occurrence is initiated at run time when a block is entered.

11.Explain about static and stack allocation in storage allocation strategies

In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:

1. The size of each object must be known at compile time.

2. Only one occurrence of each object is allowable at a given moment during program execution.

A static storage-allocation strategy is very simple to implement.

An object address can be either an absolute or a relative address

Implicit parameters
Actual parameters
Simple variables, aggregates, and temporaries

12.Explain any 4 issues in storage allocation (4). (April/May 2015)

13.Give a Syntax directed Definitions to differentiate expressions formed by applying the arithmetic operators + and * to the variable X and constants ; expression :X*(3*X+X*X). (April/May 2015) (8)

- A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions.

- The semantic rules define values for attributes associated with the symbols of the productions.
- These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass.
- SDDs are useful for specifying translations.

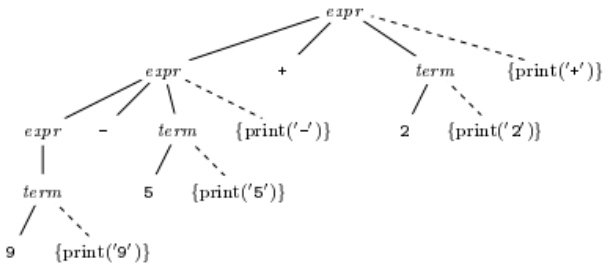
CFG + semantic rules = Syntax Directed Definitions

Example

E → E1 + T { E.val = E1.val + T.val; }
E → T { E.val = T.val; }
T → (E) { T.val = E.val; }
T → digit { T.val = digit.lexval; }

14. 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)

Annotated parse Tree:



translation scheme

expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ - *term*	{print('-')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		. . .	
term	→	9	{print('9')}

(ii) Write the 3-address code(6)

In three-address code, this would be broken down into several separate instructions. These instructions translate more easily to assembly language. It is also easier to detectcommon sub-expressions for shortening the code.

Example:

t1 := b * b

t2 := 4 * a

t3 := t2 * c

t4 := t1 - t3

t5 := sqrt(t4)

t6 := 0 - b

t7 := t5 + t6

t8 := 2 * a

t9 := t7 / t8

x := t9

(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) (April/May 2015)

- Array indexing- In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value.
- Base address consists of the address of first element in an array.
- Other elements of the array can be accessed using the base address and offset value.

Example: x = y[i]
Memory location m = Base address of y + Displacement i

- x = contents of memory location m similarly x[i] = y
- Memory location m = Base address of x + Displacement i.
- The value of y is stored in memory location m

18(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;
Write the semantic rule and derive the Parse tree for the given code (12)

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression B in if (B) S contains a jump, for when B is false, to the instruction following the code for S. In a one-pass translation, B must be translated before S is examined. What then is the target of the goto that jumps over the code for S? In Section 6.6 we

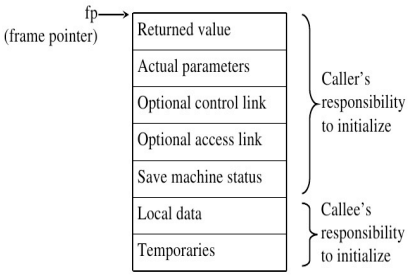
addressed this problem by passing labels as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses. This section takes a complementary approach, called backpatching, in which lists of jumps are passed as synthesized attributes. Specifically, when a jumpis generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

(ii) What is an Activation Record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations. (4) (April/May 2015)

Modern imperative programming languages typically have local variables.
– Created upon entry to function.
– Destroyed when function returns.
Each invocation of a function has its own instantiation of local variables.
– Recursive calls to a function require several instantiations to exist simultaneously.
– Functions return only after all functions it calls have returned last-in-first-out (LIFO) behavior.
– A LIFO structure called a stack is used to hold each instantiation.
The portion of the stack used for an invocation of a function is called the function's stack frame or activation record.

11

Activation Records
(Subroutine Frames)



The Stack

- Used to hold local variables.
- Large array which typically grows downwards in memory toward lower addresses,
- shrinks upwards.
- Push(r1):
  stack_pointer--;
  M[stack_pointer] = r1;
- r1 = Pop():
  r1 = M[stack_pointer];
  stack_pointer++;
- Previous activation records need to be accessed, so push/pop not sufficient.
  - Treat stack as array with index off of stack pointer.
  - Push and pop entire activation records.

Unit- V 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.

1.What is meant by optimization?
It is a program transformation that made the code produced by compiling algorithms run faster or takes less space.

2.What are the principle sources of optimization?
The principle sources of optimization are, Optimization consists of detecting patterns in the program and replacing these patterns by equivalent but more efficient constructs. The richest source of optimization is the efficient utilization of the registers and instruction set of a machine.

3.Mention some of the major optimization techniques.

- Local optimization
- Loop optimization
- Data flow analysis
- Function preserving transformations
- Algorithm optimization.

4.What are the methods available in loop optimization?
Code movement - Strength reduction- Loop test replacement- Induction variable elimination

5.What is the step takes place in peephole optimization?
It improves the performance of the target program by examining a short sequence of target instructions. It is called peephole. Replace this instructions by a shorter or faster sequence whenever possible. It is very useful for intermediate representation.

6.What are the characteristics of peephole optimization?
a. Redundant instruction elimination. b. Flow of control optimization c. Algebraic simplifications d. Use of machine idioms

7.What are the various types of optimization?
The various type of optimization is, 1)Local optimization,2)Loop optimization,3)Data flow analysis,4)Global optimization.

8.List the criteria for selecting a code optimization technique.
The criteria for selecting a good code optimization technique are, It should capture most of the potential improvement without an unreasonable amount of effort. It should preserve the meaning of the program. It should reduce the time or

space taken by the object program.

9.What is meant by U-D chaining?
It is Use-Definition chaining. It is the process of gathering information about how global data flow analysis can be used id called as use-definition (UD) chaining.

10.What do you mean by induction variable elimination?
It is the process of eliminating all the induction variables , except one when there are two or more induction variables available in a loop is called induction variable elimination.

11.List any two structure preserving transformations adopted by the optimizer?
The structure preserving transformations adopted by the optimizer are, Basic blocks.-Flow graphs.

12.What are dominators?
A node of flow graph is said to be a dominator, i.e one node dominates the other node if every path from the initial node of the flow graph to that node goes through the first node.(d Dom n).when d-node dominates n-node.

13.What is meant by constant folding?
Constant folding is the process of replacing expressions by their value if the value can be computed at complex time.

14.Define optimizing compilers.(Nov/Dec 2013)
Compilers that apply code-improving transformations are called optimizing compilers.

15.When do you say a transformation of a program is local?
A transformation of a program is called local, if it can be performed by looking only at the statement in a basic block.

16.Write a note on function preserving transformation.
A complier can improve a program by transformation without changing the function it compliers.

17.List the function –preserving transformation.
1)Common subexpression elimination.2)Copy propagation.3)Dead code elimination.4)Constant folding.

18.Define common subexpression.
An occurrence of an expression E is called a common subexpression if E was previously computed and the values of variables in E have not changed since the previous computation.

19.What is meant by loop optimization?
The running time of a program may be improved if we decrease the number of instructions in a inner loop even if we increase the amount of code outside that loop.

20.What do you mean by data flow equations?
A typical equation has the form out[s] = gen[s] U (in[s]-kill[s])
It can be read as 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.
State the meaning of in[s], out[s], kill[s], gen[s].
in[s]-The set of definitions reaching the beginning of S.out[s]-End of S.gen [s]-The set of definitions generated by S.kill[s]-The set of definitions that never reach the end of S.

21.What is data flow analysis? (Nov/Dec 2012)
The data flow analysis is the transmission of useful relationships from all parts of the program to the places where the information can be of use.

22.Define code motion and loop-variant computation.
Code motion: It is the process of taking a computation that yields the same result independent of the number of times through the loops and placing it before the loop. Loop –variant computation: It is eliminating all the induction variables, except one when there are two or more induction variables available in a loop

23.Define loop unrolling with example.(Nov/Dec 2013)
Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.
Example:
In the code fragment below, the body of the loop can be replicated once and the number of iterations can be reduced from 100 to 50.

```
for (i = 0; i < 100; i++)
  g ();
```

Below is the code fragment after loop unrolling.

```
for (i = 0; i < 100; i += 2)
{
  g ();
  g ();
}
```

23.What is constant folding?(May/June 2013)
Constant folding is the process of replacing expressions by their value if the value can be computed at complex time.

24.How would you represent the dummy blocks with no statements indicated in global data flow nalysis?(May/June 2013)
Dummy blocks with no statements are used as technical convenience (indicated as open circles).

25.What is meant by copy propagation or variable propagation?

One concerns assignments of the form f:=g called copy statements or copies for short
Eg: it means the use of variable V1 in place of V2
1.V1:=V2
2.f:V1+f
3.g:=v2+f-6
In statement 2, V2 can be used in place of V1 by copy propagations.
So,
1.V1=V2;
….
…
….
4.f:=V2+f
5.g:V2+f-6
This leads for common sub expression elimination further (V2+f is a common sub expression).

26.Define busy expressions.
An expression E is busy at a program point if and only if

- An evaluation of E exits along some path P1,P2,….Pn starting at program point P1 and,
- No definition of any operand of E exists before its evaluation along the path.

27.Define code generation. or What role does the target machine play on the code generation phase of the compiler?(April/May 2015).
The code generation is the final phase of the compiler. It takes an intermediate representation of the source program as the input and produces an equivalent target program as the output.

28.Define Target machine.
The target computer is byte-addressable machine with four bytes to a word and n-general purpose registers. R0, R1… … … .Rn-1. It has two address instructions of the form Op, source, destination in which Op is an op-code, and source and destination are data fields.

29.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          cost is 1
MOV R1,M           cost is 2
SUB 5(R0),*10(R1)  cost is 3
```

30.Define basic block. (Nov/Dec 2013)
A basic block contains sequence of consecutive statements, which may be entered only at the beginning and when it is entered it is executed in sequence without halt or possibility of branch.

31.What are the rules to find " leader" in basic block?

- It is the first statement in a basic block is a leader.
- Any statement which is the target of a conditional or unconditional goto is a leader.
- Any statement which immediately follows a conditional goto is a leader.

32.Define flow graph.(Nov/Dec 2013)
Relationships between basic blocks are represented by a directed graph called flow graph.

33.What do you mean by DAG?
It is Directed Acyclic Graph. In this common sub expressions are eliminated. So it is a compact way of representation.

34.List the advantages of DAGs (Nov/Dec 2012)
It automatically detects common sub expression.
We can determine which identifiers have their values used in the block.
We can determine which statements compute values, and which could be used outside the block. It reconstruct a simplified list of quadruples taking advantage of common sub expressions and not performs assignments of the form a=b unless necessary.

35.What is meant by registers and address descriptor?
Register descriptor: It contains information' s about,1.What registers are currently in use. 2.What registers are empty.
Address descriptor: It keeps track of the location where the current value of the name can be found at run time.

36.Explain heuristic ordering for DAG?
It is based on Node Listing algorithm.

```
While unlisted interior nodes remain do
begin
        Select an unlisted node n , all of whose parents have been listed;
list n;
        While the leftmost child 'm' of 'n' has no unlisted parents and is not a
leaf do
        do
```

```
                {
                        /*since n was just listed, surely m is not yet listed */
                }
        begin
                list m
                n=m;
        end
end
```

37.Write labeling algorithm.

```
if n is a leaf then
        if n is the leftmost child of its parent then
                LABEL(n)=1
        else
                LABEL(n)=0
        else
begin
        let n1 ,n2 ,n3,,,nk be the children of n ordered by LABEL , so
        LABEL(n1)>=LABEL(n2)>=… … … LABEL(nk)
        LABEL(n)=max(LABEL(ni)+i –1)
End
```

38.Define live variable.(Nov/Dec 2012)

A variable is live at a point in a program if its value can be used subsequently.

39.What are the different storage allocation strategies?

1) Static allocation.-It lays out storage for all data objects at compile time.
2) Stack allocation.-It manages the runtime storage as a stack.
3) Heap allocation.-It allocates and de-allocates storage as needed at runtime from a data area.

40.What is the use of Next-use information?(Nov/Dec 2013)

- If a register contains a value for a name that is no longer needed, we should re-use that register for another name (rather than using a memory location)
- So it is useful to determine whether/when a name is used again in a block
- Definition: Given statements i, j, and variable x,
  - If i assigns a value to x, and
  - j has x as an operand, and
  - No intervening statement assigns a value to x,
  - Then j uses the value of x computed at i.

41.How liveness variable calculated.(April/May 2015)

A variable is live if it holds a value that will/might be used in the future. The representation of the program that we use for liveness analysis (determining which variables are live at each point in a program), is a control flow graph. The nodes in a control flow graph are basic statements (instructions). There is an edge from statement x to statement y if x can be immediately followed by y (control flows from x to y).

42.Write the algorithm that orders the DAG nodes for generating optimal target code?.(April/May 2015)

```
(1)     while unlisted interior nodes remain do begin
(2)         select an unlisted node n, all of whose parents have been listed ;
(3)           list n;
(4)           while the leftmost child m of n has no unlisted parents
                and is not a leaf do
                    /* since n was just listed , m is not yet listed*/
                     begin
(5)                          list m;
(6)                           n = m
                end
    end
```

PART –B

1.Explain the principle sources of optimization in detail.

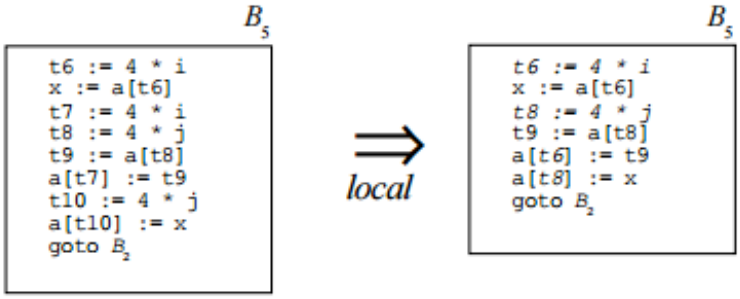
Transformations can be
- Local : look within basic block
- Global : look across blocks

• Transformations should preserve function of program.
• Function-preserving transformations include
- Common sub expression elimination

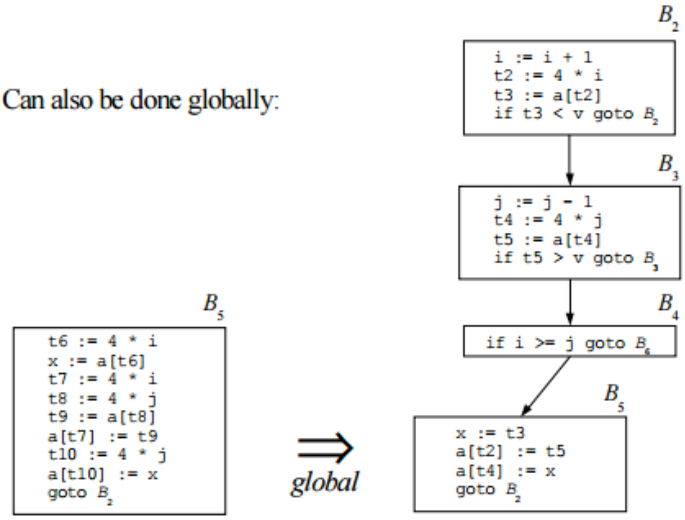
– Copy propagation
– Dead-code elimination
– Constant-folding

Common Sub expression Elimination

• Occurrence of expression E is called common sub expression if
– E was previously computed, and
– values of variables in E have not changed since previous Computation

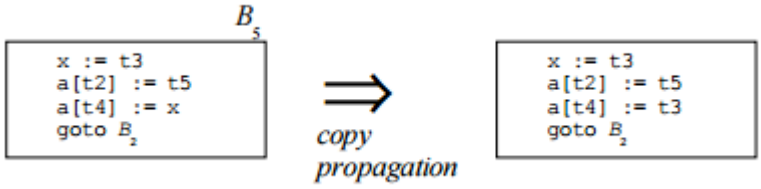


Can also be done globally:



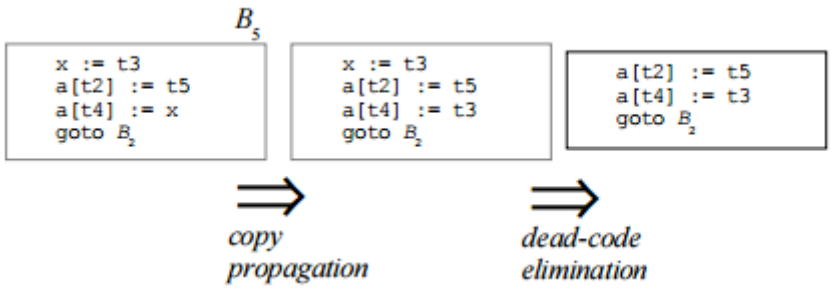
Copy Propagation

• Statement of form f := g is called a copy statement
• Idea isto use g instead of f in subsequent statements
• Doesn't help by itself, but can combine with other transformations to help eliminate code:
\



Dead-Code Elimination

• Variable that is no longer live (subsequently used) is called dead.
• Copy propagation often turns copy statement into dead code:



Loop Optimizations

• Biggest speedups often come from moving code out of inner loop
• Three techniques
– Code motion
– Induction-variable elimination
– Reduction in strength

Code Motion

• Expression whose value doesn't change inside loop is called a loop-invariant
• Code motion moves loop-invariants outside loop

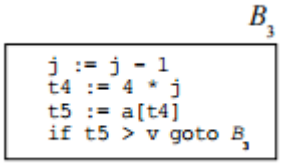
```
while (i <= limit-2)
```

⇓ *code motion*

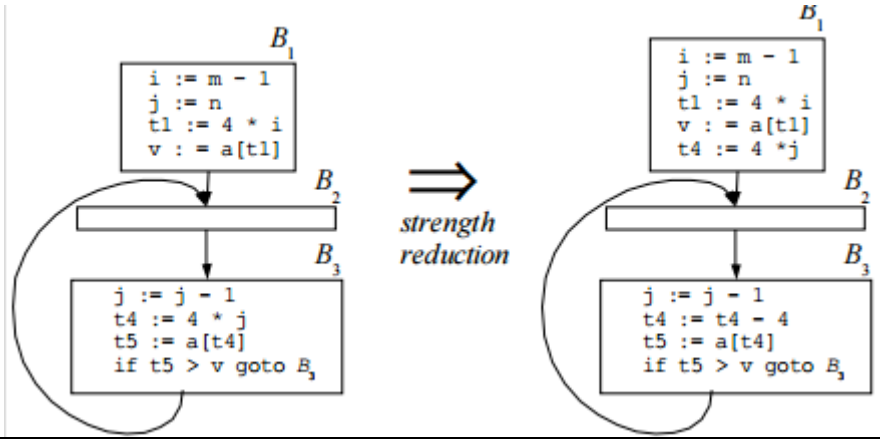
```
t = limit - 2;
while (i <= t)
```

Induction Variables and Reduction in Strength

• Variables that remain "in lock step" with each other inside a loop are called induction variables
• E.g., decreasing array byte-offset index by 4 as loop variable decreases by 1:



• Addition is like multiplication, "reduced in strength" (less costly)
• Exploit induction variables and reduction -in-strength to make loop code more efficient

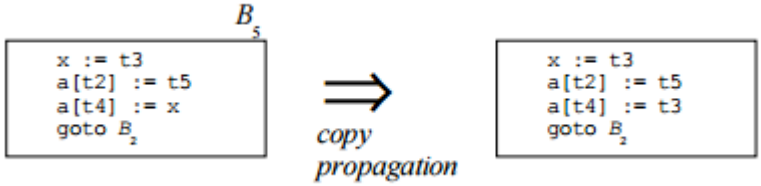


2. Discuss about the following:

i). Copy Propagation
ii) Dead-code Elimination
iii) Code motion.

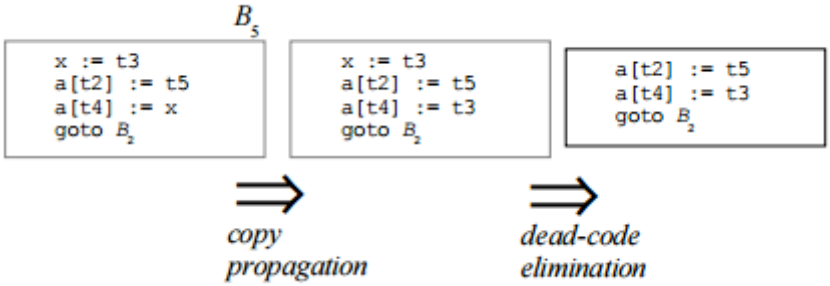
Copy Propagation

• Statement of form f := g is called a copy statement
• Idea is to use g instead of f in subsequent statements
• Doesn't help by itself, but can combine with other transformations to help eliminate code:
\



Dead-Code Elimination

• Variable that is no longer live (subsequently used) is called dead.
• Copy propagation often turns copy statement into dead code:



Code Motion

• Expression whose value doesn't change inside loop is called a loop-invariant
• Code motion moves loop-invariants outside loop

```
while (i <= limit-2)
```

code motion

```
t = limit - 2;
while (i <= t)
```

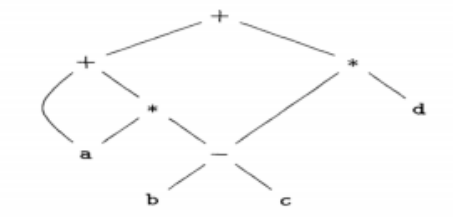
3(i).Explain optimization of basic blocks.

- It is a linear piece of code.
- Analyzing and optimizing is easier.
- Has local scope - and hence effect is limited.
- Substantial enough, not to ignore it.
- Can be seen as part of a larger (global) optimization problem

DAG representation of basic blocks

DAG representation of expressions leaves corresponding to atomic operands, and interior nodes corresponding to operators. A node N has multiple parents - N is a common sub expression.

Example: (a + a * (b - c)) + ((b - c) * d)



- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block.
- The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labelled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block;

(ii)Explain redundant common subexpression elimination.

common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyses whether it is worthwhile replacing them with a single variable holding the computed value.

In the following code:

*a = b * c + g;*

*d = b * c * e;*

it may be worth transforming the code to:

*tmp = b * c;*

a = tmp + g;

*d = tmp * e;*

if the time cost (savings) of storing and retrieving "tmp" outweighs the cost of calculating "b * c" an extra time.

The possibility to perform CSE is based on available expression analysis (a data flow analysis). An expression b*c is available at a point p in a program if: every path from the initial node to p evaluates b*c before reaching p, and there are no assignments to b or c after the evaluation but before p. The cost/benefit analysis performed by an optimizer will calculate whether the cost of the store to tmp is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.Compiler writers distinguish two kinds of CSE:

- local common sub expression elimination works within a single basic block
- global common sub expression elimination works on an entire procedure,

Both kinds rely on data flow analysis of which expressions are available at which points in a program.

4.Write about data flow analysis of structural programs.

Data-flow analysis derives information about the dynamic behavior of a program by only examining the static code.

Liveness Analysis

Definition

– A variable is live at a particular point in the program if its value at that point will be used in the future (dead, otherwise).

∴ To compute liveness at a given point, we need to look into the future Motivation: Register Allocation

– A program contains an unbounded number of variables

– Must execute on a machine with a bounded number of registers

– Two variables can use the same register if they are never in use at the same time (i.e, never

simultaneously live).

∴ Register allocation uses liveness information

```
1       a := 0
2  L1:  b := a + 1
3       c := c + b
4       a := b * 2
5       if a < 9 goto L1
6       return c
```

5.Optimize the following code using various optimization techniques:

i=1,s=0;
for(i=1;i<=3;i++)
for(j=1;j<=3;j++)
c[i][j]=c[i][j]+a[i][j]+b[i][j];

```
L1:  t1 = i * N
     t2 = t1 + j
     t3 = t2 * 4
     t4 = &c + t3
     t12 = t1 + k
     t13 = t12 * 4
     t14 = &a + t13
     t21 = k * N
     t22 = t21 + j
     t23 = t22 * 4
     t24 = &b + t23
     t31 = *t14 * *t24
     *t4 = *t4 + t31
     k = k + 1
     if( k < N) goto L1
    t1 = i * N
    t2 = t1 + j
    t3 = t2 * 4
    t4 = &c + t3
L1: t12 = t1 + k
    t13 = t12 * 4
    t14 = &a + t13
    t21 = k * N
    t22 = t21 + j
    t23 = t22 * 4
    t24 = &b + t23
    t31 = *t14 * *t24
    *t4 = *t4 + t31
    k = k + 1
    if( k < N) goto L1
```

6.(i)Explain the issues in design of code generator.

Issues in the Design of Code generator

- Memory management.
- Instruction Selection.

- Register Utilization (Allocation).
- Evaluation order.

1. Memory Management

Mapping names in the source program to address of data object is cooperating done in pass 1 (Front end) and pass 2 (code generator).

Quadruples → address Instruction.

Local variables (local to functions or procedures) are stack-allocated in the activation record while global variables are in a static area.

2. Instruction Selection

The nature of instruction set of the target machine determines selection.

-"Easy" if instruction set is regular that is uniform and complete.

Uniform: all triple addresses

all stack single addresses.

Complete: use all register for any operation.

If we don't care about efficiency of target program, instruction selection is straight forward.

For example, the address code is:

a := b + c

d := a + e

Inefficient assembly code is:

```
MOV b, R0        R0 ← b
ADD  c, R0         R0 ← c + R0
MOV R0, a        a  ← R0
MOV a, R0        R0 ← a
ADD  e, R0       R0 ← e + R0
MOV R0 , d       d  ← R0
```

Here the fourth statement is redundant, and so is the third statement if 'a' is not subsequently used.

3. Register Allocation

Register can be accessed faster than memory words. Frequently accessed variables should reside in registers (register allocation). Register assignment is picking a specific register for each such variable.

Formally, there are two steps in register allocation:

Register allocation (what register?)

This is a register selection process in which we select the set of variables that will reside in register.

Register assignment (what variable?)

Here we pick the register that contain variable. Note that this is a NP-Complete problem.

Some of the issues that complicate register allocation (problem).

1. Special use of hardware for example, some instructions require specific register.
2. Convention for Software:

For example

Register R6 (say) always return address.

Register R5 (say) for stack pointer.

Similarly, we assigned registers for branch and link, frames, heaps, etc.,

3. Choice of Evaluation order

Changing the order of evaluation may produce more efficient code.

This is NP-complete problem but we can bypass this hindrance by generating code for quadruples in the order in which they have been produced by intermediate code generator.

```
ADD x, Y, T1
ADD a, b, T2
```

is legal because X, Y and a, b are different (not dependent).

(ii)Explain peephole optimization.

Constant folding – Evaluate constant subexpressions in advance.
Strength reduction – Replace slow operations with faster equivalents.
Null sequences – Delete useless operations.
Combine operations – Replace several operations with one equivalent.
Algebraic laws – Use algebraic laws to simplify or reorder instructions.
Special case instructions – Use instructions designed for special operand cases.
Address mode operations – Use address modes to simplify code.

7.Explain the simple code generator with a suitable example

code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the completed processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the backend) needs to change from target to target.

Example

```
MOV R0 x,R0
MOV R1 y,R1
MUL R0,R1
MOV t1, R0
MOV R0 t1,R0
MOV R1 z,R1
ADD R0,R1
MOV t2, R0
MOV R0 x,R0
MOV R1 x,R1
ADD R0,R1
MOV x, R0
MOV R0 y,R0
MOV R1 y,R1
SUB R0,R1
MOV y, R0
OUT x
MOV z,y
OUT z
```

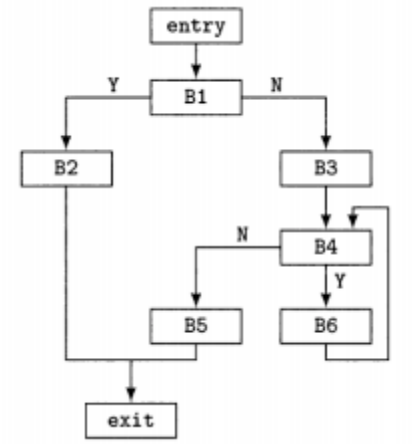
8.Write detailed notes on Basic blocks and flow graphs.

Basic blocks

A graph representation of intermediate code.

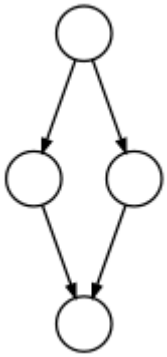
Basic block properties

- The flow of control can only enter the basic block through the first instruction in the block.
- No jumps into the middle of the block.
- Control leaves the block without halting / branching (except may be the last instruction of the block).
- The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.



flow graphs

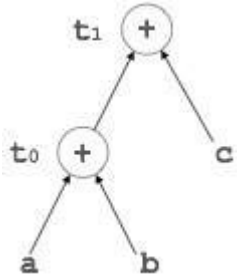
A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.



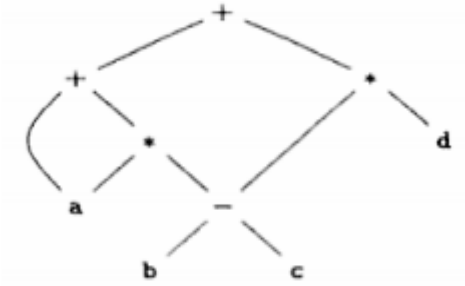
9.Define a Directed Acyclic Graph. Construct a DAG and write the sequence of instructions for the expression a+a*(b-c)+(b-c)*d. (May/June 2014)

DAG

- A representation to assist in code reordering.
  - Nodes are operations
  - Edges represent dependences
- Nodes are labeled as follows:
- Leaves with variables or constants – subscript 0 are used to distinguish initial value of the variable from other values.
- Interior nodes with operators and list of variables whose values are computed by the node.



DAG for a+a*(b-c)+(b-c)*d



10.Explain register allocation and assignment.

Register Allocation

• Better approach = register allocation: keep variable values in registers as long as possible

• Best case: keep a variable's value in a register throughout the lifetime of that variable

– In that case, we don't need to ever store it in memory

– We say that the variable has been allocated in a register

– Otherwise allocate variable in activation record

– We say that variable is spilled to memory

Register Allocation Algorithm

Hence, basic algorithm for register allocation is:

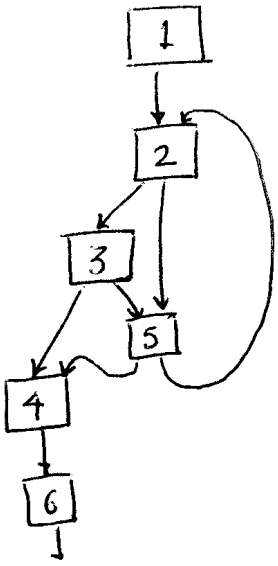
1. Perform live variable analysis (over abstract assembly code!)
2. Inspect live variables at each program point
3. If two variables are ever in same live set, they can't be allocated to the same register – they interfere with each other
4. Conversely, if two variables do not interfere with each other, they can be assigned the same register. We say they have disjoint live ranges.

11(i).Explain loops in flow graphs.

A subgraph of CFG with the following properties:

– Strongly Connected: there is a path from any node in the loop to any other node in the loop; and

– Single Entry: there is a single entry into the loop from outside the loop. The entry node of the loop is called the loop header.



Loop nodes: 2, 3, 5

Header node: 2

Loop back edge: 5→2

Tail→Head

(ii).Explain Local optimization.

Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for i = 1 to N
x = x + 1
for j = 1 to N
a(i,j) = 100*N + 10*i + j + x
```

• If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
x = x + 1
for j = 1 to N
a(i,j) = 100*N + 10*i + j + x
```

12.(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)

Goal: Generate optimal code for broad class of register machines

Machine Model:

- k interchangeable registers r0, r1, . . . , rk−1.
- Instructions are of the form ri := E, where E is an expression containing operators, registers, and memory locations (denoted M)
- Every instruction has an associated cost, measured by C()

Cost Vector: C(E) = (c0 c1 · · · cr) — it's defined for an expression E, where:

- C0: cost of computing E into memory, with the use of unbounded number of regs
- Ci: cost of computing E into a register, with the use of up to i regs

(ii)What are the advantages of DAG representation ? Give example.(4) (April/May 2015)

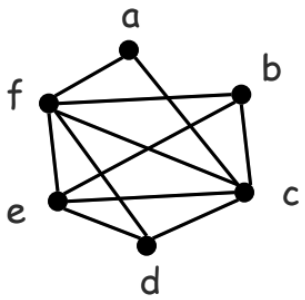
- Determining the common sub-expressions.
- Determining which names are used inside the block and computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- Simplifying the list of quadruples by eliminating the common su-expressions and not performing

13.(i)Write the procedure to perform Register Allocation and Assignment with Graph Coloring.(8)

- Two passes are used
  - Target-machine instructions are selected as though there are an infinite number of symbolic registers
  - Assign physical registers to symbolic ones
    - Create a register-interference graph
    - Nodes are symbolic registers and edges connects two nodes if one is live at a point where the other is defined.
    - For example in the previous example an edge connects a and d in the graph

Use a graph coloring algorithm to assign registers.**The Register Interference Graph**

• Two temporaries that are live simultaneously cannot be allocated in the same register
• We construct an undirected graph
– A node for each temporary
– An edge between t1 and t2 if they are live simultaneously at some point in the program
• This is the register interference graph (RIG)
– Two temporaries can be allocated to the same register if there is no edge connecting them
example:

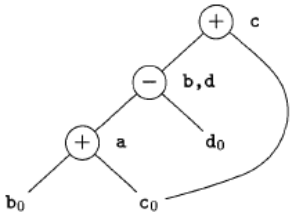


E.g., b and c cannot be in the same register
• E.g., b and d can be in the same register

(ii)Construct DAG and optimal target code for the expression X=((a+b)/(b-c))-(a+b)*(b-c)+f. (April/May 2015).(8)

- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block.

```
a = b + c
b = a - d
c = b + c
d = a - d
```



- In this section we assume we are using an n-register machine with instructions of the form
    - LD reg, mem
    - ST mem, reg
    - OP reg, reg, reg

to evaluate expressions.

- An expression tree is a syntax tree for an expression.

o Numbers, called Ershov numbers, can be assigned to label the nodes of an expression tree. A node gives the minimum number of registers needed to evaluate on a register machine the expression generated by that node with no spills. A spill is a store instruction that gets generated when there are no empty registers and a register is needed to perform a computation.

o Algorithm to label the nodes of an expression tree

1. Label all leaves 1.
2. The label of an interior node with one child is the label of its child.

3. The label of an interior node with two children is the larger of the labels of its children if these labels are different; otherwise, it is one plus the label of the left child.

14.Perform analysis of available expressions on the following code by converting into basic blocks and compute global common sub expression elimination.

- ***I:=0***
- ***A:=n-3***
- ***If i<a then loop else end***
- ***Label loop***
- ***B:=i_4***
- ***E:=p+b***
- ***D:-m[c]***
- ***E:=d-2***
- ***F:=I-4***
- ***G:=p+f***
- ***M[g]:=e***
- ***I:=i+1***
- ***A:=n-3***
- ***If i<a then loop else end***
- ***Label end*** ***(April/May 2015)***

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

```
main()
{
 int x, y, z;

 x = (1+20)* -x;
 y = x*x+(x/y);
 y = z = (x/y)/(x*x);
}
```

straight translation:

```
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
```

tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
Here is an optimized version, after constant folding andpropagation and elimination of common sub-expressions:
tmp2 = -x ;
x = 21 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
z = tmp5 / tmp3 ;
y = z ;

15.(i) Explain Loop optimization in details and apply it to the code (10)

I:=0
A:=n-3
If i<a then loop else end
Label loop
B:=i_4
E:=p+b
D:-m[c]
E:=d-2
F:=I-4
G:=p+f
M[g]:=e
I:=i+1
A:=n-3

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform.

Induction variable analysis

If a variable in a loop is a simple linear function of the index variable, such as j := 4*i + 1, it can be updated appropriately each time the loop variable is changed. This is a strength reduction, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.

(ii)What are the optimization technique applied on procedure calls? Explain with example .(6) (April/May 2015)

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack. We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.