MODULE 1

Overview of System Software

1.1 Motivation:

To provide the students with the knowledge of

- How to design & implement of various types of system software
- Differentiation between application software & system software.
- Different types of system software
- Application of system software & application software
- Study of different types of Translators

1. 2. Learning Objective:

- To learn various system software's such as assemblers, loaders, linkers, macro processors, compiler, interpreters, operating systems, device drivers.
- Differentiate between application software & system software.

1.3 Syllabus:

Lecture	Content	Duration	Self Study
1	**Introduction to System Software** with examples, Software Hierarchy, Differentiate between system software and application software	2 hrs	2 hrs
2	**Introduction to Language Processors**: Compiler, Assembler, Interpreter.	2 Hrs	2 Hrs

1.4 Learning Outcomes:

After studying this chapter, students able to:

- Compare between system software & application software.
- Differentiate between system software's & application software are available in the market.
- Understand the application of all these software's.

1.5 Definitions

- **Compiler:** is a translator (program) which convert source program (Written in C,C++ etc) into machine code.
- **Assembler:** is a translator (program) which convert source program (Written in Assembly) into machine code.
- **Interpreter:** is a translator (program) which convert source program line by line into Intermediate code, which it then executes.
- **Operating System:** is a program which provide interface or communication between hardware & software.
- **Loader:** A program routine that copies a program into memory for execution.
- **Linker:** is a program that combines object modules to form an executable program.
- **Macro Processor: is a program which is responsible for processing the macros.**
- **Device Drivers:** is a program that controls a particular type of device that is attached to your computer.
- **Software:** is a program. Program is a group of instructions.

1.6 Key Definitions:

OS: Operating System

HLL: High Level Language

ALP: Assembly Language Programming

1.7 Course Content:

Lecture 1 & 2

1.7.1 Types of Software:

Software is generally divided into:

a) System software

b) Application software

a) System software are programs which help in the running of a computer system

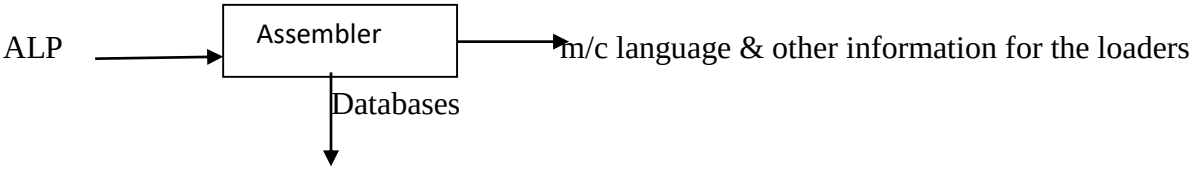
E.g. Disc operating programs, OS, Compiler etc.

b) Application software are programs which perform specific tasks for the user.

E.g. Word processing software, Graphics package, Theatre booking software.

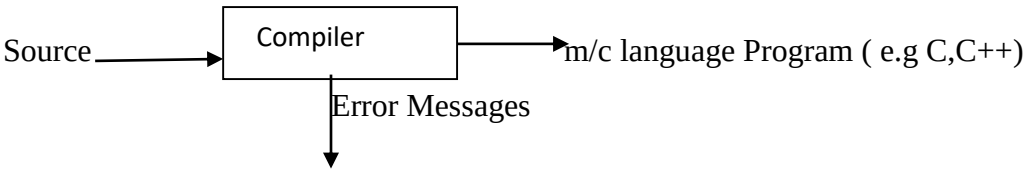
Assembler:

An assembler is a program that accepts as input an assembly language program & produces its machine language equivalent along with information for the loader.



Compiler:

A compiler is a program that reads an input in HLL & translates it into an equivalent program in machine language.



Phases of Compiler: Compiler operates in different phases.\

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator

Other- Symbol table & Error Handler

Interpreter:

An interpreter is a translator that reads the program written in HLL & translate line by line into an intermediate form, which it then executes.

1.7.2 Difference between Compiler, Interpreter & Assembler:

Sr. No	Assembler	Compiler	Interpreter
1.	It converts ALP into m/c language.	It converts HLL program into m/c language.	It converts HLL program line by line in intermediate form, which it then executes.
2.	It assemble whole source program.	It compile whole source program	It translate line by line.
3.	Speed of execution is fast.	Speed of execution is fast.	Speed of execution is slow.
4.	Program need to assemble only once & can be executed repeatedly.	Program need to compiled only once & can be executed repeatedly	For every run of program, program need to be translated.
5.	It creates an object file.	It creates an object file.	It does not creates an object file.
6.	It require large memory space to store (But less than compiler).	It require large memory space to store.	It require less memory space to store.
7.	e.g. Microsoft assembler (MASM)	MS-DOC C compiler.	Basic Interpreter.
8.	Source language- Assembly	Source language- C, C++	Source language- BASIC, LISP

Loader:

Loader is a system program which is responsible for preparing the object program for execution & initiates the execution. OR

A program routine that copies a program into memory for execution.

OR

Operating System utilities that copy program from a storage device to main memory, where they can be execute. In addition to copying a program into main memory, the loader can also replace virtual addresses with physical addresses.

Function of Loader:

a) Allocation
b) Linking
c) Relocation
d) Loading

Types of Loaders:

a) Assemble (Compile) & go loader
b) Absolute loader
c) Bootstrap loader
d) Direct linking loader

Linker:

Linker is a program that combines object modules to form an executable program.

Also called Link editor & binder.

Many programming languages allow you to write different pieces of code, called modules, separately.

In addition to put all the modules, a linker also replaces symbolic addresses with real addresses. Therefore, you need to link a program even if it contains only one.

Operating System:

Operating system is system software, consisting of program and data that runs on computer and manage the computer hardware.

It is an integrated set of programs that controls the resources of a computer system and provides its users with an interface that is easier to use.

Objective of OS:

-Make a computer system easier to use

-Manage the resources of a computer system

Function of OS

a) Process management: It takes care of creation and deletion of processes.

b) Memory management: It takes care of allocation and de-allocation of memory space to programs in need of this resource.

c) File management: It takes care of file-related activities.

d) Security: It protects the information of a computer system against unauthorized access.

e) I/ O Management.

f) Communication

g) Accounting

(1)Process management: A process is a program in execution. It is the job, which is currently being executed by the processor. During its execution a process would require certain system resources such as processor, time, main memory, files etc. OS supports multiple processes simultaneously. The process management module of the OS takes care of the creation and termination of the processes, assigning resources to the processes, scheduling processor time to different processes and communication among processes.

(2)Memory management module: It takes care of the allocation and de-allocation of the main memory to the various processes. It allocates main and secondary memory to the system/user program and data. To execute a program, its binary image must be loaded into the main memory. Operating System decides.

(a) Which part of memory are being currently used and by whom.

(b) which process to be allocated memory.

(c) Allocation and de allocation of memory space.

(3)I/O management: This module of the OS co-ordinates and assigns different I/O devices namely terminals, printers, disk drives, tape drives etc. It controls all I/O devices, keeps track of I/O request, issues command to these devices.

I/O subsystem consists of

(i) Memory management component that includes buffering, caching and spooling.

(ii) Device driver interface

(iii) Device drivers specific to hardware devices.

(4)File management: Data is stored in a computer system as files. The file management module of the OS would manage files held on various storage devices and transfer of files from one device to another. This module takes care of creation, organization, storage, naming, sharing, backup and protection of different files.

(5)Scheduling: The OS also establishes and enforces process priority. That is, it determines and maintains the order in which the jobs are to be executed by the computer system. This is so because the most important job must be executed first followed by less important jobs.

(6)Security management: This module of the OS ensures data security and integrity.

That is, it protects data and program from destruction and unauthorized access. It keeps different programs and data which are executing concurrently in the memory in such a manner that they do not interfere with each other.

(7)Processor management: OS assigns processor to the different task that must be performed by the computer system. If the computer has more than one processor idle, one of the processes waiting to be executed is assigned to the idle processor.

OS maintains internal time clock and log of system usage for all the users. It also creates error message and their debugging and error detecting codes for correcting programs

Types of OS:

1. Multiprogramming: More than one job reside in the memory & all are ready for execution.

2. Multiprocessing: is the simultaneous execution of two or more processes by a computer system having more than one CPU.

3. Multitasking: Switch from one task to another in small fraction of time.

Technically it is same as multiprogramming. But multiprogramming for multi-user systems (systems that are uses simultaneously by many users such as mainframe system) and multitasking for single-user systems (systems that are used by only one user at a time).

4. Batch processing: batch processing system is a one where programs data are collected together in a batch before processing start.

5. Multithreading: Allows different parts of a single program to run concurrently.

6. Real-time: Responds to input instantly.

7.5 Device Drivers:

- A device driver is a program that controls a particular type of device that is attached to your computer.
- There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on.
- When you buy an OS, many device drivers are built into the product.

- A device driver essentially converts the more general input/output instructions of the operating system to messages that the device type can understand.
- Some Windows programs are Virtual Device Driver. These programs interface with the Windows Virtual Machine Manager. There is a virtual device driver for each main hardware device in the system, including the hard disk drive controller, keyboard, and serial and parallel ports. They're used to maintain the status of a hardware device that has changeable settings. Virtual device drivers handle software interrupts from the system rather than hardware interrupts.
- In Windows operating systems, a device driver file usually has a file name suffix of DLL or EXE. A virtual device driver usually has the suffix of VXD.

1.8 . References:

D. M. Dhamdhere, "Systems programming & Operating Systems".

1.9 Multiple Choice Questions.

Q.1 Translator for low level programming language termed as

(A) Assembler **(B)** Compiler

(C) Linker **(D)** Loader

Q.2 The translator which perform macro expansion is called a

(A) Macro processor **(B)** Macro pre-processor

(C) Micro pre-processor **(D)** assembler

Q.3 Shell is the exclusive feature of

(A) UNIX **(B)** DOS

(C) System software **(D)** Application software

Q.4 An assembler is

(A) programming language dependent. **(C)** machine dépendant.

(B) syntax dépendant. **(D)** data dependant.

Q.5 Which of the following loader is executed when a system is first turned on or restarted

(A) Boot loader **(B)** Compile and Go loader

(C) Bootstrap loader **(D)** Relating loader

Q.6 A linker program

(A) Places the program in the memory for the purpose of execution.

(B) Relocates the program to execute from the specific memory area allocated to it.

(C) Links the program with other programs needed for its execution**.**

(D) Interfaces the program with the entities generating its input data.

Q. 7 An assembly language is a

(A) low level programming language

(B) Middle level programming language

(C) High level programming language

(D) Internet based programming language

Q.8 Which of the following are language processors?

(A) Assembler **(B)** Compiler

(C) Interpreter **(D)** All of the above

Q.9 Which is not a computer translator?

(A) Compiler (B)Assembler

© Interpreter (D) Word processor

Q.10 Does Interpreter generate object file?

a) Yes

b) No

1.10 Short questions

Q.1 Differentiate between system software & application software.

a) System software are programs which help in the running of a computer system e.g. Disc operating programs, OS, Compiler etc.

b) Application software are programs which perform specific tasks for the user. e.g. Word processing software, Graphics package, Theatre booking software.

Q.2 Define system software.

A.2 Refer Answer No.1

Q.3 Define OS. Which are the different functions of OS?

A. Refer topic no. 7.2

Q.4 Give examples of application and system software

A. For application software examples are web browser, editors.

For system software examples are assemblers, macro-processor linkers, loaders, interpreters, compilers, operating system, device drivers.

Q.5 Which are the different types of OS?

Refer answer No. 3

Q.6 Explain all system software in detail.

Q.7 What are the basic functions of language translator.

1.14. Long Questions:

Q.1 what is system programming? Explain the evolution of system software.

Ans: System software is collection of system programs that perform a variety of functions, viz file editing, recourse accounting, IO management, storage management etc.System programming is the activity of designing and implementing SPs.System programs which are the standard component of the s/w of most computer systems; The two fold motivation mentioned above arises out of single primary goal via of making the entire program execution process more effective.

Q.2 Differentiate between application program and system program. Indicate the order in which following system program are used, from developing program upto its execution. Assembler, loaders, Linkers, macroprocessor, compiler, editor.

Module 2

Introduction to Compilers and Lexical Analysis

4.1. Motivation:

- To provide the students with the knowledge about compiler and its phases

4.2. Learning Objective: Students will be able to :

- Define and compare the Compiler and Interpreter
- Illustrate the different phases of Compiler with the help of example

4.3. Syllabus:

Prerequisites	Syllabus	Duration	Self Study
Fundamental of Programming language	**Introduction to Compilers**: Design issues, passes, phases. **Lexical Analysis:** The Role of a Lexical analyzer, Input buffering, specification and recognition of tokens, Automatic construction of lexical analyzer using LEX	2Hr	2 Hr

4.4. Learning Outcomes: Students should be able to:

- Differentiate between compiler and Interpreter
- Learn/Understand working of compiler

Lecture 20

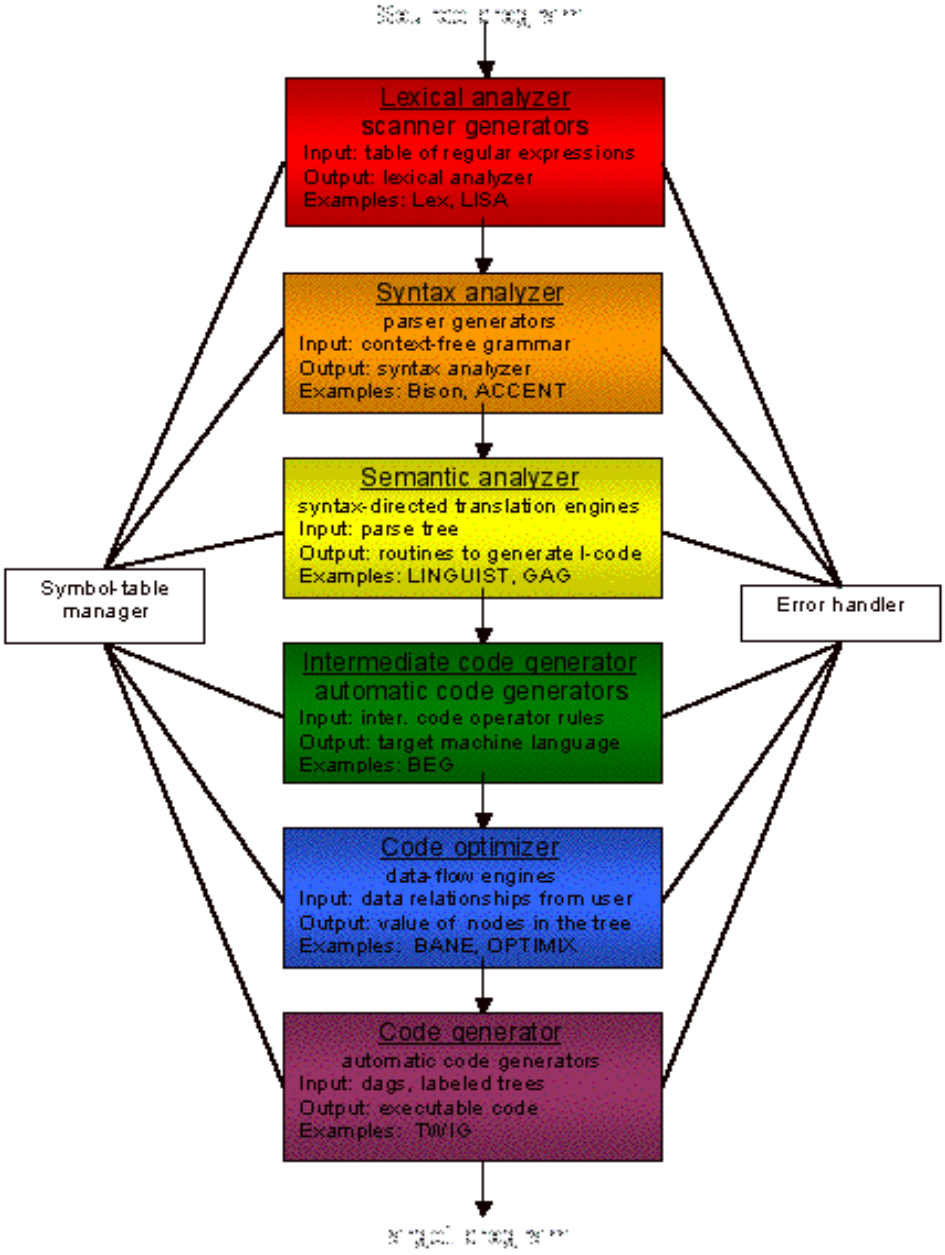
Compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code)*. The most common reason for wanting to transform source code is to create an executable program.

Interpreters

An *interpreter* is also a program that translates a high-level language into a low-level one, but it does it at the moment the program is run. You write the program using a text editor or something similar, and then instruct the interpreter to run the program. It takes the program, one line at a time, and translates each line before running it: It translates the first line and runs it, then translates the second line and runs it etc. The interpreter has no "memory" for the translated lines, so if it comes across lines of the program within a loop, it must translate them afresh every time that particular line runs.

Lecture 21 Phases of Compiler



1. **Analysis Phase :**

Analysis Phase performs 3 actions namely

a) Lexical analysis - it contains a sequence of characters called tokens. Input is source program & the output is tokens.

b) Syntax analysis - input is token and the output is parse tree

c) Semantic analysis - input is parse tree and the output is expanded version of parse tree

2 .Synthesis Phase :

Synthesis Phase performs 3 actions namely

d) Intermediate Code generation - Here all the errors are checked & it produce an intermediate code.

e)Code Optimization - the intermediate code is optimized here to get the target program

f) Code Generation - this is the final step & here the target program code is generated.

Lecture: 22

Role of a Lexical analyzer, input buffering, specification and recognition of tokens

Learning Objective: In this lecture student will able to design Lexical Analyzer.

4.9.1 Role of lexical Analyzer:

Lexical Analyzer is a program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**.

Lexical grammar

The specification of a programming language will often include a set of rules which defines the lexer. These rules are usually called regular expressions and they define the set of possible character sequences that are used to form individual tokens or lexemes**.**

Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type

Scanner

The **scanner**, is usually based on a finite state machine. It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes)

Tokenization

It is the process of demarcating and possibly classifying sections of a string of input characters

Finite-state machine (**FSM**)

F**inite-state automaton** or simply a **state machine**, is a mathematical abstraction sometimes used to design digital logic or computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similar to a flow graph in which one can inspect the way logic runs when certain conditions are met. It has finite internal memory, an input feature that reads symbols in a sequence, one at a time without going backward; and an output feature, which may be in the form of a user interface, once the model is implemented. The operation of an FSM begins from one of the states (called a *start state*), goes through transitions depending on input to different states and can end in any of those available, however only a certain set of states mark a successful flow of operation (called *accept states*).

Deterministic finite state machine

It also known as **deterministic finite automaton** (**DFA**)—is a finite state machine accepting finite strings of symbols. For each state, there is a transition arrow leading out to a next state for each symbol. Upon reading a symbol, a DFA jumps *deterministically* from a state to another by following the transition arrow. A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.

Nondeterministic finite state machine

Nondeterministic finite automaton (NFA) is a finite state machine where for each pair of state and input symbol there may be several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. Although the DFA and NFA have distinct definitions, it may be shown in the formal theory that they are equivalent, in that, for any given NFA, one may construct an equivalent DFA, and vice-versa.

Let's check the take away from this lecture

Q 1. Postponing of the resolving of some undefined symbols until a program is run is called as

A. Dynamic Linking

B. Dynamic loading

C. Linking

D. Loading

Q 2 Automaton where the next possible state is uniquely determined is called as

A. NFA

B. DFA

C. Turing Machine

Answers: 1) D 2) B

L21 & 22. Exercise:

1) Explain with example NFA and DFA

2) Define Finite state automata. What is their role compiler theory? Explain in Detail

3)Write Short note on Lexical Analysis

Questions/problems for practice:
GATE Exam Questions

1. The number of tokens in the following C statement is

printf("i = %d, &i = %x", i, &i);

(A) 3
(B) 26
(C) 10
(D) 21

Answer: (C)

2.In a compiler, keywords of a language are recognized during
(A) parsing of the program

(B) the code generation
(C) the lexical analysis of the program
(D) dataflow analysis

Answer: (C)

3. The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense?
(A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-Deterministic pushdown automata
(D) Turing Machine

Answer: (A)

Learning from the lecture 'Role of a Lexical analyzer, input buffering, specification and recognition of tokens". Student will able to list and explain the role of Lexical Analyzer in compiler Design.

4.8 **References:**

1. A.V. Aho, a nd J.D.Ullman: Principles of compiler construction,

Pearson Education

2 . A.V. Aho, R. Shethi and Ulman; Compilers - Principles, Techniques and

Tools , *Pearson Education*

3 Leland Beck " System Software " *Addision Wesley*

4. D. M. Dhamdhere; Systems programming & Operating systems , *Tata*

McGraw Hill

4.9. Question Bank

- **Objective Questions**

Q1 In a compiler the module that checks every character of the source text is called

A the code generator

B the code optimizer

C the lexical analyzer

D the syntax analyzer.

2. Which one is not function of compiler?

A. A compiler does a conversion line by line as the program is run

B. A compiler converts the whole of a higher level program code into machine code in one step

C. A compiler is a general purpose language providing very efficient execution

D. It does not report the error

Q 3. Which one is not the stage in the compilation process?

A. Syntax analysis

B. Symantic analysis

C. Code generation

D. Error reporting

Q 4. Following is not purpose of interpreter

A. An interpreter does the conversion line by line as the program is run

B. An interpreter is a representation of the system being designed

C. An interpreter is a general-purpose language providing very efficient execution

D. An interpreter does not perform the conversion line by line as the program is run

Q 5 Following is not the token

- Identifier

- Keyword
- Number
- Function

Answer: Q1-C, Q-2 D, Q3-D, Q4-D Q5-D

4.9 . Short Questions

1) What is Compiler? Explain its phases
2) What is Dynamic linking and loading
3) Compare compiler and interpreter
4) What is compiler? Draw and Explain structure of compiler
5) What are the phases of Compiler? Explain

4.10 Long Questions

1. What are the different phases of compiler? Illustrate compilers internal presentation of source program for following statement after each phase [Nov 2016]

Position = initial + Rate * 60

2. What are the different phases of compiler? Illustrate compilers internal presentation of source program for following statement after each phase. [May 2016]

Amount = P+ P * N* R/100

3. What is the difference between compiler and interpreter? [May 2016]

Module 3

Parsing

5.1 Motivation:

- Motivation of this chapter is to study & design of different Top-Down & Bottom-Up parsing techniques.
- After studying this module students can easily develop a parser or a syntax analyzer phase of a compiler.
- How to design intermediate code by using syntax directed translation
- Detail study of syntax directed definitions & translation scheme.
- The fundamental knowledge about Lexical analysis.
-

5.2. Syllabus:

Lecture	Content	Duration	Self Study
23	**Syntax Analysis:** Role of Parser	1 Lecture	2 hours
24	Top-down parsing	1 Lecture	2 hours
25 & 26	Recursive descent and predictive parsers (LL)	2 Lecture	4 hours
27 & 28	Bottom-Up parsing	2 Lecture	4 hours
29	Operator precedence parsing	1 Lecture	2 hours
30	LR parsers	1 Lecture	2 hours
31 & 32	SLR and LALR parsers	2 Lecture	2 hours
33	Automatic construction of parsers using YACC.	2 Lecture	2 hours
34	Introduction to Semantic Analysis: Need of semantic analysis, type checking and type conversion	1 Lecture	2 hours

4.3. Weightage: 20 Marks

4.4. Learning Objectives: Students should be able to-

1. List the functions of Lexical Analyzer. Describe the role of Lexical Analyzer in Compiler Design. (R)

2. Design and Develop hand written Lexical Analyzer and show the Demonstration of working of lexical analyzer in compiler design . (A)

3. Describe the role of parser in compilation process. Explain different top down and Bottom-up parsing techniques. (E)

4. Specify various parsing techniques to design new language structures with the help of grammars.**(C)**

5. Explain the construction and role of the syntax tree in the context of Parse tree.**(U)**

6. Distinguish between Parse tree , Syntax tree and DAG for graphical representation of the source program. (U)

7. Summarize different Compiler Construction tools and Describe the structure of Lex specification. (AN)

8. Apply LEX Compiler for Automatic Generation of Lexical Analyzer and Construct Lexical analyzer using open source tool for compiler design.(C)

9. Define Context Free Grammar and Describe the structure of YACC specification and Apply YACC Compiler for Automatic Generation of Parser Generator. (U)

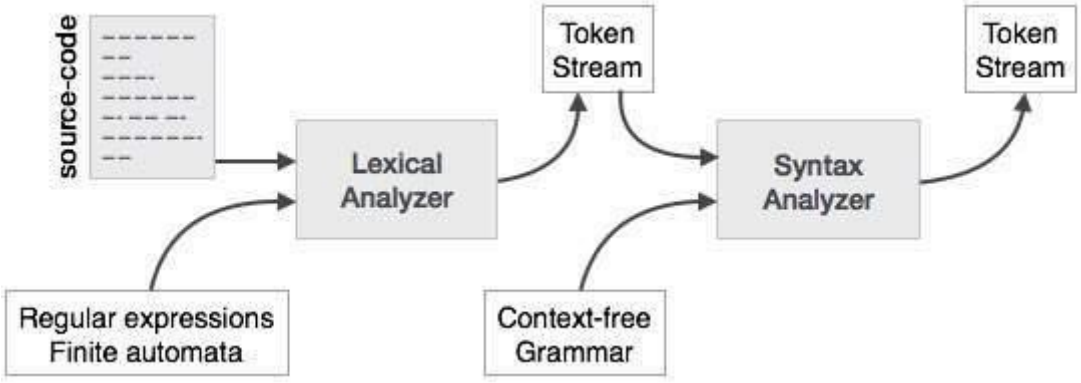
4.5. Theoretical Background:

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase. Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies.

4.6. Abbreviations:

LL(1):

"L" – left-to-right scan of input

"L" – leftmost derivation

"1" – predict based on one token look-ahead

For every non-terminal and token **predict** the next production

LR(k):

"L" – left-to-right scan of input

"L" – rightmost derivation

"k" – predict based on token look-ahead

For every non-terminal and token **predict** the next production.

SLR: simple LR parser

LR: most general LR parser(canonical LR)

LALR: intermediate LR parser (look-head LR parser)

CFG: Context Free Grammar

CLR: Canonical LR

SDD: **Syntax-Directed Definitions**

4.7. Formulae: Nil

4.8. Key Definitions:

Parsing: Parsing is the process of determining, if a string of tokens can be generated by a grammar. Or **Parsing** is the task of determining the syntax or structure of a program for this reason it is called as a syntax analysis.

Grammar: Grammar is a set of rules which check correctness of sentences.

Syntax: The rules used to form sentence is called syntax.

Semantics:-The meaning of the sentence.

Language: The set of rules denote set of valid sentence, such a set of valid sentence is called language.

Leftmost Derivation: Derivation in which only the leftmost non terminal in any sentential form is replaced at each step. Such derivation is called leftmost derivation.

Rightmost Derivation: Derivation in which only the rightmost nonterminal in any sentential form is replaced at each step. Such derivation is called rightmost derivation.

Handle of a string: Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.

Parse tree: Graphical representation of a derivation ignoring replacement order.

Annotated Parse Tree : A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

Annotating (or decorating) of the parse tree: The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

A syntax directed definition : specifies the values of attributes by associating semantic rules with the grammar productions.

Production	Production
E->E1+T	E.code=E1.code||T.code||'+'

Detail Syntax-Directed Definition:-

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form: $b=f(c_1,c_2,\ldots,c_n)$ where f is a function, and b can be one of the followings:

  - ➔ b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

  - ➔ b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production.

  ($A \rightarrow \alpha$).

Inherited Attributes: An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node.

A**ttribute grammar:-** An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

S-Attributed Definitions: only synthesized attributes used in the syntax-directed definitions.

L-Attributed Definitions: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

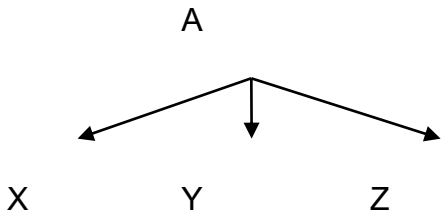
Syntax Tree: A **syntax tree** is a more condensed version of the parse tree useful for representing language constructs.

Detail definition of Parse Tree:

Given a CFG, a parse tree according to the grammar is a tree with following properties.

- The root of the tree is labeled by the start symbol
- Each leaf of the tree is labeled by a terminal (=token) or ε
  Each interior node is labeled by a nonterminal
- If $A \rightarrow X_1\ X_2\ \ldots\ X_n$ is a production, then node A has immediate children $X_1, X_2, \ldots, X_n$ where X_i is a (non)terminal or ε (ε denotes the empty string)

Example :$A \rightarrow XYZ$



4.9. Course Content:

Lecture: 23

Syntax Analysis: Role of Parser

Learning Objective: In this lecture student will able to list different role of Parser.

4.9.2 The role of parser:

1) Determine the syntactic structure of a program from the tokens produces by the scanner i.e. to check validity of source program.

2) For valid string build a parse tree.

3) For invalid string diagnostic error messages for the cause & nature of error.

Parsing technique:

a) Top Down parsing

b) Bottom up parsing

Let's check the take away from this lecture

1) Syntax analysis also called…

a)parsing b)scanning

2) Which is the Top down parsing technique?

a) LR(0) b)Recursive decent parser

c)LALR parser d)LL(1)

L23. Exercise:

Q.1) Differentiate between top-down parser & Bottom-up parser.

Q.2) Write Role of Parser in compiler Design.

Questions/problems for practice:

Q. 1 List the Phases of Compiler.

Learning from the lecture 'Role of Parser':

Student will able to list and explain the role of Parser in compiler Design.

Lecture: 24

Top-down parsing

Learning objective: In this lecture students will able to design Top Down Parser.

4.9.3 Top Down parser:

- Builds parse tree from top (root) to work down to the bottom(leaves).
- Top down parsing techniques:

  1) Recursive Decent parser

  2) Predictive 0r LL (1) parser

Rules of top down parser:

1) Grammar should not be left recursive.

2) It should be left factor.

Left Recursion:

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \Rightarrow A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

  Rule-

  $A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A'\ |\ \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\ \alpha_1 \mid ... \mid A\ \alpha_m \mid \beta_1 \mid ... \mid \beta_n$ where $\beta_1 ... \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\ A' \mid ... \mid \beta_n\ A'$

$A' \rightarrow \alpha_1\ A'\ \mid ... \mid \alpha_m\ A'\ \mid \varepsilon$ an equivalent grammar

Left-Recursion – Example

$E \rightarrow E+T\ |\ T$

$T \rightarrow T*F\ |\ F$

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

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T'\ \mid \varepsilon$

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

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

  stmt $\rightarrow$ if expr then stmt else stmt | if expr then stmt

- when we see if, we cannot now which production rule to choose to re-write *stmt* in the derivation.

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or A to $\alpha\beta_2$

  But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

  $A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_m$

convert it into

$A \rightarrow \alpha A' \mid \gamma_1 \mid ... \mid \gamma_m$

$A' \rightarrow \beta_1 \mid ... \mid \beta_n$

Left-Factoring-example

S --> iEtS | iEtSeS | a

E->b

can be rewritten as

S -->iEtSS' | a

S'-->ε | eS

E → b

Let's check the take away from this lecture

1) Which is the most powerful parsing technique?

a)LL(1) b)LR(0)

c)LR(1) d)LALR

2) Predictive parser support backtracking method..

a)Yes b)No

3) Which Grammar is called LL(1) Grammar?

a) two adjacent non-terminals at the right side.

b) parsing table with no multiply-defined entries

c)Ambiguous grammars

d)Non-ambiguous grammars

4) Which data structure used to parse the string?

a)Stack b)Queue

c)Array d)LinkList

L24. Exercise

Q.1) Check following grammar is Left recursive or not? If yes remove left recursion.

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

A.1) This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}da$ or

$\underline{A} \Rightarrow Sd \Rightarrow Aad$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar.

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$\Downarrow$ eliminate left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Q.2)Check following grammar is left recursive & Left factored or not.

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

A.2) Grammar is not left recursive .

But Grammar is not left factored. So make left factored.

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Learning from this lecture 'Top Down Parsing':
Students will able to find Left recursion and Left Factored of the given Grammar.

Lecture: 25 & 26

Recursive descent and predictive parsers (LL)

Learning Objective: In this lecture students will able to design Recursive Decent and Predictive Parser(LL).

4.9.3 Example on Predictive or LL(1) Parser:-

a. Steps:

1. Check grammar is left recursive or not.
2. Check grammar is left factored or not.
3. Find FIRST & FOLLOW set to construct Predictive parser table.
4. Constuct Predictive parser table.
5. Check grammar is LL(1) or not.
6. Parse i/p string

b. Compute FIRST as follows:

- Let a be a string of terminals and non-terminals.
- First (a) is the set of all terminals that can begin strings derived from a.

Compute FIRST(X) as follows:

a) if X is a terminal, then FIRST(X)={X}

b) if $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)

c) if X is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

c. Compute FOLLOW as follows:

a) FOLLOW(S) contains EOF

b) For productions $A \rightarrow \alpha B \beta$, everything in FIRST($\beta$) except ε goes into FOLLOW(B)

c) For productions $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

d .Constructing LL(1) Parsing Table – Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)

    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A) ➔ add $A \rightarrow \alpha$ to M[A,\$]
  - All other undefined entries of the parsing table are error entries.

e. Example on LL(1) or Predictive Parser:

Original grammar:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

This grammar is left-recursive, ambiguous and requires left-factoring. It needs to be modified before we build a predictive parser for it:

Step 1: Remove Ambiguity.

$E \rightarrow E+T$

$T \rightarrow T*F$

$F \rightarrow (E)$

$F \rightarrow id$

Grammar is left recursive hence Remove left recursion:

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E)$

$F \rightarrow id$

Step 2: Grammar is already left factored.

Step 3: Find First & Follow set to construct predictive parser table:-

FIRST(E) = FIRST(T) = FIRST(F) = {**(**, **id**}

FIRST(E') = {+, ε}

FIRST(T') = {***, ε}

FOLLOW(E) = FOLLOW(E') = {**$**, **)**}

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

FOLLOW(F) = {***, +, **$**, **)**}

Step 4: Now, we can either build a Predictive parser table:

Parsing table

	Id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Step 5: Parser table not contain any multiple defined entries. Hence, given grammar is LL(1) grammar.

Step 6: Parse the input id*id using the parse table and a stack

Step	Stack	Input	Next Action
1	**$**E	**id*id$**	E→TE'
2	**$**E'T	**id*id$**	T→FT'
3	**$**E'T'F	**id*id$**	F→**id**
4	**$**E'T'**id**	**id*id$**	match **id**
5	**$**E'T'	***id$**	T'→***FT'

6	**$**T'F*****	***id$**	match *****
7	**$**T'F	**id$**	F→**id**
8	**$**T'**id**	**id$**	match **id**
9	**$**T'	**$**	T'→ε
10	**$**	**$**	accept

Let's check the take away from this lecture

1) A left recursive grammar:

a) cannot be LL(1) b) cannot be LR(1)

c) an infinite set d) none of the above

2) True| false questions:

a) Every regular grammar is context free grammar.- **TRUE**

b)Every LL(1) grammar is SLR(1) also. - **FALSE**

c) Every unambiguous grammar belong to the class of either SLR, CLR or LALR.- **FALSE**

d)If a grammar G is SLR(1) then it is definitely LALR(1).-**TRUE**

L24 Exercise:

Q.1 Check following grammar is LL(1) or not.

$S \rightarrow i\ C\ t\ S\ E \mid a$

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

Questions/problems for practice:

Q.1 For the grammar having productions:

$A \rightarrow (A)A \mid \varepsilon$

Compute FIRST & FOLLOW set of A.

A.1 FIRST (A) = { (, ε }

FOLLOW (A) = {$,)}

Q.2 Is the following grammar LL(1)

S → aSa | ε

Learning from the lecture 'Recursive Decent and Predictive Parser(LL)':   Student will able to design Top Down Parser.

Lecture: 27 & 28

Bottom-Up parsing and Operator Precedence Parser

Learning Objective: In this lecture students will able to design Bottom – Up Parsing.

4.9.5 Bottom Up Parsing:

- A general style of bottom-up syntax analysis, known as shift-reduce parsing.
- Two types of bottom-up parsing:

  7.3.1 Operator-Precedence parsing

  7.3.2 LR parsing

Operator-Precedence Parser:

Operator-Precedence Parsing Algorithm

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

set p to point to the first symbol of w$;

repeat forever

if ($ is on top of the stack **and** p points to $) **then return**

else {

let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;

if (a $\lessdot$ b or a $\doteq$ b) **then** { /* SHIFT */

push b onto the stack;

advance p to the next input symbol;

}

else if (a $\gtrdot$ b) **then** /* REDUCE */

repeat pop stack

until (the top of stack terminal is related by $\lessdot$ to the terminal most recently popped);

else error();

}

- **Operator grammar**

  In an *operator grammar*, no production rule can have:

  1)ε at the right side

  2) Two adjacent non-terminals at the right side.

- Example

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |

B→b	O→+|*|/	E/E | id
not operator grammar	not operator grammar	perator grammar

- **Precedence Relations:**

1) In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a $<\cdot$ b b has higher precedence than a

a $=\cdot$ b b has same precedence as a

a $\cdot>$ b b has lower precedence than a

2) The determination of correct precedence relations between terminals are based on the traditional notions of associatively and precedence of operators. (Unary minus causes a problem).

3) The intention of the precedence relations is to find the handle of a right-sentential form,

$<\cdot$ with marking the left end,

$=\cdot$ appearing in the interior of the handle, and

$\cdot>$ marking the right hand.

4) In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

The partial operator-precedence table for this grammar

5) Then the input string id+id*id with the precedence relations inserted will be: \$ $<\cdot$ id $\cdot>$ + $<\cdot$ id $\cdot>$ * $<\cdot$ id $\cdot>$ \$

	id	+	*	$
Id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Advantages & Disadvantages of Operator Precedence Parsing:-

Disadvantages:

1) It cannot handle the unary minus (the lexical analyzer should handle the unary minus).

2) Small class of grammars.

3) Difficult to decide which language is recognized by the grammar.

Advantages:

1) Simple

2) Powerful enough for expressions in programming languages

Let's check the take away from this lecture

1. Analysis which determines the meaning of a statement once its grammatical structure becomes known is termed as

(A) Semantic analysis **(B)** Syntax analysis

(C) Regular analysis **(D)** General analysis

L26 & 27 Exercise:

Q.1 Explain Operator Precedence Parser with the help of Example.

Q.2 Write short note on : Virtual 8086 mode

Questions/problems for practice:

Q.1 Design Operator Precedence Parser for Arithmetic Operator.

Learning from the lecture 'Bottom up Parsing and Operator Precedence Parsing':
Student will able to Design Operator Precedence Parser.

Lecture: 29

LR parsers

Learning Objective: In this lecture students will able to Design LR Parser.

4.9.6 LR Parser:

a) LR (0) or SLR

b) LR (1) or Canonical LR

c) LALR or Look ahead LR

A)LR(0) or SLR Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

Construction of The Canonical LR(0) Collection:

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:

  C is { closure({S'→.S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

  goto function is a DFA on the sets in C.

  Steps:

  1) Find Augmented Grammar G'.

  2) Find item I.

  3) Find Closure operation.

  4) Find Goto operation.

  5) Construct Canonical Collection or LR(0) collection.

  6) Draw DFA.

  7) Find FIRST & FOLLOW set.

  8) Design SLR table.

  9) Check grammar is SLR or not.

  10) Parse the i/p string.

Step1.Augmented Grammar:

G’ is G with a new production rule S’→S where S’ is the new starting symbol.

E’-> E

E → E+T

E → T

T → T*F

T → F

F → (E)

F → id

Step 2.LR (0) Item:

An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side

- Ex: A → XYZ Possible LR(0) Items: A → .XYZ

  (four different possibility) A → X.YZ

  A → XY.Z

  A → XYZ.

- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items(**the canonical LR(0) collection**) is the basis for constructing SLR parsers.

Step 3.The Closer Operation:

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

    1. Initially, every LR (0) item in I is added to closure(I).
    2. If **$A \rightarrow \alpha.B\beta$** is in closure (I) and **$B \rightarrow \gamma$** is a production rule of G; then **$B \rightarrow .\gamma$** Will be in the closure (I). We will apply this rule until no more new LR (0) items can be added to closure (I).

The Closure Operation -- Example

$E' \rightarrow E$	closure ($\{E' \rightarrow .E\}$) =	
$E \rightarrow E+T$	{ $E' \rightarrow .E$	kernel items
$E \rightarrow T$	$E \rightarrow .E+T$	
$T \rightarrow T*F$	$E \rightarrow .T$	
$T \rightarrow F$	$T \rightarrow .T*F$	
$F \rightarrow (E)$	$T \rightarrow .F$	
$F \rightarrow id$	$F \rightarrow .(E)$	
	$F \rightarrow .id$	

Step 4.Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
    - If $A \rightarrow \alpha.X\beta$ in I then every item in **closure($\{A \rightarrow \alpha X.\beta\}$)** will be in goto(I,X).
        - If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

Example:

I ={ E' → .E, E → .E+T, E → .T,

T → .T*F, T → .F,

F → .(E), F → .id }

goto(I,E) = { E' → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

goto(I,F) = {T → F. }

goto(I,() = { F → (.E), E → .E+T, E → .T, T → .T*F, T → .F,

F → .(E), F → .id }

goto(I,id) = { F → id. }

Step 5.The Canonical LR(0) Collection – Example

I_0: E' → .E E → .E+T E → .T T → .T*F T → .F F → .(E) F → .id	I_1: E' → E. E → E.+T	I_6: E → E+.T T → .T*F T → .F F → .(E) F → .id	I_9: E → E+T. T → T.*F
	I_2: E → T. T → T.*F		I_{10}: T → T*F.
	I_3: T → F.	I_7: T → T*.F F → .(E)	I_{11}: F → (E).

I_4: $F \rightarrow (.E)$ $F \rightarrow .id$

$E \rightarrow .E+T$

$E \rightarrow .T$ I_8: $F \rightarrow (E.)$

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

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_5: $F \rightarrow id.$

Step 6. FIRST & FOLLOW set.

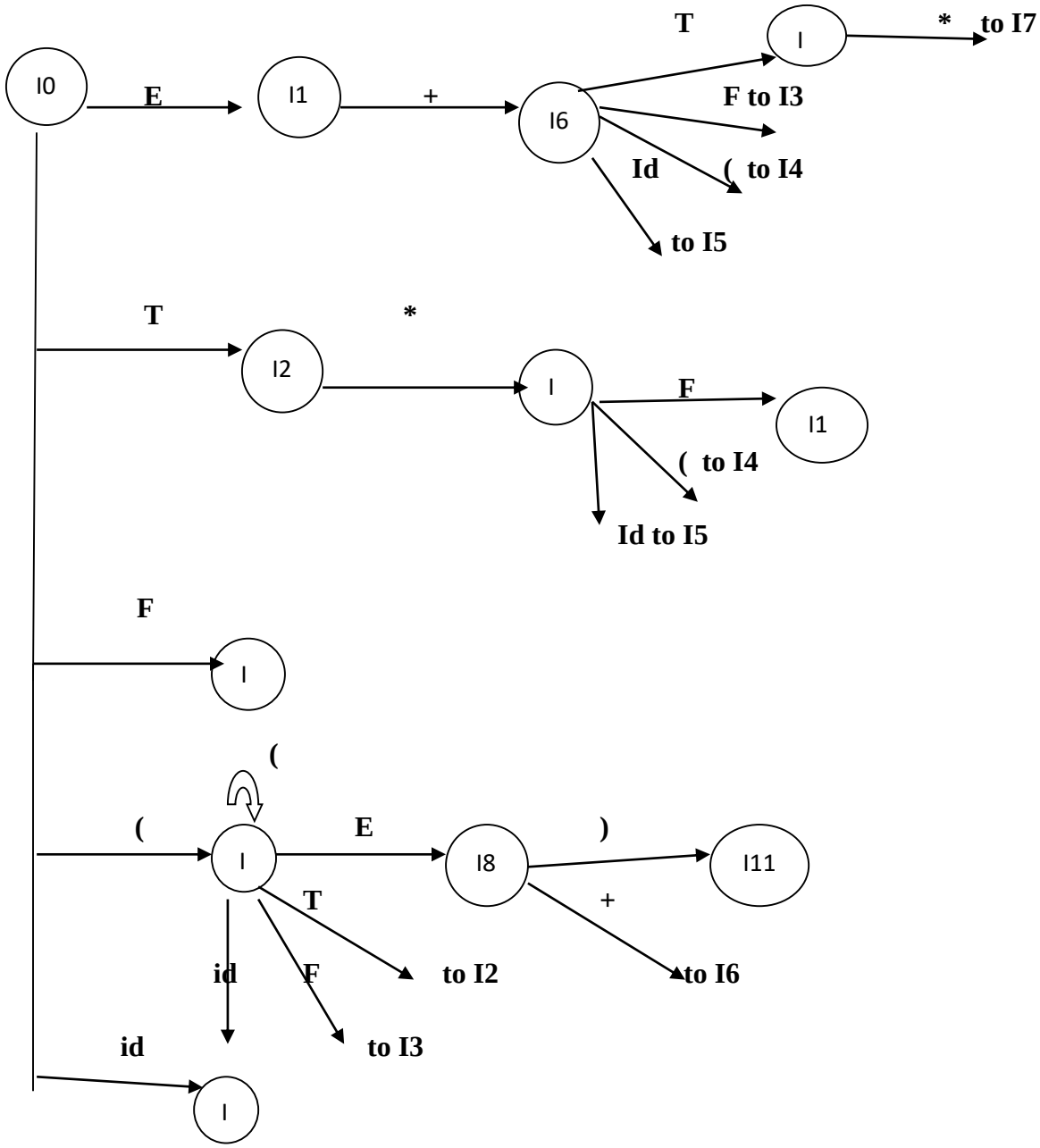
FIRST(E)= FIRST(T)= FIRST(F)={ (, id) }

FOLLOW(E)= { $, +,) }

FOLLOW(T)= {*, $, +,) }

FOLLOW(F)= {*, $, +,) }

Step 7: Transition Diagram (DFA) of Goto Function



Step 8. Design SLR parser table.

Action Table Goto Function

state	**id**	**+**	*****	**(**	**)**	**$**		**E**	**T**	**F**
0	S5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Step 9: There is no Multiple defined shift/Reduce entries in the parser table hence given grammar is LR(0) Grammar.

Step 10:Actions of a (S)LR-Parser -- Example

stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Let's check the take away from this lecture

1) ***yacc*** creates a ….. parser for the given grammar.

1)LALR 2)LR(0)

3)LR(1) 4)LL(1)

2) Which of the following conflicts can not arise in LR parsing:

a)shift-reduce b)reduce-reduce

c)shift-shift d)none of the above

3) If the grammar is LALR(1) then it is necessarily:

a)SLR(1) b)LR(1)

c)LL(1) d)None of the above

4) LR grammar is a:

a) Context free grammar b) Context sensitive grammar

c) Regular Grammar d) None of the above

5) YACC is a:

a) Lexical analyzer generator b) A parser generator

c) Semantic analyzer d) None of the above

L27 Exercise:

Q.1 Construct the LR(0) collection for following Arithmetic Grammar & construct LR(0) parser table:-

$E \rightarrow E+T$

$T \rightarrow T*F$

$F \rightarrow (E)$

$F \rightarrow id$

Question/ problems for practice

Q.1 Explain the LR parsers with suitable examples.

Learning from the Lecture 'LR Parser':
Student will able to design LR (0) Parser.

Lecture: 30 & 31

LR (1) and LALR parsers

Learning Objective: In this lecture students will able to Design LR(1) and LALR Parser

4.9.7 Example on LR(1) parser:

Steps:

1)Find Augmented Grammar G'.

2)Find LR(1) item.

3)Find Closure operation.

4)Find Goto operation.

5)Construct Canonical Collection.

6)Draw DFA.

7)Find FIRST & FOLLOW set.

8)Design LR(1) table.

9)Check grammar is LR1 or not.

10)Parse the i/p string.

LR (1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item

(**a** is a terminal or end-marker.)

- Such an object is called LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect in an item of the form [$A \rightarrow \alpha.\beta,a$], where β is not $\in$.
  - But an item of the form [$A \rightarrow \alpha.,a$] calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha.B\beta,a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation:

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

  If $A \rightarrow \alpha.X\beta,a$ in I

  then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection:

- ***Algorithm***:

  C is { closure({S'$\rightarrow$.S,\$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

if goto(I,X) is not empty and not in ***C***

add goto(I,X) to ***C***

goto function is a DFA on the sets in C.

An Example of LR(1) PARSER:

Grammar G:

$S \rightarrow C\ C$

$C \rightarrow c\ C$

$C \rightarrow d$

Step1:Augmented Grammar G'

1. $S' \rightarrow S$
2. $S \rightarrow C\ C$
3. $C \rightarrow c\ C$
4. $C \rightarrow d$

Step 2,3,4,5: Item & Closure & goto operations i.e. LR(1) Collection:

I_0: closure({$(S' \rightarrow \bullet\ S, \$)$}) =

$(S' \rightarrow \bullet\ S, \$)$

$(S \rightarrow \bullet\ C\ C, \$)$

$(C \rightarrow \bullet\ c\ C, c/d)$

$(C \rightarrow \bullet\ d, c/d)$

I_1: goto(I_1, S) = $(S' \rightarrow S \bullet , \$)$

I_2: goto(I_1, C) =

($S \rightarrow C \bullet C, \$$)

($C \rightarrow \bullet c\ C, \$$)

($C \rightarrow \bullet d, \$$)

I_3: goto(I_1, c) =

($C \rightarrow c \bullet C, c/d$)

($C \rightarrow \bullet c\ C, c/d$)

($C \rightarrow \bullet d, c/d$)

I_4: goto(I_1, d) =

($C \rightarrow d \bullet, c/d$)

I_5: goto(I_3, C) =

($S \rightarrow C\ C \bullet, \$$)

I_6: goto(I_3, c) =

($C \rightarrow c \bullet C, \$$)

($C \rightarrow \bullet c\ C, \$$)

($C \rightarrow \bullet d, \$$)

I_7: goto(I_3, d) =

($C \rightarrow d \bullet, \$$)

I_8: goto(I_4, C) =

($C \rightarrow c\ C \bullet, c/d$)

: goto(I_4, c) = I_4

: goto(I_4, d) = I_5

I_9: goto(I_7, c) =

($C \rightarrow c\ C \bullet$, $)

: goto(I_7, c) = I_7

: goto(I_7, d) = I_8

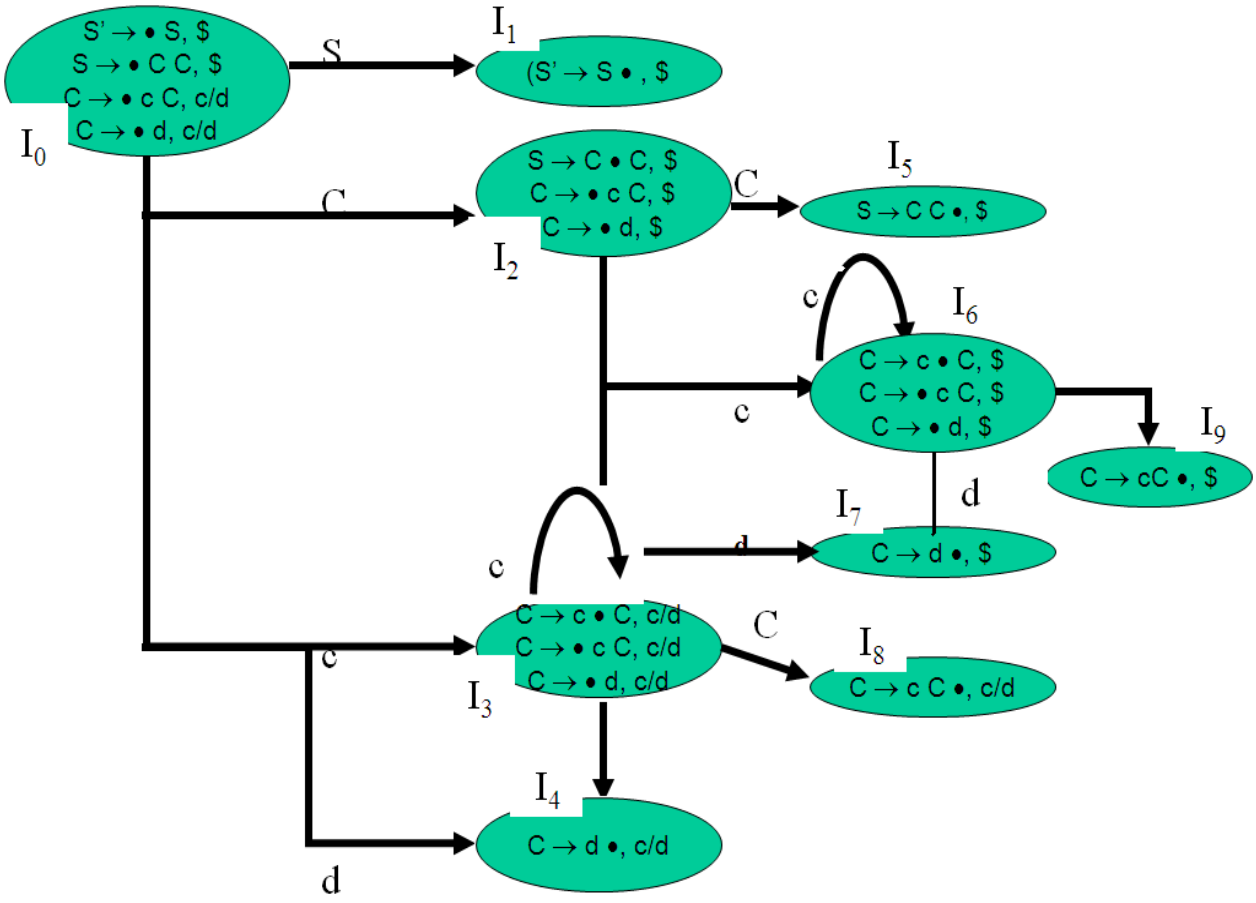
Step 6: Find FIRST & FOLLOW set.

FIRST (S)=FIRST(C)={ c , d }

FOLLOW(S)={ $ }

FOLLOW(C)={ c,d }

Step7: Construct DFA:



Step 8: Construct LR(1) Parsing Table:

Action Table Goto Function

States	c	D	$	S	C
I0	s3	s4		g1	g2
I1			acc		
I2	S6	S7			g5
I3	S3	S4			g8
I4	r3	r3			

I5			r1		
I6	S6	S7			g9
I7			r3		
I8	r2	r2			
I9			r2		

Step 9:There is no multiple defined entries in the LR(1) table hence this grammar is LR(1) grammar.

3)LALR or Lookahead parser:

LALR stands for **Lookahead LR.**

1. LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
2. The number of states in SLR and LALR parsing tables for a grammar G are equal.
3. But LALR parsers recognize more grammars than SLR parsers.
4. ***yacc*** creates a LALR parser for the given grammar.

A state of LALR parser will be again a set of LR(1) items.

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.
- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

  Consider I3 & I6 and replaced by their union:

I_3: goto(I_1, c) =

($C \rightarrow c \bullet C$, c/d)

($C \rightarrow \bullet c\ C$, c/d)

($C \rightarrow \bullet d$, c/d) =➔ I36: ($C \rightarrow c \bullet C$, c/d/$)

I_6: goto(I_3, c) = ($C \rightarrow \bullet c\ C$, c/d/$)

($C \rightarrow c \bullet C$, $) ($C \rightarrow \bullet d$, $)

($C \rightarrow \bullet c\ C$, $)

($C \rightarrow \bullet d$, $)

Consider I4 & I7 and replaced by their union:

I_4: goto(I_1, d) =

($C \rightarrow d \bullet$, c/d) ➔ I47: ($C \rightarrow d \bullet$, c/d/$)

I_7: goto(I_3, d) =

($C \rightarrow d \bullet$, $)

Consider I8 & I9 and replaced by their union:

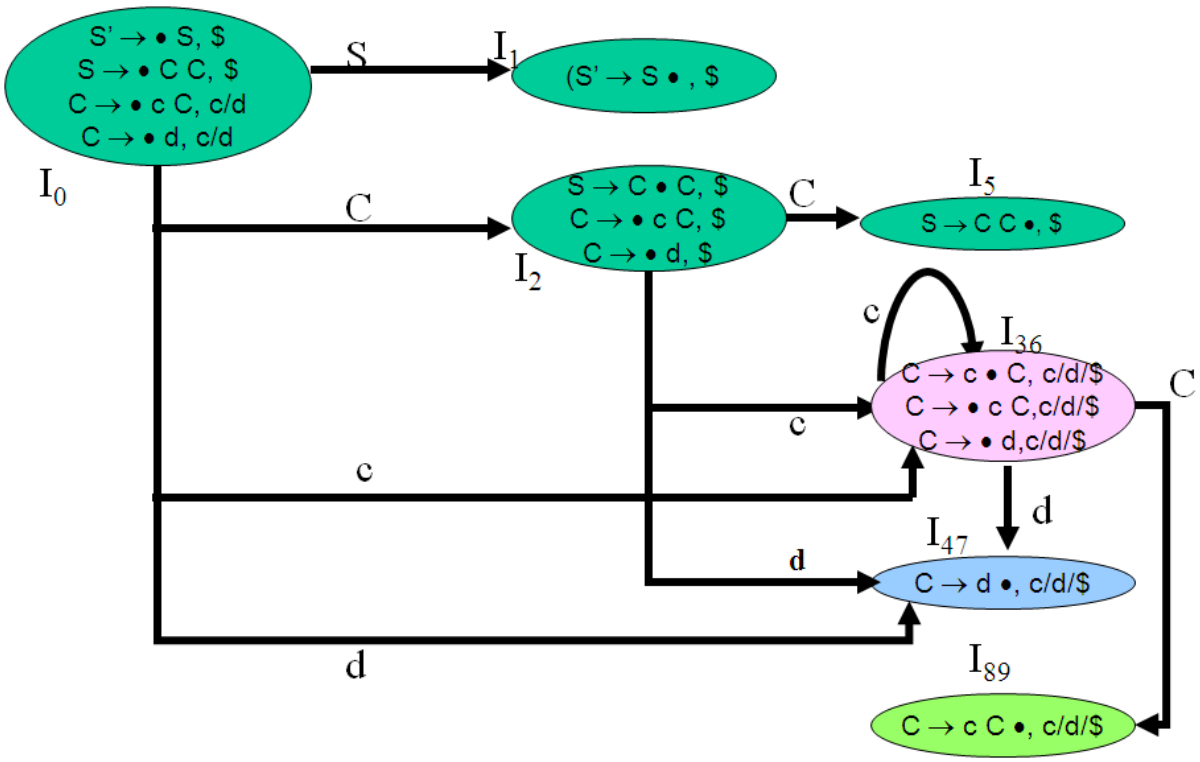
I_8: goto(I_4, C) =

($C \rightarrow c\ C \bullet$, c/d) ➔ I89: $C \rightarrow c\ C \bullet$, c/d/$

I_9: goto(I_7, c) =

($C \rightarrow c\ C \bullet$, $)

DFA:-



Creation of LALR Parsing Tables:

Action Table Goto Function

States	c	D	$	S	C
0	s36	s47		1	2
1			acc		

2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

If there is no parsing action conflicts ,then the given grammar is said to be an LALR Grammar.

Let's check the take away from this lecture

1) Which of the following conflicts can not arise in LR parsing:

a)shift-reduce b)reduce-reduce

c)shift-shift d)none of the above

2) If the grammar is LALR(1) then it is necessarily:

a)SLR(1) b)LR(1)

c)LL(1) d)None of the above

Exercise:

Q 1. Cosider the following grammar and construct the LALR parsing table.

S->AA

A-> aA | b

(Dec 2007) (10M)

Questions/problems for Practice

Q.1. Construct the LALR parsing table for the following grammar.

S->S

S-> CC

C-> cC | d

Learning from the lecture 'LR (1) and LALR': Student will able to design and Implement Bottom up Parser.

Let's check the take away from this lecture

Q.1 Which of the following eliminate common sub expression?

a)Parse tree

b) Syntax tree

c)DAG

Q.2 An inherited attribute is the one whose initial value at a parse tree node is defined in terms of:

a) attributes at the parent and /or siblings of that node

b) attributes at the children nodes only

c) attributes at both children nodes & parent and / or siblings of that node

d) none of the above

Exercise:

Q 1. Cosider the following grammar and construct the LALR parsing table.

S->AA

A-> aA | b

(Dec 2007) (10M)

Questions/problems for Practice

Q.1. Construct the LALR parsing table for the following grammar.

S->S

S-> CC

C-> cC | d

Learning from the lecture 'Syntax directed definitions' : Student will able to design and Implement Bottom up Parser.

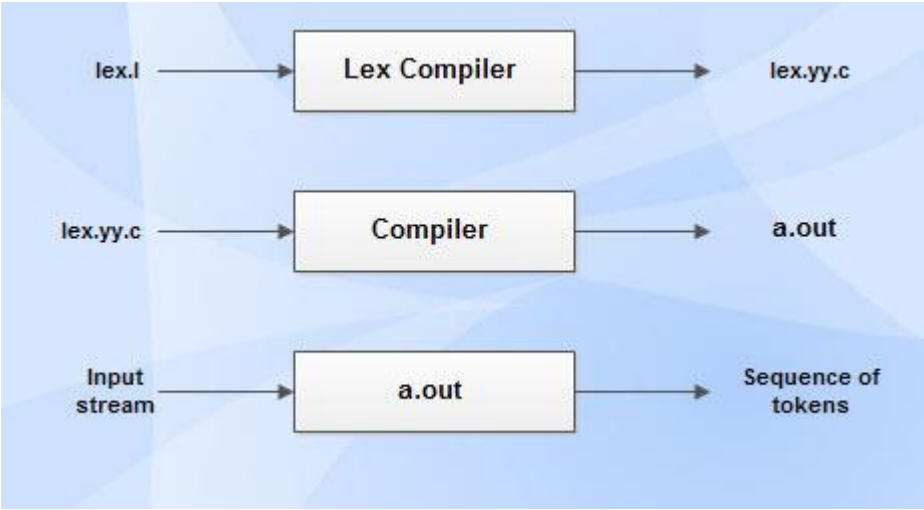
Lecture: 33

LEX Compiler

Learning Objective: In this lecture students will able to understand the working of LEX Compiler

6.9.6 LEX Compiler:

Lex is a tool in lexical analysis phase to recognize tokens using regular expression. Lex tool itself is a lex compiler.



Steps to generate LEXER

- **lex.l:** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- **lex.yy.c:** is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- **yylval** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

Structure of Lex Programs

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

Declarations This section includes declaration of variables, constants and regular definitions.

Translation rules It contains regular expressions and code segments.

Form : Pattern {Action}

Pattern is a regular expression or regular definition.

Action refers to segments of code.

Auxiliary functions This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.

Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.

Once a match is found, the associated action takes place to produce token.

The token is then given to parser for further processing.

6.9. 7 Let's check the take away from this lecture

- LEX tool is used to generate :
    a) Lexical Analyzer
    b) Syntax Analyzer
    c) ICG

L23 Exercise:

Q.2 **Write short note on: LEX Compiler**

Lecture: 34

YACC Compiler- Compiler

Learning Objective: In this lecture students will able to understand the working of YACC Compiler

6.9.6 YACC Compiler:

Why use Lex & Yacc ?

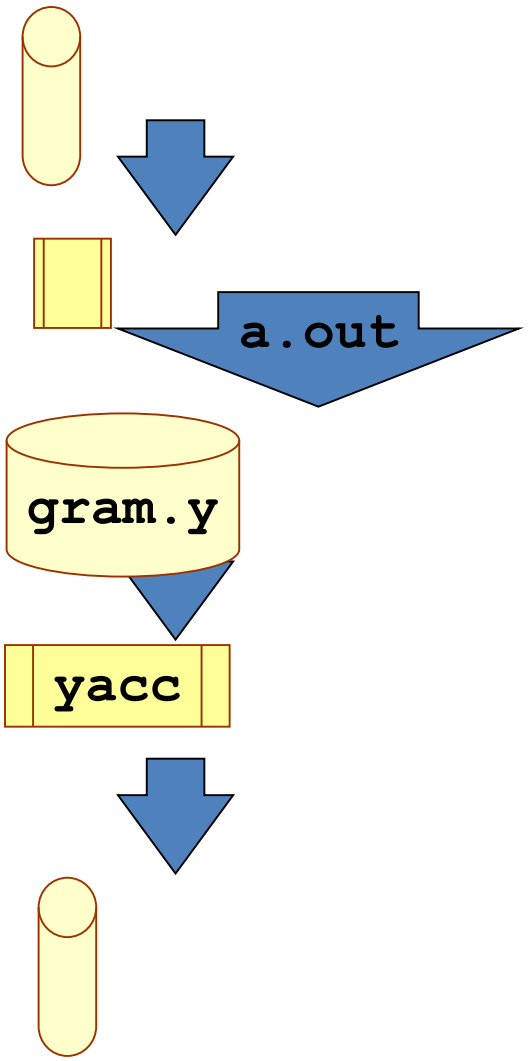
- Writing a compiler is difficult requiring lots of time and effort. Construction of the scanner and parser is routine enough that the process may be automated.

- **What is YACC ?**
  - Tool which will produce a parser for a given grammar.
  - YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.

- Original written by Stephen C. Johnson, 1975.
- Variants:
  - lex, yacc (AT&T)
  - bison: a yacc replacement (GNU)
  - flex: fast lexical analyzer (GNU)
  - BSD yacc

PCLEX, PCYACC (Abraxas Software)

How YACC Works?



File containing desired grammar in yacc format

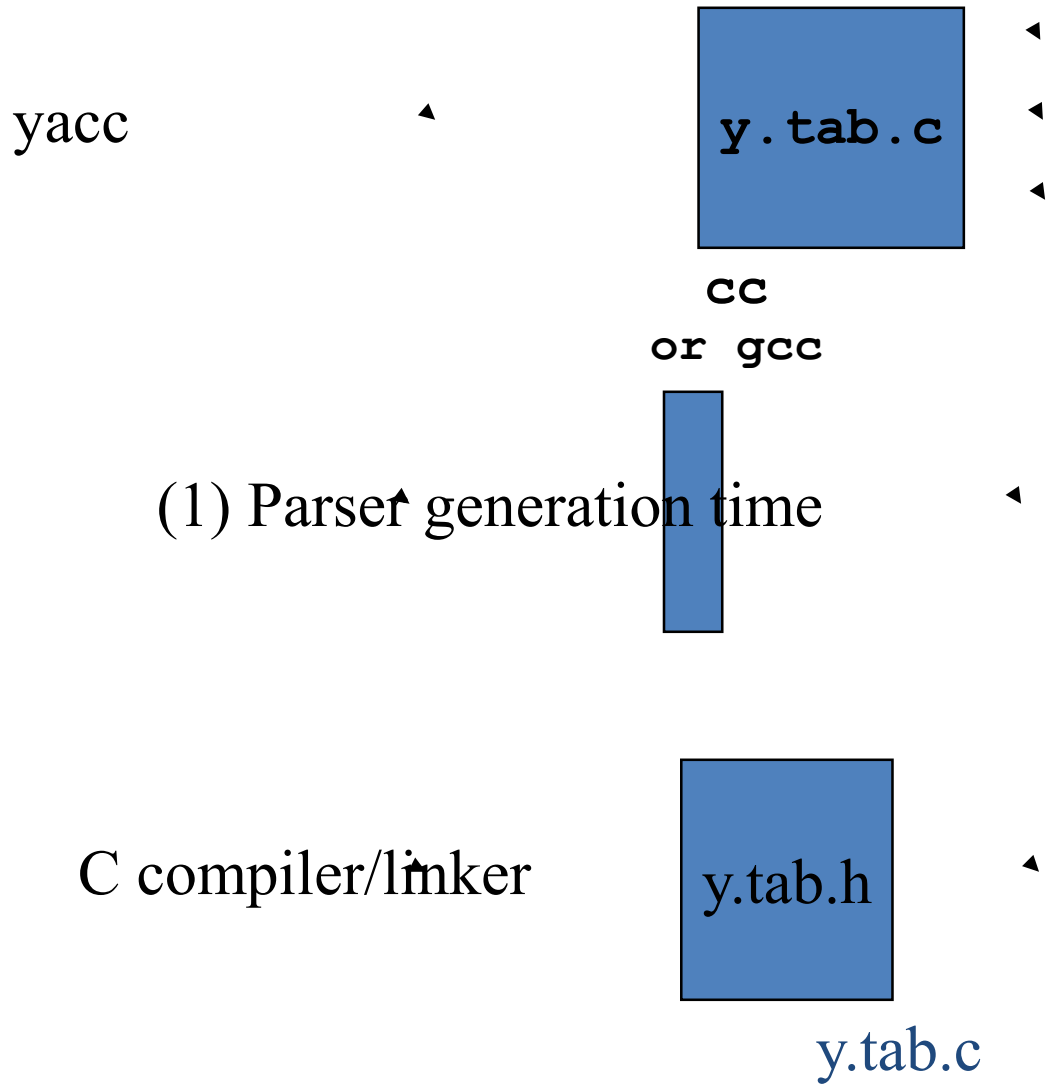
yacc program

C source program created by yacc

C compiler

Executable program that will parse grammar given in gram.y

How YACC Works



An YACC File Example

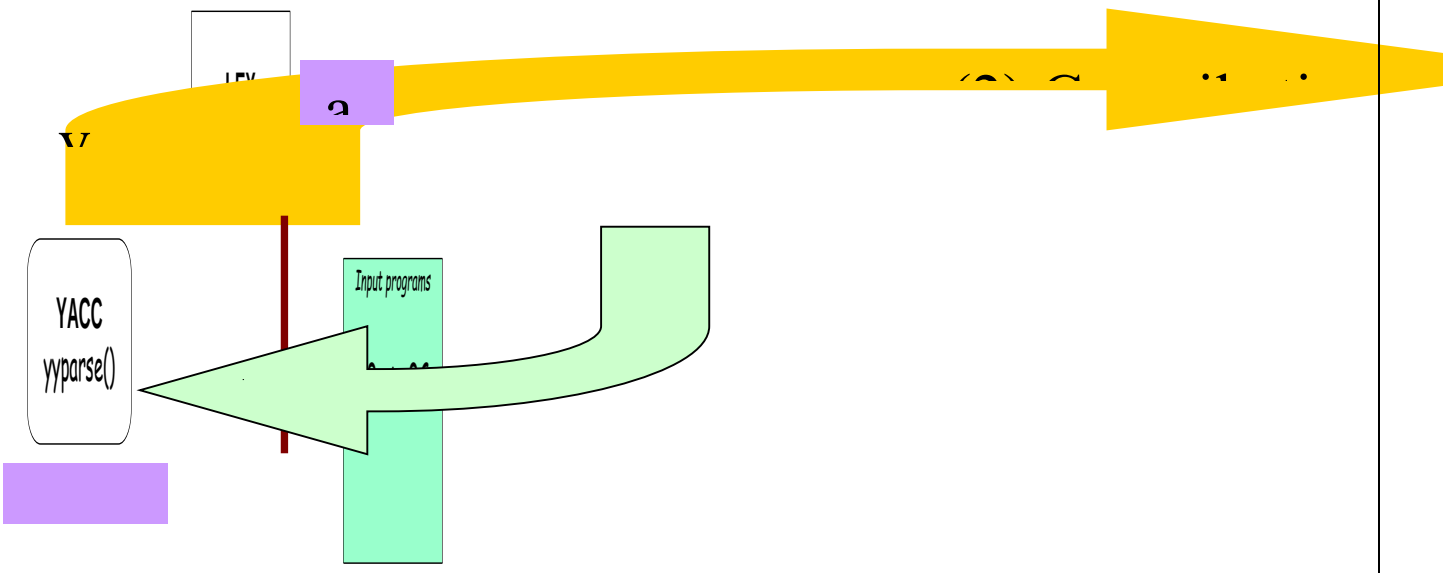
```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression            { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
```

Works with Lex



YACC File Format

%{

C declarations

%}

yacc declarations

%%

Grammar rules

%%

Additional C code

- **Comments enclosed in /* ... */ may appear in any of the sections.**

Add to Knowledge (Content Beyond Syllabus)

Chomsky Hierarchy

A grammar can be classified on the basis of production rules Chomsky classified grammars into following types.

Type and Grammar name	Language used	Forms of production	Accepting device
Type-0 –unrestricted grammar	Recursively enumerable	X1->X2 where X1,X2Є(V U T)* Where V-is a variable set T-is a Terminal set	Turing Machine
Type -1 –context sensitive grammar	Context sensitive	X1->X2 X1,X2Є(V U T)* Where V-is a variable set T-is a Terminal set & |X1|<=|X2|	Turing Machine with bounded tape & length of tape is finite
Type-2-Context free grammar	Context free language	Y->X1 Where YЄV & X1Є(VUT)*	PDA (Push down automata)
Type-3-Regular Grammar	Regular Language	X->aY|a|Ya|ε Where X,YЄV	FA(Finite Automata)

		& a Є T	

4.10 Learning Outcomes:

1. Know:

a) Student should be able to differentiate between Top Down and Bottom up Parser.
b) Understand the role of Lexical and Syntax analyzer in Compiler Design.

2. Comprehend:

a) Student should be able to explain and design Lexical Analyzer.
b) Student should be able to describe and design Syntax Analyzer.

3. Apply, analyze and synthesize:

Student should be able to:

1. Show the demonstration how Lexical Analyzer generate tokens and detect error.
2. Show the working of Bottom up Parser.

4.11. Short Answer questions

Q.1)Test whether the grammar is LR(1) or not.

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Problem

A reduce by $A \rightarrow \varepsilon$ b reduce by $A \rightarrow \varepsilon$

reduce by $B \rightarrow \varepsilon$ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict reduce/reduce conflict

Q.2 For the grammar having productions:

$A \rightarrow (A)A \mid \varepsilon$

Compute FIRST & FOLLOW set of A.

A.2 FIRST (A) = { (, ε }

FOLLOW (A) = {$,)}

Q.3 Is the following grammar LL(1)

S → aSa | ε

Q.4 Consider the following CFG:-

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

Remove the left recursion from above grammar.

Answer: This grammar is not immediately left-recursive,

but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar.

The resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow Aac \mid bc \mid d$

⇓ eliminate left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow bcA' \mid dA'$

$A' \rightarrow acA' \mid \varepsilon$

Q.5 Test whether the grammar is LL(1) or not and construct a predictive parsing table for it.

S → AaAb

S → BbBa

A → ε

B → ε

Q.6 What is parsing? Write down the drawback of top down parsing of backtracking.

Ans:Parsing is the process of analyzing a text, made of a sequence of tokens, to determine its grammatical structure with respect to a given formal grammar. Parsing is also known as syntactic analysis and parser is used for analyzing a text. The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. The input is a valid input with respect to a given formal grammar if it can be derived from the start symbol of the grammar.

Following are drawbacks of top down parsing of backtracking:

(i) Semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse.

(ii) Precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed.

Q.7 For the following grammar construct the predictive parsing table and explain that step by step.

Grammar G:-

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)

F→id

Q.8 Construct the LR(0) collection for following Arithmetic Grammar & construct LR(0) parser table:-

E→E+T

T→T*F

F→(E)

F→id

Q.9 Write short notes on:

i) LEX & YACC **ii) Recursive descent parser**

Q.10 Consider the following grammar:-

E→E+T | T

T→T*F | F

F→(E)

F→id

Show the shift reduce parser action for the string id+id+id*id.

(May 2007,) (10M)

Q.11 **Consider the following CFG**:-

E→E+T | T

T→T*F | F

F→(E) |I

I→a | b| c

Remove the left recursion from above grammar.

Q.12 Construct LL(1) parsing table for following grammar.

S -> aBDh

B-> cC

C->bC | **ε**

D -> EF

E-> g | **ε**

F -> f | **ε**

(Dec 2007) (10M)

Q 13. Cosider the following grammar and construct the LALR parsing table.

S->AA

A-> aA | b

(Dec 2007) (10M)

Q.14. Construct the LALR parsing table for the following grammar.

S->S

S-> CC

C-> cC | d

Q.15. Explain the LR parsers with suitable examples.

Q.16 Construct the predictive parser for the following grammar.

S → AaAb

S → BbBa

A → ε

B → ε

4.13. University Questions Sample Answers:

Q.1. Consider the following grammar-

S->A

A-> Ad|Ae|aB|aC

B->bBC | f

C->g

(Dec 2007) (10M)

Q.2. Eliminate left recursion present in following grammar (Remove direct and Indirect recursion both)

S→Aa|b

A→Ac|Sd|€ [May 2016]

Q. 3 What is Handle pruning? [Dec 2016]

Q.4 For the given grammar given below, construct operator precedence relations matrix, assuming *, + are binary operators and id as terminal symbol and E as non terminal symbol.

E→E+E

E→E*E

E→id

Apply operator precedence parsing algorithm to obtain skeletal syntax free for the statement Id+id*id. [Dec 2016, Nov 2015]

Q. 5 Construct SLR parsing table for following grammar. Show how parsing actions are done for the input string ()()$. Show stacks content, i/p buffer, action.

S-> (S)S

S->€ [Dec 2016]

Q. 6 Find First and Follow set for given grammar below:

E→TE’ E’→+TE’|€

T→FT’ T’→*FT’|€

F→(E) F→id [Nov 2015]

4.15. References

1. A.V. Aho, a nd J.D.Ullman: Principles of compiler construction,

Pearson Education

2 . A.V. Aho, R. Shethi and Ulman; Compilers - Principles, Techniques and

Tools , *Pearson Education*

4.16 Practice for Module No.4 Syntax Analyzer (Based on Gate Exam & University Patterns)

1. What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon- and unit-production (i.e., of type A -> ϵ and A -> a) to parse a string with n tokens?
(A) n/2
(B) n-1
(C) 2n-1
(D) 2^n

Answer: (B)

2. Consider the following two sets of LR(1) items of an LR(1) grammar.

X -> c.X, c/d
X -> .cX, c/d
X -> .d, c/d
X -> c.X, $
X -> .cX, $
X -> .d, $

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?

1. Cannot be merged since look aheads are different.
2. Can be merged but will result in S-R conflict.
3. Can be merged but will result in R-R conflict.
4. Cannot be merged since goto on c will lead to two different sets.

(A) 1 only
(B) 2 only
(C) 1 and 4 only
(D) 1, 2, 3, and 4

Answer: (D)

3. For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E1, E2, and E3. is the empty string, $ indicates end of input, and, | separates alternate right hand sides of productions.

S → a A b B | b A a B | ε
A → S
B → S

	a	b	$
S	**E1**	**E2**	S → ε
A	A → S	A → S	error
B	B → S	B → S	**E3**

(A) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b, $}

(B) FIRST(A) = {a, b, $}
FIRST(B) = {a, b, ε}
FOLLOW(A) = {a, b}
FOLLOW(B) = {$}

(C) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = ∅

(D) FIRST(A) = {a, b} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

(A) A
(B) B
(C) C
(D) D

Answer: (A)

4. Consider the data same as above question. The appropriate entries for E1, E2, and E3 are

(A) E1: S → aAbB, A → S
E2: S → bAaB, B → S
E3: B → S

(B) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: S → ε

(C) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: B → S

(D) E1: A → S, S → ε
E2: B → S, S → ε
E3: B → S

(A) A
(B) B
(C) C
(D) D

Answer: (C)

5. The grammar S → aSa | bS | c is
(A) LL(1) but not LR(1)
(B) LR(1)but not LR(1)
(C) Both LL(1)and LR(1)
(D) Neither LL(1)nor LR(1)

Answer: (C)

6. Match all items in Group 1 with correct options from those given in Group 2.

Group 1	Group 2
P. Regular expression	1. Syntax analysis
Q. Pushdown automata	2. Code generation
R. Dataflow analysis	3. Lexical analysis
S. Register allocation	4. Code optimization

(A) P-4. Q-1, R-2, S-3
(B) P-3, Q-1, R-4, S-2
(C) P-3, Q-4, R-1, S-2
(D) P-2, Q-1, R-4, S-3

Answer: (B)

7. Which of the following describes a handle (as applicable to LR-parsing) appropriately?
(A) It is the position in a sentential form where the next shift or reduce operation will occur

(B) It is non-terminal whose production will be used for reduction in the next step
(C) It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur
(D) It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found

Answer: (D)

8. An LALR(1) parser for a grammar G can have shift-reduce (S-R) conflicts if and only if
(A) the SLR(1) parser for G has S-R conflicts
(B) the LR(1) parser for G has S-R conflicts
(C) the LR(0) parser for G has S-R conflicts
(D) the LALR(1) parser for G has reduce-reduce conflicts

Answer: (B)

9. Which one of the following is a top-down parser?
(A) Recursive descent parser.
(B) Operator precedence parser.
(C) An LR(k) parser.
(D) An LALR(k) parser

Answer: (A)

10. Consider the grammar with non-terminals N = {S,C,S1 },terminals T={a,b,i,t,e}, with S as the start symbol, and the following set of rules:

S --> $iCtSS_1|a$
S_1 --> $eS|\epsilon$
C --> b

The grammar is NOT LL(1) because:
(A) it is left recursive
(B) it is right recursive
(C) it is ambiguous
(D) It is not context-free.

Answer: (C)

11. Consider the following two statements:

P: Every regular grammar is LL(1)
Q: Every regular set has a LR(1) grammar

Which of the following is TRUE?
(A) Both P and Q are true
(B) P is true and Q is false
(C) P is false and Q is true
(D) Both P and Q are false

Answer: (C)

12. Consider the following grammar.

S -> S * E
S -> E
E -> F + E
E -> F
F -> id

Consider the following LR(0) items corresponding to the grammar above.

(i) S -> S * .E
(ii) E -> F. + E
(iii) E -> F + .E

Given the items above, which two of them will appear in the same set in the canonical sets-of-items for the grammar?
(A) (i) and (ii)
(B) (ii) and (iii)
(C) (i) and (iii)
(D) None of the above

Answer: (D)

13. A canonical set of items is given below

S --> L. > R
Q --> R.

On input symbol < the set has
(A) a shift-reduce conflict and a reduce-reduce conflict.
(B) a shift-reduce conflict but not a reduce-reduce conflict.
(C) a reduce-reduce conflict but not a shift-reduce conflict.
(D) neither a shift-reduce nor a reduce-reduce conflict.

Answer: (D)

14. Consider the grammar defined by the following production rules, with two operators $*$ and +

S --> T * P
T --> U | T * U
P --> Q + P | Q
Q --> Id
U --> Id

Which one of the following is TRUE?

(A) + is left associative, while ∗ is right associative
(B) + is right associative, while ∗ is left associative
(C) Both + and ∗ are right associative
(D) Both + and ∗ are left associative

Answer: (B)

15. Which one of the following grammars is free from left recursion?

(A)
$$S \rightarrow AB$$
$$A \rightarrow Aa \mid b$$
$$B \rightarrow c$$

(B)
$$S \rightarrow Ab \mid Bb \mid c$$
$$A \rightarrow Bd \mid \varepsilon$$
$$B \rightarrow e$$

(C)
$$S \rightarrow Aa \mid B$$
$$A \rightarrow Bb \mid Sc \mid \varepsilon$$
$$B \rightarrow d$$

(D)
$$S \rightarrow Aa \mid Bb \mid c$$
$$A \rightarrow Bd \mid \varepsilon$$
$$B \rightarrow Ae \mid \varepsilon$$

(A) A
(B) B
(C) C
(D) D

Answer: (B)

16. Consider the following grammar:

S → FR
R → S | ε
F → id

In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, $] respectively.
(A) {S → FR} and {R → ε }
(B) {S → FR} and { }
(C) {S → FR} and {R → *S}
(D) {F → id} and {R → ε}

Answer: (A)

17. The grammar A → AA | (A) | ε is not suitable for predictive-parsing because the grammar is
(A) ambiguous
(B) left-recursive
(C) right-recursive
(D) an operator-grammar

Answer: (B)

18. Consider the grammar

S → (S) | a

Let the number of states in SLR(1), LR(1) and LALR(1) parsers for the grammar be n1, n2 and n3 respectively. The following relationship holds good
(A) n1 < n2 < n3 **(B)** n1 = n3 < n2 **(C)** n1 = n2 = n3
(D) n1 ≥ n3 ≥ n2

Answer: (B)

19. Which of the following grammar rules violate the requirements of an operator grammar ? P, Q, R are nonterminals, and r, s, t are terminals.

1. P → Q R
2. P → Q s R
3. P → ε
4. P → Q t R r

(A) 1 only
(B) 1 and 3 only
(C) 2 and 3 only
(D) 3 and 4 only

Answer: (B)

20. Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar?
(A) Removing left recursion alone
(B) Factoring the grammar alone

(C) Removing left recursion and factoring the grammar
(D) None of these

Answer: (D)

Self-Assessment

Q.1 What is Lexical Analysis? What are the functions of Lexical Analyzer?
Q.2 What is Grammar? Write Different types of grammar.
Q.3 Write Predictive Parser algorithm to design LL(1) Parser.
Q.4 Write Operator Precedence Algorithm.
Q.5 Differentiate between Top Down and Bottom up Parser.

Self-Evaluation

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		**Tick Your choice**
1.	Do you understand the role of Syntax Analyzer in Compiler Design?	○ Yes ○ No
2.	Do you able to differenciate different Types of parser?	○ Yes ○ No
3.	Do you able to Design Top down and Bottom Up Parser?	○ Yes ○ No
4.	Do you able to identify different types of compilation error ?	○ Yes ○ No
5.	Do you Differenciate Between Top down and Bottom up parser?	○ Yes ○ No
6.	Do you understand module ?	○ Yes, Completely. ○ Partialy. ○ No, Not at all.