NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT 332

COURSE TITLE: Survey of Programming Languages

NUMBER OF UNITS: Four Units

COURSE GUIDE

Course Code	**CIT 332**
Couse Title	**Survey of Programming Languages**
Couse Developer/Writer	**Dr. Kehinde Adebola Sotonwa** **Computer Science & Information Technology Department,** **Bells University of Technology, Otta, Ogun State.**
Course Editor	
Programme Leader	
Course Coordinator	

National Open University of Nigeria

National Open University of Nigeria Headquarters
University Village, Plot 91 Cadastral Zone, Nnnamdi Azikiwe Expressway Jabi, Abuja

LAGOS OFFICE
14/16 Ahmadu Bello Way
Victoria Island , Lagos

e-mail: centrainforAnou.edu.ng
URL: www.nou.edu.ng

Published by:
National Open University of Nigeria

ISBN: 978-058-470-6

MAIN COURSE

CONTENTS **PAGE**

Introduction

CIT 332: Survey of Programming Languages is a Second Semester course. It is a four (4) credit degree course available to three hundred level students offering Computer Science and all related courses, studying towards acquiring a Bachelor of Science in Computer Science and other related disciplines.

The course is divided into four (4) modules and 15 study units. It entails the concept of programming languages. It further deals with general overview of programming languages. The course also introduces language description, evolution of level of programming languages and language comparison. concepts underlying modern programming languages: C/C++, C#, Java, Python, LISP, PERL, ALGOL and PROLOG.

What you will Learn in this Course

The overall aim of this course is to teach you general overview of programming languages: evolution, generation, concepts, application domain and criteria for language evaluation. The course is designed to describe the fundamental concepts of programming language by discussing the design issues of the various language constructs, examining the design choices for these constructs in some of the most common languages, and critically comparing design alternatives, Fundamental syntactic and semantic concepts underlying modern programming languages, different modern programming languages: C/C++, C#, Java, Python, LISP, PERL, ALGOL and PROLOG, their syntax: bindings and scope, data types and type checking, functional scheme, expression of assignments, control structure, program statements, program units etc., and finally language comparison.

Course Aim

This course aims to take a step further in teaching you the basic and best approach to survey programming languages. It is hoped that the knowledge would enhance both the programmer/developer expertise and Students to be familiar with popular programming languages and the advantages they have over each other.

Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

The general objective of the course as an integral part of the Bachelor Degree for Computer Science Students in National Open University, Abeokuta, is to:

- Demonstrate understanding of the evolution of programming languages and relate how this history has led to the paradigms available today.

- Identify at least one outstanding and distinguishing characteristic for each of the programming paradigms covered in this unit.
- Evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression.
- Identify at least one distinguishing characteristic for each of the programming paradigms covered in this unit.
- Describe the importance and power of abstraction in the context of virtual machines.
- Explain the benefits of intermediate languages in the compilation process.
- Evaluate the tradeoffs in reliability vs. writability.
- Compare and contrast compiled and interpreted execution models, outlining the relative merits of each.
- Describe the phases of program translation from source code to executable code and the files produced by these phases.
- Explain the differences between machine-dependent and machine-independent translation and where these differences are evident in the translation process.
- Explain formal methods of describing syntax (backus-naur form, context-free grammars, and parser tree).
- Describe the meanings of programs (dynamic semantics, weakest precondition).
- Identify and describe the properties of a variable such as its associated address, value, scope, persistence, and size.
- Explain data types: primitive types, character string types, user-defined ordinal types, array types, associative arrays, point and reference types
- Demonstrate different forms of binding, visibility, scoping, and lifetime management.
- Demonstrate the difference between overridden and overloaded subprograms
- Explain functional side effects.
- Demonstrate the difference between pass-by-value, pass-by-result, pass-by-value-result, pass-by-reference, and pass-by-name parameter passing.
- Explain the difference between the static binding and dynamic binding.
- Discuss evolution, history, program structure and features of some commonly used programming languages paradigm such as C/C++, C#, Java, Python, LISP, PERL, ALGOL and PROLOG.
- Examine, evaluate and compare these languages
- To increase capacity of computer science students to express ideas
- Improve their background for choosing appropriate languages
- Increase the ability to learn new languages
- To better understand the significance of programming implementation
- Overall advancement of computing.

Working through this Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self-assessment exercises and tutor - marked assignments, and at some point in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Justification

Any serious study of programming languages requires an examination of some related topics among which are formal methods of describing the syntax and semantics of programming languages and its implementation techniques. The need to use programming language to solve our day-to-day problems grows every year. Students should be able to familiar with popular programming languages and the advantage they have over each other. They should be able to know which programming language solves a particular problem better. The theoretical and practical knowledge acquired from this course will give the students a foundation from which they can appreciate the relevant and the interrelationships of different programming languages.

Course Materials

The major components of the course are:

1. Course Guide
2. Study Units
3. Text Books
4. Assignment Files
5. Presentation Schedule

Course Requirements

This is a compulsory course for all computer science students in the University. In view of this, students are expected to participate in all the course activities and have minimum of 75% attendance to be able to write the final examination.

Study Units

There are 4 modules and 15 study units in this course. They are:

Module 1 Concept of Programming Languages

Unit 1 History of Programming Languages
Unit 2 Reasons for Studying Concepts of Programming Language
Unit 3 Application Domains
Unit 4 Criteria for Language Evaluation

Module 2 Influence of Language Design

Unit 1 Computer Architecture
Unit 2 Language Paradigms
Unit 3 Language Design Trade-offs,
Unit 4 Implementation Method

Module 3 Language Description

Unit 1 Fundamental Syntactic Analysis Concept on Underlying Modern Programming Language
Unit 2 Fundamental Semantic Analysis Concept on Underlying Modern Programming Language
Unit 3 Formal Language and Grammars
Unit 4 The Basic Element of Programming Languages

Module 4 Evolution of Programming Language Levels

Unit 1 Brief History on Level of Programming Languages
Unit 2 Evolution of Different Languages
Unit 3 Language Evaluation and Comparison

Textbooks and References

Bjarne, Dines; Cliff, B. Jones (1978). The Vienna Development Method: The Meta-Language, *Lecture Notes in Computer Science 61*. Berlin, Heidelberg, New York: Springer.

Maurizio Gabbrielli and Simone Martini (2010). Programming Languages: Principles and Paradigm, Springer-Verlag London Limited, 2010.

Robert W. Sebesta. Concepts of Programming Languages. Tenth Edition, University of Colorado at Colorado Spring Pearson, 2011.

Bernd Teufel, Organization of Programming Languages, Springer-Verlag/Wien, 1991

Michael L. Scoot. Programming Language Pragmatics. Morgan Kaufmann, 2006.

Anoemuah RosemaryAruorezi, Michael and Cecilia Ibru University, 2018

Allen B. Tucker and Robert Noonan. Programming Languages Principles and Paradigm. Mcgraw-Hill Higher Education, 2006.

Robert Harle, Object Oriented Programming IA NST CS and CST Lent 2009/10.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.

Kasabov NK, (1998) Foundations of Neural Networks, *Fuzzy Systems and Knowledge Engineering*. The MIT Press Cambridge.

Bezem M (2010) A Prolog Compendium. Department of Informatics, University of Bergen. Bergen, Norway.

Rowe NC. (1988) Artificial Intelligence through Prolog. Prentice-Hall.

Olusegun Folorunso, Organization of Programming Languages, *The Department of Computer Science, University of Agriculture,* Abeokuta.

Bramer M (2005) Logic Programming with Prolog. Springer. USA 10. *Prolog Development Center* (2001) Visual Prolog Version 5.x: Language Tutorial. Copenhagen, Denmark

Bergin, Thomas J. and Richard G. Gibson, eds. *History of Programming Languages-II*. New York: ACM Press, 1996.

Christiansen, Tom and Nathan Torkington. *Perlfaq1 Unix Manpage*. Perl 5 Porters, 1997-1999.

Java History." http://ils.unc.edu/blaze/java/javahist.html. Cited, March 29, 2000.

Programming Languages." *McGraw-Hill Encyclopedia of Science and Technology*. New York: McGraw-Hill, 1997.

Wexelblat, Richard L., ed. *History of Programming Languages*. New York: Academic Press, 1981.

A History of Computer Programming Languages (onlinecollegeplan.com)

A History of Computer Programming Languages (brown.edu).

A Brief History of Programming Languages." http://www.byte.com/art/9509/se7/artl9.htm. Cited, March 25, 2000.

A Short History of the Computer." http://www.softlord.com/comp/. Jeremy Myers. Cited, March 25, 2000.

IT Lecture 1 History of Computers.pdf (webs.com). Miss N. Nembhard, Source (1995)

Robers, Eric S. *The Art and Science of C. Addison-Wesley Publishing Company*. Reading: 1995.

Rowe NC. (1988) Artificial Intelligence through Prolog. Prentice-Hall.

Why Study Programming Languages? (lmu.edu), Hipster Hacker https://cs.Imu.edu/ray/notes/whystudyprogramminglanguages/.

Reasons Why You Should Learn BASIC Programming | UoPeople https://www.uoppeople.edu/blog/6-reasons-why-you-should-learn-basic-programming/

Programming Language Concepts Test Questions/Answers, Dino Cajic, (dinocajic.blogspot.com), https://dinocajic.blogspot.com/2017/02/programming-language-concepts-test,html

50+ Best Programming Interview Questions and Answers in 2021 (hackr.io) https://hackr.io/blog/programming-interview-questions.

ide.geeksforgeeks.org, 5th Floor, A-118, Sector-136, N.oida, Uttar Pradesh – 201305.

Concepts of Programming Languages, Tenth Edition, Robert W, Sebesta, University of Colorado, Colorado Springs Pearson.

Louden: Programming Languages: *Principles and Practices,* Cengage Learning

Tucker: Programming Languages: *Principles and paradigms,* Tata McGraw –Hill.
E Horowitz: Programming Languages, 2nd Edition, Addison Wesley

Programming Language Evaluation Criteria Part 1: Readability | by ngwes | Medium, *Language Evaluation Criteria* (gordon.edu)

Emmett Boudreau. What Is A Programming Paradigm. *An overview of programming paradigms.* | Towards Data Science

Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning.

Questions-on-principle-of-programming-language.pdf (oureducation.in), https://blog/oureducation.in/wp-content/uploads/2013/Questions-on-principle -of-programming-language.pdf#.

Language Evaluation Criteria PPL by Jayesh Umre

Chapter_3_Von_Neumanns_Architecture.pdf (weebly.com), https://viennaict.weebly.com.

Michael L. Scott, Programming Language Pragmatics, *Third Edition, Morgan Kaufmann Publishers, Elsevier*.

Semantics in Programming Languages - INFO4MYSTREY , BestMark Mystery2020 (info4mystery.com), https://www.info4mystery.com/2019/01/semantics-in-profgramming-languages.html.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.

J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985.

Semantic Analysis in Compiler Design - GeeksforGeeks, https://www.geeksforgeeks.org/semantic-analysis-in-computer-design/.

Compiler Design - Semantic Analysis (tutorialspoint.com), https://www.tutorialspoint.com/compiler_design_semantic_analysis.html.

Semantics of Programming Languages Computer Science Tripos, Part 1B, *Peter Sewell Computer Laboratory*, University of Cambridge, 2009.

UTD: Describing Syntax and Semantics: Dr. Chris Davis, cid021000@utdallas.edu

Hennessy, M. (1990). The Semantics of Programming Languages. *Wiley. Out of print, but available on the web* at http://www.cogs.susx.ac.uk/users/matthewh/ semnotes.ps.gz12. Boston: Addison-Wesley, 2007.

Pierce, B. C. (2002) Types and Programming Languages. *MIT Press*

Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

Winskel, G. (1993). The Formal Semantics of Programming Languages. MIT Press. An introduction to both operational and denotational semantics; recommended for the Part II Denotational Semantics course.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

Programming Languages: Types and Features - Chakray, https://www.chakay.com/programmin-languages-types-and-features.

A. W. Appel. Modern Compiler Implementation in Java. *Cambridge University Press, Cambridge*, 1998. 3. J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, 3/E. Addison Wesley, Reading, 2005. Available online at http://java.sun.com/docs/books/jls/index.html. References 55

J. E. Hopcroft, R. Motwani, and J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, 2001.

C. Laneve. La Descrizione Operazionale dei Linguaggi di Programmazione. Franco Angeli, Milano, 1998 (in Italian).

C. W. Morris. Foundations of the theory of signs. In Writings on the Theory of Signs, pages 17–74. Mouton, The Hague, 1938.

Programming Data Types & Structures | Computer Science (teachcomputerscience.com), https://teachcomputerscience.com/programming-data-types/

Beginner's Guide to Passing Parameters | Codementor, Michael Safyan, https://www.codementor.io/@michaelsafyan/computer-different-ways-to-pass-parameters-du107xfer.

Parameter Passing | Computing (glowscotland.org.uk), Stratton, https://blogs.glowscotland.org.uk
61. Parameter Passing Techniques in C/C++ - GeeksforGeeks, https://www.geeksforgeeks.org/

Parameter passing - Implementation (computational constructs) - Higher Computing Science Revision - BBC Bitesize, https://www.bbc.co.uk/

Definition of Parameters in Computer Programming (thoughtco.com), David Bolton, 2017.

Modern Programming Language, Lecture 18-24: LISP Programming

LISP - Data Types (tutorialspoint.com)

Introduction to LISP, CS 2740, Knowledge Representation Lecture 2 by Milos Hauskrecht, 5329 Sennott Square

The Evolution of Lisp: ACM History of Programming Languages Gabriel and Steele's (FTP).

Common LISP: A Gentle Introduction to Sybbolic Computattion by Davis S. Touretzky

History of Python Programming Language - Python Pool

History of Python - GeeksforGeeks by Sohom Pramanick

Perl Programming Language: history, features, applications, Why Learn? - Answersjet

History of Python (tutorialspoint.com), History Of Python Programming Language – Journal Dev.

A brief history of the Python programming language | by Dan Root | Python in Plain English

History and Development of Python Programming Language (analyticsinsight.net)

The ALGOL Programming Language (umich.edu)

Programming History: The Influence of Algol on Modern Programming Languages (Part 1) | by Mike McMillan | Better Programming

History Of Algol 68 | programming language (wordpress.com) https://www.javapont.com classification –of-programming-languages, https//www.w3schools.in/category/java-tutorial, https://www.totorialspoint.com/difference-between-hihg-level-language-and-low-level-language.https://byjus.com/gate/difference-between-high-level-and-low-level-languages/

What is a Low-Level Language? - Definition from Techopedia, https://www.tutorialandexample.com/input-devies-of-computer/

Java Syntax - A Complete Guide to Master Java - DataFlair (data-flair.training), https://datra-flair.training/blog/basic-jaja-syntax/

C++ Language Tutorial by Juan Soulie, http://www. cplusplus.com./doc/tutorial/

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 15 tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavor to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor -Marked Assignments (TMAs)

There are 15 TMAs in this course. You need to submit all the TMAs. The best 5 will therefore be counted. The total marks for the best five (5) assignments will be 40% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination for the course will carry 60% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your language learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the contents, then a set of objectives and then the dialogue and so on. The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

Course Marking Scheme

The following table includes the course marking scheme

Table 1: Course Marking Scheme

Assessment	Marks
Assignments 1-15	15 assignments, 40% for the best 5 total = 8% X 5 = 40%
Final Examination	60% of overall course marks
Total	100% of Course Marks

Course Overview

This table indicates the units, the number of weeks required to complete them and the assignments.

Unit	Title of the Work	Weeks	Assessment (End of Unit)
	Course Guide		
Module 1	**Concept of Programming Languages**		
1	History of Programming Languages	Week 1	Assessment 1
2	Reasons for Studying Concepts of Programming Languages	Week 1	Assessment 2
3	Application Domains	Week 2	Assessment 3
4	Criteria for Language Evaluation	Week 2	Assessment 4
Module 2	**Influence Language Design**		
1	Computer Architecture	Week 3	Assessment 5

2	Language Paradigm	Week 3	Assessment 6
3	Language Design Trade-offs	Week 4	Assessment 7
4	Implementation Method	Week 4	Assessment 8
Module 3 Language Description			
1	Fundamental Syntactic Analysis Concepts on Underlying Modern Programming Languages	Week 5	Assessment 9
2	Fundamental Semantic Analysis Concept on Underlying Modern Programming Language	Week 6	
3	Formal Language and Grammars	Week 7	
4	The Basic Element of Programming Languages	Week 8	Assessment 10
Module 4 Evolution of Programming Language Levels			
1	Brief History on Level of Programming Languages	Week 9	Assessment 11
2	Evolution of Different Languages	Week 10 &11	
3	Language Evaluation and Comparison	Week 12	Assessment 12

HOW TO GET THE BEST FROM THIS COURSE

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

1. Read this Course Guide thoroughly.

2. Organize a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units.

Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.

3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.

4. Turn to Unit 1 and read the introduction and the objectives for the unit.

5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time

.

6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.

7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.

8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.

9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.

10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this Course Guide).

TUTORS AND TUTORIALS

There are 12 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail or submit your tutor-marked assignments to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

- You do not understand any part of the study units or the assigned readings
- You have difficulty with the self-tests or exercises
- You have a question or problem with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face to face contact with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively.

TABLE OF CONTENTS **PAGE**

MODULE 1 CONCEPT OF PROGRAMMING LANGUAGES

Unit 1 History Programming Language
Unit 2 Reasons for studying concepts of programming language
Unit 3 Application domains
Unit 4 Criteria for language evaluation

UNIT 1 HISTORY OF PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

Ever since the invention of Charles Babbage's difference engine in 1822, computers have required a means of instructing them to perform a specific task. This means is known as a programming language. Computer languages were first composed of a series of steps to wire a particular program; these morphed into a series of steps keyed into the computer and then executed; later these languages acquired advanced features such as logical branching and object orientation. The computer languages of the last fifty years have come in two stages, the first major languages and the second major languages, which are in use today.

Most computer programming languages were inspired by or built upon concepts from previous computer programming languages. Today, while older languages still serve as a strong foundation for new ones, newer computer programming languages make programmers' work simpler. Businesses rely heavily on programs to meet all of their data, transaction, and customer service needs. Science and medicine need accurate and complex programs for their research. Mobile applications must be updated to meet consumer demands. And all of these new and growing needs ensure that computer programming languages, both old and new, will remain an important part of modern life.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Define programming languages
- Distinguished between different generation of programming languages
- Describe the characteristics of different generation of programming languages

3.0 MAIN CONTENT

3.1 What is Programming Language

Before getting into computer programming, what is computer programs and what they do. A computer program is a sequence of instructions written using a Computer Programming Language to perform a specified task by the computer. The two important terms used in the above definition are:

- Sequence of instructions
- Computer Programming Language

To understand these terms, consider a situation when someone asks you about how to go to a nearby KFC. What exactly do you do to tell him the way to go to KFC? You will use Human Language to tell the way to go to KFC, something as follows:

First go straight, after half kilometer, take left from the red light and then drive around one kilometer and you will find KFC at the right. Here, you have used English Language to give several steps to be taken to reach KFC. If they are followed in the following sequence, then you will reach KFC:

1. Go straight
2. Drive half kilometer
3. Take left
4. Drive around one kilometer
5. Search for KFC at your right side

Now, try to map the situation with a computer program. The above sequence of instructions is actually a Human Program written in English Language, which instructs on how to reach KFC from a given starting point. This same sequence could have been given in Spanish, Hindi, Arabic, or any other human language, provided the person seeking direction knows any of these languages. Now, let's go back and try to understand a computer program, which is a sequence of instructions written in a Computer Language to perform a specified task by the computer. Following is a simple program written in Python Programming Language − print "Hello, World!"

The above computer program instructs the computer to print "Hello, World!" on the computer screen. A computer program is also called a computer software, which can range from two lines to millions of lines of instructions.

- Computer program instructions are also called program source code and computer programming is also called program coding.
- A computer without a computer program is just a dump box; it is programs that make computers active.

Computer programming is the process that professionals use to write code that instructs how a computer, application or software program performs. At its most basic, computer programming is a set of instructions to facilitate specific actions. A programming language is a computer language that is used by programmers (developers) to communicate with computers. It is a set of instructions written in any specific language to perform a specific task. It is made up of a series of symbols that serves as a bridge that allow humans to translate our thoughts into instructions computers can understand. Human and machines process information differently, and programming languages are the key to bridging the gap between people and computers therefore the first computer programming language was created.

3.2 Classification of Programming Language Generation

Generations of computers have seen changes based on evolving technologies. With each new generation, computer circuitry, size, and parts have been miniaturized, the processing and speed doubled, memory got larger, and usability and reliability improved. Note that the timeline specified for each generation is tentative and not definite. The generations are actually based on evolving chip technology rather than any particular time frame.

The five generations of computers are characterized by the electrical current flowing through the processing mechanisms listed below:

- The first is within vacuum tubes
- Transistors
- Integrated circuits
- Microprocessors
- Artificial intelligence

3.2.1 First Generation (1GL)

Machine-level programming language used to program first-generation computers. Originally, no translator was used to compile or assemble the first-generation language. Computers usually operate on a binary level which are sequences of 0's and 1 'so, in the early days of computer programming binary machine code was used to program computers. Binary programming means that the program directly reflects the hardware structure of the computer. Sequences of 0's and 1 's cause certain actions in processing, control, or memory units, i.e. programming was done on the flip-flop level. It was recognized very rapidly that programming computers on binary machine code level is very troublesome. Thus, it is obvious that binary programming could not successfully be applied to solve complex problems.

3.2.2 Second Generation Languages (2GL)

Assembly languages: the obvious drawbacks of binary programming became smaller by the introduction of second generation languages (2GL). These languages allowed mnemonic abbreviations as symbolic names and the concept of commands and operands was introduced. A programmer's work became much easier, since the symbolic notation and addressing of instructions and data was now possible. Compilation systems, called assemblers, were developed

to translate the assembly language/symbolic programs into machine code. Assembly languages still reflect the hardware structure of the target machine - not on the flip-flop level, but on the register level, i.e. the abstraction has changed from the flip-flop to the register level. The instruction set of the target computer directly determines the scope of an assembly language. With the introduction of linking mechanisms, assembling of code separately became possible and the first steps towards program structuring were recognizable, although the term structured programming cannot be used for programming assembly code. The major disadvantage of assembly languages is that programs are still machine dependent and, in general, only readable by the authors.

3.2.3 Third Generation Languages (3GL)

Third generation languages /high level languages/Problem oriented languages: these languages allow control structures which are based on logical data objects: variables of a specific type. They provide a level of abstraction allowing machine-independent specification of data, functions or processes, and their control. A programmer can now focus on the problem to be solved without being concerned with the internal hardware structure of the target computer. When considering high level programming languages, there are four groups of programming languages: imperative, declarative, object oriented and functional languages.

3.2.4 Fourth Generation Languages (4GL)

Fourth generation languages/Non procedural languages deal with the following two fields which become more and more important: database and query languages, and program or application generators. The steadily increasing usage of software packages like database systems, spread sheets, statistical packages, and other (special purpose) packages makes it necessary to have a medium of control available which can easily be used by non-specialists. In fourth generation languages the user describes what he wants to be solved, instead of how he wants to solve a problem - as it is done using procedural languages. In general, fourth generation languages are not only languages, but interactive programming environments. E.g. SOL (Structured Query Language): a query language for relational databases based on Codd's requirements for non-procedural query languages. Another example is NATURAL emphasizing on a structured programming style. Program or application generators are often based on a certain specification method and produce an output (e.g. a high level program) to an appropriate specification. There exist already a great number of fourth generation languages:

3.2.5 Fifth Generation Language (5GL)

5GL is a programming language based around solving problems using constraints given to program rather using an algorithm written by a programmer. 5GL allows computers to have their own ability to think and their own inferences can be worked out by using the programmed information in large databases. 5GL gave birth to the dream of robot with AI and Fuzzy Logic. The fifth-generation languages are also called 5GL. It is based on the concept of artificial intelligence. It uses the concept that that rather than solving a problem algorithmically, an application can be built to solve it based on some constraints, i.e., we make computers learn to solve any problem. Parallel Processing & superconductors are used for this type of language to make real artificial intelligence. Advantages of this generation is that machines can make

decisions, it reduces programmer effort to solve a problem and very easier than 3GL or 4GL to learn and use. Examples are: PROLOG, LISP, etc.

SELF-ASSESSMENT EXERCISE

1. What is programming language?
2. List the major component of computer generation
3. Explain different generation of computer you know.

3.2 Characteristics of each generation

A) Computer Characteristics and Capabilities for 1GL

- **Size** – Relatively big size. Size was equivalent to a room.
- **Speed** – slow speed, hundred instructions per second.
- **Cost** – cost was very high.
- **Language**– Machine and Assembly Language.
- **Reliability** – high failure rate, Failure of circuits per second.
- **Power**– high power Consumption and it generated much heat.

B) Trends and Developments in Computer Hardware

- **Main Component** – based on vacuum tubes
- **Main memory –**Magnetic drum
- **secondary Memory** – Magnetic drum & magnetic tape.
- **Input Media** – Punched cards & paper tape
- **Output Media** – Punched card & printed reports.
- **Example –** ENIAC, UNIVAC, Mark –I,mark-III , IBM 700 series , IBM 700 series ,IBM 701 series IBM 709 series etc.

A) Computer Characteristics and Capabilities 2GL

- **Size** – Smaller than first generation Computers.
- **Speed** – Relatively fast as compared to first generation, thousand instructions per second.
- **Cost** – cost Slightly lower than first generation.
- **Language** – Assembly Language and High level languages like FORTRAN, COBOL, BASIC.
- **Reliability** – Failure of circuits per days.
- **Power**– Low power Consumption.

B) Trends and Developments in Computer Hardware

- **Main Component** – Based on Transistor.
- **Main Memory** – Magnetic core.
- **Secondary Memory** – Magnetic tape & magnetic Disk.
- **Input Media** – Punched cards
- **Output Media** – Punched card & printed reports.

- **Example –** IBM-7000, CDC 3000 series, PDP1,PDP3,PDP 5 ,PDP8 ,ATLAS,IBM-7094 etc.

A) Computer Characteristics and Capabilities for 3GL

- **Size** – Smaller than Second Generation Computers. Disk size mini computers.
- **Speed** – Relatively fast as compared to second generation, Million instructions per second (MIPS).
- **Cost** – cost lower than Second generation.
- **Language**– High level languages like PASCAL, COBOL, BASIC, C etc.
- **Reliability** – Failure of circuits in Weeks.
- **Power**– Low power Consumption.

B) Trends and Developments in Computer Hardware

- **Main Component** – Based on Integrated Circuits (IC)
- **Primary Memory** – Magnetic core.
- **Secondary Memory**– Magnetic Tape & magnetic disk.
- **Input Media** – Key to tape & key to disk
- **Output Media** – Printed reports & Video displays.
- **Example** – IBM-307 Series, CDC 7600 series, PDP (Personal Data processer) II etc.

A) Computer Characteristics and Capabilities 4GL

- **Size** – Typewriter size micro Computer.
- **Speed** – Relatively fast as compared to Third generation, Tens of Millions instructions per second.
- **Cost** – Cost lower than third generation.
- **Language**– High level languages like C++, KL1, RPG, SQL.
- **Reliability** – Failure of circuits in months.
- **Power**– Low power Consumption.

B) Trends and Developments in Computer Hardware

- **Main Component** – Large scale integrated (LSI) Semiconductor circuits called MICRO PROCESSOR or chip and VLSI (Very Large scale integrated).
- **Main Memory** – Semiconductor memory like RAM, ROM and cache memory is used as a primary memory.
- **Secondary Memory** – Magnetic disk, Floppy disk, and Optical disk (CD, DVD).
- **Input Media** – keyboard.
- **Output Media** – Video displays, Audio responses and printed reports.
- **Example** – CRAY 2, IBM 3090/600 Series, IBM AS/400/B60 etc**.**

A) Computer Characteristics and Capabilities 5GL

- **Size** –Credit card size microcomputers.
- **Speed** – Billions instructions per second.

- **Cost** – Cost Slightly lower than first generation.
- **Language**– Artificial Intelligence (AI) Languages like LISP, PROLOG etc
- **Reliability** – Failure of circuits in year.
- **Power**– Low power Consumption.

B) Trends and Developments in Computer Hardware

- **Main Component** – based on ULSI (Ultra Large scale integrated) Circuit .that is also called Parallel Processing method.
- **Memory** – Optical disk and magnetic disk.
- **Input Media** – Speech input, Tactile input.
- **Output Media** – Graphics displays, Voice responses.
- **Example –** Lap-Tops, palm –Tops, Note books, PDA (personal Digital Assistant) etc.

4.0 CONCLUSION

The study of programming languages is valuable for some important reasons: It gives insight to generation of programming languages, enables to know background of computer as a whole, and the components attached to each of the generation.

5.0 SUMMARY

This unit has explained what programming language is, classification and explanation of different programming language generation, basic components of each computer programming generation. You also saw different characteristics of each programming language generation in terms of computer characteristic, their capabilities, trend and development in computer hardware for different generation.

6.0 TUTOR-MARKED ASSIGNMENT

1. Categorized each level of programming language on different generation of today?
2. Discuss the capabilities that distinguished on generation from another
3. Explain the development of computer hardware in each generation of programming languages
4. Describe the advantages and disadvantages of some programming environment you have used.
5. What is the name of the category of programming languages whose structure is dictated by the von Neumann computer architecture?
6. What two programming language deficiencies were discovered as a result of the research in software development in the 1970s?

7.0 REFERENCES/FURTHER READING

1. A History of Computer Programming Languages (brown.edu)
2. A Brief History of Programming Languages." http://www.byte.com/art/9509/se7/artl9.htm. Cited, March 25, 2000.

3. A Short History of the Computer." http://www.softlord.com/comp/. Jeremy Myers. Cited, March 25, 2000.
4. Bergin, Thomas J. and Richard G. Gibson, eds. *History of Programming Languages-II*. New York: ACM Press, 1996.
5. Christiansen, Tom and Nathan Torkington. *Perlfaq1 Unix Manpage*. Perl 5 Porters, 1997-1999.
6. Java History." http://ils.unc.edu/blaze/java/javahist.html. Cited, March 29, 2000.
8. Programming Languages." *McGraw-Hill Encyclopedia of Science and Technology*. New York: McGraw-Hill, 1997.
9. Wexelblat, Richard L., ed. *History of Programming Languages*. New York: Academic Press, 1981.
10. A History of Computer Programming Languages (onlinecollegeplan.com)
11. IT Lecture 1 History of Computers.pdf (webs.com) Prepared by Miss N. Nembhard, Source: Robers, Eric S. The Art and Science of C. Addison-Wesley Publishing Company. Reading: 1995.

Unit 2 REASONS FOR STUDYING CONCEPTS OF PROGRAMMING LANGUAGE

CONTENTS

1.0 INTRODUCTION

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. The following is what we believe to be a compelling list of potential benefits of studying concepts of programming languages.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- To increase the student's capacity to use different constructs and develop effective algorithms
- To improve use of existing programming and enable students to choose language more intelligently
- To make learning new languages easier and design your own language
- To encounter fascinating ways of programming you might never have imagined before.

3.0 MAIN CONTENT

3.1 Application Domains

3.1.1 Increased ability to express ideas/algorithms

In Natural language, the depth at which people think is influenced by the expressive power of the language they use. In programming language, the complexity of the algorithms that people Implement is influenced by the set of constructs available in the programming language. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, limiting the forms of algorithms they can construct. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs. In other words, the study of programming language concepts builds an appreciation for valuable language features and constructs and encourages programmers to use them, even when the language they are using does not directly support such features and constructs.

3.1.2 Improved background for choosing appropriate Languages

Many programmers use the language with which they are most familiar, even though poorly suited for their new project. It is ideal to use the most appropriate language. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

3.1.3 Increased ability to learn new languages

For instance, knowing the concept s of object oriented programming OOP makes learning Java significantly easier and also, knowing the grammar of one's native language makes it easier to learn another language. If thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned therefore it is essential that practicing programmers know the vocabulary and fundamental concepts of programming languages so they can read and understand programming language descriptions and evaluations, as well as promotional literature for languages and compilers.

3.1.4. Better Understanding of Significance of implementation

This leads to understanding of why languages are designed the way they are. This is an ability to use a language more intellectually, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices. Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. It allows visualization of how a computer executes various language constructs. It provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

3.1.5. Better use of languages that are already known

Many contemporary programming languages are large and complex. It is uncommon for a programmer to be familiar with and use all of the features of a language uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

3.16. The overall advancement of computing

The study of programming language concepts should be justified and the choose languages should be well informed so that better languages would eventually squeeze out poorer ones.

SELF-ASSESSMENT EXERCISE

1. Itemize the reasons responsible for program languages concept
2. Discuss four (4) out the reasons mentioned

4.0 CONCLUSION

This unit has explained different reasons for studying concepts of programming languages, enables students to know language more intelligently, make programming easy to learn, enhanced problem solving skills, help to choose language appropriate for a particular project, internally diverse networking, it has also created opportunities for invention and innovation which enable to learning new languages.

5.0 SUMMARY

Like any language (spoken or written), it is easier to learn earlier in life. Computer languages teach us logical skills in thinking, processing and communicating. Combined with creative visions, some of the most influential products were designed around a programming language. The best inventions are born where skills and creativity meet. After all, who is better than the young generations of the world to imagine the future if our technology? One of the benefit of learning how to code at a young age is enhanced academic performance. Lear ning how this technology works will prepare children for a quick advancing world in which computers and smartphones are utilized for almost every function of our daily lives.

6.0 TUTOR-MARKED ASSIGNMENT

1. What makes a programming language successful?
2. Enumerate and explain the various types of errors that can occur during the execution of a computer program?
3. Please explain an algorithm. What are some of its important features?
4. What do you understand by machine code?
5. What do you mean by "beta version" of a computer program?

7.0 REFERENCES/FURTHER READING

1. Michael L. Scoot. Programming Language Pragmatics. Morgan Kaufmann, 2006.
2. Rowe NC. (1988) Artificial Intelligence through Prolog. Prentice-Hall
3. Olusegun Folorunso, Organization of Programming Languages, The Department of Computer Science, University of Agriculture, Abeokuta.
4. Why Study Programming Languages? (lmu.edu), Hipster Hacker https://cs.Imu.edu/ray/notes/whystudyprogramminglanguages/
5. Reasons Why You Should Learn BASIC Programming | UoPeople https://www.uoppeople.edu/blog/6-reasons-why-you-should-learn-basic-programming/
6. Programming Language Concepts Test Questions/Answers, Dino Cajic, (dinocajic.blogspot.com), https://dinocajic.blogspot.com/2017/02/programming-language-concepts-test,html
7. 50+ Best Programming Interview Questions and Answers in 2021 (hackr.io) https://hackr.io/blog/programming-interview-questions
8. ide.geeksforgeeks.org, 5th Floor, A-118, Sector-136, Noida, Uttar Pradesh - 201305
9. Bernd Teufel, Organization of Programming Languages, Springer-Verlag/Wien, 1991

UNIT 3 APPLICATION DOMAIN

CONTENTS

1.0 INTRODUCTION

Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones. Because of this great diversity in computer use, programming languages with very different goals have been developed. This discuss areas of computer applications and their associated languages.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Discuss different area of application domain
- Highlight associated languages that is applicable to each domain
- Describe fundamental properties of programming languages.

3.0 MAIN CONTENT

3.1 Application Domains

One of the prerequisites for the development of a programming language is that we have a definition and a clear understanding of the contents of the application domain concerned. This is the part of an organization for which application software is developed. This means that the application domain is our starting point and the context for programming language to software development. Many development methodologies take this understanding of the application domain for granted. They assume that the developers somehow know what domain they have to deal with. In respect of this, application domain is divided into categories:

3.1.1 Scientific Applications

Scientific application can be characterized as those which predominantly manipulate numbers and arrays of numbers, using mathematical and statistical principles as a basis for the algorithms. These algorithms encompass such problem as statistical significance test, linear programming, regression analysis and numerical approximations for the solution of differential and integral equations. Example of such programming languages are FORTRAN, Pascal, Meth lab.

3.1.2 Data processing Applications

This can be characterized as those programming problems whose predominant interest is in the creation, maintenance, extraction and summarization of data in records and files. COBOL is a programming language is the first high level language used for business application and still commonly used language. Business languages are characterized by facilities for producing elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify decimal arithmetic operations.

3.1.3 Text processing Applications

These are characterized as those whose principal activity involves the manipulation of natural language text, rather than numbers as their data. SNOBOL and C language have strong text processing capabilities

3.1.4 Artificial intelligence Applications

These are characterized as those programs which are designed principally to emulate intelligent behavior. They include game playing algorithms such as chess, natural language understanding programs, computer vision, robotics and expert systems. LISP has been the predominant AI programming language, and also PROLOG using the principle of ‘’Logic programming’’ Lately AI applications are written in Java, C++ and python.

3.15 Systems Programming Applications

System programming applications involve developing those programs that interface the computer system (the hardware) with the programmer and the operator. These programs include compilers, assembles, interpreters, input-output routines, program management facilities and schedules for utilizing and serving the various resources that comprise the system. Ada, C and Modula – 2 are examples of programming languages used.

3.1.6 Web software

The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript or PHP. There are also some markup-like languages that have been extended to include constructs that control document processing, collection of languages includes: Markup (e.g. XHTML) - Scripting for dynamic content under which there are: Client side, using scripts embedded in the XHTML documents e.g. JavaScript, PHP Server side, using the common Gateway interface e.g. JSP, ASP, PHP General-purpose, executed on the web server through e.g. Java, C++

SELF-ASSESSMENT EXERCISE

1. Discuss other application domain you know that is not mention here
2. What is the disadvantages of having too many features in a language?

4.0 CONCLUSION

An application domain is a mechanism (similar to a process in an operating system) used within the Common Language Infrastructure (CLI) to isolate executed software applications from one another so that they do not affect each other. Categories of application domains are discussed extensively based on scientific application, data processing

application, test processing application, artificial intelligence application, system programming application and web software with different choice of programming languages that can be used for them.

5.0 SUMMARY

Computers are used in a wide variety of problem-solving domains. This unit has explained how the design and evaluation of a particular programming language is highly dependent on the domain in which it is to be used, it has enables students to know language more intelligently and be able to choose language appropriate for a particular project and to make learning new languages easier.

6.0 TUTOR-MARKED ASSIGNMENT

1. What makes a programming language successful?
2. What programming language has dominated scientific computing over the past 50 years?
3. What programming language has dominated business applications over the past 50 years?
4. What programming language has dominated artificial intelligence over the past 50 years?
5. In what language is most of UNIX written?

7.0 REFERENCES/FURTHER READING

1. Michael L. Scoot. Programming Language Pragmatics. Morgan Kaufmann, 2006.
2. Rowe NC. (1988) Artificial Intelligence through Prolog. Prentice-Hall
3. ide.geeksforgeeks.org, 5th Floor, A-118, Sector-136, Noida, Uttar Pradesh - 201305
4. Bernd Teufel, Organization of Programming Languages, Springer-Verlag/Wien, 1991
5. Concepts of Programming Languages, Tenth Edition, Robert W, Sebesta, University of Colorado, Colorado Springs Pearson.
6. Olusegun Folorunso, Organization of Programming Languages, The Department of Computer Science, University of Agriculture, Abeokuta

UNIT 4 LANGUAGE EVALUATION CRITERIA

CONTENTS

1.0 INTRODUCTION

The design and evaluation of programming languages is a challenging area because; as discussed earlier and seen there is no such thing as a "best" language. Instead, existing languages are strong

by some criteria and weak by the others, so that the choice o f a language for a particular purpose is tied to a decision a s t o which criteria are most important. In order to understand the various constructs of a programming language and its capabilities, it is useful to know some evaluation criteria. Each of these criteria is determined and influenced by a certain number of language qualities, determined and implemented during the design of the language. Therefore, design determines language characteristics that make it good for a specific task resolution.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- To consider criteria for evaluating programming languages
- Highlight associated criteria
- Show characteristics that affect those criteria

3.0 MAIN CONTENT

3.1 Language evaluation criteria

1. **Expressivity:** means the ability of a language to clearly reflect the meaning intended by the algorithm designer (the programmer). Thus an "expressive" language permits an utterance to be compactly stated, and encourages the use of statement forms associated with structured programming (usually "while "loops and "if – then – else" statements).

2. **Well-Definedness:** By "well-definiteness", we mean that the language's syntax and semantics are free of ambiguity, are internally consistent and complete. Thus the implementer of a well-defined language should have, within its definition a complete specification of all the language's expressive forms and their meanings. The programmer, by the same virtue should be able to predict exactly the behavior of each expression before it is actually executed.

3. **Data types and structures:** By "Data types and Structures", we mean the ability of a language to support a variety of data values (integers, real, strings, pointers etc.) and non-elementary collect ions of these.

4. **Readability:** One of the most important criteria for judging a programming language is the ease with which programs can be read and understood. Maintenance was recognized as a major part of the cycle, particularly in terms of cost and once the ease of maintenance is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages.
5. **Overall Simplicity:** The overall simplicity of a programming language strongly affects its readability. A language with a large number of basic constructs is more difficult to learn than one with a smaller number.

6. **Modularity:** Modularity has two aspects: the language's support for sub-programming and the language's extensibility in the sense of allowing programmer – defined operators and data types. By sub programming, we mean the ability to define independent procedures and

functions (subprograms), and communicate via parameters or global variables with the invoking program.

7. **Input-Output facilities:** In evaluating a language's "Input-Output facilities" we are looking at its support for sequential, indexed, and random access files, as well as its support for database and information retrieval functions.

8. **Portability:** A language which has "portability" is one which is implemented on a variety of computers. That is, its design is relatively "machine – independent". Languages which are well- defined tend to be more portable than others.

9. **Efficiency:** An "efficient" language is one which permits fast compilation and execution on the machines where it is implemented. Traditionally, FORTRAN and COBOL have been relatively efficient languages in their respective application areas.

10. **Orthogonality:** in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful.

11. **Pedagogy:** Some languages have better "pedagogy" than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

12. **Generality:** Generality: Means that a language is useful in a wide range of programming applications. For instance, APL has been used in mathematical applications involving matrix algebra and in business applications as well. The table below show language evaluation criteria and the characteristics that affect them.

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

SELF-ASSESSMENT EXERCISE

1. Why is readability important to writability?

2. How is the cost of compilers for a given language related to the design of that language?

4.0 CONCLUSION

Among the most important criteria for evaluating languages are readability, writability, reliability, and overall cost. These will be the basis on which we examine and judge the various language features that were discussed.

5.0 SUMMARY

Evaluation results are likely to suggest that your program has strengths as well as limitations, which should not be a simple declaration of program success or failure. Evidence that your EE program is not achieving all of its ambitious objectives can be hard to swallow, but it can also help to learn where to best to put limited resources. A good evaluation is one that is likely to be replicable, meaning that the same evaluation should be conducted and get the same results. The higher the quality of the evaluation design, its data collection methods and its data analysis, the more accurate its conclusions and the more confident others will be in its findings.

6.0 TUTOR-MARKED ASSIGNMENT

1. What are some features of specific programming languages you know whose rationales are a mystery to you?
2. What common programming language statement, in your opinion, is most detrimental to readability?
3. Explain the different aspects of the cost of a programming language.?
4. In your opinion, what major features would a perfect programming language include
5. Write an evaluation of some programming language you know, using the criteria described

7.0 REFERENCES/FURTHER READING

1. Sebesta: Concept of programming Language, Pearson Edu
2. Louden: Programming Languages: *Principles and Practices,* Cengage Learning
3. Tucker: Programming Languages: *Principles and paradigms,* Tata McGraw –Hill.
4. E Horowitz: Programming Languages, 2nd Edition, Addison Wesley
5. ide.geeksforgeeks.org, 5th Floor, A-118, Sector-136, Noida, Uttar Pradesh – 201305
6. Programming Language Evaluation Criteria Part 1: Readability | by ngwes | Medium
7. Language Evaluation Criteria (gordon.edu)
8. Language Evaluation Criteria PPL by Jayesh Umre

MODULE 2 INFLUENCE OF LANGUAGE DESIGN

Unit 1 Computer architecture
Unit 2 Language paradigms
Unit 3 Language design trade-offs,
Unit 4 Implementation method

UNIT 1 COMPUTER ARCHITECTURE

CONTENTS

1.0 INTRODUCTION

Primary influences on language design has a profound effect on the basic Computer architecture in which languages are developed around computer architecture, known as the Von Neumann architecture (the most prevalent computer architecture). Pedagogy: Some languages have better "pedagogy" than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

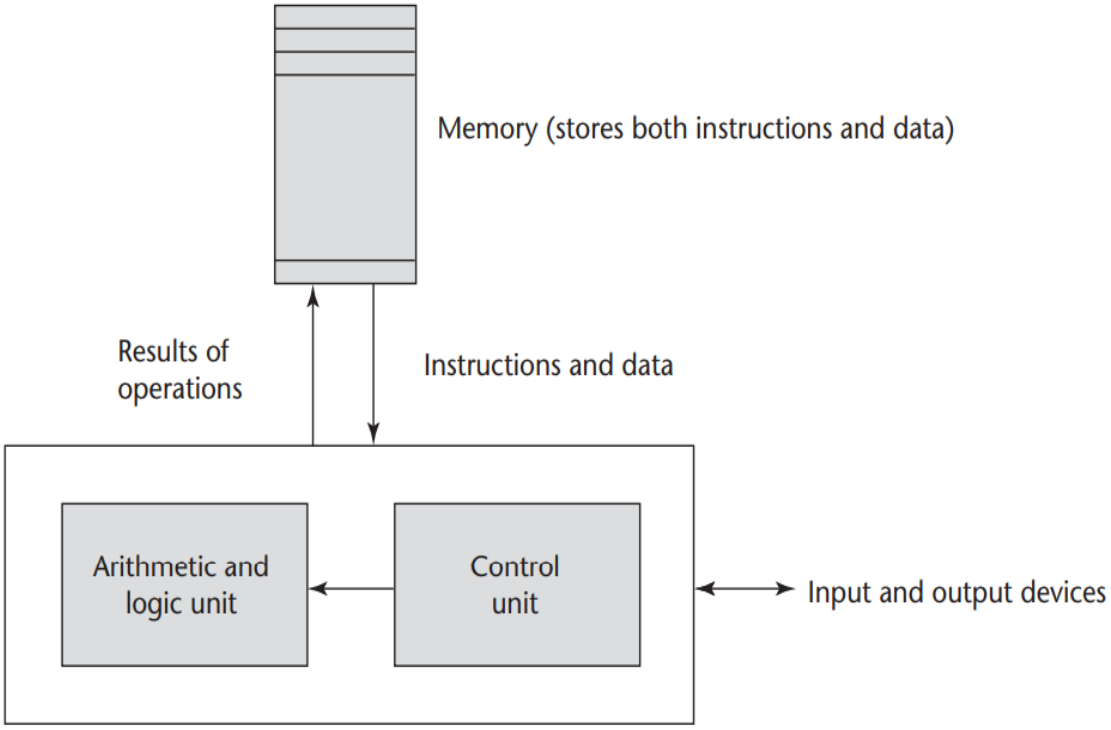
- Ability to store instructions in the memory along with the data on which the instructions operate
- To know how to partition the overall memory into two smaller memories, one containing instructions and the other containing data.

3.0 MAIN CONTENT

3.1 Computer Architecture

A computer scientist John von Neumann in 1945 described a design architecture for an electronic digital computer with subdivisions of a central arithmetic part, a central control part, a memory to store both data and instructions, external storage, and input and output mechanisms. John Von Neumann introduced the idea of the stored program which is used to keep programmed

instructions, as well as its data, in read-write, random-access memory (RAM). Previously data and programs were stored in separate memories. Von Neumann realized that data and programs are indistinguishable and can, therefore, use the same memory. On a large scale, the ability to treat instructions as data is what makes assemblers, compilers and other automated programming tools possible. One can "write programs which write programs". This led to the introduction of compilers which accepted high level language source code as input and produced binary code as output.



According to Von Neumann Architecture, the basic function performed by a computer is the execution of a program. A program is a set of machine instructions**.** An instruction is a form of control code, which supplies the information about an operation and the data on which the operation is to be performed. The Von Neumann architecture uses a single processor which follows a linear sequence of fetch-decode-execute. In order to do this, the processor has to use some special registers, which are discrete memory locations with special purposes attached. These are:

Register	Meaning
PC	Program Counter
CIR	Current Instruction Register
MAR	Memory Address Register
MDR	Memory Direct Register
IR	Index Register
Accumulator	Hold Result

- The program counter keeps track of where to find the next instruction so that a copy of the instruction can be placed in the current instruction register. Sometimes the program counter

is called the Sequence Control Register (SCR) as it controls the sequence in which instructions are executed.

- The current instruction register holds the instruction that is to be executed. The memory address register is used to hold the memory address that contains either the next piece of data or an instruction that is to be used.
- The memory data register acts like a buffer and holds anything that is copied from the memory ready for the processor to use it.
- The central processor contains the arithmetic-logic unit (also known as the arithmetic unit) and the control unit.
- The arithmetic-logic unit (ALU) is where data is processed. This involves arithmetic and logical operations. Arithmetic operations are those that add and subtract numbers, and so on. Logical operations involve comparing binary patterns and making decisions.
- The control unit fetches instructions from memory, decodes them and synchronises the operations before sending signals to other parts of the computer.
- The accumulator is in the arithmetic unit, the program counter and the instruction registers are in the control unit and the memory data register and memory address register are in the processor.
- An index register is a microprocessor register used for modifying operand addresses during the run of a program, typically for doing vector/array operations. Index registers are used for a special kind of indirect addressing (covered in 3.5 (i)) where an immediate constant (i.e. which is part of the instruction itself) is added to the contents of the index register to form the address to the actual operand or data.

3.2 The Fetch-Decode-Execute-Reset Cycle

The following is an algorithm that shows the steps in the cycle. At the end the cycle is reset and the algorithm repeated.

- Load the address that is in the program counter (PC) into the memory address register (MAR). 2. Increment the PC by 1.
- Load the instruction that is in the memory address given by the MAR into the memory data register (MDR).
- Load the instruction that is now in the MDR into the current instruction register (CIR).
- Decode the instruction that is in the CIR.
- If the instruction is a jump instruction, then a. Load the address part of the instruction into the PC b. Reset by going to step 1.
- Execute the instruction.
- Reset by going to step 1.

Steps 1 to 4 are the fetch part of the cycle. Steps 5, 6a and 7 are the execute part of the cycle and steps 6b and 8 are the reset part.

Step 1 simply places the address of the next instruction into the memory address register so that the control unit can fetch the instruction from the right part of the memory. The program counter is then incremented by 1 so that it contains the address of the next instruction, assuming that the instructions are in consecutive locations. The memory data register is used whenever anything is

to go from the central processing unit to main memory, or vice versa. Thus the next instruction is copied from memory into the MDR and is then copied into the current instruction register. Now that the instruction has been fetched the control unit can decode it and decide what has to be done. This is the execute part of the cycle. If it is an arithmetic instruction, this can be executed and the cycle restarted as the PC contains the address of the next instruction in order. However, if the instruction involves jumping to an instruction that is not the next one in order, the PC has to be loaded with the address of the instruction that is to be executed next. This address is in the address part of the current instruction, hence the address part is loaded into the PC before the cycle is reset and starts all over again.

4.0 CONCLUSION

Von Neumann architecture describes a design architecture for an electronic digital computer with subdivisions of a central arithmetic part, a central control part, a memory to store both data and instructions, external storage, and input and output mechanisms**.** The meaning of the phrase has evolved to mean a stored-program computer. A stored-program digital computer is one that keeps its programmed instructions, as well as its data, in read-write, random-access memory (RAM). So he introduced the idea of the stored program. Previously data and programs were stored in separate memories. Von Neumann realized that data and programs are indistinguishable and can, therefore, use the same memory. On a large scale, the ability to treat instructions as data is what makes assemblers, compilers and other automated programming tools possible. One can "write programs which write programs". This led to the introduction of compilers which accepted high level language source code as input and produced binary code as output.

5.0 SUMMARY

The major influences on language design have been machine architecture and software design methodologies which is a theoretical design for a stored program computer that serves as the basis for almost all modern computers.

6.0 Tutor-Marked Assignment

1. Explain what is meant by the term Von Neumann Architecture.
2. Describe the fetch/decode part of the fetch/decode/execute/reset cycle, explaining the purpose of any special registers that you have mentioned.
3. Describe two ways in which the program counter can change during the normal execution of a program, explaining, in each case, how this change is initiated.
4. What is the name of the category of programming languages whose structure is dictated by the von Neumann computer architecture?

7.0 REFERENCE/FURTHER READING

1. Chapter_3_von_Neumanns_Architecture.pdf (weebly.com), https://viennaict.weebly.com.
2. What Is A Programming Paradigm?. An overview of programming paradigms… | by Emmett Boudreau | Towards Data Science
3. Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning.

4. Michael L. Scott, Programming Language Pragmatics, Third Edition, Morgan Kaufmann Publishers, Elsevier.
5. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.
6. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
7. Questions-on-principle-of-programming-language.pdf (oureducation.in), https://blog/oureducation.in/wp-content/uploads/2013/Questions-on-principle -of-programming-language.pdf#

UNIT 2 LANGUAGE PARADIGMS

CONTENTS

1.0 INTRODUCTION

When programs are developed to solve real-life problems like inventory management, payroll processing, student admissions, examination result processing, etc. they tend to be huge and complex. The approach to analyzing such complex problems, planning for software development and controlling the development process is called programming methodology. New software development methodologies (e.g. object Oriented Software Development) led to new paradigms in programming and by extension, to new programming languages. A programming paradigm is a pattern of problem-solving thought that underlies a particular genre of programs and languages. Also a programming paradigm is the concept by which the methodology of a programming language adheres to.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- To introduces several different paradigms of programming
- To explain the history and evolution of different languages paradigm
- To gain experience with these paradigms by using examples programming languages
- Describe fundamental properties of programming languages
- To better understand the significance of programming implementation

- To understand how to break problems of different paradigm into smaller units and place them into different approach.

3.0 MAIN CONTENT

3.1 Language Paradigm

Paradigm is a model or world view. Paradigms are important because they define a programming language and how it works. A great way to think about a paradigm is as a set of ideas that a programming language can use to perform tasks in terms of machine-code at a much higher level. These different approaches can be better in some cases, and worse in others. A great rule of thumb when exploring paradigms is to understand what they are good at. While it is true that most modern programming languages are general-purpose and can do just about anything, it might be more difficult to develop a game, for example, in a functional language than an object-oriented language. Many people classify languages into these main paradigms:

3.1.1 Imperative/Procedural programming languages

These are mostly influenced by the von Neumann computer architecture. Problem is broken down into procedures, or blocks of code that perform one task each. All procedures taken together form the whole program. It is suitable only for small programs that have low level of complexity. Typical elements of such languages are assignment statements, data structures and type binding, as well as control mechanisms; active procedures manipulate passive data objects. Example − For a calculator program that does addition, subtraction, multiplication, division, square root and comparison, each of these operations can be developed as separate procedures. In the main program each procedure would be invoked on the basis of user's choice. E.g. FORTRAN Algol, Pascal, C/C++, C#, Java, Perl, JavaScript, Visual BASIC.NET.

3.1.2. Functional/applicative programming languages

These type of languages have no assignment statements. Their syntax is closely related to the formulation of mathematical functions. Thus, functions are central for functional programming languages. Here the problem, or the desired solution, is broken down into functional units. Each unit performs its own task and is self-sufficient. These units are then stitched together to form the complete solution. Example − A payroll processing can have functional units like employee data maintenance, basic salary calculation, gross salary calculation, leave processing, loan repayment processing, etc. E.g. LISP, Scala, Haskell, Python, Clojure, Erlang. It may also include OO (Object Oriented) concepts.

3.1.3 Logic/declarative/ruled based programming languages

Facts and rules (the logic) are used to represent information (or knowledge) and a logical inference process is used to produce results. In contrast to this, control structures are not explicitly defined in a program, they are part of the programming language (inference mechanism). Here the problem is broken down into logical units rather than functional units. Example: In a school management system, users have very defined roles like class teacher, subject teacher, lab assistant, coordinator, academic in-charge, etc. So the software can be divided into units depending on user roles. Each

user can have different interface, permissions, etc. e.g. PROLOG, PERL, this may also include OO concepts.

3.1.4. Object-oriented programming languages

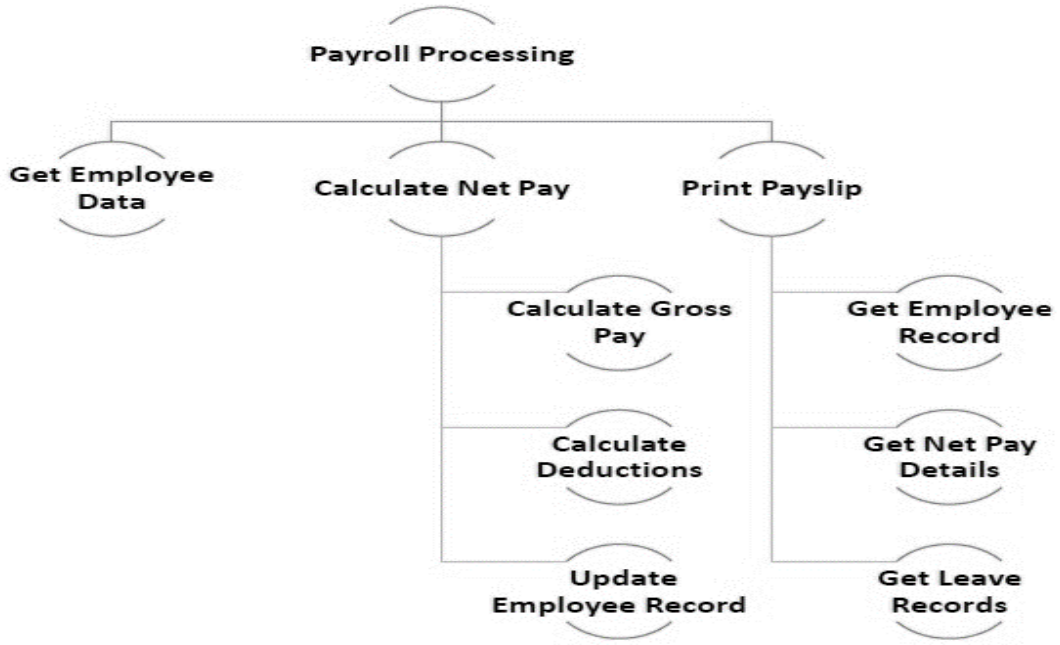
know the object representing data as well as procedures. Data structures and their appropriate manipulation processes are packed together to form a syntactical unit. Here the solution revolves around entities or objects that are part of problem. The solution deals with how to store data related to the entities, how the entities behave and how they interact with each other to give a cohesive solution. Example − If we have to develop a payroll management system, we will have entities like employees, salary structure, leave rules, etc. around which the solution must be built. E.g. SIMULA 67, SMALLTALK, C++, Java, Python, C#, Perl, Lisp or EIFFEL.

3.1.5. Scripting Language

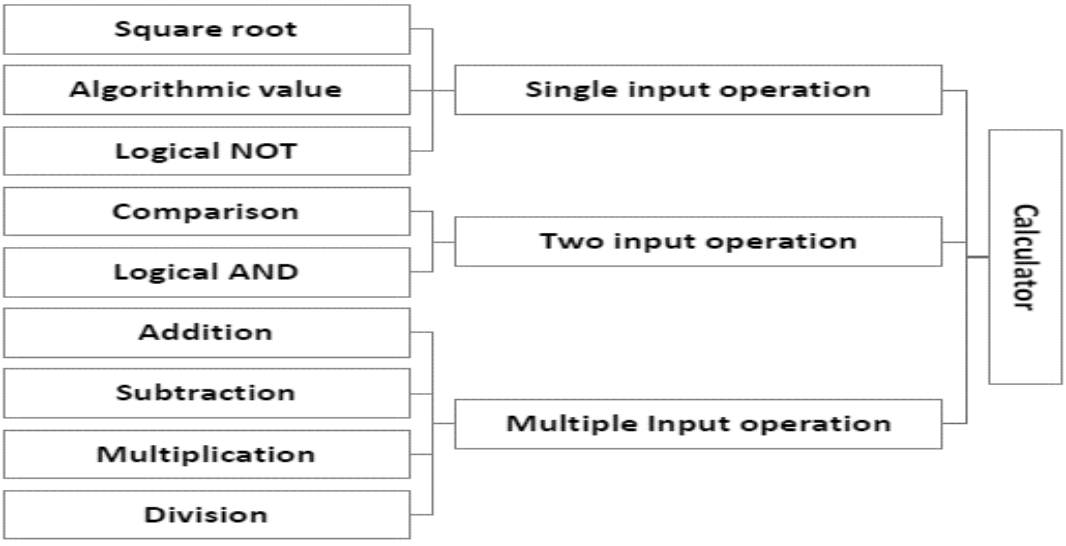
A scripting language or script language is a programming language for a runtime system that automates the execution of tasks that would otherwise be performed individually by a human operator. Scripting languages are usually interpreted at runtime rather than compiled. Scripting languages are a popular family of programming languages that allow frequent tasks to be performed quickly. Early scripting languages were generally used for niche applications – and as "glue languages" for combining existing systems. With the rise of the World Wide Web, a range of scripting languages emerged for use on web servers. Since scripting languages simplify the processing of text, they are ideally suited to the dynamic generation of HTML pages**.** The user can learn to code in scripting languages quickly, not much knowledge of web technology is required. It is highly efficient with the limited number of data structures and variables to use, it helps in adding visualization interfaces and combinations in web pages. There are different libraries which are part of different scripting languages. They help in creating new applications in web browsers and are different from normal programming languages. Examples are Node js, JavaScript, Ruby, Python, Perl, bash, PHP etc.

Software developers may choose one or a combination of more than one of these methodologies to develop a software. Note that in each of the methodologies discussed, problem has to be broken down into smaller units. To do this, developers use any of the following two approaches:

Top-down approach: The problem is broken down into smaller units, which may be further broken down into even smaller units. Each unit is called a module. Each module is a self-sufficient unit that has everything necessary to perform its task. The following illustration shows an example of how you can follow modular approach to create different modules while developing a payroll processing program.



Bottom-up Approach: In bottom-up approach, system design starts with the lowest level of components, which are then interconnected to get higher level components. This process continues till a hierarchy of all system components is generated. However, in real-life scenario it is very difficult to know all lowest level components at the outset. So bottoms up approach is used only for very simple problems. Let us look at the components of a calculator program.



SELF-ASSESSMENT EXERCISE

1. What was the first functional language?
2. Name three languages in each of the following categories: von Neumann, functional, object oriented.
3. Name two logic languages.

4. Name two widely used concurrent languages.

4.0 CONCLUSION

All programming paradigms have their benefits to both education and ability. Functional languages historically have been very notable in the world of scientific computing. Of course, taking a list of the most popular languages for scientific computing today, it would be obvious that they are all multi-paradigm. Object-oriented languages also have their fair share of great applications. Software development, game development, and graphics programming are all great examples of where object-oriented programming is a great approach to take.

The biggest note one can take from all of this information is that the future of software and programming language is multi-paradigm. It is unlikely that anyone will be creating a purely functional or object-oriented programming language anytime soon. If you ask me, this isn't such a bad thing, as there are weaknesses and strengths to every programming approach that you take, and a lot of true optimization is performing tests to see which methodology is more efficient or better than the other overall. This also puts a bigger thumbtack into the idea that everyone should know multiple languages from multiple paradigms. With the paradigms merging using the power of generics, it is never known when one might run into a programming concept from an entirely different programming language

5.0 SUMMARY

The unit has explained what programming language is, classification and explanation of different programming language generation, basic components of each computer programming generation. You also saw different characteristics of each programming language generation in terms of computer characteristic, their capabilities, trend and development in computer hardware for different generation.

6.0 TUTOR-MARKED ASSIGNMENT

1. What are the three fundamental features of an object-oriented programming language? 23.
2. What language was the first to sup
3. What distinguishes declarative languages from imperative languages?
4. What is generally considered the first high-level programming language?
5. Why aren't concurrent languages listed as a category

7.0 REFERENCES/FURTHER READING

1. What Is A Programming Paradigm?. An overview of programming paradigms… | by Emmett Boudreau | Towards Data Science
2. Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning
3. Michael L. Scott, Programming Language Pragmatics, Third Edition, Morgan Kaufmann Publishers, Elsevier.
4. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.

5. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
6. Questions-on-principle-of-programming-language.pdf (oureducation.in), https://blog/oureducation.in/wp-content/uploads/2013/Questions-on-principle -of-programming-language.pdf#

UNIT 3 LANGUAGE TRADE-OFFS DESIGN

CONTENTS

1.0 INTRODUCTION

Programming paradigms, like software architecture, have trade-offs. In fact, many of the same methods for comparing architectural designs apply just as well to language design. Conceptual design involves a series of trade-off decisions among significant parameters such as operating speeds, memory size, power, and I/O bandwidth - to obtain a compromise design which best meets the performance requirements. Both the uncertainty in these requirements and the important trade-off factors should be ascertained. Those factors can be used to evaluate the design trade-offs (usually on a qualitative basis).

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- To be able to express algorithms in a relatively simple way
- Prioritizing between stability and evolution of successful language
- To be able to develop a general purpose production focused programming language
- Improve the background for choosing appropriate programming languages for certain classes of programming problems.

3.0 MAIN CONTENT

3.1 Language Trade-Offs Design

A trade-off is made when using an interpreted language. One can trade speed of development for higher execution costs**.** Because each line of an interpreted program must be translated each time it is executed, there is a higher overhead. Thus, an interpreted language is generally more suited to ad hoc requests than predefined requests. That is especially true for programs that are based around

manipulating state over a long term. Trade-off is a clear, logically simple structure that makes complex algorithms easy to build right and scales well but makes stateful systems harder to build. There are many trade-offs in language design such as:

- **Reliability:** this take into account the time required for malfunction detection and reconfiguration or repair.

- **Expandability:** measures the computer system's ability to conveniently accommodate increased requirements by higher speed or by physical expansion without the cost of a major redesign. Modularity is a desirable method for providing expandability and should be incorporated whenever feasible.

- **Programmability:** there should be a balance between programming simplicity and hardware complexity to prevent the cost of programming from becoming overwhelming. The degree of software sophistication and the availability of support software should be considered during the design.

- **Maintainability:** should not be neglected when designing the computer, repair should be readily accomplished during ground operation.

- **Compatibility:** this should be developed between computer and interfaces, software, power levels to facilitate programming.

- **Adaptability:** is defined as the ability of the system to meet a wide range of functional requirements without requiring physical modifications.

- **Availability:** is the portability that the computer is operating satisfactorily at a given time. It is closely related to reliability.

- **Development status and cost:** are complex management factors which can have significant effects on the design as well. They require the estimation of a number of items such as the extent off off-the-shelf hardware use, design risks in developing new equipment using advanced technologies, potential progress in the state of the art during the design and development.

SELF ASSESSMENT EXERCISE

1. What language used orthogonality as a primary design criterion?
2. What does it mean for a program to be reliable?

4.0 CONCLUSION

All programming paradigms have their benefits to both education and ability. Functional languages historically have been very notable in the world of scientific computing. Of course, taking a list of the most popular languages for scientific computing today, it would be obvious that they are all multi-paradigm. Object-oriented languages also have their fair share of great applications. Software

development, game development, and graphics programming are all great examples of where object-oriented programming is a great approach to take.

The biggest note one can take from all of this information is that the future of software and programming language is multi-paradigm. It is unlikely that anyone will be creating a purely functional or object-oriented programming language anytime soon. If you ask me, this isn't such a bad thing, as there are weaknesses and strengths to every programming approach that you take, and a lot of true optimization is performing tests to see which methodology is more efficient or better than the other overall. This also puts a bigger thumbtack into the idea that everyone should know multiple languages from multiple paradigms. With the paradigms merging using the power of generics, it is never known when one might run into a programming concept from an entirely different programming language

5.0 SUMMARY

The major influences on language design have been machine architecture and software design methodologies. Designing a programming language is primarily an engineering feat, in which a long list of trade-offs must be made among features, constructs, and capabilities.

6.0 TUTOR-MARKED ASSIGNMENT

1. What is trade's off of translation**?**
2. What executes programming codes line by line, rather than the whole program
3. Describe some design trade-offs between efficiency and safety in some language you know.
4. In your opinion, what major features would a perfect programming language include?
5. Was the first high-level programming language you learned implemented with a pure interpreter, a hybrid implementation system, or a compiler? (You may have to research this.)

7.0 REFERENCES/FURTHER READING

1. Programming Language Evaluation Criteria Part 1: Readability | by ngwes | Medium
2. Language Evaluation Criteria (gordon.edu)
3. Language Evaluation Criteria PPL by Jayesh Umre
4. Sebesta: Concept of programming Language, Pearson Edu
5. Louden: Programming Languages: Principles and Practices, Cengage Learning
6. Tucker: Programming Languages: Principles and paradigms, Tata McGraw –Hill.
7. E Horowitz: Programming Languages, 2nd Edition, Addison Wesley

UNIT 4 IMPLEMENTATION METHOD

CONTENTS

1.0 INTRODUCTION

A programming language implementation is a system for executing computer programs. There are four approaches or methods to programming language implementation which will be discussed.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Be able in principle to program in any programming paradigm language.
- Understand the significance of an implementation of a programming language in a compiler or interpreter.
- Increase the capacity to express programming concepts and choose among alternative ways to express things
- Simulate useful features in languages that lack them

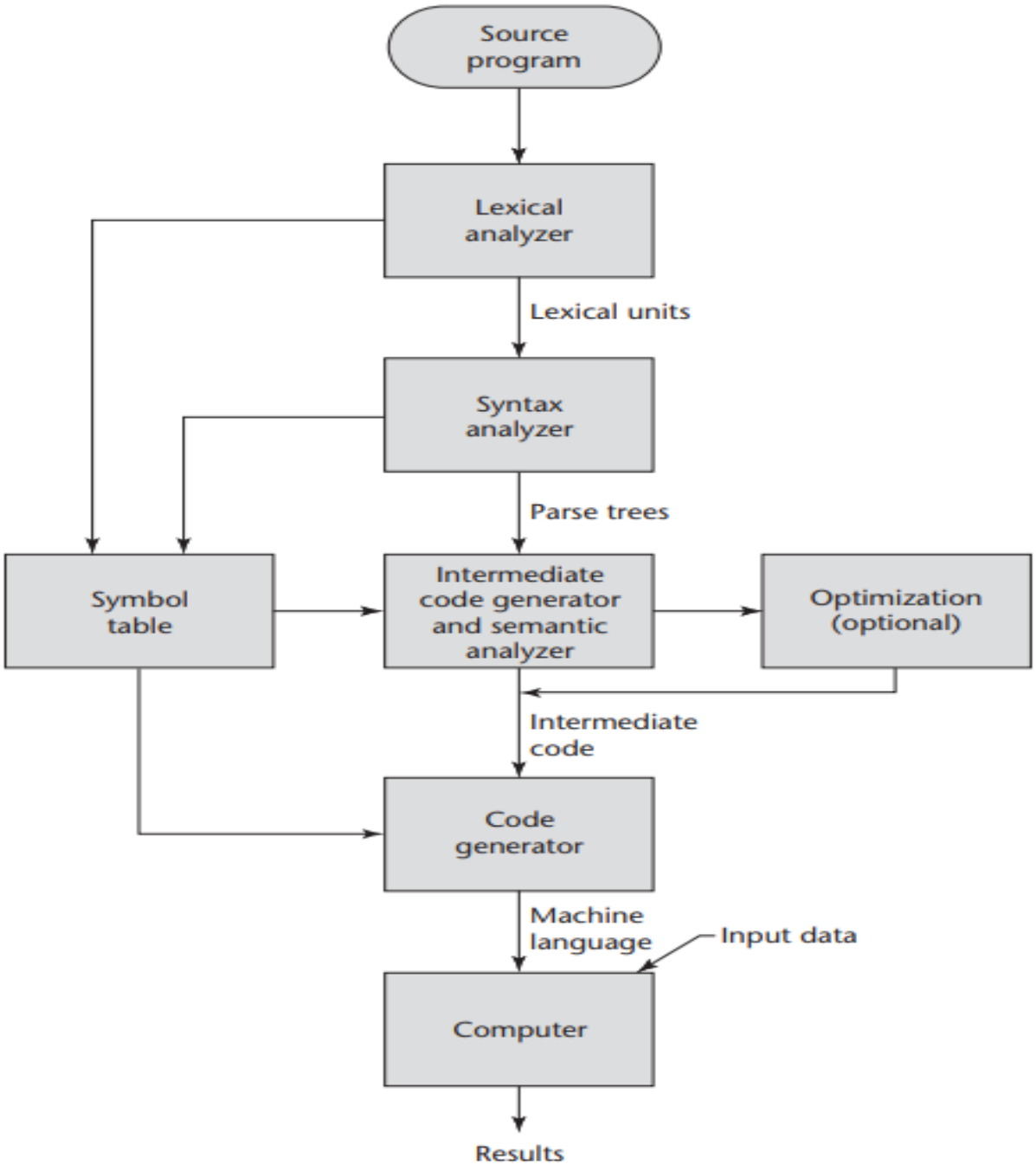
3.0 MAIN CONTENT

3.1 Implementation Method

Explain implementation process in programming language such as:

3.1.1 Compilation

Programs can be translated into machine language, which can be executed directly on the computer. This method is called a compiler implementation and has the advantage of very fast program execution, once the translation process is complete. The language that a compiler translates is called the source language. The process of compilation and program execution takes place in several phases, the most important of which are shown in Figure below:



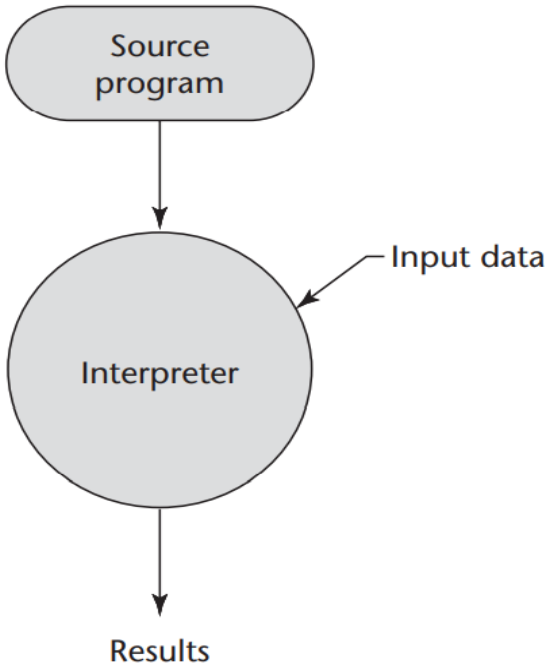
3.1.1.1 Compilation Process

- Lexical analysis converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
- Syntactic analysis: transforms lexical units into parse trees which represent the syntactic structure of the program.
- Semantics analysis check for errors hard to detect during syntactic analysis; generate intermediate code.

- Code generation – Machine code is generated

3.1.2 Interpretation

Pure interpretation lies at the opposite end (from compilation) of implementation methods. With this approach, programs are interpreted by another program called an interpreter, with no translation whatever. The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions. The process of pure interpretation is shown in Figure below:



3.1.2.1 Phases of Interpretation

- Programs are interpreted by another program (an interpreter)
- Easier implementation of programs (run-time errors can easily and immediately be displayed).
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more memory space and is now rare3 for traditional high level languages.
- Significant comeback with some Web scripting languages like PHP and JavaScript.

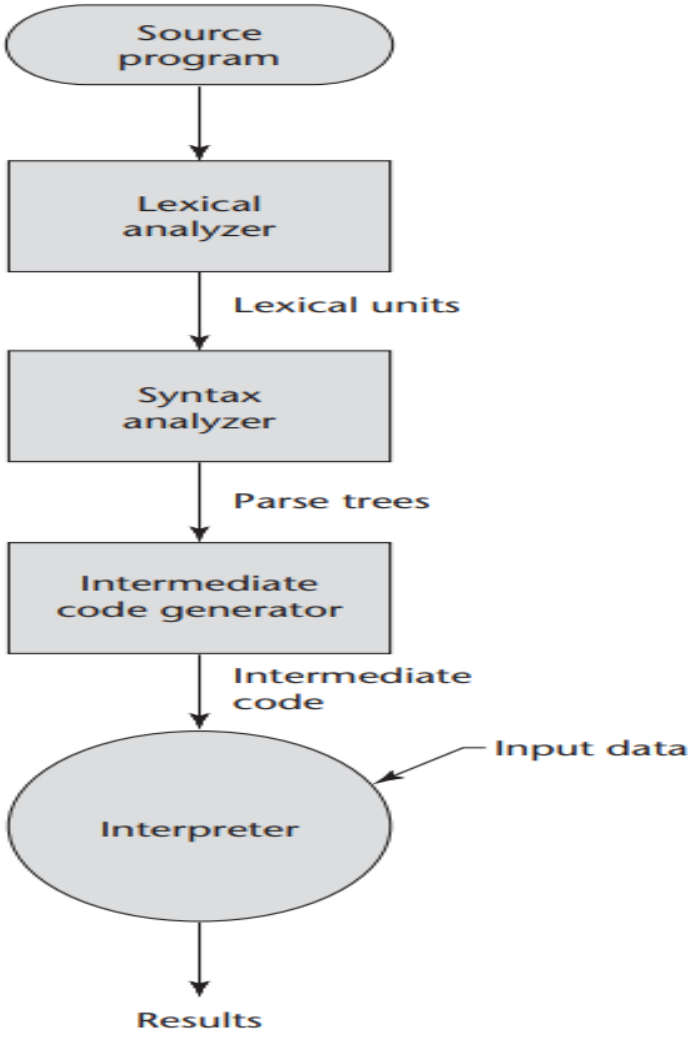
- Interpreters usually implement as a read-eval-print loop:
  - Read expression in the input language (usually translating it in
  - some internal form)
  - Evaluates the internal forms of the expression
  - Print the result of the evaluation
  - Loops and reads the next input expression until exit

- Interpreters act as a virtual machine for the source language:
  - Fetch execute cycle replaced by the read-eval-print loop

- Usually has a core component, called the interpreter "run-time" that is a compile program running on the native machine.

3.1.3 Hybrid

Some language implementation systems are a compromise between compilers and pure interpreters; they translate high-level language programs to an intermediate language designed to allow easy interpretation. This method is faster than pure interpretation because the source language statements are decoded only once. Such implementations are called hybrid implementation systems. The process used in a hybrid implementation system is shown in Figure below:



Instead of translating intermediate language code to machine code, it simply interprets the intermediate code.

- Programs translated into an intermediate language for easy Interpretation
- This involves a compromise between compilers and pure interpreters. A high level program is translated to an intermediate language that allows easy interpretation.

- Hybrid implementation is faster than pure interpretation. Examples of the implementation occur in Perl and Java.
  - Perl programs are partially compiled to detect errors before interpretation.
  - Initial implementat6ions of Java were hybrid. The intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run time system (together, these are called Java Virtual Machine).

3.1.4 Just –in-time

After hybrid, then compile sub programs code the first time they are called. This implementation initially translates programs to an intermediate language then compile the intermediate language of the subprograms into machine code when they are called.

- Machine code version is kept for subsequent calls. Just-in-time systems are widely used for Java programs. Also .NET languages are implemented with a JIT system.

SELF ASSESSMENT EXERCISE

1. What does a linker do?
2. What are the advantages in implementing a language with pure interpreter?
3. What are the three general methods of implementing a programming language?
4. Which produces faster program execution, a compiler or a pure interpreter and how?

4.0 Conclusion

Language implementation are intimately related to one another. Obviously an implementation must conform to the rules of the language. At the same time, a language designer must consider how easy or difficult it will be to implement various features, and what sort of performance is likely to result for programs that use those features. Language implementations are commonly differentiated into those based on interpretation and those based on compilation. However, the difference between these approaches is fuzzy, and that most implementations include a bit of each. As a general rule, a language is compiled if execution is preceded by a translation step that fully analyzes both the structure and meaning of the program, and produces an equivalent program in a significant different form.

5.0 Summary

The major methods of implementing programming languages are compilation, pure interpretation, and hybrid implementation. Programming environments have become important parts of software development systems, in which the language is just one of the components. Implementation method will acquaint you with the fundamental ideas surrounding the design and implementation of high-level programming languages. The course stresses underlying theoretical concepts as well as a significant, practical course project. At the same time, the course focuses on making the material accessible to students of varied backgrounds.

6.0 Tutor-Marked Assignment

1. Give three examples, in different contexts, of abstract machines.
2. Describe the functioning of the interpreter for a generic abstract machine
3. Describe the differences between the interpretative and compiled implementations of a programming language, emphasizing the advantages and disadvantages.
4. What are the advantages in using an intermediate machine for the implementation of a language?
5. What is a just-in-time compiler
6. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?
7. What is a just-in-time compiler?

7.0 References/Further Reading

1. Hal Abelson, Jerry Sussman and Julie Sussman, Structured and Interpretation of Computer Program., Resource for Functional Programming and the Lambda Calculus.
2. Aho, Sethi and Ullman's Compilers: Principles, Techniques and Tools (Dragon book) The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
3. Andrew Appel's Modern Compiler Impementation in Java.
4 Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning
5. Michael L. Scott, Programming Language Pragmatics, Third Edition, Morgan Kaufmann Publishers, Elsevier.
6. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.

MODULE 3 LANGUAGE DESCRIPTION

UNIT 1 FUNDAMENTAL SYNTACTIC ANALYSIS CONCEPT ON UNDERLYING MODERN PROGRAMMING LANGUAGE

CONTENTS

1.0 INTRODUCTION

The task of providing a concise yet understandable description of a programming language is difficult but essential to the language's success has some languages have suffered the problem of having many slightly different dialects, a result of a simple but informal and imprecise definition. One of the problems in describing a language is the diversity of the people who must understand the description. Among these are initial evaluators, implementers, and users. Most new programming languages are subjected to a period of scrutiny by potential users, often people within the organization that employs the language's designer, before their designs are completed. These are the initial evaluators. The success of this feedback cycle depends heavily on the clarity of the description. Programming language implementers obviously must be able to determine how the expressions, statements, and program units of a language are formed, and also their intended effect when executed. The difficulty of the implementers' job is, in part, determined by the completeness and precision of the language description.

Finally, language users must be able to determine how to encode software solutions by referring to a language reference manual. Textbooks and courses enter into this process, but language manuals are usually the only authoritative printed information source about a language. The study of programming languages, like the study of natural languages, can be divided into examinations of syntax and semantics. The syntax of a programming language is the form of its expressions, statements, and program units. Its semantics is the meaning of those expressions, statements, and program units.

For example, the syntax of a Java while statement is:

```
while (boolean_expr) statement
```

The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed. Otherwise, control continues after the while construct. Then control implicitly returns to the Boolean expression to repeat the process. Although they are often separated for discussion purposes, syntax and semantics are closely related. In a well-designed programming language, semantics should follow directly from syntax; that is, the appearance of a statement should strongly suggest what the statement is meant to accomplish. Describing syntax is easier than describing semantics, partly because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- How to learn a syntactical analysis;
- how learn new grammar and syntax of grammar
- understand generative grammar
- to consider rules of grammar in order to define the logical meaning as well as correctness of the sentences.

3.0 MAIN CONTENT

3.1 Syntactic Analysis

A language, whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The strings of a language are called sentences or statements. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language. English, for example, has a large and complex collection of rules for specifying the syntax of its sentences. By comparison, even the largest and most complex programming languages are syntactically very simple.

Syntax is the set of rules that define what the various combinations of symbols mean. This tells the computer how to read the code. Syntax refers to a concept in writing code dealing with a very specific set of words and a very specific order to those words when we give the computer instructions. This order and this strict structure is what enables us to communicate effectively with a computer. Syntax is to code, like grammar is to English or any other language. A big difference though is that computers are really exacting in how we structure that grammar or our syntax. This syntax is why we call programming coding. Even amongst all the different languages that are out there. Each programming language uses different words in a different structure in how we give it information to get the computer to follow our instructions. Syntax analysis is a task performed by a compiler which examines whether the program has a proper associated derivation tree or not. The syntax of a programming language can be interpreted using the following formal and informal techniques:

- Lexical syntax for defining the rules for basic symbols involving identifiers, literals, punctuators and operators.
- Concrete syntax specifies the real representation of the programs with the help of lexical symbols like its alphabet.
- Abstract syntax conveys only the vital program information.

The Syntax of a programming language is used to signify the structure of programs without considering their meaning. It basically emphasizes the structure, layout of a program with their appearance. It involves a collection of rules which validates the sequence of symbols and instruction used in a program. In general, languages can be formally defined in two distinct ways: by recognition and by generation.

3.1.1 Language Recognizer

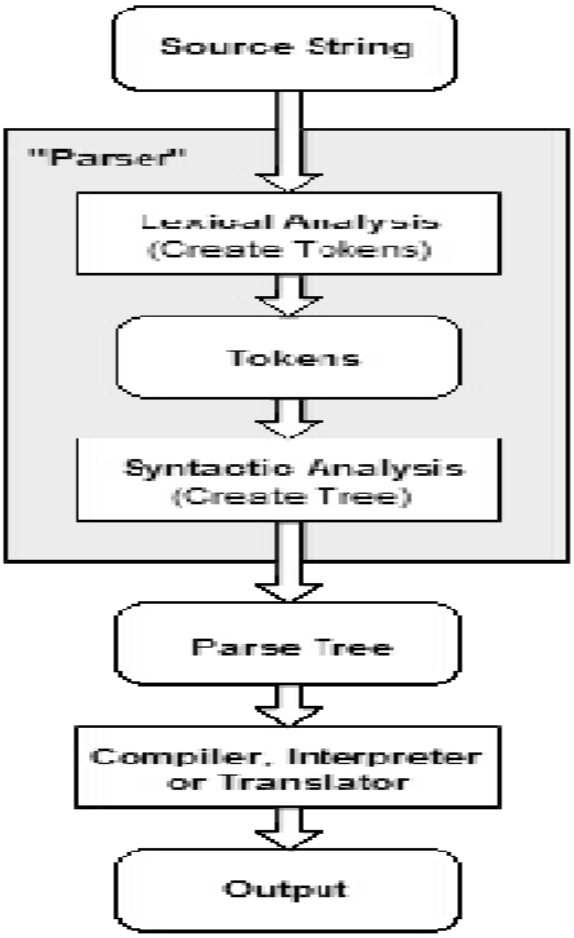
The syntax analysis part of a compiler is a recognizer for the language the compiler translates. In this role, the recognizer need not test all possible strings of characters from some set to determine whether each is in the language. Rather, it need only determine whether given programs are in the language. In effect then, the syntax analyzer determines whether the given programs are syntactically correct. The structure of syntax analyzers, also known as parsers as discussed before. Language recognizer is like a filters, separating legal sentences from those that are incorrectly formed.

3.1.2 Language Generator

A language generator is a device that can be used to generate the sentences of a language. a generator seems to be a device of limited usefulness as a language descriptor. However, people prefer certain forms of generators over recognizers because they can more easily read and understand them. By contrast, the syntax-checking portion of a compiler (a language recognizer) is not as useful a language description for a programmer because it can be used only in trial-and-error mode. For example, to determine the correct syntax of a particular statement using a compiler, the programmer can only submit a speculated version and note whether the compiler accepts it. On the other hand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator. There is a close connection between formal generation and recognition devices for the same language which led to formal languages.

3.1.3 Parsing

In linguistics, parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, especially in reference to how phrases are divided up in garden path sentences. The diagram below shows overview process.

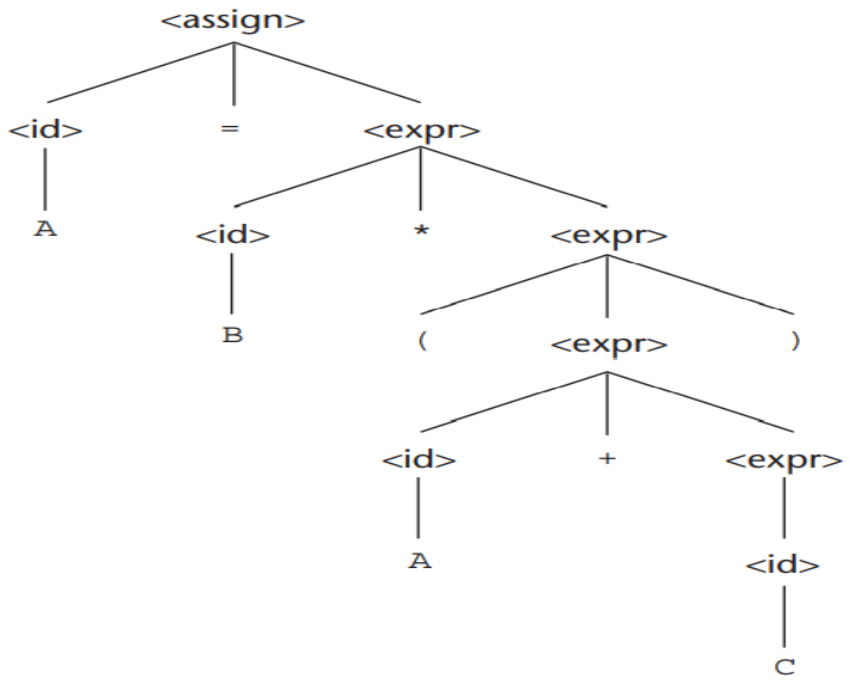


Overview of process

3.1.3.1 Parse Trees

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called parse trees. For example, the parse tree in Figure below showed the structure of the assignment statement derived. Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence. For example:
A parse tree for the simple statement

A = B * (A + C)



3.1.3.2 Parser

In computing, a parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyzer to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool.

3.1.3.3 Types of parser

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- **Top-down parsing:** Top-down parsing can be viewed as an attempt to find leftmost derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Examples includes: Recursive descent parser, LL parser (Left-to-right, Leftmost derivation), and so on.

- **Bottom-up parsing:** A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

This is the process of recognizing an utterance (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language. Most languages

have the meanings of their utterances structured according to their syntax—a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down part by part and look at its analyzed form (known as its parse tree in computer science, and as its deep structure in generative grammar) as discussed earlier.

3.1.4 Syntactic Ambiguity

Syntactic ambiguity is a property of sentences which may be reasonably interpreted in more than one way, or reasonably interpreted to mean more than one thing. Ambiguity may or may not involve one word having two parts of speech or homonyms. Syntactic ambiguity arises not from the range of meanings of single words, but from the relationship between the words and clauses of a sentence, and the sentence structure implied thereby. When a reader can reasonably interpret the same sentence as having more than one possible structure, the text is equivocal and meets the definition of syntactic ambiguity.

3.1.5 Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained. When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

Arithmetic	Comparison	Logical
Exponentiation (^)	Equality (=)	Not
Negation (-)	Inequality (<>)	And
Multiplication and division (*, /)	Less than (<)	Or
Integer division (\)	Greater than (>)	XOR
Modulus arithmetic (Mod)	Less than or equal to (<=)	Eqv
Addition and subtraction (+, -)	Greater than or equal to (>=)	Imp
String concatenation (&)	Is	&

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right. The string concatenation operator (&) is not an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. The Is operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

SELF ASSESSMENT ASSIGNMENT

1. Discuss the general overview of programming process

2. Explain the two types of parser

4.0 Conclusion

Syntax analysis is a common part of language implementation, regardless of the implementation approach used. Syntax analysis is normally based on a formal syntax description of the language being implemented. A context-free grammar, which is also called BNF, is the most common approach for describing syntax. Syntax analyzers have two goals: to detect syntax errors in a given program and to produce a parse tree, or possibly only the information required to build such a tree, for a given program. Syntax analyzers are either top-down, meaning they construct leftmost derivations and a parse tree in top-down order, or bottom-up, in which case they construct the reverse of a rightmost derivation and a parse tree in bottom-up order. Parsers that work for all unambiguous grammars have complexity O(n3). However, parsers used for implementing syntax analyzers for programming languages work on subclasses of unambiguous grammars and have complexity O(n).

5.0 Summary

Syntax analysis is another phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

6.0 Tutor-Marked Assignment

1. Who are language descriptions for?
2. Describe the operation of a general language generator.
3. Describe the operation of a general language recognizer.
4. The two mathematical models of language description are generation and recognition. Describe how each can define the syntax of a programming language

7.0 References/Further Reading

1. Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning
2. Michael L. Scott, Programming Language Pragmatics, Third Edition, Morgan Kaufmann Publishers, Elsevier.
3. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.
4. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
5. Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License
6. Introduction to Linguistic Theory: Adam Szczegielniak, Syntax: The Sentence Pattern of Languages, copyright in part: Cengage learning, https://scholar.harvard.edu/files/adam/files/syntax.ppt.pdf

7. An introduction to Syntactic and Theory, Hiada Koopman, Dominique Sportiche and Edward Stabler: https://linguistic.ucla.edu/people/stabler/isat.pdf
8. Tech Differences online
9. Specifying Syntax
10. UTD: Describing Syntax and Semantics: Dr. Chris Davis, cid021000@utdallas.edu

UNIT 2 FUNDAMENTAL SEMANTIC ANALYSIS CONCEPT ON UNDERLYING MODERN PROGRAMMING LANGUAGE

CONTENTS

1.0 INTRODUCTION

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now we'll move forward to semantic analysis, where we delve even deeper to check whether they form a sensible set of instructions in the programming language. Whereas any old noun phrase followed by some verb phrase makes a syntactically correct English sentence, a semantically correct one has subject-verb agreement, proper use of gender, and the components go together to express an idea that makes sense. For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth. Semantic analysis is the front end's penultimate phase and the compiler's last chance to weed out incorrect programs. We need to ensure the program is sound enough to carry on to code generation.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Be familiar with rule-based presentations of the operational semantics and type systems for some simple be able to prove properties of an operational semantics using various forms of induction
- To understand the fundamental semantic issues of variables, nature of names and special words in programming languages
- Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
- Be familiar with some operationally-based notions of semantic equivalence of program phrases and their basic properties

3.0 MAIN CONTENT

3.1 Semantic Analysis

Semantics term in a programming language is used to figure out the relationship among the syntax and the model of computation. It emphasizes the interpretation of a program so that the programmer could understand it in an easy way or predict the outcome of program execution. An approach known as syntax-directed semantics is used to map syntactical constructs to the computational model with the help of a function.

Semantic analysis is to provide the task acknowledgment and statements of a semantically correct program. There are following styles of semantics.

3.1.1 Operational

Determining the meaning of a program in place of the calculation steps which are necessary to idealized execution. Some definitions used structural operational semantics which intermediate state is described on the basis of the language itself others use abstract machine to make use of more ad-hoc mathematical constructions. With an operational semantics of a programming

language, one usually understands a set of rules for its expressions, statements, programs, etc., are evaluated or executed. These guidelines tell how a possible implementation of a programming language should be working and it is not difficult to give skills an implementation of an interpreter of a language in any programming languages simply by monitoring and translating it operational semantics of the language destination deployment.

3.1.2 Denotational

Determining the meaning of a program as elements of a number of abstract mathematical structures e.g. with regard to functions such as programming language specific mathematical functions.

3.1.3 Axiomatic or logical

The definition of a program defining indirectly, by providing the axioms of logic to the characteristics of the program. Compare with specification and verification.

3.2 Types of Sematic Analysis

Types of semantic analysis involves the following: static and dynamic semantics.

3.2.1 Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

3.2.3 Dynamic semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

3.3 Semantic Analyzer

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

3.4 Semantic Errors

Some of the semantics errors that the semantic analyzer is expected to recognize:

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

3.5 Functions of Semantic Analysis

3.5.1 Type Checking

Ensures that data types are used in a way consistent with their definition.

3.5.2 Label Checking

A program should contain labels references.

3.5.3 Flow Control Check

Keeps a check that control structures are used in a proper manner (example: no break statement outside a loop)

3.6 Fundamental Semantic Issues of Variables, Nature of Names and Special Words in Programming Languages

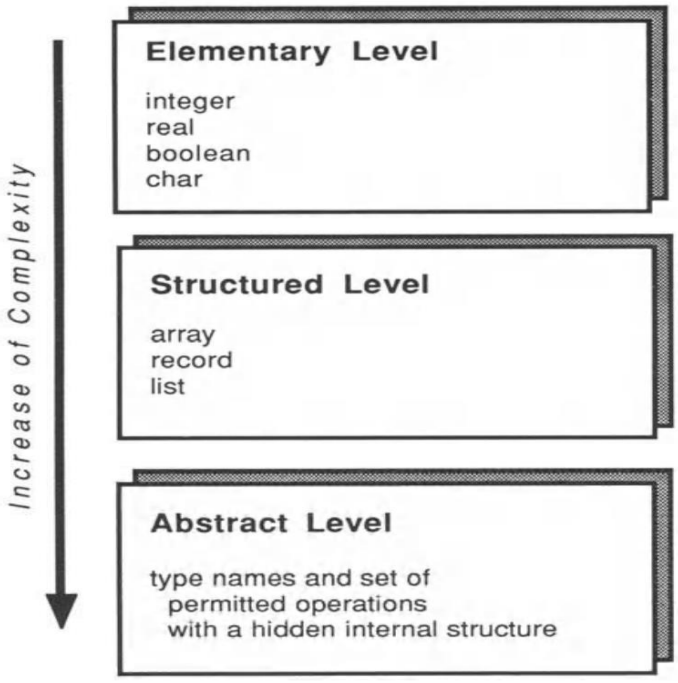
Attributes of variables, including type, address and value will be discussed.

3.6.1 Variables

Variables in programming tells how the data is represented which can be range from very simple value to complex one. The value they contain can be change depending on condition. When creating a variable, we also need to declare the data type it contains. This is because the program will use different types of data in different ways. Programming languages define data types differently. Data can hold a very simplex value like an age of the person to something very complex like a student track record of his performance of whole year. It is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. Compilers have to replace variables' symbolic names with the actual locations of the data. While the variable name, type, and location generally remain fixed, the data stored in the location may get altered during program execution.

For example, almost all languages differentiate between ‘integers’ (or whole numbers, eg 12), ‘non-integers’ (numbers with decimals, eg 0.24), and ‘characters’ (letters of the alphabet or words). In programming languages, we can distinguish between different type levels which from the user's point of view form a hierarchy of complexity, i.e. each level allows new data types or operations of greater complexity.

- **Elementary level:** Elementary (sometimes also called basic or simple) types, such as integers, reals, boo leans, and characters, are supported by nearly every programming language. Data objects of these types can be manipulated by well-known operators, like +, - , *, or /, on the programming level. It is the task of the compiler to translate the operators onto the correct machine instructions, e.g. fixed-point and floating-point operations.



Type levels in programming languages

- **Structured level:** Most high level programming languages allow the definition of structured types which are based on simple types. We distinguish between static and dynamic structures. Static structures are arrays, records, and sets, while dynamic structures are a b it more complicated, since they are recursively defined and may vary in size and shape during the execution of a program. Lists and trees are dynamic structures.

- **Abstract level:** Programmer defined abstract data types are a set of data objects with declared operations on these data objects. The implementation or internal representation of abstract data types is hidden to the users of these types to avoid uncontrolled manipulation of the data objects (i.e the concept of encapsulation).

3.6.2 Naming conventions

Unlike their mathematical counterparts, programming variables and constants commonly take multiple-character names, e.g. COST or total. Single-character names are most commonly used only for auxiliary variables; for instance, i, j, k for array index variables. Some naming conventions are enforced at the language level as part of the language syntax and involve the format of valid identifiers. In almost all languages, variable names cannot start with a digit (0-9) and cannot contain whitespace characters. Whether, which, and when punctuation marks are permitted in variable names varies from language to language; many languages only permit the underscore (_) in variable names and forbid all other punctuation. In some programming languages, specific (often punctuation) characters (known as sigils) are prefixed or appended to variable identifiers to indicate the variable's type. Case-sensitivity of variable names also varies between languages and some languages require the use of a certain case in naming certain entities; most modern languages are case-sensitive; some older languages are not. Some languages reserve certain forms of variable

names for their own internal use; in many languages, names beginning with 2 underscores ("__") often fall under this category.

3.6.3 Binding

Binding describes how a variable is created and used (or "bound") by and within the given program, and, possibly, by other programs, as well. There are two types of binding; Dynamic, and Static binding.

3.6.3.1 Dynamic Binding

Also known as Dynamic Dispatch) is the process of mapping a message to a specific sequence of code (method) at runtime. This is done to support the cases where the appropriate method cannot be determined at compile-time. It occurs first during execution, or can change during execution of the program.

3.6.3.2 Static Binding

It occurs first before run time and remains unchanged throughout program execution

3.6.4 Scope

The scope of a variable describes where in a program's text, the variable may be used, while the extent (or lifetime) describes when in a program's execution a variable has a (meaningful) value. Scope is a lexical aspect of a variable. Most languages define a specific scope for each variable (as well as any other named entity), which may differ within a given program. The scope of a variable is the portion of the program code for which the variable's name has meaning and for which the variable is said to be "visible". It is also of two type; static and dynamic scope.

3.6.4.1 Static Scope

The static scope of a variable is the most immediately enclosing block, excluding any enclosed blocks where the variable has been re-declared. The static scope of a variable in a program can be determined by simply studying the text of the program. Static scope is not affected by the order in which procedures are called during the execution of the program.

3.6.4.2 Dynamic Scope

The dynamic scope of a variable extends to all the procedures called thereafter during program execution, until the first procedure to be called that re-declares the variable.

3.6.5 Referencing

The referencing environment is the collection of variable which can be used. In a static scoped language, one can only reference the variables in the static reference environment. A function in a static scoped language does have dynamic ancestors (i.e. its callers), but cannot reference any variables declared in that ancestor.

3.2 Difference Between Syntax and Semantics

1. Syntax refers to the structure of a program written in a programming language. On the other hand, semantics describes the relationship between the sense of the program and the computational model.

2. Syntactic errors are handled at the compile time. As against, semantic errors are difficult to find and encounters at the runtime.

3. For example, in c++ a variable "s" is declared as "int s;", to initialize it we must use an integer value. Instead of using integer we have initialized it with "Seven". This declaration and initialization is syntactically correct but semantically incorrect because "Seven" does not represent integer form.

4. In relation syntactic interpretation must have some distinctive meaning, while semantic component is associated with a syntactic representation.

SELF EXERCISE ASSIGNMENT

1. List some error sematic errors that you know
2. What is the main function of semantic analyzer
3. Itemize function of sematic analysis?
4. In a tabular form state out the difference between syntax and semantics

4.0 Conclusion

A brief introduction to three methods of semantic description: operational, denotational, and axiomatic. Operational semantics is a method of describing the meaning of language constructs in terms of their effects on an ideal machine. In denotational semantics, mathematical objects are used to represent the meanings of language constructs. Language entities are converted to these mathematical objects with recursive functions. Axiomatic semantics, which is based on formal logic, was devised as a tool for proving the correctness of programs.

5.0 Summary

Static and dynamic sematic were discussed with different types of semantic error, binding and scope were discussed with their types as well and finally, fundamental semantic issues of variables, nature of names and special words in programming languages were analyzed.

6.0 Tutor-Marked Assignment

1. Some programming languages are type less. What are the obvious advantages and disadvantages of having no types in a language?
2. What are the advantages and disadvantages of dynamic scoping?
3. Define static binding and dynamic binding
4. In what ways are reserved words better than keywords?
5. Define lifetime, scope, static scope, and dynamic scope

7.0 References/Further Reading

1. Semantics in Programming Languages - INFO4MYSTREY , BestMark Mystery2020 (info4mystery.com), https://www.info4mystery.com/2019/01/semantics-in-profgramming-languages.html
2. A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.
3. J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.
4. A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.
5. J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985.
6. Semantic Analysis in Compiler Design - GeeksforGeeks, https://www.geeksforgeeks.org/semantic-analysis-in-computer-design/
7. Compiler Design - Semantic Analysis (tutorialspoint.com), https://www.tutorialspoint.com/compiler_design_semantic_analysis.html
8. Semantics of Programming Languages Computer Science Tripos, Part 1B, Peter Sewell Computer Laboratory, University of Cambridge, 2009.
9. Specifying Syntax
10. UTD: Describing Syntax and Semantics: Dr. Chris Davis, cid021000@utdallas.edu
11. Hennessy, M. (1990). The Semantics of Programming Languages. Wiley. Out of print, but available on the web at http://www.cogs.susx.ac.uk/users/matthewh/ semnotes.ps.gz12.
12. Boston: Addison-Wesley, 2007.
13. Pierce, B. C. (2002) Types and Programming Languages. MIT Press
14. Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
15. Pierce, B. C. (ed) (2005) Advanced Topics in Types and Programming Languages. MIT Press.
16. Winskel, G. (1993). The Formal Semantics of Programming Languages. MIT Press. An introduction to both operational and denotational semantics; recommended for the Part II Denotational Semantics course.
17. Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

UNIT 3 FORMAL LANGUAGE AND GRAMMARS

CONTENTS

1.0 INTRODUCTION

The formal language-generation mechanisms, usually called grammars, are commonly used to describe the syntax of programming languages. A formal grammar is a set of rules of a specific kind, for forming strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar describes only the form of the strings and not the meaning or what can be done with them in any context. A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting must start. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a recognizer.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Explain formal methods of describing syntax (backus-naur form, context-free grammars, and parser tree).
- To be able to recognize and understand the meaning of targeted grammatical structures in written and spoken form.
- Students will be able to build an elementary understanding of form, meaning and use in varied discourse settings.

3.0 MAIN CONTENT

3.1 Grammar

The grammar of a language establishes the alphabet and lexicon. Then by means of a syntax, it defines those sequences of symbols corresponding to well-formed phrases and sentences. A grammar mainly consists of a set of rules for transforming strings. To generate a string in the language, one begins with a string consisting of a single start symbol. The production rules are then applied in any order, until a string that contains neither the start symbol nor designated nonterminal symbols is produced. The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.

A grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer"—a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognizers, formal language theory uses separate formalisms, known as automata theory. One of the interesting results of automata theory is that it is not possible to design a recognizer for certain formal languages

3.1.1 The Syntax of Grammars

In the classic formalization of generative grammars, a grammar *G* consists of the following components:

- a finite set *N* of *non-terminal symbols*, none of which appear in strings formed from *G*.
- a finite set of *terminal symbols* that is disjoint from *N*.
- a finite set *P* of *production rules*, each rule of the form
- where * is the Kleene star operator and denotes set union.

That is, each production rule maps from one string of symbols to another, where the first string (the "head") contains at least one non-terminal symbol. In the case that the second string (the "body") consists solely of the empty string – i.e. it contains no symbols at all, it may be denoted with a special notation (often _, *e* or _) in order to avoid confusion.

A distinguished symbol, that is, the *start symbol*. A grammar is formally defined as the tuple (*N*, _, *P*, *S*). Such a formal grammar is often called a rewriting system or a phrase structure grammar in the literature A grammar can be formally written as four tuple element i.e. G = {N, ∑, S, P}

where N is the set of variable or non-terminal symbols
∑ is a set of terminal symbol known as the alphabet
G = (N, ∑, S, P)
N → {S, A} where S is the start symbol
∑ = {λ, a}

Example

Assuming the alphabet consists of *a* and *b*, the start symbol is *S* and we have the following production rules:

1. S→aSb
2. S→ ba

then we start with *S*, and can choose a rule to apply to it. If we choose rule 1, we obtain the string *aSb*. If we choose rule 1 again, we replace *S* with *aSb* and obtain the string *aaSbb*. If we now choose rule 2, we replace *S* with *ba* and obtain the string *aababb*, and are done. We can write this series of choices more briefly, using symbols: S→aSb →aaSbb →aababb. The language of the grammar is then the infinite set $\{a^n bab^n \mid n \geq 0\}$ = {ba, abab, aababb, aaababbb, ------}, where a^k is *a* repeated *k* times (and *n* in particular represents the number of times production rule 1 has been applied).

Example 1

Consider the grammar G where N = {S, B} = {a, b, c}, S is the start symbol and P consists of the following production rules:

1. S→ aBSc
2. S→ abc
3. Ba → aB
4. Bb → bb

This grammar defines the language L(G) = $a^n b^n c^n \mid n \geq 1\}$ where a^n denotes a string of n consecutive a's. Thus the language is the set of strings that consist of one or more a's, followed by the same number of b's and then by the same number of c's.

Solution

S $\rightarrow_2$abc
S $\rightarrow$ $_1$aBSc $\rightarrow_2$aBabcc $\rightarrow_3$aaBbcc $\rightarrow_4$aabbcc
S $\rightarrow_1$aBSc $\rightarrow_1$aBaBScc $\rightarrow_2$ aBaBabccc $\rightarrow_3$aaBBabaccc $\rightarrow_3$aaBaBbccc $\rightarrow_3$aaaBBbccc $\rightarrow_4$aaaBbbccc $\rightarrow_4$aaabbbccc

Example 2

Consider G = ({S, A, B}, {a, b}, S, P) N = {S, A, B} produce 2 sample strings from the grammar and a^2b^4 is contained in the grammar?

1. P: S → AB
2. A → Aa
3. B → Bb
4. A → a
5. B → b

Solution

S $\rightarrow$ $_1$AB
S $\rightarrow$ $_1$AB $\rightarrow$ $_2$AaB $\rightarrow$ $_3$AaBb $\rightarrow$ $_3$AaBbb $\rightarrow$ $_3$AaBbbb $\rightarrow$ $_4$aaBbbb $\rightarrow$ $_5$aabbbb

Example 3

Analyze the structure of this grammar G = {V, T, S, P} where V is used in place of N and T is used in place of ∑ and produce 4 sample strings from the grammar?
G = (V, T, S, P), V = (A, S, B), T = (λ, a, b)

1. P: S → ASB
2. A → a
3. B → b
4. S → λ

Solution

S $\rightarrow$ $_4\lambda$
S $\rightarrow$ $_1$ASB $\rightarrow$ $_2$aSB $\rightarrow$ $_3$aSb $\rightarrow$ $_4$ab
S $\rightarrow$ $_1$ASB $\rightarrow$ $_4$AB

$S \rightarrow {}_1ASB \rightarrow {}_2aSB \rightarrow {}_4aB$

SELF ASSESSMENT EXERCISE

1. Exercise: Obtain a gramma that generates a language L (G) = $\{a^n b^{n+1}: n \geq 0\}$
2. Using the parse tree figure, show a parse tree and a leftmost derivation for each of the following statements:
   a. A = A * (B + (C * A))
   b. B = C * (A * C + B)
   c. A = A * (B + (C)

3.1.2 Noam Chomsky and John Backus

In the middle to late 1950s, two men, Noam Chomsky and John Backus, developed the same syntax description formalism, which subsequently became the most widely used method for programming language syntax. Four classes of generative devices or grammars that define four classes of languages were described. Two of these grammar classes, named context-free and regular, turned out to be useful for describing the syntax of programming languages. The forms of the tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars. It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammar and context-free grammars simply refer to as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. Two important types are *context-free grammars* (Type 2) and *regular grammars* (Type 3). The languages that can be described with such a grammar are called *context-free languages* and *regular languages*, respectively. Although much less powerful than unrestricted grammars (Type 0), which can in fact express any language that can be accepted by a Turing machine, these two restricted types of grammars are most often used because parsers for them can be efficiently implemented.

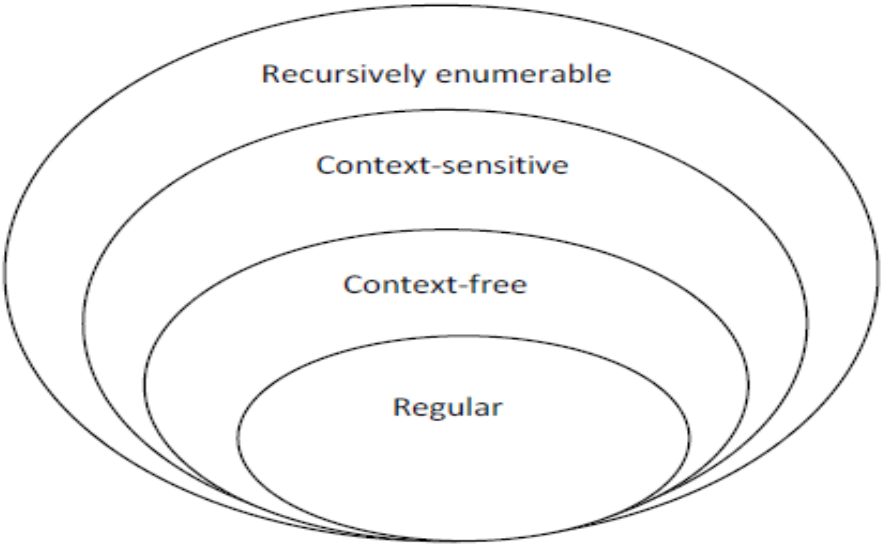
3.1.3 Types of Grammars and Automata

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.

- Type-0 grammars (unrestricted grammars) include all formal grammars.
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages.
- Type-2 grammars (context-free grammars) generate the context free languages.
- Type-3 grammars (regular grammars) generate the regular languages.

Grammar Type	Grammar Accepted	Language Accepted	Automation
Type 0	Unrestricted grammar	Recursively enumerable language needs counters, registers, selection and	Turing machine $(a^n c^m b^n)^k$

		memory to recognized the grammar	
Type 1	Context sensitive grammar	Context sensitive language needs counter, register and selection to recognized the grammar	Linearly bounded $a^nc^mb^n$
Type 2	Context free grammar	Context free language needs a counter and register to recognized the grammar	Push down automata $(ab)^n = a^nb^n$
Type 3	Regular expression	Regular expression, regular language only need counters to recognize grammar	Finite state automaton $(ab)^2 = ab$



4.0 Conclusion

Backus-Naur Form and context-free grammars are equivalent metalanguages that are well suited for the task of describing the syntax of programming languages. Not only are they concise descriptive tools, but also the parse trees that can be associated with their generative actions give graphical evidence of the underlying syntactic structures. Furthermore, they are naturally related to recognition devices for the languages they generate, which leads to the relatively easy construction of syntax analyzers for compilers for these languages. An attribute grammar is a descriptive formalism that can describe both the syntax and static semantics of a language. Attribute grammars are extensions to context-free grammars. An attribute grammar consists of a grammar, a set of attributes, a set of attribute computation functions, and a set of predicates, which together describe static semantics rules.

5.0 Summary

A formal grammar is a set of rules of a specific kind, for forming strings in a formal language. It has four components that form its syntax and a set of operations that can be performed on it, which form its semantic. An attribute grammar is a descriptive formalism that can describe both the

syntax and static semantics of a language. Attribute grammars are extensions to context-free grammars. An attribute grammar consists of a grammar, a set of attributes, a set of attribute computation functions, and a set of predicates, which together describe static semantics rule

6.0 Tutor-Marked Assignment

1. What is the primary use of attribute grammars?
2 Which of the following sentences are in the language generated by this grammar Baab, bbbab, bbaaaaa, bbaab
3. Define a grammar that generates the language {anbm | n,m ≥ 1} using only productions of the form N → tM or N → t, where N and M are any non-terminal symbol and t is any terminal symbol. Try to give an intuitive explanation of why there exists no grammar with these characteristics which generates the language {anbn | n ≥ 1}.
4. List the four classes of grammar?

7.0 References/Further Reading

1. Introduction to Linguistic Theory: Adam Szczegielniak, Syntax: The Sentence Pattern of Languages, copyright in part:Cengage learning, https://scholar.harvard.edu/files/adam/files/syntax.ppt.pdf
2. An introduction to Synstatic and Theory, Hiada Koopman, Dominique Sportiche and Edward Stabler: https://linguistic.ucla.edu/people/stabler/isat.pdf
3. Tech Differences online
4. Specifying Syntax
5. UTD: Describing Syntax and Semantics: Dr. Chris Davis, cid021000@utdallas.edu
6. Learning Programming Methodologies: Absolute Beginners, Tutorials Point, Simply Easy Learning
7. Michael L. Scott, Programming Language Pragmatics, Third Edition, Morgan Kaufmann Publishers, Elsevier.
8. Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2011.
9. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms. Boston: Addison-Wesley, 2007.
10. Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License

UNIT 4 THE BASIC ELEMENTS OF PROGRAMMING LANGUAGE

CONTENTS

1.0 INTRODUCRTION

syntax in computer programming as the concept of giving specific word sets in specific orders to computers so that they do what we want them to do. Every programming language uses different word sets in different orders, which means that each programming language uses its own syntax. But, no matter the programming language, computers are really exacting in how we structure our syntax. Programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about computational processes. Programs serve to communicate those ideas among the members of a programming community. Thus, programs must be written for people to read, and only incidentally for machines to execute. When we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three such mechanisms:

- **primitive expressions and statements**, which represent the simplest building blocks that the language provides,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of elements: functions and data. (Soon we will discover that they are really not so distinct.) Informally, data is stuff that we want to manipulate, and functions describe the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive functions, as well as have some methods

for combining and abstracting both functions and data. To develop any instruction there are some elements needed or we can essentially present in all language.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- To encourage the students to understand control flow, arithmetic expressions and design issues in arithmetic expression.to determine whether the text string on input is a sentence in the given (natural) language.
- To introduce students to control flow and execution sequence in different programming languages.
- To introduce students to subprograms in different programming languages as the fundamental building blocks of programs
- To explore the design concepts including parameter-passing methods, local referencing environment, overload subprograms, generic subprograms and the aliasing and side effect problems.
- To explore programming language constructs that support data abstraction and discuss the general concept of abstraction in programming and programming languages.

3.0 MAIN CONTENT

3.1 Primitive Data type

This is a classification identifying one of various types of data, such as floating point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. There are several classifications of data types, some of which include: It is a basic data type which is provided by a programming language as a basic building block. Most languages allow more complicated composite types to be recursively constructed starting from basic types. It also a built-in data type for which the programming language provides built-in support.

3.1.1 Common Data Types

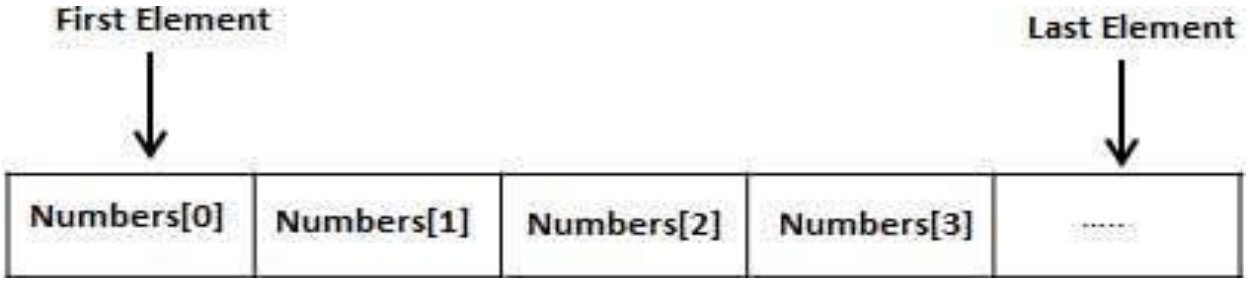
- **Integer**: a whole number that can have a positive, negative, or zero value. It cannot be a fraction, nor can it include decimal places. It is commonly used in programming, especially for increasing values. Addition, subtraction, and multiplication of two integers results in an integer. However, division of two integers may result in either an integer or a decimal. The resulting decimal can be rounded off or truncated in order to produce an integer.

- **Character**: any number, letter, space, or symbol that can be entered in a computer. Every character occupies one byte of space.

- **String**: used to represent text. It is composed of a set of characters that can include spaces and numbers. Strings are enclosed in quotation marks to identify the data as strings, and not as variable names, nor as numbers.

- **Floating Point Number**: a number that contains decimals. Numbers that contain fractions are also considered floating-point numbers.
- **Varchar**: as the name implies, a varchar is a variable character, on account of the fact that the memory storage has variable length. Each character occupies one byte of space, plus 2 bytes additional for length information.

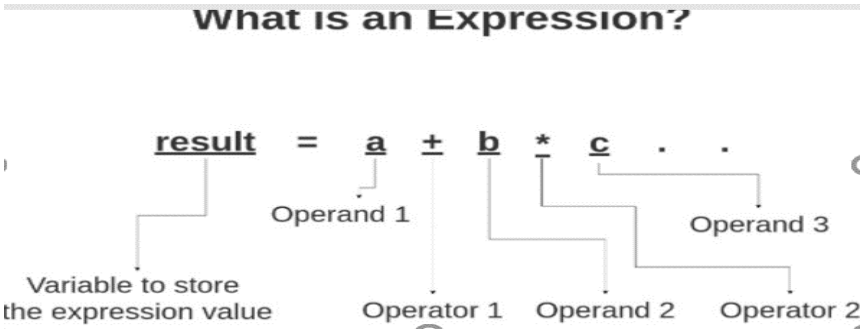
- **Array**: a kind of a list that contains a group of elements which can be of the same data type as an integer or string. It is used to organize data for easier sorting and searching of related sets of values. An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset. Each variable, called an element, is accessed using a subscript (or index). All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



3.2 Expressions

Expressions are the fundamental means of specifying computations in a programming language. An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

Programming languages generally support a set of operators that are similar to operations in mathematics. A language may contain a fixed number of built-in operators (e.g. + - * = in C and C++), or it may allow the creation of programmer defined operators (e.g. Haskell). Some programming languages restrict operator symbols to special characters like + or := while others allow also names like div (e.g. Pascal).



3.2.1 Overloaded operators

In some programming languages an operator may be ad-hoc polymorphic, that is, have definitions for more than one kind of data, (such as in Java where the + operator is used both for the addition of numbers and for the concatenation of strings). Such an operator is said to be overloaded. In languages that support operator overloading by the programmer but have a limited set of operators, operator overloading is often used to define customized uses for operators.

3.2.2 Short-Circuit Evaluation

Short-circuit evaluation, minimal evaluation, or McCarthy evaluation denotes the semantics of some Boolean operators in some programming languages in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true. In some programming languages (Lisp), the usual Boolean operators are short - circuit. In others (Java, Ada), both short-circuit and standard Boolean operators are available.

3.2.3 Categories of Expressions

- **Boolean Expression** - an expression that returns a Boolean value, either true or false. This kind of expression can be used only in the IF-THEN-ELSE control structure and the parameter of the display condition command. A relational expression has two operands and one relational operator. The value of a relational expression is Boolean. In programming languages that include a distinct Boolean data type in their type system, like Java, these operators return true or false, depending on whether the conditional relationship between the two operands holds or not.

- **Numeric Expression -** an expression that returns a number. This kind of expression can be used in numeric data fields. It can also be used in functions and commands that require numeric parameters.

- **Character Expression** - an expression that returns an alphanumeric string. This kind of expression can be used in string data fields (format type Alpha). They can also be used in functions and command that require string parameters.

- **Relational Expression:** Relational operator is a programming language construct or operator that test or define some kind of relation between two entities. These include numerical equality (e.g., $5 = 5$) and inequalities (e.g., $4 \geq 3$). An expression created using a relational operator forms what is known as a relational expression or a condition. Relational operators are also used in technical literature instead of words. Relational operators are usually written in infix notation, if supported by the programming language, which means that they appear between their operands (the two expressions being related).

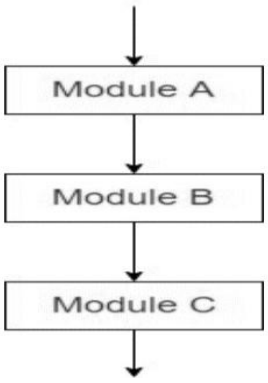
3.3 Control structures

Control structures allow the programmer to define the order of execution of statements. The availability of mechanisms that allow such a control makes programming languages powerful in their usage for the solution of complex problems. Usually problems cannot be solved just by sequencing some expressions or statements, rather they require in certain situations decisions - depending on some conditions - about which of some alternatives has to be executed, and/or to repeat or iterate parts of a program an arbitrary time - probably also depending on some conditions. Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

3.3.1 Sequential Logic (Sequential Flow)

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



Sequential Flow Control

3.3.2 Selection Logic (Conditional Flow)

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as Conditional Structures. These structures can be of three types:

Single Alternative: This structure has the form:

```
If (condition) then:
      Module A]
[End of If structure]
```

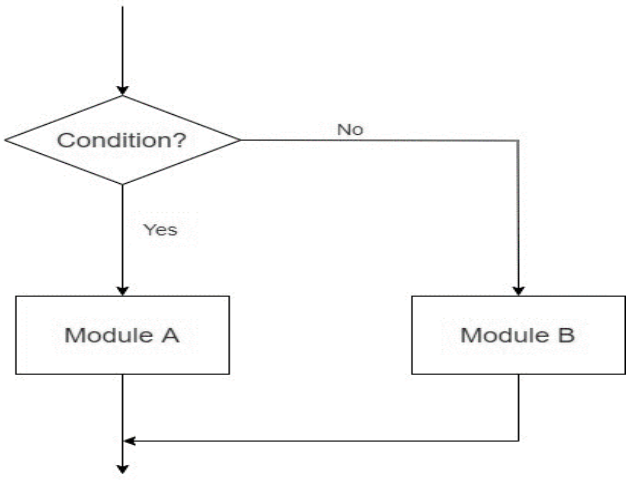
Double Alternative: This structure has the form:

```
If (Condition), then:
      [Module A]
Else:
      [Module B]
[End if structure]
```

Multiple Alternatives: This structure has the form:

```
If (condition A), then:
      [Module A]
Else if (condition B), then:
      [Module B]
      ..
      ..
Else if (condition N), then:
      [Module N]
[End If structure]
```

In this way, the flow of the program depends on the set of conditions that are written. This can be more understood by the following flow charts:



Double Alternative Control Flow

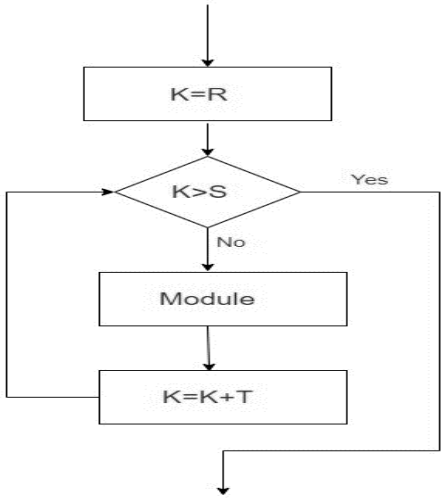
3.3.3 Iteration Logic (Repetitive Flow)

The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop. The two types of these structures are:

Repeat-For Structure**:** This structure has the form:

```
Repeat for i = A to N by I:
      [Module]
[End of loop]
```

Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.

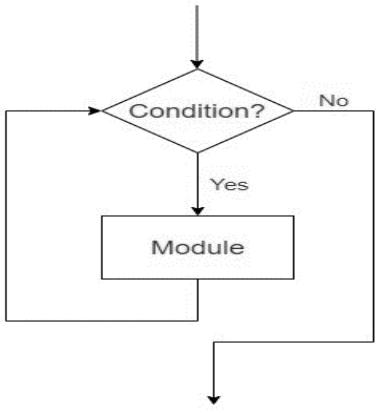


Repeat-For Flow

Repeat-While Structure: It also uses a condition to control the loop. This structure has the form:

```
Repeat while condition:
      [Module]
[End of Loop]
```

In this, there requires a statement that initializes the condition controlling the loop, and there must also be a statement inside the module that will change this condition leading to the end of the loop.



Repeat While Flow

3.4 Subroutine and Functions

The element of the programming allows a programmer to use snippet of code into one location which can be used over and over again. The primary purpose of the functions is to take arguments in numbers of values and do some calculation on them after that return a single result. Functions are required where you need to do complicated calculations and the result of that may or may not be used subsequently used in an expression. If we talk about subroutines that return several results. Where calls to subroutines cannot be placed in an expression whether it is in the main program where subroutine is activated by using CALL statement which include the list of inputs and outputs that enclosed in the open and closed parenthesis and they are called the arguments of the subroutines. There are some of the rules follow by both to define name like less than six letters and start with the letters. The name should be different that used for variables and functions.

Subroutine (also called procedure, function, routine, method, or subprogram) is a portion of code within a larger program that performs a specific task and is relatively independent of the remaining code. A subroutine is often coded so that it can be started ("called") several times and/or from several places during a single execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the "call" once the subroutine's task is done. A subroutine may be written so that it expects to obtain one or more data values from the calling program (its parameters or arguments). It may also return a computed value to its caller (its return value), or provide various result values or out(put) parameters. Indeed, a common use of subroutines is to implement mathematical functions, in which the purpose of the subroutine is purely to compute one or more results whose values are entirely determined by the parameters passed to the subroutine. (Examples might include computing the logarithm of a number or the determinant of a matrix.)

3.4.1 Advantages subroutines

- Decomposition of a complex programming task into simpler steps: this is one of the two main tools of structured programming, along with data structures.
- Reducing the duplication of code within a program,
- Enabling the reuse of code across multiple programs,
- Hiding implementation details from users of the subroutine.

3.4.2 Disadvantages

- The invocation of a subroutine (rather than using in-line code) imposes some computational overhead in the call mechanism itself.
- The subroutine typically requires standard housekeeping code—both at entry to, and exit from, the function (function prologue and epilogue—usually saving general purpose registers and return address as a minimum).

3.5 Parameter Passing Methods

There are some issues to consider when using local variables instead of global variables.

- How can the value of a variable in one sub-program be accessible to another sub-program?

- What happens if the value of a variable needed in more than one subprogram is to change within one subprogram but not in the main program?

The solution to these issues is to use parameter passing. Parameter passing allows the values of local variables within a main program to be accessed, updated and used within multiple sub-programs without the need to create or use global variables.

Parameter is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine. These pieces of data are called arguments. An ordered list of parameters is usually included in the definition of a subroutine, so that, each time the subroutine is called, its arguments for that call can be assigned to the corresponding parameters.

Parameters identify values that are passed into a function. For example, a function to add three numbers might have three parameters. A function has a name, and it can be called from other points of a program. When that happens, the information passed is called an argument.

3.5.1 Function Parameters

Each function parameter has a type followed by an identifier, and each parameter is separated from the next parameter by a comma. The parameters pass arguments to the function. When a program calls a function, all the parameters are variables. The value of each of the resulting arguments is copied into its matching parameter in a process call *pass by value*. The program uses parameters and returned values to create functions that take data as input, make a calculation with it and return the value to the caller.

3.5.2 The Difference Between Functions and Arguments

The terms parameter and argument are sometimes used interchangeably. However, parameter refers to the type and identifier, and arguments are the values passed to the function. In the following C++ example, *int a* and *int b* are parameters, while *5* and *3* are the arguments passed to the function.

```
int addition (int a, int b)
{
     int r;
     r=a+b;
      return r;
}
int main ()
{
     int z;
     z = addition (5,3);
     cout << "The result is " << z;
}
```

3.5.3 Value of Using Parameters

- Parameters allow a function to perform tasks without knowing the specific input values ahead of time.
- Parameters are indispensable components of functions; which programmers use to divide their code into logical blocks.

3.5.4 Parameter Passing Techniques

There are a number of different ways a programming language can pass parameters:

1. Pass-by-value
2. Pass-by-reference
3. Pass-by-value-result
4. Pass-by-name

The most common are pass-by-value and pass-by-reference.

3.5.4.1 Pass-by-value

This method uses *in-mode* semantics. Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. Example of C++ code:

```
#include <iostream>
#include <cstdlib>

void AssignTo(int param) {
        param = 5;
}

int main(int argc, char* argv[]) {
        int x = 3;
        AssignTo(x);
        std::cout << x << std::endl;
        return 0;
}
```

In a pass-by-value system, the statement `AssignTo(x)` creates a copy of the argument `x`. This copy has the same value as the original argument (hence the name "pass-by-value"), but it does not have the same identity. Thus the assignment within `AssignTo` modifies a variable that is distinct from the supplied parameter. Thus, the output in this passing style is `3` and not `5`.

3.5.4.2 Pass-by-reference (aliasing)

This technique uses *in/out-mode* semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. Example:

```
#include <iostream>
#include <cstdlib>

void AssignTo(int& param) {  // use of ampersand (&)
implies reference
        param = 5;
}

int main(int argc, char* argv[]) {
        int x = 3;
        AssignTo(x);
        std::cout << x << std::endl;
        return 0;
}
```

In a pass-by-reference system, the function invocation does not create a separate copy of the argument; rather, a reference to the argument (hence the name "pass-by-reference") is supplied into the `AssignTo` function. Thus, the assignment in `AssignTo` modifies not just the variable named `param`, but also the variable `x` in the caller, causing the output to be `5` and not `3`.

3.6 Concept of Abstraction

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects. In programming we do apply the same meaning of abstraction by making classes those are not associated with any specific instance. The abstraction is done when we need to only inherit from a certain class, but not need to instantiate objects of that class. In such case the base class can be regarded as "Incomplete". Such classes are known as "Abstract Base Class".

An Abstract Data Type is a user-defined data type that satisfies the following two conditions: – The representation of, and operations on, objects of the type are defined in a single syntactic unit – The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition

Abstract classes and Abstract methods:

1. An abstract class is a class that is declared with an abstract keyword.
2. An abstract method is a method that is declared without implementation.

3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

Advantages of Abstraction

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only important details are provided to the user.

3.7 Input/output

The element of computer programming allows interaction of the program with the external entities. Example of input/output element are printing something out to the terminal screen, capturing some text that user input on the keyboard and can be include reading and writing files. Let's take a language example to understand the concept of input and output. C++ use streams to perform input and output operation in sequential media in terms of screen, the keyboard or a file. We can define stream as an entity that can insert or extract characters and there is no need to know details about the media associated to the stream or any of its internal specification. We need to know about streams is that they are source or destination of characters and the characters are accepted sequentially.

3.8 Features of Programming Languages

A programming language consists of a vocabulary containing a set of grammatical rules intended to convey instructions to a computer or computing device to perform specific tasks. Each programming language has a unique set of keywords along with a special syntax to organize the software's instructions. A programming language's feature include orthogonality or simplicity, available control structures, data types and data structures, syntax design, support for abstraction, expressiveness, type equivalence and strong versus weak type checking, exception handling and restricted aliasing. The performance of a program, including reliability, readability, writability, reusability and efficiency, is largely determine by the way the programmer writes the algorithm and selects the data structures, as well as other implementation details.

However, the features of the programming language are vital in supporting and enforcing programmers in using proper language mechanisms in implementing the algorithms and data structures. The table below indicates that simplicity, control structures, data types and data structures have significant impact on all aspects of performance. Syntax design and the support for abstraction are important for readability, reusability, writability and reliability. However, they do not have a significant impact on the efficiency of the program. Expressiveness supports writability,

but it may have a negative impact on the reliability of the program. Strong type checking and restricted aliasing reduce the expressiveness of writing programs, but are generally considered to produce more reliable programs. Exception handling prevents the program from crashing due to unexpected circumstances and semantic errors in the program. The table below showing the performance evaluation of features of programming language.

Performance / Language features	Efficiency	Readability / Reusability	Writeability	Reliability
Simplicity/Orthogonality	✓	✓	✓	✓
Control structures	✓	✓	✓	✓
Typing and data structures	✓	✓	✓	✓
Syntax design		✓	✓	✓
Support for abstraction		✓	✓	✓
Expressiveness			✓	✓
Strong checking				✓
Restricted aliasing				✓
Exception handling				✓

SELF ASSESSMENT ASSIGNMENT

1. Write a program in the language of your choice that behaves differently if the language used name equivalence than if it used structural equivalence

4.0 Conclusion

The data types of a language are a large part of what determines that language‘s style and usefulness. The primitive data types of most imperative languages include numeric, character, string, floating points, arrays and varchar. Case sensitivity and the relationship of names to special words represent design issues of names. Variables are characterized by the sextuples: name, address, value, type, lifetime, scope. Binding is the association of attributes with program entities

Expression are of 5 categories: Boolean, character, numerical and relational expression and so also control structure are of 3 types sequence, selection and iteration. A subprogram definition describes the actions represented by the subprogram. Subprograms can be either functions or procedures. Local variables in subprograms can be stack-dynamic or static. Techniques of parameter passing: are pass-by-value and pass-by-reference. Some languages allow operator overloading. A co-routine is a special subprogram with multiple entries

5.0 Summary

Various features of language design can have a major impact on the complexity of syntax analysis. In many cases, features that make it difficult for a compiler to scan or parse also make it difficult for a human being to write correct, maintainable code.

6.0 Tutor-Marked Assignment

1. What is the definition of control structure?
2. Rewrite the following pseudocode segment using a loop structure in the specified languages:
   K = (j + 13)/27
   Loop:
   If k > 10 then goto out
   K = k + 1
   I = 3 * k-1
   Goto loop
   Out: ...
   a. Fortran 95
   b. Ada
   c. C
   d. C++,
   e. Java
   f. C#
   g. Python
3. Define operator precedence and operator associativity
4. What is overloaded operator?
5. Describe a situation in which the operator in a programming language would not be
   i] Associative
   ii] Commutative

7.0 References/Further Reading

1. Programming Languages: Types and Features - Chakray, https://www.chakay.com/programmin-languages-types-and-features.
2. A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, 1988.
3. A. W. Appel. Modern Compiler Implementation in Java. Cambridge University Press, Cambridge, 1998. 3. J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, 3/E. Addison Wesley, Reading, 2005. Available online at http://java.sun.com/docs/books/jls/index.html. References 55
4. J. E. Hopcroft, R. Motwani, and J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, 2001.
5. C. Laneve. La Descrizione Operazionale dei Linguaggi di Programmazione. Franco Angeli, Milano, 1998 (in Italian).
6. C. W. Morris. Foundations of the theory of signs. In Writings on the Theory of Signs, pages 17–74. Mouton, The Hague, 1938.

7. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
8. G. Winskel. The Formal Semantics of Programming Languages. MIT Press, Cambridge, 1993.
9. Programming Data Types & Structures | Computer Science (teachcomputerscience.com), https://teachcomputerscience.com/programming-data-types/
10 Beginner's Guide to Passing Parameters | Codementor, Michael Safyan, https://www.codementor.io/@michaelsafyan/computer-different-ways-to-pass-parameters-du107xfer.
11. Parameter Passing | Computing (glowscotland.org.uk), Stratton, https://blogs.glowscotland.org.uk
12. Parameter Passing Techniques in C/C++ - GeeksforGeeks, https://www.geeksforgeeks.org/
13. Parameter passing - Implementation (computational constructs) - Higher Computing Science Revision - BBC Bitesize, https://www.bbc.co.uk/
14, Definition of Parameters in Computer Programming (thoughtco.com), David Bolton, 2017.

MODULE 4 EVOLUTION OF PROGRAMMING LANGUAGE LEVELS

UNIT 1 BRIEF HISTORY ON LEVEL OF PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

Till now, thousands of programming languages have come into form. All of them have their own specific purposes. All of these languages have a variation in terms of the level of abstraction that they all provide from the hardware. A few of these languages provide less or no abstraction at all, while the others provide a very high abstraction.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Demonstrate understanding of the evolution of programming languages and relate how this history has led to the paradigms available today.
- Identify at least one outstanding and distinguishing characteristic for each of the programming paradigms covered in this unit

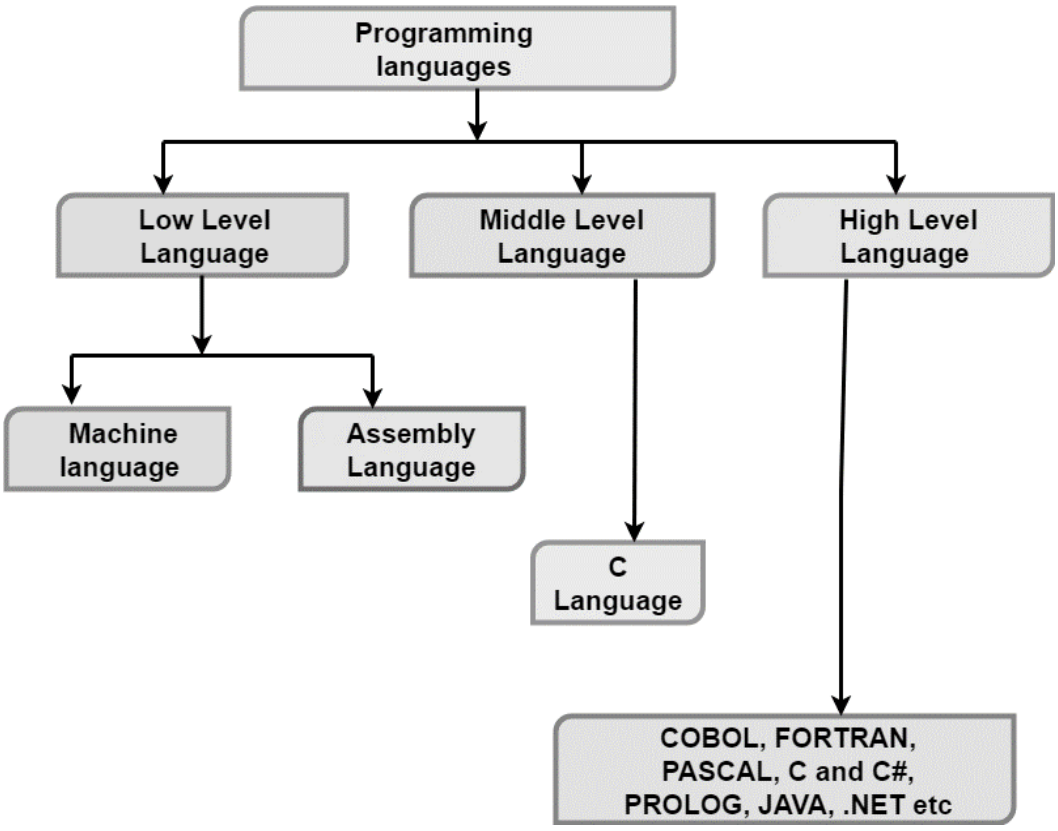
3.0 MAIN CONTENT

3.1 CLASSIFICATION ON LEVEL OF PROGRAMMING LANGUAGES

On the basis of the programming language level of abstraction, there are two types of programming languages: Low-level language and High-level language.

The first two generations are called low-level languages. The next three generations are called middle level language and high-level languages. Both of these are types of programming languages that provide a set of instructions to a system for performing certain tasks. Though both of these have specific purposes, they vary in various ways. The primary difference between low and high-level languages is that any programmer can understand, compile, and interpret a high-

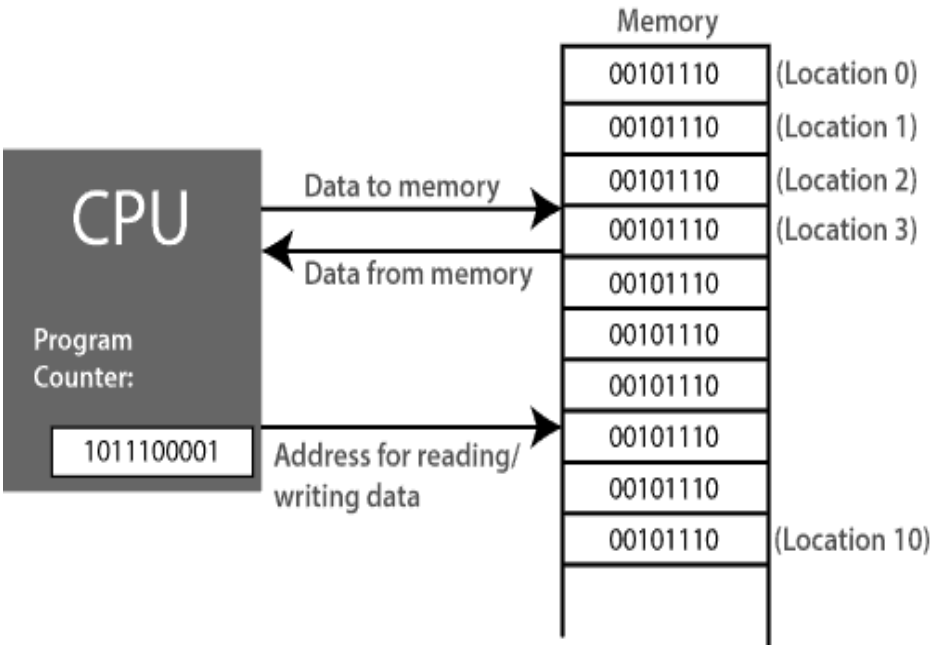
level language feasibly as compared to the machine. The machines, on the other hand, are capable of understanding the low-level language more feasibly compared to human beings.



3.1.1 Low-Level Language

The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions. The languages that come under this category are the Machine level language and Assembly language. A low-level programming language provides little or no abstraction from a computer's instruction set architecture—commands or functions in the language map that are structurally similar to processor's instructions. Generally, this refers to either machine code or assembly language. Low-level languages are considered to be closer to computers. In other words, their prime function is to operate, manage and manipulate the computing hardware and components. Programs and applications written in a low-level language are directly executable on the computing hardware without any interpretation or translation.

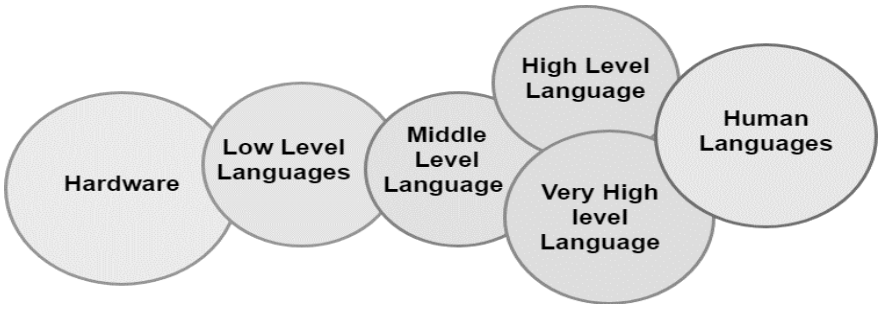
It is hardware dependent language. Each processor has kept its own instruction set, and these instructions are the patterns of bits. There is the class of processors using the same structure, which is specified as an instruction set — any instruction can be divided into two parts: the operator or opcode and operand. The starting bits are known as the operator or opcode whose role is to identify the kind of operation that are required to be performed. The remaining bits are called operand, whose purpose is to show the location of activity. The programs are written in various programming languages like C, C++. Java python etc. The computer is not capable of understanding these programming languages therefore, programs are compiled through compiler that converts into machine language.



3.1.2 Middle-Level Language in Computer

The middle-level language lies in between the low level and high-level language. C language is the middle-level language. By using the C language, the user is capable of doing the system programming for writing operating system as well as application programming. The Java and C++ are also middle-level languages. The middle-level programming language interacts with the abstraction layer of a computer system. It serves as the bridge between the raw hardware and programming layer of the computer system. The middle-level language is also known as the intermediate programming language and pseudo-language.

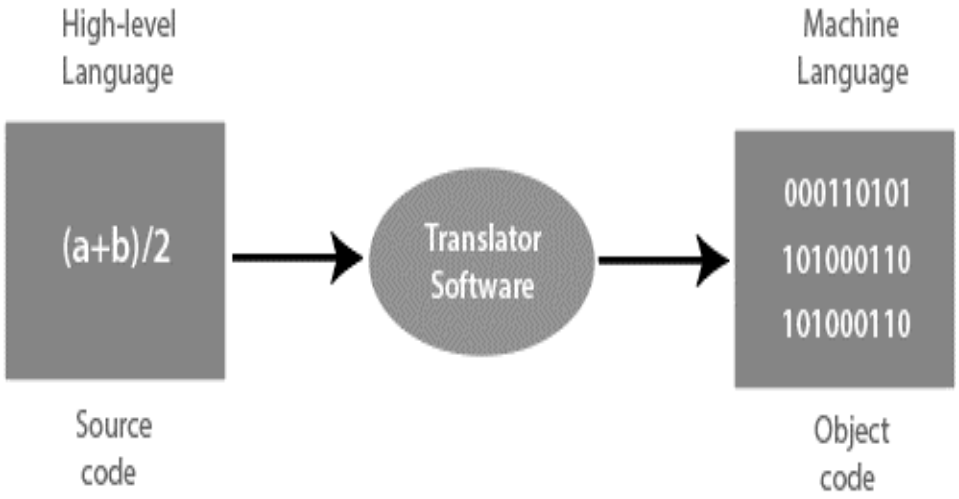
The middle-level language is an output of any programming language, which is known as source code. The source code is written in a high-level language. This kind of middle-level language is designed to improve the translated code before the processor executes it.
The improvement schedule helps to adjust the source code according to the computational framework of the target machine of various languages. The processor of CPU does not directly execute the source code of the middle-level language. It needs interpretation into binary code for the execution. The improvement schedule helps to adjust the source code according to the computational framework of the target machine of various languages. The processor of CPU does not directly execute the source code of the middle-level language. It needs interpretation into binary code for the execution.



Overlapping of Middle Level Language

3.1.3 High-Level Language

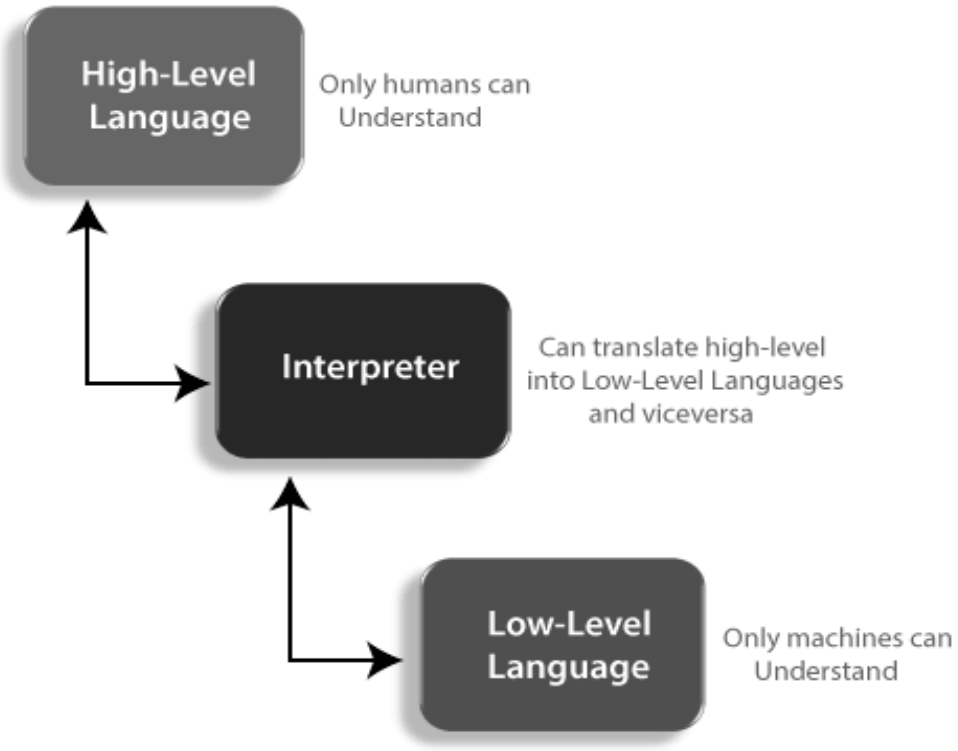
The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine level language. When writing a program in a high-level language, then the whole attention needs to be paid to the logic of the problem. A compiler is required to translate a high-level language into a low-level language.



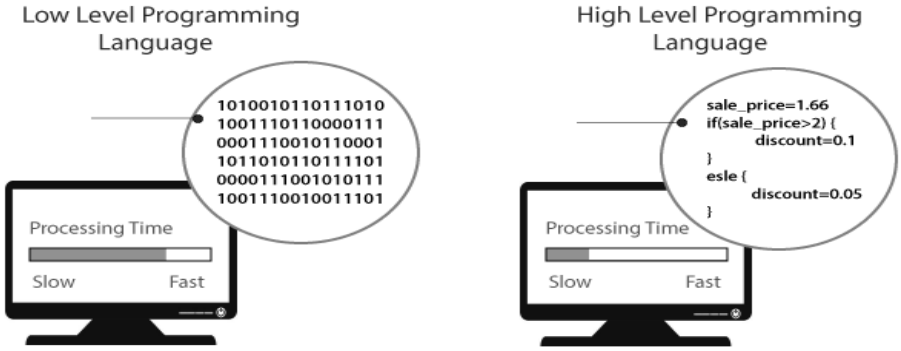
The program of high-level language must be interpreted before the execution. The high-level language deal with the variables, arrays, objects, complex arithmetic or Boolean expression, subroutines and functions, loops, threads, locks, etc. The high-level languages are closer to human languages and far from machine languages. It is similar to human language, and the machine is not able to understand this language.

```
#include<stdio.h>
int main()
{
Printd("hello");
getch()
return 0;
}
```

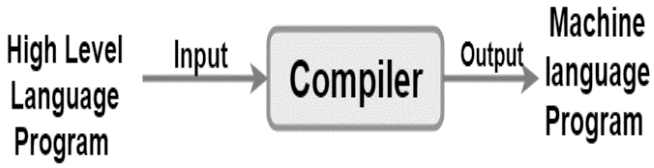
This is the example of C language, which is a middle-level language because it has the feature of both the low and high-level language. The human can understand this example easily, but the machine is not able to understand it without the translator. Every high-level language uses a different type of syntax. Some languages are designed for writing desktop software programs, and other languages are used for web development.



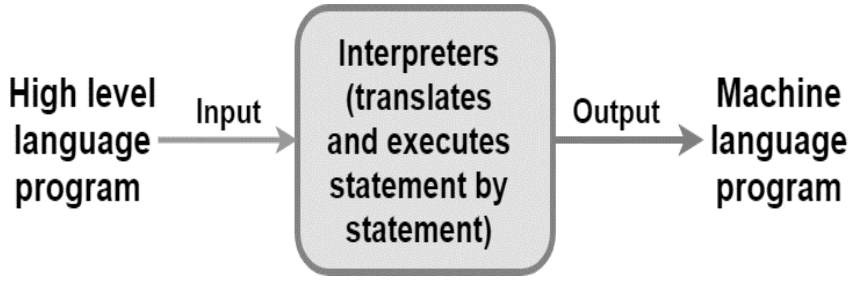
These all languages are considered as the high-level language because they must be processed with the help of a compiler or interpreter before the code execution. The source code is written in scripting languages like Perl and PHP can be run by the interpreter. These languages can convert the high-level code into binary code so that the machine can understand.



The compiler is the translator program software. This software can translate into its equivalent machine language program. The compiler compiles a set of machine language instructions for every program in a high-level language.



The linker is used for the large programs in which we can create some modules for the different task. When we call the module, the whole job is to link to that module and program is processed. We can use a linker for the huge software, storing all the lines of program code in a single source file. The interpreter is the high-level language translator. It takes one statement of the high-level language program and translates it into machine level language instruction. Interpreter immediately executes the resulting machine language instruction. The compiler translates the entire source program into an object program, but the interpreter translates line by line.



The high-level language is easy to read, write, and maintain as it is written in English like words. The languages are designed to overcome the limitation of low-level language, i.e., portability. The high-level language is portable; i.e., these languages are machine-independent.

3.2 Difference Between Low Level and High Level Languages

Parameter	High-Level Language	Low-Level Language
Basic	These are programmer-friendly languages that are manageable, easy to understand, debug, and widely used in today's times.	These are machine-friendly languages that are very difficult to understand by human beings but easy to interpret by machines.
Ease of Execution	These are very easy to execute.	These are very difficult t
Process of Translation	High-level languages require the use of a compiler or an interpreter for their translation into the machine code.	Low-level language requires an assembler for directly translating the instructions of the machine language.
Efficiency of Memory	These languages have a very low memory efficiency. It means that they consume more memory than any low-level language.	These languages have a very high memory efficiency. It means that they consume less energy as compared to any high-level language.
Portability	These are portable from any one device to another.	A user cannot port these from one device to another.
Comprehensibility	High-level languages are human-friendly. They are, thus, very easy to understand and learn by any programmer.	Low-level languages are machine-friendly. They are, thus, very difficult to understand and learn by any human.
Dependency on Machines	High-level languages do not depend on machines.	Low-level languages are machine-dependent and thus very difficult to understand by a normal user.

Debugging	It is very easy to debug these languages.	A programmer cannot easily debug these languages.
Maintenance	High-level languages have a simple and comprehensive maintenance technique.	It is quite complex to maintain any low-level language.
Usage	High-level languages are very common and widely used for programming in today's times.	Low-level languages are not very common nowadays for programming.
Speed of Execution	High-level languages take more time for execution as compared to low-level languages because these require a translation program.	The translation speed of low-level languages is very high.
Abstraction	High-level languages allow a higher abstraction.	Low-level languages allow very little abstraction or no abstraction at all.
Need of Hardware	One does not require a knowledge of hardware for writing programs.	Having knowledge of hardware is a prerequisite to writing programs.
Facilities Provided	High-level languages do not provide various facilities at the hardware level.	Low-level languages are very close to the hardware. They help in writing various programs at the hardware level.
Ease of Modification	The process of modifying programs is very difficult with high-level programs. It is because every single statement in it may execute a bunch of instructions.	The process of modifying programs is very easy in low-level programs. Here, it can directly map the statements to the processor instructions.
Examples	Some examples of high-level languages include Perl, BASIC, COBOL, Pascal, Ruby, etc.	Some examples of low-level languages include the Machine language and Assembly language.

4.0 Conclusion

In terms of speed, programs written in low-level languages are faster than those written in middle and high-level languages. This is because these programs do not need to be interpreted or compiled. They interact directly with the registers and memory. On the other hand, programs written in a high-level language are relatively slower.

5.0 Summary

There are clear differences between high-level, mid-level, and low-level programming languages. We can also point out that each type of programming language is designed to serve its specific purpose. For this reason, depending on the purpose each programming language is serving is what make one type of programming better and prefer over the other.

6.0 Tutor-Marked Assignment

1. What are some features of specific programming languages you know whose rationales are a mystery to you?

2. Name and explain another criterion by which languages can be judged
3. In your opinion, what major features would a perfect programming language include?
4. Was the first high-level programming language you learned implemented with a pure interpreter, a hybrid implementation system, or a compiler? (You may have to research this.)
5. Describe the advantages and disadvantages of low level, middle level and high level languages

7.0 References/Further Reading

1. A History of Computer Programming Languages (brown.edu)
2. A Brief History of Programming Languages." http://www.byte.com/art/9509/se7/artl9.htm. Cited, March 25, 2000.
3. A Short History of the Computer." http://www.softlord.com/comp/. Jeremy Myers. Cited, March 25, 2000.
4. Bergin, Thomas J. and Richard G. Gibson, eds. *History of Programming Languages-II*. New York: ACM Press, 1996.
5. Christiansen, Tom and Nathan Torkington. *Perlfaq1 Unix Manpage*. Perl 5 Porters, 1997-1999.
6. Java History." http://ils.unc.edu/blaze/java/javahist.html. Cited, March 29, 2000.
8. Programming Languages." *McGraw-Hill Encyclopedia of Science and Technology*. New York: McGraw-Hill, 1997.
9. Wexelblat, Richard L., ed. *History of Programming Languages*. New York: Academic Press, 1981.
10. A History of Computer Programming Languages (onlinecollegeplan.com)
11. IT Lecture 1 History of Computers.pdf (webs.com) Prepared by Miss N. Nembhard, Source: Robers, Eric S. The Art and Science of C. Addison-Wesley Publishing Company. Reading: 1995.

UNIT 2 EVOLUTION OF PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Describes the development of a collection of programming languages.
- To understand the evolution of various programming languages, compare and contrast and examine the advantages and disadvantages one language has over the other.
- Student should be able to write simple programs in various languages, focuses on the contributions of the language and the motivation for its development.
- to introduce the concepts of logic programming and logic programming languages, including a brief description of a subset of prolog.

3.0 MAIN CONTENT

3.1 C/C+ Languages

The programming language was created, designed and developed by a Danish Computer Scientist named Bjarne Stroustrup at Bell Telephone Laboratories (now known as Nokia Bell Labs) in Murray Hill, New Jersey in 1979. As he wanted a flexible and a dynamic language which was similar to C with all its features, but with additional of active type checking, basic

inheritance, default functioning argument, classes, in-lining, etc. and hence C with Classes (C++) was launched. C++ was initially known as "C with classes, and was renamed C++ in 1983. ++ is shorthand for adding one to variety in programming; therefore, C++ roughly means that "one higher than C."

The C++ language is a procedural and as well as object-oriented programming language and is a combination of both low-level and high-level language, regarded as (Intermediate/Middle-Level Language). C++ is sometimes called a hybrid language because it is possible to write object oriented or procedural code in the same program in C++. This has caused some concern that some C++ programmers are still writing procedural code, but are under the impression that it is object orientated, simply because they are using C++. Often it is an amalgamation of the two. This usually causes most problems when the code is revisited or the task is taken over by another coder.

C/C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. C/C++ is one of the most popular programming languages and its application domains include systems software (such as Microsoft Windows), application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. It was designed for UNIX system environment to enable programmer to improve the quality of code produced thus making reusable code easier to write. C/C++ offers broad range of OOP features such as multiple inheritance, strong typing, dynamic memory management, template, polymorphism, exception handling and overloading. C/C++ also support usual features such as a variety of data type including strings, arrays, structures, full of input and output features, data pointers and typed conversion.

C/C++ continues to be used and is one of the preferred programming languages to develop professional applications. C/C++ has dynamic memory allocation but does not have garbage collection: that allow program to misuse and lack memory. It also supports dangerous raw memory pointers and pointer arithmetic which decrease the time needed for software development. C/C++ has no official home in web as it follows the old tradition of translating C++ into C and others are native compiler. C/C++ is the mother of all high level languages. Example of C++ compiler are GNU, C/C++ compiler, GCC.

3.1.1 C/C++ Program Structure

Let us look at a simple code that would print the words *Hello World*.

```
#include <iostream>
Using namespace std;

// main() is where program execution begins
Int main() {
   Cout<<"Hello World";//prints Hello World
Return 0;
}
```

Let us look at the various parts of the above program

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.

- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.

- The next line '**// main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.

- The line **int main()** is the main function where program execution begins.

- The next line **cout << "Hello World";** causes the message "Hello World" to be displayed on the screen.

- The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

- **Semicolons and Blocks in C++:** In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity. For example, following are three different statements:

```
x = y;
y = y + 1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

```
{
       cout << "Hello World"; // prints Hello World
       return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

```
x = y;
y = y + 1;
add(x, y);
```

is the same as `x = y; y = y + 1; add(x, y);`

- **C++ Identifiers:** A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline

character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit. Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are reserved keywords. The standard reserved keywords are:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum,
explicit, export, extern, false, float, for, friend, goto, if, inline,
int, long, mutable, namespace, new, operator, private, protected,
public, register, reinterpret_cast, return, short, signed, sizeof,
static, static_cast, struct, switch, template, this, throw, true, try,
typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t,
```

while Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
```

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the `RESULT` variable is not the same as the `result` variable or the `Result` variable. These are three different variable identifiers.

- **Fundamental data types:** When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way. A summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Ranger*
char	Character or small integer	1byte	Signed: -128 - 127 Unsigned: 0 t- 255
short int (short)	Short integer	2bytes	Signed: -32768 – 32767 Unsigned: 0 - 65535
int	Integer	4bytes	Signed: -2147483648-2147483647 Unsigned: 0 - 4294967295
long int (long)	Long integer	4bytes	Signed: -2147483648-2147483647 Unsigned: 0 – 4294967295
bool	Boolean value. It can take one of two values: true or false	1byte	True or false
float	Floating point number	4bytes	+/-3.4e_/-38(~7 digits)
double	Double precision floating point number	8bytes	+/-1.7e_/-308(~15 digits)

long double	Long double precision floating point number	8bytes	+/-1.7e_/-308(~15 digits)
Whar_t	Wide character	2 or 4 bytes	1 wide character

- **Declaration of variables:** In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

3.1.2 Features of C/C++

- **Simplicity:** It is a simple programming language in the sense of a structured approach that is the program can be broken down into parts and can be resolved modularly. This helps the programmer to recognize and understand the problem easily.

- **Platform or Machine Independent:** A language is said to be platform dependent whenever the program is executing in the same operating system where that was developed and compiled but not run and execute on other operating system. C++ is platform dependent language.

- **High-Level Language**: C++ is a high level language, making it easier to understand as it is closely associated with the human-comprehensible language that is English. It also supports the feature of low-level language that is used to develop system applications such as kernel, driver, etc. That is why it is also known as mid-level language as C++ has the ability to do both low-level & high-level programming.

- **Multi-paradigm programming language:** C++ is a language that supports object-oriented, procedural and generic programming. It supports at least 7 different styles of programming. This makes it very versatile.

- **Case Sensitive:** Similar to C, C++ also treats the lowercase and uppercase characters in a different manner. For example, the keyword "cout' (for printing) changes if we write it as 'Cout' or "COUT"**.** But this problem does not occur in other languages such as HTML and MySQL.

- **Compiler-based:** C++ is a compiler-based programming language, unlike Java or Python which are interpreter-based. This feature makes C++ comparatively faster than Java or Python language.

- **Existence of Libraries:** The C++ programming language offers a library full of in-built functions that make things easy for the programmer. These functions can be accessed by including suitable header files.

- **Dynamic memory allocation:** Since C++ supports the use of pointers, it allows us to allocate memory dynamically. Constructors and destructors can even be used while working with classes and objects in C++.

- **Syntax based language:** C++ is a strongly tight syntax based programming language. If any language, follow rules and regulation very strictly known as strongly tight syntax based language. Example C, C++, Java, .net etc. If any language not follow rules and regulation very strictly known as loosely tight syntax based language. Example HTML.

3.2 C# Language

C# During the development of the .NET Framework, the class libraries were originally written using a managed code compiler system called Simple Managed C (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called Cool, which stood for "C-like Object Oriented Language". Microsoft had considered keeping the name "Cool" as the final name of the language, but chose not to do so for trademark reasons. By the time the .NET project was publicly announced at the July 2000 Professional Developers Conference, the language had been renamed C#, and the class libraries and ASP.NET runtime had been ported to C#.

3.2.1 Some notable distinguishing features of C# are:

- It has no global variables or functions. All methods and members must be declared within classes. Static members of public classes can substitute for global variables and functions.
- Local variables cannot shadow variables of the enclosing block, unlike C and C++. Variable shadowing is often considered confusing by C++ texts.

- C# supports a strict Boolean data type, bool. Statements that take conditions, such as while and if, require an expression of a type that implements the true operator, such as the boolean type. While C++ also has a boolean type, it can be freely converted to and from integers, and expressions such as if(a) require only that a is convertible to bool, allowing a to be an int, or a pointer. C# disallows this "integer meaning true or false" approach, on the grounds that forcing programmers to use expressions that return exactly bool can prevent certain types of common programming mistakes in C or C++ such as if (a = b) (use of assignment = instead of equality ==).

- In addition to the try...catch construct to handle exceptions, C# has a try...finally construct to guarantee execution of the code in the finally block.

- C# currently (as of version 4.0) has 77 reserved words.

- Multiple inheritances are not supported, although a class can implement any number of interfaces. This was a design decision by the language's lead architect to avoid complication and simplify architectural requirements throughout CLI.

3.2.2 Common Type System (CTS)

C# has a unified type system. This unified type system is called Common Type System (CTS). A unified type system implies that all types, including primitives such as integers, are subclasses of the System.Object class. For example, every type inherits a ToString() method. For performance reasons, primitive types (and value types in general) are internally allocated on the stack.

3.2.3 Categories of data types

CTS separate data types into two categories:

- **Value types:** they are plain aggregations of data. Instances of value types do not have referential identity nor a referential comparison semantics - equality and inequality comparisons for value types compare the actual data values within the instances, unless the corresponding operators are overloaded. Value types are derived from System. ValueType, always have a default value, and can always be created and copied. Some other limitations on value types are that they cannot derive from each other (but can implement interfaces) and cannot have an explicit default (parameterless) constructor. Examples of value types are all primitive types, such as int (a signed 32-bit integer), float (a 32-bit IEEE floating-point number), char (a 16- bit Unicode code unit), and System.DateTime (identifies a specific point in time with nanosecond precision). Other examples are enum (enumerations) and struct (user defined structures).

- **Reference types:** they have the notion of referential identity - each instance of a reference type is inherently distinct from every other instance, even if the data within both instances is the same. This is reflected in default equality and inequality comparisons for reference types, which test for referential rather than structural equality, unless the corresponding operators are overloaded (such as the case for System.String). In general, it is not always possible to create an instance of a reference type, nor to copy an existing instance, or perform a value comparison on two existing instances, though specific reference types can provide such services by exposing a public constructor or implementing a corresponding interface (such as ICloneable or IComparable). Examples of reference types are object (the ultimate base class for all other C# classes), System.String (a string of Unicode characters), and System.Array (a base class for all C# arrays). Both type categories are extensible with user-defined types

3.2.4 C# Program Structure

```
using system;

namespace HelloWorldApplication {
     class HelloWorld {
          static void Main (string[] args) {
               /* my first program in C# */
               console.WritteLine ("Hello World");
               concole.ReadKey();
          }
     }
}
```

- The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.

- The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
- The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
- The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class does when executed.
- The next line /*...*/ is ignored by the compiler and it is put to add **comments** in the program.
- The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");** *WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
- The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

3.2.5 Features of C#

- **Simple:** C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.
- **Modern Programming Language:** C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.
- **Object Oriented**: C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.
- **Type Safe:** C# type safe code can only access the memory location that it has permission to execute. Therefore, it improves a security of the program.
- **Interoperability:** Interoperability process enables the C# programs to do almost anything that a native C++ application can do.
- **Scalable and Updateable:** C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.
- **Component Oriented**: C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

- **Structured Programming Language:** C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

- **Rich Library:** C# provides a lot of inbuilt functions that makes the development fast.

- **Fast Speed:** The compilation and execution time of C# language is fast.

3.3 Java Language

Java was developed by James Gosling from SunMicrosys by his team in 1991. In 1995 Netscape Incorporated released its version of Netscape browser capable of running Java programs. Original name of Java was Oak and the new name was inspired by a coffee bean. Java is a general all-purpose language which can be used independent of the internet. Java borrowed idea from Modula – 3 Mesa and objective – C.

Java has features of inheritance, strong type checking, modularity, exceptional handling, polymorphism, dynamic loading of libraries, arrays, strings handling, concurrency etc. Fundamental component of Java is class, Java is quite verbose and one need to understand OOP to make the most effective use of Java. Java seems restructure since it does not use pointers and does not provide the raw power of C/C++

3.3.1 Syntax of Java

```
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Prints the string
to the console.
    }
}
```

It is very important to keep in mind the following points in Java program:

- **Case Sensitivity**: Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

- **Java Identifiers:** All Java components require names. Names used for classes, variables, and methods are called identifiers. In Java, there are several points to remember about identifiers. *All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
  - ✓ After the first character, identifiers can have any combination of characters.
  - ✓ A key word cannot be used as an identifier.
  - ✓ Most importantly, identifiers are case sensitive.
  - ✓ Examples of legal identifiers: age, $salary, _value, __1_value.
  - ✓ Examples of illegal identifiers: 123abc, -salary.

- **Java Modifiers:** Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

- ✓ Access Modifiers − default, public, protected, private
- ✓ Non-access Modifiers − final, abstract, strictfp.

- **Java Variables:** Following are the types of variables in Java −
  - ✓ Local Variables
  - ✓ Class Variables (Static Variables)
  - ✓ Instance Variables (Non-static Variables)

- **Java Arrays:** Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap.

- **Literals:** These are the identifiers which have a particular value in itself. These can be assigned to variables. Literals can also be thought of as constants. These are of different types such as numeric, characters, strings etc.

  a. Numeric Literals: For numeric literals there are 4 kinds of variations:
  - ✓ Decimal(Any number of base 10), Example- 87,53
  - ✓ Binary(Any number with a base 2), Example- 1011,110
  - ✓ Octal Point(Any number with base 8), Example= 1177
  - ✓ Hexadecimal Point(Any number with base 16), Example- A54C

  b. Floating point Literals: We can specify the numeric values only with the use of a decimal point(.). These numbers represent fractional numbers which cannot be expressed as whole integers. For Example: 10.876

  c. Character Literals: These are the literals which deal with characters i.e inputs which are not numeric in type.
  - ✓ Single quoted character: This encloses all the uni-length characters enclosed within single quotes. Example- 'a','j'.
  - ✓ Escape Sequences**:** These are the characters preceded by a backslash which perform a specific function when printed on screen such as a tab, creating a new line, etc. Example- '\n'
  - ✓ Unicode Representation**:** It can be represented by specifying the concerned unicode value of the character after '\u'. Example- "\u0054'

- Comments in Java: Comments are needed whenever the developer needs to add documentation about a function which is defined within the program. This is to enhance the code-readability and understandability. Comments are not executed by the compiler and simply ignored during execution. The comments are of the following types:

  a. Single Line Comments in Java: These comments, as the name suggests, consist of a single line of comment generally written after a code line to explain its meaning. They are marked with two backslashes (//) and are automatically terminated when there is a new line inserted in the editor. For Example:

```
int i = 6;
```

```
String s = "DataFlair"; // The value of i is set to 6
initially.The string has value "DataFlair"
```

b. **Multi Line Comments in Java:** These comments span for multiple lines throughout the codebase. They are generally written at the beginning of the program to elaborate about the algorithm. These are also used by developers to comment out blocks of code during debugging. They comprise of a starting tag(/*) and an ending tag(*/). For Example:

```
class Comment {
   public static void main(String[] args) throws IOException {
        /* all this is under a multiline comment
        These wont be executed by compiler
        Thank you*/
}
}
```

c. **Documentation Comment:** The javadoc tool processes these comments while generating documentation.

- **Java Keywords:** The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names. Keywords are the identifiers which have special meaning to the compiler. These cannot be used to name variables, classes, functions etc. These are reserved words.

abstract	assert	boolean	break	byte
case	class	catch	char	const
continue	default	continue	default	do
else	double	enum	extends	final
extends	finally	float	for	goto
if	implements	import	instanceof	int
long	native	new	package	interface
private	protected	public	return	private
short	static	strictfp	super	short
switch	synchronized	this	throw	volatile
throws	transient	try	Void	while

3.3.2 Basic structure of Java program

There are two basic parts of a Java program namely, Packages and Main Method.

a. **Package:** This is the same thing as a folder in your computer. It contains classes, interfaces and many more. It organizes the classes into namespaces. The classes which are of the same package can access each other's protected and private members. They are generally imported by using the import keyword i.e, import java.util.* where we are importing java's util package

b. **Main Method:** The main method marks the entry point of the compiler in the program. The main method must always be static. For Example:

```
public class DataFlair {
     void teachJava() {
          System.out.println("Teaching Java by DataFlair");
}
public static void main(String[] args) throws IOException {
          System.out.println("In main");
          DataFlair ob = new DatFlair();
          ob.teachJava();
     }
}
```

Output is:

```
In main
Teaching Java by DataFlair
```

- **Control statements:** The syntax of control statements in Java are pretty straightforward. Let's take a deeper look into the control statements in Java.

  a. Conditional Statements: These statements are purely based on condition flow of the program. Its divided into the following 3 parts:

  ✓ if statement: The statement suggests that if a particular statement yields to true then the block enclosed within the if statement gets executed. Exsmple:

```
if (condition) {
     action to be performed
}
```

  ✓ if else statement: This statement is of the format that if a condition enclosed is true then the if block gets executed. If not the else block gets executed**.** Example:

```
if (condition) {
     action1;
}
else {
     action2
}
```

  ✓ Else if statement: This statement encloses a if statement in an else block. Example

```
if (condition) {
     action 1
}
else if (condition2) {
```

```
     action 2
}
```

- ✓ Switch case: The switch case is used for multiple condition checking. It's based on the value of the variable. The value of the variable mentioned marks the flow of the control to either of the case blocks mentioned. Example:

```
switch (var_name)
case value1:
     action1;
     break;
case value2:
     action2;
     break;
default:
     action3;
     break;
```

b. **Iteration Statements:** These are the statements which are primarily known as loops in programming which run a particular set of programs a fixed number of times, Some of the types of iterative statements are:

- ✓ For loop: The for loop is responsible for running the snippet of code inside it for a predetermined number of times. Example:

```
for (i = 0; i < 5; i++) {
     System.out.println("Hi");
}
```

- ✓ While loop: This type of loop runs indefinitely until the condition is false. Example:

```
while (i < 6) {
          System.out.println("Hi");
i++;
}
```

This prints Hi on the screen five times until the value of i becomes 6

- ✓ Do-while loop: This is the same as the while loop. The only difference lies in the fact that the execution occurs once even if the condition is false. Example:

```
do {
     System.out.println("Hi");
}
while ( i > 6 );
```

c. **Control Flow Statements/Jump Statements:** Sometimes we need to discontinue a loop during execution.

- ✓ Break statement: This breaks the nearest loop inside which is mentioned. The execution continues from the next line just when the current scope ends.

- ✓ Continue Statement: This continues the execution from the next iteration of the loop and skips the current execution. Example:

```
while (i < 10) {
     if (i == 3) continue;
     i++;
     }
```

- ✓ Return statement: The return statements are generally useful in methods when returning a value when the function is done executing. After the return statement executes, the remaining function does not execute.

- **Exception Handling in Java:** Exception handling is important to custom output the errors during the unfortunate case of an error occurrence. Syntax of the exception handling is fairly simple and structured. It goes as below:

```
try {
     // Code block within which error can occur
}
catch(Exception e) {
     //Code block to perform when error thrown
}
Finally {
     //Code to be executed after the end of try block. This is
executed even if there is no error.
}
```

There is a special keyword called throws, it is useful to throw custom exceptions. For Example- **throw new ArithmeticException();**

- ✓ **Try:** This block houses the code which is responsible for an error thrown. Generally programmers enclose the code which they think may throw an error in the try block.
- ✓ **Catch:** This block houses the code to be performed when a particular exception is found. There can be custom messages defining what kind of error has occurred for better documentation and flow of the program.
- ✓ **Finally:** The finally block executes whether or not there is any error faced by the compiler. This part is generally used to enclose the code that has to be executed irrespective of the errors occurred during compilation/execution of the program. Example program to evaluate exception handling in Java

```
import java.io. * ;
class ExcptionHandle {
     public static void main(String[] args) throws IOException {
```

```
            int a = 10,
            b = 0;
            int c;
            try {
                c = a / b;
            } catch(Exception e) {
                System.out.println(e);
                //TODO: handle exception
            }
}
```

Output: `java.lang.ArithmeticException: / by zero`

- **Operator in Java:** As the name suggests, operators are the ones for performing operations on two or more entities. They are of multiple types as below:

  a. **Arithmetic:** These are the operators which are solely for performing arithmetic operations. These include addition (+), subtraction(-), multiplication (*), division (/), modulo(%) and many more.

  b. **Relational:** These are the operators which obtain the relation between two different entities in a program. These include less than(<), greater than(>), less than or equals to(<=), greater than or equals to(>=) ,equals to(==), not equals to(!=) Example:

```
if (a < b) {
   System.out.println("A greater");
}
```

  c. **Bitwise:** These operators are useful for performing bitwise operations on an entity. These include AND(&), bitwise OR(|), bitwise XOR(^), bitwise complement (~), bitwise left shift(<<) and so on. Example: (A&B) will give 12 if a = 0000 and b= 1100

  d. **Logical:** These operators are useful to check the logic of a particular operation of two operands. These include Logical AND(&&), Logical OR(||), logical NOT(!) and so on. For Example: `if (a == 6 && b == 5)`

  e. **Assignment:** These operators are responsible for assigning variables to variables. These include equal(=),add AND(+=), subtract AND operator(-=) and so on. Example:

```
int x = 65;
int y +=6

int y+=6(equivalent to int y=y+6;)
```

- **Object in Java:** Objects are created from classes in Java. Once we define a class, we can create the object of a class by the following simple syntax. These are the instances of the class:

```
< class - name > <object - name > =new < class - name of instance creation > ()
```

Example: `DataFlair java = new DataFlair();`

- **Class in Java:** Classes generally start with the class name which has its first letter capital. Generally the case is CamelCase for class names. It has a very simple syntax as below:

```
class < Class - name > {
     instance variables;
          class method1() {}
class method2() {}
} //end class
```

```
Example:
class DataFlair {
     int a;
     void teach() {
         System.out.println("Learning java from DataFlair");
     }
}
```

- **Methods in Java:** Methods or functions are specific entities which return a value only when they are called. They have a syntax similar to classes.

```
<
return type > <
function - name > {
     action1;
     action2;
}
```

For Example:

```
void print() {
     System.out.println("hey I am learning Java at DataFlair");
}
```

- **Interfaces:** Interfaces are a collection of abstract methods in Java. We will learn more about Java in the following articles. We define interfaces as below;

```
interface < interface - name > {
     static functions;
     abstract methods;
}
```

Example:

```
interface DataFlair {
     void teach();
     static void evaluate();
}
```

- **Access modifiers:** Access modifiers as the name suggests, limits the access of the entities they are defined with. The access modifiers used by Java are:
  - ✓ **Public** – Accessible to every other class or interface. There is no restriction of access.
  - ✓ **Private**– This keyword renders all entities to be accessible only inside the class they are declared.
  - ✓ **Protected**– The protected members of the class are accessible to classes within the same package or subclasses of different packages.
  - ✓ **Default**– If no access modifier is mentioned then the default access modifier is invoked. This limits the access of the particular entity within the same package. For Example**:**

```
public int a=8
```

- **Arrays:** Arrays are consecutive data items of the same datatype. They have a fairly simple syntax of declaration. If the array has to be declared explicitly it has the syntax of:

```
< data type > <array - name > [ < array - size > ] = {
     data1,
     data2,
     data3...
};
```

For Example:

```
int arr[3] = {
     1,
     2,
     3
};
```

else if the array has to be declared during runtime.

```
< data type > <array - name > [] = new < array - name > [ < sizeofArray > ]
```

Example: `int arr[] = new int[10];`

- **Variables:** The variables concept has been explained in the following articles,however the syntax of variables is simple and easy to learn.

```
Java Syntax for variables:
< data type > <variable - name >
```

```
Example:
String s = ”DataFlair”;
```

- **Datatype:** The datatypes come before variables to define the type of data they would be storing. These include int,short,byte,float,double;

Syntax

```
< datatype > <
var - name >
```

Example: `int b = 12;`

3.3.3 Features of Java

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.

- **Distributed** − Java is designed for the distributed environment of the internet.

- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

3.4 Python Language

Python was imagined in the late 1980s by Guido van Rossum at Centrum Wiskunde and Informatica in the Netherlands as a successor to the ABC language, equipped for a particular case dealing with and interfacing with the Amoeba working framework. Python usage started in December 1989 firstly as a hobby project because he was looking for an interesting project to keep him occupied during Christmas. He declared his perpetual excursion from his duties as Python's Benevolent Dictator For Life (BDFL), a title the Python people group presented to him to mirror his extended haul responsibility as the undertaking's central chief in 12th July, 2018. He presently shares his initiative as an individual from a five-person directing council and in January 2019, dynamic Python center designers chose Brett Cannon, Nick Coghlan, Barry Warsaw, Carol Willing and Van Rossum to a five-part "Controlling Council" to lead the task and the language was finally released. For quite some time he used to work for Google, but currently, he is working at Dropbox.

Python was created through some ABC syntax and its good features by creating a good scripting language which had removed all the flaws and fixed the issues in ABC. The inspiration for the name came from BBC's TV Show: 'Monty Python's Flying Circus'. Python language is a dynamically typed and garbage-collected one. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python was often described as a "batteries included" language due to its comprehensive standard library. It used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C. Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity. When it was released it had more than enough capability to provide classes with inheritance, several core data types exception handling and functions.

Python's strength as a new language was how easy it was to extend it support multiple platforms. It is capable of communicating with different file formats and libraries. Van Rossum also aimed for making it a programming language for everybody. So the language's design is simple yet powerful. As it grew in popularity and adoption. Python does not appear to be going away anytime soon, as its user base is large and increasing. Python is used by many well-known organizations, and numerous OS developers support it, making Python's future look bright. In 2007, 2010, and 2018, Python was named TIOBE's Programming Language of the Year. This award is given to the language that has had the greatest increase in popularity over the course of the year.

3.4.1 Python Program Structure

Properties

Python is implicitly and dynamically typed, so you do not have to declare variables. The types are enforced, and the variables are also case sensitive, so var and VAR are treated as two separate variables.

Data types

The data structures in Python are dictionaries, tuples and lists. Python has five standard data types

- Numbers
- String
- List
- Tuple
- Dictionary

Python Lists

Lists are similar to one-dimensional arrays, although you can also have lists of other lists. Dictionaries are essentially associative arrays or hash tables.
Tuples are one-dimensional arrays. Python arrays can be of any type, and the types are always zero. Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

Operators are the constructs which can manipulate the value of operands.
Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators

- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Strings

Python strings can either use single or double quotation marks, and you can use quotation marks
Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

- **Identifiers in Python**: The python identifier is a name to identify a variable, function, class, module or any other object.

  - It is case sensitive.
  - Reserved keywords cannot be used as an identifier
  - Special characters like @,! #,$,% cannot be used as an identifier
  - There are some naming conventions similar to other programming languages
    - Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
    - Starting an identifier with a single leading underscore indicates that the identifier is private.
    - Starting an identifier with two leading underscores indicates a strongly private identifier.
    - If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

- **Reserved Words:** The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not	assert	finally
or	break	for	pass	class
from	print	continue	global	raise
def	if	return	del	import
try	elif	in	while	else
is	with	except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

- Its definition should be independent of any particular hardware/OS
- Its definition should ne standardized and compile implementation should comply with this standard.
- It should support software engineering technology
- It should discourage / prohibiting poor practices and promoting or support maintenance activities.
- It should effectively support the application domains of interest.
- It should support the requirements level of system reliability and safety.
- Its compiler implementation should be commensurate with the current state of technology.
- Appropriate software engineering based support tools and environment should have considered.

3.4.2 Features of Python Language

- **Simple Syntax:** When it comes to programming, understanding the syntax of the language is very important. The program will not work if it's not syntactically correct. With different languages, the ease with which you code, comes with practice. In Python, the developers of the language wanted to make sure that the language would be as close to the human language as possible. This was done while keeping the programming rules in mind and hence the language has a simple syntax.

- **Easy to Learn:** Python being simple to code is one of the main reasons the language has become popular over time. The simple syntax allows people to work with it easily. It allows people who are not familiar with programming to pick up a new language quickly. It also gives an advantage to developers to code without having to debug or have errors in the code too often.

- **Free and Open Source:** When Guido Van Rossum was developing the language, he was set on the idea that the language should be available to all. This allowed people from all around the world to access the language with ease and also help build the language further. Python is a free/libre and open-source software (FLOSS), which makes reading the source code, distributing the software, and making changes to it quite easy.

- **High-level Language:** Python is a high-level programming language, which focuses on making the code easier for the programmer. High-Level languages are easier to comprehend and work with. High-level languages are slower than low-level languages because their syntax is not understood by the machine. In high-level languages, there comes a step where you need to convert this code into a code the computer can understand (low-level code) either during or before runtime. However, people still prefer high-level languages like Python because of the ease and simplicity of working with them.

- **Interpreted:** In interpreted languages, the step of first converting the main code into a binary code does not take place. The code is then converted to machine code during runtime, which makes the language easy to debug and portable, as seen in the flowchart below.

- **Memory Allocation:** In Python, memory allocation becomes easier because when you assign a value to a variable, automatic memory allocation takes place during run time.

- **Portable:** Portability is one of the major advantages of using Python. Python is free for all open-source languages and high-level languages which allows you to work on multiple platforms and operating systems. You can use python on operating systems such as Linux, UNIX, Windows, Mac OS X / OS X / macOS, etc.

- **Object-Oriented:** When we talk about object-oriented programming, the main goal here is to focus on the data and functionality and build the program around the object. Let us look at some of the advantages of object-oriented programming languages. OOP allows the reusability of codes. It also works on data redundancy, which allows you to store the data in multiple memory locations. Data redundancy provides memory backup in case you lose the memory at one location. Having OOP, in turn, allows the concept of inheritance, polymorphism, and encapsulation to come into play.

- **Procedure-oriented**: Python is one of the few programming languages that supports both object-oriented programming as well as Procedural Programming. This feature allows you to work on the reusable functions that focus on solving the problem at hand. One can also use this feature to call the same function in other areas of the code as well.

- **Classes**: Python is an object-oriented programming language that allows the use of classes. OOP can be used to create multiple parent and child classes. These classes follow the class inheritance rules. This allows the parent class to override any method of the child class. The mechanism also allows the child to call a method from the parent class.

- **Extensible:** Python has a very helpful feature that allows developers who are comfortable and confident with C/C++/Java code to write code in these languages. This makes Python extensible as you can extend these languages onto other programming languages. Since C and C++ are not interpreted languages, the code is first converted to machine code and then run on Python.

- **Embeddable:** Python also has a feature that allows you to generate codes in other languages and embed them into python code or vice-versa.

- **Extensive Standard Library:** Python is free and open-source, which led people from all around the world to come up with libraries focusing on various areas. Over time, there has been a generation of a vast range of libraries, starting from libraries for documentation, regulation to machine learning, natural language processing, web scraping, etc.

- **Wide Range of in-built functions:** Other than the functions that these wide ranges of libraries provide, Python has a set of extensive inbuilt functions that can be used without accessing any library.

- **Supports exception handling:** When programs are created, a good number of them will have certain functions, inputs or problems that, when faced, can disrupt the flow of the

program. These are considered to be exceptions. If they are identified in advance can prevent the program from either crashing or going into an infinite loop. Python is one of the few languages that are capable of supporting exception handling to a great extent which makes it a developer and programmer's favorite language.

- **Robust:** Python's ability to work well with exceptions and exception handling, as well as its ease of use and comprehension has made the error rate in the code reduce. This makes the language a very reliable and robust language to work with.

- **Advance features:** Python has a wide range of other advanced features such as list, dataframes, dictionary, and generator comprehensions.

- **GUI Programming Support:** Now, we have established that Python has a vast range of libraries. One of their unique libraries is the Graphic User Interface library which developers use for developing Desktop applications.

3.5 LISP Language

Lisp (derives from "LISt Processing") is one of the oldest programming languages. It was invented in 1958, with the language being conceived by John McCarthy and is based on his paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine". Over the years, Lisp has evolved into a family of programming languages.

List Processing is functional or lambda that is based language extremely rich and powerful programming language that has enjoyed continuous use and popularity since mid 1960. It is a weakly types language with excellent support for reflection and on-the fly code generation and interpreting. The languages are extreme flexibility, expressive power and has ability to treat code as data which make it the undisputed language of AI research for all the 1970s and 1980s. All LISP implementation offered a set of programming features tough to equal in any language. Some today features are macros, string handling, recursion, closures, reflection, arrays, extensive input and output facilities. LISP is the second oldest high level language, it is extremely well documented and the most wide spread dialect.

It has simple data structure (atoms and lists) and makes heavy use of recursion. It is an interpretive language and has many variations. The most commonly used general-purpose dialects are Common Lisp and Scheme. Other dialects include Franz Lisp, Interlisp, Portable Standard Lisp, XLISP and Zetalisp. It is the most widely used AI programming language.

3.5.1 Structure of LISP

Valid Objects: A LISP program has two types of elements: atoms and lists.

Atoms: Atoms include numbers, symbols, and strings. It supports both real numbers and integers.

- symbols: a symbol in LISP is a consecutive sequence of characters (no space). For example a, `x,` and `price-of-beef` are symbols. There are two special symbols: T and NIL for logical true and false.

- strings: a string is a sequence of characters bounded by double quotes. For example "`this is red`" is a string.

S-expression: An S-expression (S stands for symbolic) is a convention for representing data or an expression in a LISP program in a text form. It is characterized by the extensive use of prefix notation with explicit use of brackets (affectionately known as Cambridge Polish notation). S-expressions are used for both code and data in Lisp. S-expressions were originally intended only as machine representations of human-readable representation of symbols, but Lisp programmers soon started using S-expressions as the default notation. S-expressions can either be single objects or atoms such as numbers, or lists.

Evaluation of S-expression

1) Evaluate an atom
- Numerical and string atoms evaluate to themselves;
- Symbols evaluate to their values if they are assigned values, return Error, otherwise;
- The values of T and NIL are themselves.

2) Evaluate a list
- Evaluate every top element of the list as follows, unless explicitly forbidden:
- The first element is always a function name; evaluating it means to call the function body;
- Each of the rest elements will then be evaluated, and their values returned as the arguments for the function. Examples:

```
>(+ (/ 3 5) 4) 23/5
```

The first element is + so we make a call to + function with (/ 3 5) and 4 as arguments. Now (/ 3 5) is evaluated and the answer is 3/5. After that 4 is evaluated to itself and thus returns 4. So, 3/5 and 4 are added to return 23/5.

Lists: List is the most important concept in LISP. A list is a group of atoms and/or lists, bounded by (and). For example, (a b c) and (a (b c)) are two lists. In these examples the list (a b c) is a list of atoms a, b, and c, whereas (a (b c)) is a list of two elements, the first one an atom a, and the second one a list (b c).

Top elements of a list: The first-level elements in LISP are called top-level elements. For example, top elements of list (a b c) are a, b, and c. Similarly, top elements of list (a (b c)) are a and (b c). An empty list is represented by nil. It is the same as ().

Function calls: In LISP, a function and a function call is also a list. It uses prefix notation as shown below: `(`

```
function-name arg1 ... argn)
```

A function call returns function value for the given list of arguments. Functions are either provided by LISP function library or defined by the user. There are many built-in function in LISP. This includes math functions as well as functions for manipulating lists.

The math functions include: `- +, -, *, /, exp, expt, log, sqrt, sin, cos, tan, max, min` with the usual meanings and function for manipulating lists are list constructor (cons, list and append), list selectors (first (car) and rest (cdr), predicate: (A predicate is a special function which returns NIL if the predicate is false, T or anything other than NIL, otherwise. Predicates are used to build Boolean expressions in the logical statements) and set operations: (A list can be viewed as a set whose members are the top elements of the list. The list membership function is a basic function used to check whether an element is a member of a list or not). In order to select elements from a list, selectors functions are used.

Defining LISP functions: In LISP, defun is used to write a user-defined function. Note that different dialects of LISP may use different keywords for defining a function. The syntax of defun is as below:

```
(defun func-name (arg-1 ... Arg-n) func-body)
```

That is, a function has a name, list of arguments, and a body. A function returns the value of last expression in its body.

The Concept of Local and Global Variables: Local variables are defined in function body. Everything else is global.

Conditional control: LISP has multiple conditional control statements. The set includes `if, when, and cond.`

if statement: The if statement has the following syntax:

```
(if <test> <then> <else>)
```

That is, an if statement has three parts: the test, the then part, and the else part. It works almost exactly like the if statement in C++. If the test is TRUE, then the then part will be executed otherwise the else part will be executed. If there is no else part, then if the test is not true then the if statement will simply return NIL. Here is an example that shows the if statement:

```
> (setq SCORE 78)
> 78
> (if (> score 85) 'HIGH
          (if (and (< score 84) (> score 65)) 'MEDIUM 'LOW))
> MEDIUM
```

In the above **if** statement, the **then** part contains 'HIGH and the **else** part is another **if** statement. So, with the help of nested **if** statements we can develop code with multiple branches.

cond statement: The **cond** statement in LISP is like the switch statement in C++. There is however a slight different as in this case each clause in the **cond** requires a complete Boolean test. That is,

just like multiple else parts in the **if** statement where each needs a separate condition. Syntax of **cond** is as shown below:

```
>(cond (<test-1 <action n-1>)
       (<test-2> <action-2>)
          ...
       (test-k> <action-k>))
```

Each `(<test-i> <action-i>)` is called a clause. If `test-i (start with i=1) returns T` (or anything other than NIL), this function returns the value of `action-i,` else, it goes to the next clause. Usually, the last test is T, which always holds, meaning otherwise. Just like the **if** statement, **cond** can be nested `(action-i may contain (cond ...))`

Recursion: Recursion is the main tool used for iteration. In fact, if you don't know recursion, you won't be able to go too far with LISP. There is a limit to the depth of the recursion, and it depends on the version of LISP. Following is an example of a recursive program in LISP. We shall be looking at a number of recursive functions in the examples.

```
(defun power (x y) (if (= y 0) 1 (* x (power x (1- y)))))
```

This function computes the power of x to y. That is, it computes xy by recursively multiplying x with itself y number of times.

Iteration: Apart from recursion, in LISP we can write code involving loops using iterative non-recursive mechanism. There are two basic statements for that purpose: **dotimes** and **dolist.**

dotimes: dotimes is like a counter-control for loop. Its syntax is given as below:

```
(dotimes (count n result) body)
```

It executes the body of the loop n times where count starts with 0, ends with n-1. The result is optional and is to be used to hold the computing result. If result is given, the function will return the value of result. Otherwise it returns NIL. The value of the count can be used in the loop body.

dolist: The second looping structure is **dolist**. It is used to iterate over the list elements, one at a time. Its syntax is given below:

```
(dolist (x L result) body)
```

It executes the body for each top level element x in L. x is not equal to an element of L in each iteration, but rather x takes an element of L as its value. The value of x can be used in the loop body. As we have seen in the case of dotimes, the result is optional and is to be used to hold the computing result. If result is given, the function will return the value of result. Otherwise it returns NIL.

Property Lists: Property lists are used to assign/access properties (attribute-value pairs) of a symbol. The following functions are used to manipulate a property list.

- To assign a property: (setf (get object attribute) value)
- To obtain a property: (get object attribute)
- To see all the properties; (symbol-plist object)

Array: Although the primary data structure in LISP is a list, it also has arrays. These are data type to store expressions in places identified by integer indexes. We create arrays by using the linear-array function as shown below:

```
(setf linear-array (make-array '(4)))
```

This statement creates a single dimension array of size 4 by the name of lineararray with indices from 0 to 3.

3.5.2 What made Lisp Different

LISP was one of the earliest programming language. It was designed at MIT for artificial intelligence and it has since been the defacto standard language for the AI community, especially in the US. LISP program composed of expression. They are in fact trees of expression, each of which returns a value. It has Symbol types: symbols are effectively pointer to strings stored in a hash table. One very important aspect of LISP is that the whole language is there. That is, there is no real distinction between read-time, compile-time and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime. That is, programs are expressed directly in the parse trees that get build behind the scenes when other languages are parsed, and these trees are made of lists, which are Lisp date structures. This provides a very powerful feature allowing the programmer to write programs that write programs. From a programming language design point of view, LISP was the first to introduce the following concepts:

- Conditionals: such as if-then-else construct.
- Function types: where functions are just like integers and strings
- Recursion: first language to support it.
- Dynamic typing: all variable are pointers.
- Garbage-Collection.
- Programs composed of expressions.
- A symbol type.
- The whole program is a mathematical function

3.5.3 Important Points in LISP

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are case-insensitive, cos 45 or COS 45 are same.
- LISP tries to evaluate everything, including the arguments of a function. Only three types of elements are constants and always return their own value
  - ✓ Numbers
  - ✓ The letter **t,** that stands for logical true.
  - ✓ The value **nil,** that stands for logical false, as well as an empty list.

Naming Conventions in LISP: Name or symbols can consist of any number of alphanumeric characters other than whitespace, open and closing parentheses, double and single quotes, backslash, comma, colon, semicolon and vertical bar. To use these characters in a name, you need to use escape character (\). A name can have digits but not entirely made of digits, because then it would be read as a number. Similarly, a name can have periods, but can't be made entirely of periods.

In LISP, variables are not typed, but data objects are. LISP data types can be categorized as:

- **Scalar types** − for example, number types, characters, symbols etc.
- **Data structures** − for example, lists, vectors, bit-vectors, and strings.

Any variable can take any LISP object as its value, unless you have declared it explicitly. Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions and in method declarations.

The data types are arranged into a hierarchy. A data type is a set of LISP objects and many objects may belong to one such set.

- The **typep** predicate is used for finding whether an object belongs to a specific type.
- The **type-of** function returns the data type of a given object.

Type Specifiers in LISP

Type specifiers are system-defined symbols for data types

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream
character	keyword	readtable	string
[common]	list	sequence	[string-char]
complied-function	long-float	short-float	symbol
complex	nill	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Apart from these system-defined types, you can create your own data types. When a structure type is defined using **defstruct** function, the name of the structure type becomes a valid type symbol.

3.5.4 Features of LISP programming language

There are many features of Lisp programming language which makes it one of the most popular programming language of its time.

- **High level programming language:** Lisp programming language is one of the oldest expression based high level programming language.
- **Multi paradigm:** Lisp programming language supports multi paradigm like imperative, functional, procedural, reflective and meta.
- **Artificial intelligence programming:** Lisp programming language is mainly used for Artificial intelligence (AI) programming and to make manipulation of data string easy.
- **Typing disciplines:** Lisp programming language uses dynamic and strong typing disciplines.
- **Data types:** Lisp programming language also provides different data types such as adjustable arrays, objects, structures, symbols, vectors, lists, hash-tables, etc.
- **Dialects:** The commonly used dialects of Lisp programming language are Arc, Scheme, Clojure, Common Lisp, EuLisp, Hy, MDL, etc.
- **High level debugging:** Lisp programming language also provides the feature of high level debugging.
- **Extensible programming language**: Lisp programming language is an easily extensible programming language.
- **Machine independent:** Lisp programming is a machine independent programming language which means that it can be use by programmer in any machine.

3.5.5 Advantages Lisp programming language

- Lisp programming language is simple and easy to learn programming language.
- Lisp programming language is an easily extensible programming language.
- Lisp programming language is one of the oldest expression based high level programming language.
- Lisp programming language also provides the feature of high level debugging.
- Lisp programming is a machine independent programming language which means that it can be use by programmer in any machine.
- Lisp programming language also provides different data types such as adjustable arrays, objects, structures, symbols, vectors, lists, hash-tables, etc.
- Lisp programming language is mainly used for Artificial intelligence (AI) programming and to make manipulation of data string easy.

- Lisp programming language create easy and Powerful macros.

3.6 PERL Language

Perl is a programming language developed by Larry Wall, especially designed for text processing. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. After a year of development, Larry released the first version of Perl in 1986, alternately calling it the Practical Extraction and Report Language and the Pathologically Eclectic Rubbish Lister. Others call it the Swiss Army Chainsaw of software. There has been a revolution in Perl programming practice as well as its testing culture. CPAN (the Comprehensive Perl Archive Network) continues to grow exponentially, making it the killer feature of Perl. Perl programming language was originally developed as scripting language to make report processing easy in 1987. Larry Wall is known as the father of Perl programming language. He decided to name the programming language "Perl" because he thought it was a memorable and short name. The first version of Perl programming language i.e. version 1.0 was officially released in December 8, 1987 and the latest version of Perl is version 5.24 which was released in May 19, 2016.

The Perl language owes much of its popularity to the fact that it handles text with ease and efficiency. Once you know the language, it's far easier to whip up a quick system-administration utility in Perl than in most other languages. And with Perl's excellent internal resource management, the utilities you write in Perl are far less prone to memory leakage, stack bashing, and other problems common to lower-level languages like C. PERL was created to reduce the short coming on any language. It was developed using a mix of C/C++, Java, Sed, Grep, awk, Shell etc, with an attempt to keep the best features of them all in one language. It is the most useful in developing parser and other fast text processing application. It has significant use in web development, supports OOP but does not stipulate an object storage format. PERL does not support the traditional notion of records and struts instead associative array are provided to serve all such purpose.

Perl is a general purpose, high level interpreted and dynamic programming language. Perl was originally developed for the text processing like extracting the required information from a specified text file and for converting the text file into a different form. Perl supports both the procedural and Object-Oriented programming. Perl is a lot similar to C syntactically and is easy for the users who have knowledge of C**,** C++**.**

3.6.1 Structure of PERL

Like other programming languages, Perl also follows a basic syntax for writing programs for applications and software or writing a simple Perl program. This syntax contains some predefined words known as keywords, variables for storing values, expressions, statements for executing the logic, loops for iterating over a variable value, blocks for grouping statements, subroutines for reducing the complexity of the code, etc. All these, when put together, will make a Perl program. A Perl program whether it be a small code for addition of two numbers or a complex one for executing web scripts, uses these variables, statements and other parameters that comprise of a program’s syntax.

- **Variables:** are user-defined words that are used to hold the values passed to the program which will be used to evaluate the code. Every Perl program contains values on which the code performs its operations. These values cannot be manipulated or stored without the use of a Variable. A value can be processed only if it is stored in a variable, by using the variable's name. A value is the data passed to the program to perform manipulation operation. This data can be either number, strings, characters, lists, etc. Example**:** Above example contains one string variable and two integer variables.

```
Values:
5
geeks
15

Variables:
$a = 5;
$b = "geeks";
$c = 15;
```

- **Expressions:** Expressions in Perl are made up of variables and an operator symbol. These expressions formulate the operation that is to be performed on the data provided in the respective code. An expression in Perl is something that returns a value on evaluating. An expression can also be simply a value with no variables and operator symbol. It can be an integer or just a string with no variable. Example:

```
Value 10 is an expression, $x + $y is an expression that
returns their sum, etc.
```

- **Comments:** Perl developers often make use of the comment system as, without the use of it, things can get real confusing, real fast. Comments are useful information that the developers provide to make the reader understand the source code. It explains the logic or a part of it used in the code. Comments are usually helpful to someone maintaining or enhancing your code when you are no longer around to answer questions about it. These are often cited as a useful programming convention that does not take part in the output of the program but improves the readability of the whole program. There are two types of comment in Perl:

  ✓ **Single line comments:** Perl single line comment starts with hashtag symbol with no white spaces (#) and lasts till the end of the line. If the comment exceeds one line, then put a hashtag on the next line and continue the comment. Perl's single line comments are proved useful for supplying short explanations for variables, function declarations, and expressions. Example:

```
#!/usr/bin/perl
$b = 10;    # Assigning value to $b
$c = 30;    # Assigning value to $c

$a = $b + $c;   # Performing the operation
print "$a";     # Printing the result
```

- ✓ **Multi-line string as a comment:** Perl multi-line comment is a piece of text enclosed within "=" and "=cut". They are useful when the comment text does not fit into one line; therefore, needs to span across lines. Multi-line comments or paragraphs serve as documentation for others reading your code. Perl considers anything written after the '=' sign as a comment until it is accompanied by a '=cut' at the end. Please note that there should be no whitespace after the '=' sign. Example:

```
#!/usr/bin/perl

=Assigning values to
variable $b and $c
=cut
$b = 10;
$c = 30;

=Performing the operation
and printing the result
=cut
$a = $b + $c;
print "$a";
```

- **Statements:** this holds instructions for the compiler to perform operations. These statements perform the operations on the variables and values during the Run-time. every statement in Perl must end with a semicolon(;). Basically, instructions written in the source code for execution are called statements. There are different types of statements in the Perl programming language like Assignment statement, Conditional statement, Looping statements, etc. These all help the user to get the required output. For example, n = 50 is an assignment statement.

  Multi-Line Statements: Statements in Perl can be extended to one or more lines by simply dividing it into parts. Unlike other languages like Python, Perl looks for a semicolon to end the statement. Every line between two semicolons is considered as a single statement.

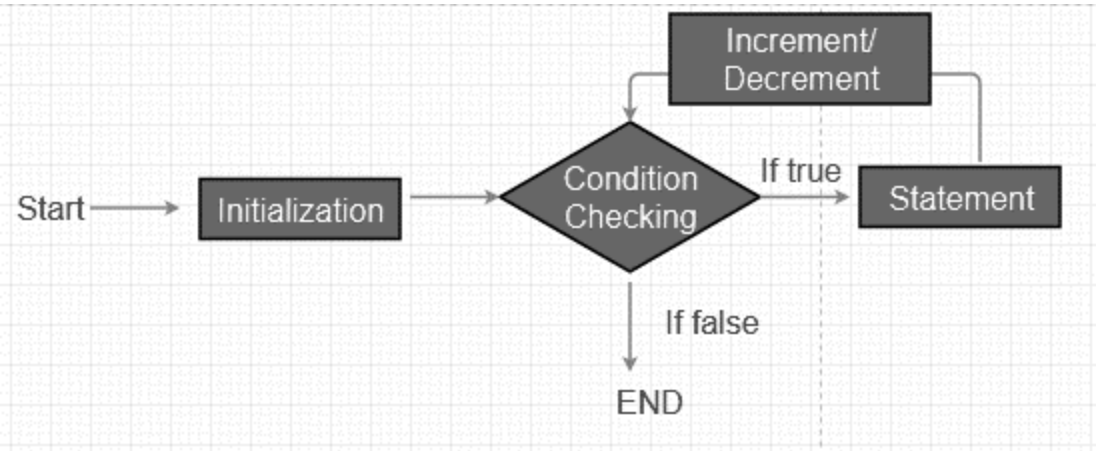
- Block: this is a group of statements that are used to perform a relative operation. In Perl, multiple statements can be executed simultaneously (under a single condition or loop) by using curly-braces ({}). This forms a block of statements which gets executed simultaneously. This block can be used to make the program more optimized by organizing the statements in groups. Variables that are declared inside a block have their scope limited to that specific block and will be of no use outside the block. They will get executed only till that specific block is getting executed. Example:

```
{
        $x = 15;
        $x = $x + 25;
        print($x);
}
```

In the above code, the variable $x will have its scope limited to this particular block only and will be of no use outside the block. Above block holds statements that have their operations related to each other.

- **Functions or Subroutines:** is a block of code written in a program to perform some specific task. We can relate functions in programs to employees in an office in real life for a better understanding of how functions work. Suppose the boss wants his employee to calculate the annual budget. So how will this process complete? The employee will take information about the statics from the boss, performs calculations and calculate the budget and shows the result to his boss. Functions work in a similar manner. They take information as a parameter, execute a block of statements or perform operations on these parameters and returns the result. Perl provides us with two major types of functions:
  - ✓ **Built-in Functions:** Perl provides us with a huge collection of built-in library functions. These functions are already coded and stored in the form of functions. To use those we just need to call them as per our requirement like sin(), cos(), chr(), return(), shift(), etc.
  - ✓ **User Defined Functions:** Apart from the built-in functions, Perl allows us to create our own customized functions called the user-defined functions or Subroutines.

- **Loops:** Like any other language, loop in Perl is used to execute a statement or a block of statements, multiple times until and unless a specific condition is met. This helps the user to save both time and effort of writing the same code multiple times. Perl supports various types of looping techniques:
  - ✓ For loop
  - ✓ For each loop
  - ✓ While loop
  - ✓ Do…. while loop
  - ✓ Until loop
  - ✓ Nested loops



- **Whitespaces and Indentation:** Whitespaces in Perl are the blanks that are used between the variables and operators or between keywords, etc. Perl has no effect of whitespaces unless they are used within quotes. Whitespaces such as spaces, tabs, newlines, etc. have the same meaning in Perl if used outside the quotes. Example:

```
$a = $b + $c;

Here, spaces are of no use,

it will cause no effect even if it is written as

$a = $b       +         $c;
```

Similarly, the process of indentation is used to arrange the code in an organized way to make it easier for the readers. Whenever a block of statements is used then the indentation will help reducing the reading complexity of the code. Example:

```
Using Indentation:
{
    $a = $b + $c;
    print "$a";
}

Without using Indentation:
{
$a = $b + $c;
print "$a";
}
```

In the above example, both of the blocks will work in the exact same way but, for codes which have a large number of statements, the use of indentation makes it more compatible with the readers. Though it is not necessary to use Whitespaces and Indentation in your Perl code, but it is a good practice to do the same.

- **Keywords:** Keywords **or** Reserved words are the words in a language that are used for some internal process or represent some predefined actions. They have a special meaning to the compiler. These words are therefore not allowed to use as variable names or objects. Doing this will result in a compile-time error. In Perl, keywords include built-in functions as well along with the control words. These keywords can sometimes be used as a variable name but that will result in confusion and hence, debugging of such a program will be difficult.
  Example: `One can use $print as a variable along with the keyword print().`

3.6.2 Features of Perl

- **Easy to learn:** Perl programming language easy to use and easy to learn programming language. The concept of developing perl programming languages taken from other programming languages and the syntax of perl programming language are also similar to

other programming languages like awk, sh, sed, C, etc. so it is easy to learn and understand perl programming language.

- **Object oriented programming:** Perl programming language also supports the features of object oriented programming. It follows all the concepts of object oriented programming approach like class, inheritance, abstraction, polymorphism, and encapsulation, etc. Features of object oriented programming makes maintenance and development easy and fast.

- **Procedural programming language:** Perl programming language also supports the feature of procedural programming style. In Procedural programming there are specified steps for each programs to solve a particular problem.

- **An open source:** Perl programming language is an open source programming language which is licensed under Artistic license or GNU General Public Licence. You can easily download Perl programming language from its official website for free. You can modify it according to your requirements and the terms of GNU General Public Licence.

- **Platform independent:** Perl programming language was originally designed for Unix operating system but now you can easily use perl programming language in any platform that is why it is a cross platform or platform independent programming language.
- **Extendable programming language**: All programming languages extendable programming language as it provides thousands of third party modules on CPAN for free. And for standard library, these modules provide various powerful extensions.

- **Interpreted programming language:** Perl programming language is a high level programming language which needs an interpreter to convert scripts written in Perl into machine readable language that is why Perl is an interpreted programming language.

3.7 ALGOL language

ALGOL (ALGOrithmic Language) is one of several high level languages designed specifically for programming scientific computations. It started out in the late 1950's. ALGOL was created in 1959 by a joint meeting in Zűrich,Switzerland by GAMM (Society of Applied Mathematics and Mechanics [German group]) and ACM (Association of Computing Machinery). The conference was one of the first formal attempts to address the issue of software portability lasted 6 days and ended with the creation of ALGOL 58. first formalized in a report titled ALGOL 58.

Originally ALGOL 58 was meant to be used as a universal algorithmic programing language, but the disputes over the implementation and other areas caused another conference to be called in 1960; this time in Paris, France that resulted in ALGOL 60. Since there were still disputes about the language, another branch of ALGOL was created ALGOL 68. But due to the difficulty of understanding its two-level grammar (hyper rules and meta rules i.e. ‘ALPHA:: beta.’), which was meant to be used like templates to enable multiple different types of production, it never caught on.

ALGOL's machine independence permitted the designers to be more creative, but it made implementation much more difficult. Although ALGOL never reached the level of commercial popularity of FORTRAN and COBOL, it is considered the most important language of its era in terms of its influence on later language development. ALGOL's lexical and syntactic structures became so popular that virtually all languages designed since have been referred to as "ALGOL - like"; that is they have been hierarchical in structure with nesting of both environments and control structures.

Like LISP, ALGOL had recursive subprograms—procedures that could invoke themselves to solve a problem by reducing it to a smaller problem of the same kind. ALGOL introduced block structure, in which a program is composed of blocks that might contain both data and instructions and have the same structure as an entire program. Block structure became a powerful tool for building large programs out of small components.

ALGOL contributed a notation for describing the structure of a programming language, Backus–Naur Form, which in some variation became the standard tool for stating the syntax (grammar) of programming languages. ALGOL was widely used in Europe, and for many years it remained the language in which computer algorithms were published. Many important languages, such as Pascal, are its descendants.

3.7.1 ALGOL Structure Program

Notable language Elements of ALGOL

- Bold symbols and reserved words: There are 61 such reserved words (some with "brief symbol" equivalents) in the standard sub-language:
  - ✓ mode, op, prio, proc,
  - ✓ flex, heap, loc, long, ref, short,
  - ✓ bits, bool, bytes, char, compl, int, real, sema, string, void,
  - ✓ channel, file, format, struct, union,
  - ✓ of, at "@", is ":=:", isnt ":/=:", ":≠:", true, false, empty, nil "○", skip "~", co comment "¢", pr, pragmat.
  - ✓ case in ouse *in* out esac "(~ | ~ |: ~ | ~ | ~)",
  - ✓ for from to by while do od,
  - ✓ if then elif *then* else fi "(~ | ~ |: ~ | ~ | ~)",
  - ✓ par begin end "(~)", go *to*, goto, exit ".".

- **Mode: Declarations:** The basic data types (called modes in ALGOL 68 parlance) are real, int, compl (complex number), bool, char, bits and bytes. For example:
  - ✓ int n = 2;
  - ✓ co n is a fixed constant of 2.co
  - ✓ int m := 3;
  - ✓ co m is a newly created local variable whose value is initially set to 3. This is short for ref int m = loc int := 3; co
  - ✓ real avogadro = 6.0221415□23; co Avogadro's number co
  - ✓ long long real pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399
  - ✓ 37510;

- ✓ compl square root of minus one = 0 ⊥ 1

However, the declaration real x; is just syntactic sugar for ref real x = loc real;. That is, x is really the *constant identifier* for a *reference to* a newly generated local real variable.

- **Special characters**

Most of Algol's "special" characters (×, ÷, ≤, ≥, ≠, ¬, ∨, ∧, □, →, ↓, ↑, ⌷, ⌊, ⌈, ⌊, ⌈, ○, ⊥ and ¢) can be found on the IBM 2741 keyboard with the APL "golf-ball" print head inserted, these became available in the mid 1960s while ALGOL 68 was being drafted.
These characters are also part of the unicode standard and most of them are available in several popular fonts.

- ✓ if then elif *then* else fi "(~ | ~ |: ~ | ~ | ~)",
- ✓ par begin end "(~)", go *to*, goto, exit ".".
- ✓ Transput: Input and output
- ✓
- ○ Transput is the term used to refer to ALGOL 68's input and output facilities. There are pre-defined procedures for unformatted, formatted and binary transput. Files and other transput devices are handled in a consistent and machine-independent manner.

The following example prints out some unformatted output to the standard output device:

```
print ((newpage, "Title", newline, "Value of i is ",
i, "and x[i] is ", x[i], newline))
```

Note the predefined procedures newpage and newline passed as arguments.

3.72 Features of ALGOL

- Algol was the first language that was designed to be machine independent. It was also the first language whose syntax was formally described.

- ALGOL 68 was the source of several new ideas in language design, some of which were subsequently adopted by other languages.

- Orthogonality: resulted in several innovative features of ALGOL 68 which is inclusion of user defined data types to provide a few primitive types and structures and allow the user to combine those primitives into a large number of different structures.

- User-defined data types are valuable because they allow the user to design data abstractions that fit particular problems very closely.

- ALGOL 68 introduced the kind of dynamic arrays that will be termed implicit heap-dynamic. A dynamic array is one in which the declaration does not specify subscript bounds.

- ALGOL 68 achieved writability by the principle of orthogonality: a few primitive concepts and the unrestricted use of a few combining mechanisms.

- ALGOL 68 was targeted to a single class: scientific applications.

3.8 PROLOG Language

Programming in logic programming languages is nonprocedural. Programs in such languages do not state exactly how a result is to be computed but rather describe the necessary form and/or characteristics of the result. What is needed to provide this capability in logic programming languages is a concise means of supplying the computer with both the relevant information and an inferencing process for computing desired results. Predicate calculus supplies the basic form of communication to the computer, and the proof method, named resolution, developed first by Robinson (1965), supplies the inferencing technique.

During the early 1970s, Alain Colmerauer and Phillippe Roussel in the Artificial Intelligence Group at the University of Aix-Marseille, together with Robert Kowalski of the Department of Artificial Intelligence at the University of Edinburgh, developed the fundamental design of Prolog. The primary components of Prolog are a method for specifying predicate calculus propositions and an implementation of a restricted form of resolution. Both predicate calculus and resolution are described in Chapter 16. The first Prolog interpreter was developed at Marseille in 1972. The version of the language that was implemented is described in Roussel (1975). The name Prolog is from programming logic.

3.8.1 PROLOG Structure Program

Prolog programs consist of collections of statements. Prolog has only a few kinds of statements, but they can be complex. One common use of Prolog is as a kind of intelligent database. This application provides a simple framework for discussing the Prolog language. The database of a Prolog program consists of two kinds of statements: facts and rules. The following are examples of fact statements:

```
mother (joanne, jake).
father(vern, joanne).
```

These state that `joanne is the mother of jake, and vern is the father of joanne.`

An example of a rule statement is

```
Grandparent (X, Z) :- parent(X, Y), parent(Y, Z).
```

This states that it can be deduced that X is the grandparent of Z if it is true that X is the parent of Y and Y is the parent of Z, for some specific values for the variables X, Y, and Z.

The Prolog database can be interactively queried with goal statements, an example of which is:

```
father(bob, darcie).
```

This asks if `bob is the father of darcie.` When such a query, or goal, is presented to the Prolog system, it uses its resolution process to attempt to determine the truth of the statement. If it can conclude that the goal is true, it displays "true." If it cannot prove it, it displays "false."

General structure of Prolog programs can be summarized by the following-four statement types:

- Facts are declared describing object and their relationships.
- Rules governing the objects and their relationships are then defined.
- Questions are then asked about the objects and their relationships.
- Commands are available at any time for system operations.

3.8.2 Features of PROLOG

- **Untyped:** Prolog programming language is an untyped fourth generation programming language.
- **Paradigm:** Prolog programming language uses declarative and logic programming paradigm.
- **Open source:** Prolog programming language is a free and open source programming language which means anyone can easily download and install Prolog programming language from its official website.
- **Applications:** Prolog programming language is mainly used in Machine learning, Robot Planning, Expert System, Problem Solving, Automated reasoning, etc.
- **Database type:** Prolog is a traditional programming language which includes single data type.
- **Artificial intelligence:** Prolog programming language is mainly related with computational linguistics and artificial intelligence.

4.0 Conclusion

The first functional programming language was invented to provide language features for list processing, the need for which grew out of the first applications in the area of artificial intelligence (AI For practical reasons, many functional languages extend the lambda calculus with additional features, including assignment, I/O, and iteration. Lisp dialects, moreover, are homoiconic: programs look like ordinary data structures, and can be created, modified, and executed on the fly. Lists feature prominently in most functional programs, largely because they can easily be built incrementally, without the need to allocate and then modify state as separate operations. Many functional languages provide other structured data types as well.

Like imperative and functional programming, logic programming is related to constructive proofs. But where an imperative or functional program in some sense is a proof (of the ability to generate

outputs from inputs), a logic program is a set of axioms from which the computer attempts to construct a proof. And where imperative and functional programming provide the full power of Turing machines and lambda calculus, respectively (ignoring hardware-imposed limits on arithmetic precision, disk and memory space, etc.), Prolog provides less than the full generality of resolution theorem proving, in the interests of time and space efficiency. At the same time, Prolog extends its formal counterpart with true arithmetic, I/O, imperative control flow, and higher-order predicates for self-inspection and modification.

Object-oriented programming is characterized in terms of three fundamental concepts: encapsulation, inheritance, and dynamic method binding. Encapsulation allows the implementation details of an abstraction to be hidden behind a simple interface. Inheritance allows a new abstraction to be defined as an extension or refinement of some existing abstraction, obtaining some or all of its characteristics automatically. Dynamic method binding allows the new abstraction to display its new behavior even when used in a context that expects the old abstraction. Different programming languages support these fundamental concepts to different degrees. an object-oriented language should present a uniform object model of computing, in which every data type is a class, every variable is a reference to an object, and every subroutine is an object method. Moreover, objects should be thought of in anthropomorphic terms: as active entities responsible for all computation.

5.0 Summary

Investigation and development environments of a number of programming languages had been reviewed. This gives the students a good perspective on current issues in language design and the stage for an in-depth discussion of the important features of contemporary languages had been set up as well.

6.0 Tutor-Marked Assignment

1. Where was LISP developed? By whom?
2. What dialect of LISP is used for introductory programming courses at some universities?
3. Why does C++ include the features of C that are known to be unsafe?
4. From what language does Objective-C borrow its syntax for method calls?
5. What was the first application for Java?
6. What characteristic of Java is most evident in JavaScript?
7. What array structure is included in C# but not in C, C++, or Java?
8. What two languages was the original version of Perl meant to replace?
9. What data structure does Python use in place of arrays?
10. What is the primary platform on which C# is used?
12. LISP began as a pure functional language but gradually acquired more and more imperative features. Why?
13. Describe in detail the three most important reasons, in your opinion,
14. Why ALGOL 60 did not become a very widely used language.
15. Are there any logic programming languages other than Prolog
16. Write a prolog program that finds the maximum of a list of numbers
17. Write a Prolog program that implement quicksort

7.0 References/Further Reading

1. Modern Programming Language, Lecture 18-24: LISP Programming
2. LISP - Data Types (tutorialspoint.com)
3. Introduction to LISP, CS 2740, Knowledge Representation Lecture 2 by Milos Hauskrecht, 5329 Sennott Square
4. The Evolution of Lisp: ACM Hisotry of Programming Languages Gabriel and Steele's (FTP).
5. Common LISP: A Gentle Introduction to Sybbolic Computattion by Davis S. Touretzky
6. History of Python Programming Language - Python Pool
7. History of Python - GeeksforGeeks by Sohom Pramanick
8. Perl | Basic Syntax of a Perl Program - GeeksforGeeks
9. Perl Programming Language: history, features, applications, Why Learn? - Answersjet
10. History of Python (tutorialspoint.com), History Of Python Programming Language – Journal Dev,,,
11. A brief history of the Python programming language | by Dan Root | Python in Plain English
12. History and Development of Python Programming Language (analyticsinsight.net)
13. A brief history of Python Programming Language (studysection.com)
14. ALGOL | computer language | Britannica
15. The ALGOL Programming Language (umich.edu)
16. Programming History: The Influence of Algol on Modern Programming Languages (Part 1) | by Mike McMillan | Better Programming
17. History Of Algol 68 | programming language (wordpress.com)
18. https://www.javapont.com classification –of-programming-languages
19. https//www.w3schools.in/category/java-tutorial
20. https://www.totorialspoint.com/difference-between-hihg-level-language-and-low-level-language.
21. https://byjus.com/gate/difference-between-high-level-and-low-level-languages/
22. What is a Low-Level Language? - Definition from Techopedia
23. https://www.tutorialandexample.com/input-devies-of-computer/
24. Java Syntax - A Complete Guide to Master Java - DataFlair (data-flair.training), https://datra-flair.training/blog/basic-jaja-syntax/
25. C++ Language Tutorial by Juan Soulie, http://www. cplusplus.com./doc/tutorial/

UNIT 3 LANGUAGE EVALUATION AND COMPARISON

CONTENTS

1.0 INTRODUCTION

The study of programming language concepts builds an appreciation for valuable language features and encourages programmers to use them. The fact that many features of languages can be simulated in other languages does not lessen significantly the importance of designing languages with the best collection of features. Each programming language contains a unique set of keywords and syntax, which are used to create a set of instructions. Thousands of programming languages have been developed till now, but each language has its specific purpose.

2.0 OBJECTIVES

- To evaluate the strengths and weaknesses of different programming languages
- To know their tools, libraries and support behind those languages.
- In this programming language comparison, we provide three direct face-offs between popular object-oriented, open source and concurrent programming languages

3.0 MAIN CONTENT

3.1 Criteria for a Good Programming language

- Its definition should be independent of any particular hardware/OS
- Its definition should ne standardized and compile implementation should comply with this standard.
- It should support software engineering technology
- It should discourage / prohibiting poor practices and promoting or support maintenance activities.
- It should effectively support the application domains of interest.
- It should support the requirements level of system reliability and safety.
- Its compiler implementation should be commensurate with the current state of technology.
- Appropriate software engineering based support tools and environment should considered.

3.2 Language Evaluation

3.2.1 C++

- C++ is completely independent of any particular hardware or software
- C++ has followed the standardization path of most languages.
- The structure and OO features of C++ provide support for the concept of encapsulation and data abstraction unlike C.
- Reliability is supported by OO features f C++.
- C++ compiler continues t improve suing current technology because of its popularity.
- A wide variety of supporting tools and environment id available for C++ development because of its popularity.
- C++ is used to create OO codes and the programmer has good OO features to facilitate maintainability.
- C++ readily use object file produced by any language compiler which composed an application.
- C++ improves with better language constitute for facilitating language interface.
- C++ improved considerably on the language characteristics of C for supporting reliability with feature such as encapsulation and improved expression.

3.2.2 C#

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a part of .Net Framework.

3.2.3 Java

- It is a completely independent of any particular language / OS.
- Java had a usual road standardization as with most language.
- Java was developed because C could not meet the support of software engineering technology.
- It is a general purpose language which support WWW application ad provide good support for any domain in which it has been tired.
- It is very sophisticated for a new language driving some current technology trends and commensurate with current technology.
- Current Java tool kits contain primarily tools to support code creation.
- Mainly features of Java support maintainability such as those which support code clarity, encapsulation and object oriented.
- Java provide interfacing with other languages by providing wrappers around the code from the other languages.

- Java requires the specification of information, the omission that can make program unreliable such as type specification.

3.2.4 Python

- Python supports both the programming methods that are functional and structured, as well as object-oriented programming.
- Python can also be used as a scripting language or compiled to byte-code for the development of huge applications.
- It allows dynamic type verification and provides very high-level dynamic data types.
- It supports garbage collection automatically.
- It integrates seamlessly with C, C++, COM, ActiveX, CORBA, and Java.

3.2.5 LISP

- LISP works on pretty much on all OS.
- It is somewhat lacking when compared to other languages in aspect o standardized and compiler implementation.
- Common LISP had CLOS which support OOP. It is extremely old language, easily extensible due to its powerful code and data philosophy.
- Though, it provides good tools for software engineering base support tool and environment, it is still not up to par with the other languages.
- It is mostly functional language, not as good as and exceptional due to its flexibility nature and clean code.
- It provides for interface to other languages.

3.2.6 PERL

- It is completely independent of any particular hardware / software.
- It has fixed standard.
- It was developed to make development simple and quick, prohibit poor practice but not unduly used for software engineering.
- It was developed for fast text processing, for server side scripting and freely available.
- Software engineering based support tools are being studied in PERL.
- It supports OO model and structured programming approach.
- It bears resemblance to C++ but not as strict as C++ in its syntax.
- Mixed language support in PERL does not provide wrapper as Java for other languages.
- It acts as a glue and used to call upon various different program from the system command line. It is flexible in its declaration of variables and can be unreliable in certain cases.

3.2.7 ALGOL

- Structural equivalence. Automatic type conversion, including de-referencing
- Flexible arrays
- Generalized loops (for-from-by-to-while-do-od), if-then-elif-fi
- Integer case statement with 'out' clause, skip statement, goto

- Blocks, procedures and user-defined operators
- Procedure parameters
- Concurrent execution (cobegin/coend) and semaphores
- Generators heap and loc for dynamic allocation
- No abstract data types, no separate compilation.

3.2.8 PROLOG

- Prolog programming language is a declarative programming language that uses logic programming paradigm and fully object oriented.
- It is related with artificial intelligence and computational linguistics.
- Prolog programming language is an open source programming language.
- Prolog programming language is a fourth generation programming language
- It supports directing linkage with C/C++ and Win32 API functions.
- Prolog is algebraic data type, patter matching and unification
- It is automatic memory management and supports parametric polymorphism

3.3 Risk Encountered if Criteria is Violated

- If a language is not independent of a particular platform, portability is compromised.
- Hardware and software option are limited for the original system and future upgrade. If compiler implementation does not comply with a standard language definition poor code characteristics will make testing and maintenance a nightmare
- Poor support for application domain compromise the case of development.
- If reliability is compromise system will perform below expectation costly and also life and property will be endangering due to safety problem.
- Out of the date compiler produced substandard of code that is difficult to use and maintain which can also prohibit the use of key language feature.
- Lack of appropriate automated development support compromise developer productivity and system quality.

3.4 Language Comparison

- The language should support effective reuse of program unit
- Language should not hinder but help to write a portae code
- Language should be designed in a way that programming error can be detected and eliminated as quickly as possible.
- Language should be capable of being implemented efficiently.
- Language should not hinder but help good programming practice.
- There should be a good quality compilation available for language as well as good quality IDE.
- Available programmer should be familiar with the language and there should e a quality training to justify itself in future project.
- Ability for a language` to express complex computation / complex data structure in appealing and intuitive ways.

Comparison between C/C++, Java and Python

C++	Java	Python
Complied programming language	Complied programming language	Interpreted programming language
Support operator overloading	Does not support operator overloading	Support operator overloading
Provide both single and multiple inheritance	Provide partial multiple inheritance using interfaces	Provide both single and multiple inheritance
Platform dependent	Platform independent	Platform dependent
Does not support threads	Has in build multithreading	Supports multithreading
Has limited number of library support	Has library support for many concepts like UI	Has a huge set of libraries that make it fir for AI, data science etc.
Code length is a bit lesser, 1.5 less than that of Java	Java has huge code	Smaller code
Function and variables are used outside the class	Every bit of code is inside a class	Function and variable can be declared and used outside the class also.
C++ program is a fast compiling programming language	Java program compiler a bit slower than C++	Due to the use of interpreter execution is slower
Strictly uses syntax norms	Strictly uses syntax norms	Use of ; is not compulsory
Like ; and {}	Like punctuations ,;.	

Comparison between LISP and PROLOG

Second oldest high level programming language after FORTRAN that has changed a great deal since its early days	Logic programming language associated with artificial intelligence and computational linguistics
Supports functional, procedural, reflective and meta paradigms	Supports logical programming paradigm
John McCarthy is the designer of LISP	Alain Colmerauer and Robert Kowalski are the designer of PROLOG
First appear in 1958	First appear in 1972

SELF ASSESSMENT EXERCISE

1. Compare PERL and ALGOL in your own opinion

4.0 Conclusion

All these programming languages support one paradigm or the others for examples C/C++, C#, Java, Algol and Perl are all procedural languages but also support object oriented language, LISP and Python are applicative languages also object oriented languages, LISP and Prolog are popular programming languages that help to develop AI-based applications. PROLOG and PERL are ruled based language and also include object oriented concept.

5.0 Summary

Programming languages continue to evolve in both industry and research, as systems and applications change. Today there is a wide variety of programming languages with different languages, syntax, and features. Developers can now use a language based on either the client's preference or their own.

6.0 Tutor-Marked Assignment

1. Make an educated guess as to the most common syntax error in Lisp programs.
2. Describe in detail the three most important reasons, in your opinion, why ALGOL 60 did not become a very used language.
3. Do you think language design committee is a good idea? Support your opinion.
4. Evaluate some other programming languages you know and itemized risk encountered if those criteria were violated?

7.0 References/Further Reading

1. History of Python (tutorialspoint.com), History Of Python Programming Language – Journal Dev,,,
2. A brief history of the Python programming language | by Dan Root | Python in Plain English
3. History and Development of Python Programming Language (analyticsinsight.net)
4. A brief history of Python Programming Language (studysection.com)
5. ALGOL | computer language | Britannica
6. The ALGOL Programming Language (umich.edu)
7. Programming History: The Influence of Algol on Modern Programming Languages (Part 1) | by Mike McMillan | Better Programming
8. History Of Algol 68 | programming language (wordpress.com)
9. https://www.javapont.com classification –of-programming-languages
10. https//www.w3schools.in/category/java-tutorial
11. https://www.totorialspoint.com/difference-between-hihg-level-language-and-low-level-language.
12. https://byjus.com/gate/difference-between-high-level-and-low-level-languages/
13. What is a Low-Level Language? - Definition from Techopedia
14. https://www.tutorialandexample.com/input-devies-of-computer/
15. Java Syntax - A Complete Guide to Master Java - DataFlair (data-flair.training), https://datra-flair.training/blog/basic-jaja-syntax/
16. C++ Language Tutorial by Juan Soulie, http://www. cplusplus.com./doc/tutorial/
17. Modern Programming Language, Lecture 18-24: LISP Programming
18. LISP - Data Types (tutorialspoint.com)
19. Introduction to LISP, CS 2740, Knowledge Representation Lecture 2 by Milos Hauskrecht, 5329 Sennott Square
20. The Evolution of Lisp: ACM Hisotry of Programming Languages Gabriel and Steele's (FTP).
21. Common LISP: A Gentle Introduction to Sybbolic Computattion by Davis S. Touretzky
22. History of Python Programming Language - Python Pool
23. History of Python - GeeksforGeeks by Sohom Pramanick

24. Perl | Basic Syntax of a Perl Program - GeeksforGeeks
25. Perl Programming Language: history, features, applications, Why Learn? - Answersjet