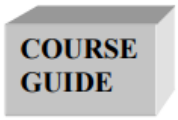
NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY~~SCHOOL~~ OF SCIENCES ~~AND TECHNOLOGY~~

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT401

COURSE TITLE: ORGANIZATION OF PROGRAMMING LANGUAGES



CIT401

ORGANIZATION OF PROGRAMMING LANGUAGES

Course Team: Dr OGUNBANWO, Afolakemi Simbo (Developer/Writer)

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

~~National Open University of Nigeria~~
~~Headquarters~~
~~14/16 Ahmadu Bello Way~~
~~Victoria Island~~
~~Lagos~~

~~Abuja Office~~
~~No. 5 Dar es Salaam Street~~
~~Off Aminu Kano Crescent~~
~~Wuse II, Abuja Nigeria~~

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

Formatted: Font: 14 pt

First Printed 20221

ISBN:

CONTENTS PAGE

Introduction:

Organization of Programming Languages is a course on the fundamental principles of programming languages, introduction to fundamental principles and techniques in programming languages design and implementation. It handles the programing paradigm and historical pattern of programming. The course elaborates on language structure, data type and data structure.

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 Objectives:

The general objective of the course is to:

- 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

CIT 401– Organization of Programming Languages is a three (3) unit course. It deals with Language definition structure. Data types and structures, Review of basic data types, including lists and tress, control structure and data flow, Run-time consideration, interpretative languages, lexical analysis and parsing.

This Course Guide gives you a brief overview of the course content, course duration, and course materials.

What You Will Learn in this Course

Course Aims

i. Introduce the concepts of programming language in preparation for the main course;

ii. to discuss structural layer of programming language and formal methods of describing syntax;

iii. Introduce lexical analysis, parsing and language processing; and

iv. explain data type and structure.

v. Identify the common error of runtime

Course Objectives

Certain objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course of the study, you will need to confirm, at the end of each unit, if you have met the objectives set at the beginning of each unit. By the end of this course you should be able to:

-

Working through this Course

In order to have a thorough understanding of the course units, you will need to read and understand the contents and be committed to learning and implementing your knowledge.

This course is designed to cover approximately sixteen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

Course Materials

These include:

1. Course Guide

2. Study Units

3. Recommended Texts

4. A file for your assignments and for records to monitor your progress.

Study Units

There are ten (10) study units in this course:

ORGANIZATION OF PROGRAMMING LANGUAGES: (3 Units)

Module 1	**Introduction to Programming Language**
Unit 1	Introduction to Programming Language
Unit 2	Programming Languages Evolution and Paradigms
Unit 3	Structure and unstructured Programming Language
Module 2	**Language Structure**
Unit 1	Language Structure
Unit 2	Syntax and Semantics
Unit 3	Lexical Analysis and Parsing
Unit 4	Language processing
Module 3	**Structuring Data**
Unit 1	Data Types and Data Structure
Unit 2	Control Structure and Data Flow
Unit 3	Run-time Consideration

Make use of the course materials, do the exercises to enhance your learning.

Textbooks and References

Chen, Y. (2020). Chapter 1 Basic Principles of Programming Languages. In *Introduction to Programming Languages* (Sixth, pp. 1–40). Kendal Hunt Plublishing

John C. Mitchell (2003). *Concepts in Programming Languages***.** Cambridge University Press © 2003 (529 pages)**.** ISBN:0521780985

Sebesta, R. W. (2016). Concepts of Programming Languages (Eleventh Edition). Pearson Education Limited.

Sebesta, R. W. (2009). Concepts of Programming Languages (Tenth Edition). Pearson Education Limited.

Jaemin Hong and Sukyoung Ryu (2010) *Introduction to Programming Languages*

Ghezzi & Jazayeri (1996.) *Programming language concepts—Third edition* John Wiley & Sons New York Chichester Brisbane Toronto Singapore 1996.

Gabbriell M. & Martini S. (2010). *Programming Languages: Principles and Paradigms*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-914-5_1, © Springer-Verlag London Limited 2010

Archana M. *Principles of Programming Languages*

https://www.integralist.co.uk/posts/data-types-and-data-structures/

https://www.geeksforgeeks.org/

https://www.sctevtservices.nic.in/docs/website/pdf/140338.pdf

https://www.scribd.com/document/70893872

http://www.tutorialsspace.com/Programming-Languages

https://www.geeksforgeeks.org/the-evolution-of-programming-languages

https://blog.stackpath.com/runtime/

http://net-informations.com/python/iq/checking.htm

Assignments File

These are of two types: the self-assessment exercises and the Tutor-Marked Assignments. The self-assessment exercises will enable you monitor your performance by yourself, while the Tutor-Marked Assignment is a supervised assignment. The assignments take a certain percentage of your total score in this course. The Tutor-Marked Assignments will be assessed by your tutor within a specified period.

The examination at the end of this course will aim at determining the level of mastery of the subject matter. This course includes twelve Tutor-Marked Assignments and each must be done and submitted accordingly. Your best scores however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

Presentation Schedule

The *Presentation Schedule* included in your course materials gives you the important dates for the completion of tutor marked assignments and attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

Assessment

There are two aspects to the assessment of the course. First are the tutor marked assignments; second, is a written examination. In tackling the assignments, you are expected to apply information and knowledge acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark. At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

Tutor-Marked Assignment

There are twelve tutor-marked assignments in this course. You need to submit all the assignments. The total marks for the best four (4) assignments will be 30% 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 it together with form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If, however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

Examination and Grading

The final examination for the course will carry 70% 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 leaming on you.

The course units are similarly designed with the introduction following the table of 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

This table shows how the actual course marking is broken down.

Assessment Marks

Assignment 1- 4 Four assignments, best three marks of the four count at 30% of course marks Final Examination 70% of overall course marks Total 100% of course marks

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 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*).

Facilitators/Tutors and Tutorials

There are 15 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.

Summary

The course presented the fundamental of programing language, evolution of programming language, programming paradigm, language structure, syntax and semantics, lexical analysis, and language processing. Also, the course intimates the leaner with data type and data structure, control structure and data flow as well as run-time consideration. Upon the completing this course, the learner will be equipped with the ability to know popular programming languages and the advantage they have over each other. Also to the appropriate programming language to use.

I wish you success with the course and hope that you will find it both interesting and useful.

Module 1: Concept of Programming Language

Introduction of Module

Unit 1 Introduction to Programming Language

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Introduction to Programming
   - 3.2. Classification of Programming Language
     - 3.2.1. Machine Language
     - 3.2.2. Assembly Language
     - 3.2.3. High Level Language
   - 3.3. Attribute of good Programming Language
     - 3.3.1. Clarity, Simplicity and Unity
     - 3.3.2. Orthogonality
     - 3.3.3. Support for Abstraction
     - 3.3.4. Programming Environment
     - 3.3.5. Ease of Program Verification/Reusability
     - 3.3.6. Portability of Programs
   - 3.4. Program Performance and Features of Programming Language
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 1 Introduction to Programming Language

1 Introduction

This unit will introduce the leaner to fundamental of programing language which serve as the foundation. Several definitions of programming language are stated. The three categories of programming language namely machine, assembly and high level languages will be briefly discussed. Also, the unit exposes the student to the attribute of good programming language as well as program performance and features of programming language.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Define Programming language
- Explain machine language, assembly language and high level language
- Discuss attribute of good programming language
- Explain performance and features of programming language

3 Main Content

3.1 Introduction to Programming language

A programming language is a language designed to communicate instructions to a machine particularly a computer. Programming languages are used to create programs that control the behavior of a machine or to express algorithms precisely. A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express all possible algorithms. Thus, Programming language is a set of commands, strings of characters readable by programmers but easily translatable to machine code; it has syntax, grammar, and semantics.

- Syntax is a set of rules that define how the commands have to be arranged to make sense and to be correctly translatable to the machine code.
- Grammar is a set of rules of using different punctuation, quotation marks, semicolons, and other symbols to divide and clarify the syntax of a particular language.
- The last component of programming language is semantics, a set of meanings assigned to every command of the language and is used to properly translate the programme to machine code.

The programmer communicates with a machine using programming languages. Most of the programs have a highly structured set of rules.

3.2 Classification of Programming language

Programming Language can be grouped into three namely; Machine Languages, Assembly Languages and High level Languages.

3.2.1 Machine Language:

Machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding. Machine level language is a language that supports the machine side of the programming or does not provide human side of the programming. It consists of (binary) zeros and ones. Each instruction in a program is represented by a numeric code, and numerical addresses are used throughout the program to refer to memory locations in the computer's memory. Microcode allows for the expression of some of the more powerful machine level instructions in terms of a set of basic machine instructions.

3.2.2 Assembly language:

Assembly language is easier to use than machine language. An assembler is useful for detecting programming errors. Programmers do not have the absolute address of data items. Assembly language encourage modular programming.

3.2.3 High level language

High level language is a language that supports the human and the application sides of the programming. A language is a machine independent way to specify the sequence of operations necessary to accomplish a task. A line in a high level language can execute powerful operations, and correspond to tens, or hundreds, of instructions at the machine level. Consequently more programming is now done in high level languages. Examples of high level languages are BASIC, FORTRAN etc

3.3 Attribute of good Programming Language

There are various factors, why the programmers prefer one language over the another. And some of very good characteristics of a good programming language are,

3.3.1 Clarity, Simplicity and Unity:

A Programming language provides both a conceptual framework for Algorithm planning and means of expressing them. It should provide a clear, simple and unified set of concepts that can be used as primitives in developing algorithms. It should have the following; Minimum number of different concepts, Rules for their combination, Simple and regular. This attribute is called conceptual integrity.

3.3.2 Orthogonality:

It is one of the most important feature of PL orthogonality is the property that means " Changing A does not change B". If I take Real world example of an orthogonal system Would be a radio, where changing the station does not change the volume and vice versa. When the features of a language are orthogonal, language is easier to learn and programs are easier to write because only few exceptions and special cases to be remembered.

3.3.3 Support for Abstraction:

There is always found that a substantial gap remaining between the abstract data structure and operations that characterize the solution to a problem and their particular data structure and operations built into a language.

3.3.4 Programming Environment:

An appropriate programming environment adds an extra utility and make language to be implemented easily such as; 1) The availability of- Reliable- Efficient - Well documentation. 2) Speeding up creation and testing by-special Editors- testing packages.

3.3.5 Ease of program verification/Reusability:

The reusability of program written in a language is always a central concern. A program is checked by various testing technique like Formal verification method Desk checking Input output test checking. We verify the program by many more techniques. A language that makes program verification difficult maybe far more troublesome to use. Simplicity of semantic and syntactic structure is a primary aspect that tends to simplify program verification.

3.3.6 Portability of programs:

Programming language should be portable means it should be easy to transfer a program from which they are developed to the other computer. A program whose definition is independent of

features of a Particular machine forms can only support Portability. Example: Ada, FORTRAN, C, c++, Java.

3.4 Program performance and features of programming languages

A programming language's features 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. While the performance of a program, including reliability, readability, writability, reusability, and efficiency, is largely determined 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. Figure 1 shows the influence of a language's features on the performance of a program written in that language.

The figure 1 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.

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

Figure 1: Impact of language features on the performance of the programs

4 Self-Assessment Exercises

- List and explain the three categories of programming language
- Discuss the attribute of a good programming language
- State five features of program
- List five program performance

5 Conclusion

In this unit, you have been introduced to the fundamental of programming language. 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.

6 Summary

Unit 2 Programming Language Evolution and Paradigms

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
    3.1. Programming Language Evolution
        3.1.1. 1883 – The Beginning
        3.1.2. 1949 – Assembly Language
        3.1.3. 1957 – FORTRAN
        3.1.4. 1958 – ALGOL
        3.1.5. 1959 – COBOL
        3.1.6. 1964 – BASIC
        3.1.7. 1970 – Pascal
        3.1.8. 1972 – C
        3.1.9. Other Popular Programming Language
    3.2. Programming Language Paradigms
        3.2.1. Categories of Programming Paradigm
        3.2.2. Overview of Main Programming Paradigm
            3.2.2.1. Imperative Paradigm
            3.2.2.2. Object-Oriented Paradigm
            3.2.2.3. Functional (Application) Paradigm
            3.2.2.4. Logic Paradigm
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 2 Programming Languages Evolution and Paradigms

1 Introduction

Programming Language is indeed an essential part of today's tech world. There are lots of programming languages which have their own syntax, sematic and features. This unit presents the evolution of programming language. And also deliberates on programming paradigm.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Have historical knowledge of programming language
- Discuss the programming paradigm
- Explain different categories of programming paradigm

3 Main Content

3.1 Programming Language Evolution

3.1.1 1883: The Beginning …!!

In the early days, Charles Babbage had made the device, but he was confused about how to give instructions to the machine, and then Ada Lovelace wrote the instructions for the analytical engine. The device was made by Charles Babbage and the code was written by Ada Lovelace for computing Bernoulli's number. First time in history that the capability of computer devices was judged.

3.1.2 1949: Assembly Language

It is a type of low-level language. It mainly consists of instructions (kind of symbols) that only machines could understand. In today's time also assembly language is used in real-time programs such as simulation flight navigation systems and medical equipment eg – Fly-by-wire (FBW) systems. It is also used to create computer viruses.

3.1.3 1957: FORTRAN

Developers are John Backus and IBM. It was designed for numeric computation and scientific computing. Software for NASA probes voyager-1 (space probe) and voyager-2 (space probe) was originally written in FORTRAN 5. In year "1954", Fortran (FORmula TRANslator) is Created in 1954 by John Backus. It is First high level language. It was developed by using the first compiler ever developed It is Machine Independent Language. In 1958 FORTRAN 2nd version introduces

subroutines, functions, loops and primitive for loop. It started as a Project later renamed ALGOL58. The theoretical definition of the language is published and no compiler is required in this.

3.1.4 **1958: ALGOL**

ALGOL stands for ALGOrithmic Language. The initial phase of the most popular programming languages of C, C++, and JAVA. It was also the first language implementing the nested function and has a simple syntax than FORTRAN. The first programming language to have a code block like “begin” that indicates that your program has started and “end” means you have ended your code. ALGOL(ALGOrithmic Language) was released in "1960"and its major releases were in "1960" and "1968". It Was first "Block Structured Language." It was Considered to be the first second generation Computer Language and Machine Independent language. It introduced concepts like: Block structure code(Marked by BEGIN and END), Scope of variables(Scope of local variables inside blocks), BNF (Backus Naur Form), Notation for defining syntax, Dynamic Arrays, Reserved words and IF THEN ELSE, FOR, WHILE loops

3.1.5 **1959: COBOL**

It stands for COmmon Business-Oriented Language. In 1997, 80% of the world’s business ran on Cobol. The US internal revenue service scrambled its path to COBOL-based IMF (individual master file) in order to pay the tens of millions of payments mandated by the coronavirus aid, relief, and economic security. COBOL(Common Business Oriented Language). It was rated in May 1959 by the ShortRange committee of the us department of DOD. The CODASYL(COnference on Data SYstems Languages) Worked from May 1959 to April 1960. ANSI standard included(cobol-68(1968), Cobol-74(1974),COBOL-85(1985), and COBOL-2002(2002) Object Oriented version of COBOL is introduced in 1997 i.e COBOL-97

3.1.6 **1964: BASIC**

BASIC (Beginner's All-purpose Symbolic Instruction Code). It was designed as a teaching language in 1963 by John George Kemeny and Thomas Eugene Kurtz of Dartmouth college. Intended to make it easy to learn programming. It stands for beginner’s All-purpose symbolic instruction code. In 1991 Microsoft released Visual Basic, an updated version of Basic. The first microcomputer version of Basic was co-written by Bill Gates, Paul Allen, and Monte Davidoff for their newly-formed company, Microsoft.

3.1.7 1970: Pascal

Pascal is named after a French religious fanatic and mathematician Blaise Pascal. It was Created in 1970 with the intension of replacing BASIC for teaching language. It was quickly developed as a general purpose language. It was Programs compiled to a platform-independent intermediate p-code. The compiler for Pascal was written in Pascal.

3.1.8 1972: C

It is a general-purpose, procedural programming language and the most popular programming language till now. All the code that was previously written in assembly language gets replaced by the C language like operating system, kernel, and many other applications. It can be used in implementing an operating system, embedded system, and also on the website using the Common Gateway Interface (CGI). C is the mother of almost all higher-level programming languages like C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, and Unix's C shell.

3.1.9 Other Programming Languages

The table 1 below listed some popular programming languages among the programmers.

3.2 Programming Language Paradigm

'The Merriam-Webster's Collegiate dictionary': *"A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated" while* 'The American Heritage Dictionary of the English Language, Third Edition': *"An example that serves as pattern or model."* A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathem*atical theory. By the word paradigm, we understand a set of patterns and practices used to achieve a certain goal. Millions of programming languages have been invented, and several thousands of them are actually in use. Compared to natural languages that developed and evolved independently, programming languages are far more similar to each other. This is because;

- different programming languages share the same mathematical foundation (e.g., Boolean algebra, logic);
- they provide similar functionality (e.g., arithmetic, logic operations, and text processing);
- they are based on the same kind of hardware and instruction sets;

- they have common design goals: find languages that make it simple for humans to use and efficient for hardware to execute;
- designers of programming languages share their design experiences.

It is worthwhile to note that many languages belong to multiple paradigms. For example, we can say that C++ is an object-oriented programming language. However, C++ includes almost every feature of C and thus is an imperative programming language too. We can use C++ to write C programs. Java is more object- oriented, but still includes many imperative features. For example, Java's primitive type variables do not obtain memory from the language heap like other objects. Lisp contains many nonfunctional features. Scheme can be considered a subset of Lisp with fewer nonfunctional features. Prolog's arithmetic operations are based on the imperative paradigm.

3.2.1 Categories of Programming Paradigm

There are many programming paradigms in use today. A main programming paradigm stems an idea within some basic discipline which is relevant for performing computations. Some programming languages, however, are more similar to each other, while other programming languages are more different from each other. Based on their similarities or the paradigms, programming languages can be divided into different classes namely;

- Imperative paradigm
- Functional paradigm,
- Logic paradigm
- Object-Oriented paradigm
- Visual paradigm
- Parallel/concurrent paradigms,
- Constraint based paradigm
- Dynamic paradigms.

3.2.2 Overview of Main Programming Paradigm

There are four main programming paradigms which are imperative paradigm functional paradigm, logical paradigm and object-oriented paradigm.

3.2.2.1 Imperative Paradigm

The imperative, also called the procedural, programming paradigm expresses computation by fully specified and fully controlled manipulation of named data in a stepwise fashion. In other words, data or values are initially stored in variables (memory locations), taken out of (read from) memory, manipulated in ALU (arithmetic logic unit), and then stored back in the same or different variables (memory locations). Finally, the values of variables are sent to the I/O devices as output. The foundation of imperative languages is the stored program concept-based computer hardware organization and architecture (von Neumann machine). The stored program concept will be further explained in the next chapter. Typical imperative programming languages include all assembly languages and earlier high-level languages like Fortran, Algol, Ada, Pascal, and C.

3.2.2.2 Object-Oriented Paradigm

The object-oriented programming paradigm is basically the same as the imperative paradigm, except that related variables and operations on variables are organized into classes of objects. The access privileges of variables and methods (operations) in objects can be defined to reduce (simplify) the interaction among objects. Objects are considered the main building blocks of programs, which support language features like inheritance, class hierarchy, and polymorphism. Typical object-oriented programming languages include Smalltalk, C++, Python, Java, and C#.

3.2.2.3 Functional (Application) Paradigm

The functional, also called the applicative, programming paradigm expresses computation in terms of mathematical functions. Since we express computation in mathematical functions in many of the mathematics courses, functional programming is supposed to be easy to understand and simple to use. However, since functional programming is very different from imperative or object-oriented programming, and most programmers first get used to writing programs in imperative or object-oriented paradigms, it becomes difficult to switch to functional programming. The main difference is that there is no concept of memory locations in functional programming languages. Each function will take a number of values as input (parameters) and produce a single return value (output of the function). The return value cannot be stored for later use. It has to be used either as the final output or immediately as the parameter value of another function. Functional programming is about defining functions and organizing the return values of one or more functions as the parameters of another function. Functional programming languages are mainly based on the lambda calculus that will be discussed in Chapter 4. Typical functional programming Languages

include ML, SML, and Lisp/Scheme. Python and C# support direct applications of lambda calculus and many functional programming features.

3.2.2.4 Logic Paradigm

The logic, also called the declarative, programming paradigm expresses computation in terms of logic predicates. A logic program is a set of facts, rules, and questions. The execution process of a logic program is to compare a question to each fact and rule in the given fact and rulebase. If the question finds a match, we receive a yes answer to the question. Otherwise, we receive a no answer to the question. Logic programming is about finding facts, defining rules based on the facts, and writing questions to express the problems we wish to solve. Prolog is the only significant logic programming language.

4 Self-Assessment Exercises

- Explain the evolution of programming language.
- What is programming language paradigm?
- List all categories of programming language paradigm.
- Compare and contrast the four programming paradigms: imperative, object-oriented, functional, and logic
- Explain in details the four common programming language paradigm.

5 Conclusion

Paradigm is a set of basic principles, concepts, and methods for how a computation or algorithm is expressed. We have several programming paradigms nowadays. Although there is similarity between some of these programming languages. It is worthy to know which paradigm the programming language you are using belong. This serves as bases and purpose of this unit.

6 Summary

In this unit, you learnt that the history of programming language right from the beginning till present. Also, the programming paradigms were discussed. Based on the similarity and differences of programming languages were program paradigm were grouped into eight classes namely; Imperative paradigm, Functional paradigm, Logic paradigm, Object-Oriented paradigm, Visual paradigm, Parallel/concurrent paradigms, Constraint based paradigm, Dynamic paradigms. The four common program paradigm were also discussed in details.

7 References/Further Reading

Ghezzi & Jazayeri (1996.) *Programming language concepts—Third edition* John Wiley & Sons New York Chichester Brisbane Toronto Singapore 1996.

https://www.geeksforgeeks.org/the-evolution-of-programming-languages

Unit 3 Structured/Unstructured Programming Language

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Structured Programming
   - 3.2. Elementary Structures of Structured Programs
   - 3.3. Different between Structured and unstructured programming language
   - 3.4. Types of Structured Programming
     - 3.4.1. Procedural Programming
     - 3.4.2. Object-oriented Programming
     - 3.4.3. Model-based Programming
   - 3.5. Components of Structured Programming
   - 3.6. Advantages and Disadvantages of Structured Programming
     - 3.6.1. Advantages of Structured Programming
     - 3.6.2. Disadvantages of Structured Programming
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 3 Structured/Unstructured Programming Language

1 Introduction

To make programs easier to understand this unit discusses structured and unstructured programming. The difference between structured and unstructured language will also be discussed. Furthermore, the unit deliberates on different types and components of structured programming language. It also presented the advantages and disadvantages of structured programming.

2 Intended Learning Outcomes (ILOs)

- At the end of the unit, students should able to
- Understand structured/ unstructured programming language
- Differentiate between structured and unstructured programming language
- Explain types and component of structured programming
- Discuss the advantages and disadvantaged od structured programming

3 Main Content

3.1 Elementary structures of structured programs?

Structured programming (sometimes known as *modular programming*) is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming, but the mechanisms of support, like the syntax of the programming languages, varies. Where modules or elements of code can be reused from a library, it may also be possible to build structured code using modules written in different languages, as long as they can obey a common module interface or application program interface (API) specification. However, when modules are reused, it's possible to compromise data security and governance, so it's important to define and enforce a privacy policy controlling the use of modules that bring with them implicit data access rights.

Structured programming encourages dividing an application program into a hierarchy of modules or autonomous elements, which may, in turn, contain other such elements. Within each element, code may be further structured using blocks of related logic designed to improve readability and maintainability. These may include case, which tests a variable against a set of values; Repeat, while and for, which construct loops that continue until a condition is met. In all structured

programming languages, an unconditional transfer of control, or goto statement, is deprecated and sometimes not even available.

- Block: It is a command or a set of commands that the program executes linearly. The sequence has a single point of entry (first line) and exit (last line).
- Selection: It is the branching of the flow of control based on the outcome of a condition. Two sequences are specified: the 'if' block when the condition is true and the 'else' block when it is false. The 'else' block is optional and can be a no-op.
- Iteration: It is the repetition of a block as long as it meets a specific condition. The evaluation of the condition happens at the start or the end of the block. When the condition results in false, the loop terminates and moves on to the next block.
- Nesting: The above building blocks can be nested because conditions and iterations, when encapsulated, have singular entry-exit points and behave just like any other block.
- Subroutines: Since entire programs now have singular entry-exit points, encapsulating them into subroutines allows us to invoke blocks by one identifier.

3.2 Difference between structured and unstructured programming languages

A structured programming language facilitates or enforces structured programming practices. These practices can also be supported with unstructured languages, but that will require specific steps in program design and implementation. Structured programming practices thus date to the emergence of structured programming languages.

The theoretical basis for structured programming goes back to the 1950s, with the emergence of the ALGOL 58 and 60 languages. Up to then, code clarity was reduced by the need to build condition/action tests by having programmers write linked tests and actions explicitly (using the goto statement or its equivalent), resulting in what was often called spaghetti code. ALGOL included block structure, where an element of code included a condition and an action.

Modular programming, which is today seen as synonymous with structured programming, emerged a decade later as it became clear that reuse of common code could improve developer productivity. In modular programming, a program is divided into semi-independent modules, each of which are called when needed. Purists argue that modular programming requires actual

independence of modules, but most development teams consider any program that divides logic into separate elements, even if those elements exist within the same program, as modular.

Modern programming languages are universally capable of producing structured code. Similarly, they're also capable of producing code fairly described as unstructured if used incorrectly. Some would say that an unstructured programming language contains goto statements and, thus, does not require a "call" to a separate module, which then returns when complete, but that definition is unnecessarily restrictive. It's better to say that the mechanisms for enforcing structure vary by language, with some languages demanding structure and other accepting less-structured code.

3.3 Types of structured programming

Structured programming can be divided into three categories, including:

3.3.1 Procedural programming.

Defines modules as "procedures" or "functions" that are called with a set of parameters to perform a task. A procedural language will begin a process, which is then given data. It is also the most common category and has recently been subdivided into the following:

- Service-oriented programming simply defines reusable modules as "services" with advertised interfaces.
- Microservice programming focuses on creating modules that do not store data internally, and so are scalable and resilient in cloud deployment.
- Functional programming, technically, means that modules are written from functions, and that these functions' outputs are derived only from their inputs. Designed for server less computing, the definition of functional programming has since expanded to be largely synonymous with microservices.

3.3.2 Object-oriented programming (OOP).

Defines a program as a set of objects or resources to which commands are sent. An object-oriented language will define a data resource and send it to process commands. For example, the procedural programmer might say "Print(object)" while the OOP programmer might say "Tell Object to Print".

3.3.3 Model-based programming.

The most common example of this is database query languages. In database programming, units of code are associated with steps in database access and update or run when those steps occur. The database and database access structure will determine the structure of the code. Another example of a model-based structure is Reverse Polish Notation (RPN), a math-problem structure that lends itself to efficient solving of complex expressions. Quantum computing, just now emerging, is another example of model-based structured programming; the quantum computer demands a specific model to organize steps, and the language simply provides it.

3.4 Components of structured programming

At the high level, structured programs consist of a structural hierarchy starting with the main process and decomposing downward to lower levels as the logic dictates. These lower structures are the modules of the program, and modules may contain both calls to other (lower-level) modules and blocks representing structured condition/action combinations. All of this can be combined into a single module or unit of code, or broken down into multiple modules, resident in libraries.

Modules can be classified as "procedures" or "functions." A procedure is a unit of code that performs a specific task, usually referencing a common data structure available to the program at large. Much of the data operated on by procedures is external. A function is a unit of code that operates on specific inputs and returns a result when called.

Structured programs and modules typically have a header file or section that describes the modules or libraries referenced and the structure of the parameters and module interface. In some programming languages, the interface description is abstracted into a separate file, which is then implemented by one or more other units of code.

3.5 Advantages and Disadvantages of structured programming

3.5.1 Advantages of structured programming

The primary advantages of structured programming are:

- It encourages top-down implementation, which improves both readability and maintainability of code.

- It promotes code reuse, since even internal modules can be extracted and made independent, residents in libraries, described in directories and referenced by many other applications.
- It's widely agreed that development time and code quality are improved through structured programming.

These advantages are normally seen as compelling, even decisive, and nearly all modern software development employs structured programming.

3.5.2 Disadvantages of structured programming

The biggest disadvantage of structured programming is a reduction in execution efficiency, followed by greater memory usage. Both these problems arise from the introduction of calls to a module or process, which then returns to the caller when it's done. System parameters and system resources are saved on a stack (a queue organized as LIFO, or last-in-first-out) and popped when needed. The more program logic is decomposed, meaning the more modules are involved, the greater the overhead associated with the module interface. All structured programming languages are at risk to "over-structuring" and loss of efficiency.

Structured programming can also be applied incorrectly if the type of structure selected isn't right for the task at hand. The best-known example is the solving of math problems. RPL is an efficient way to state and solve a math problem because it eliminates the need to explicitly state execution order and eliminates recursion in code. However, if that problem was to be posed in structured programming procedural or object form, the resulting code would be much less efficient than the RPL version.

4 Self-Assessment Exercises

- Define structured programming
- Differentiate between structured and unstructured programming
- Discuss the different categories of structured programming
- Explain the component of structured programming

5 Conclusion

Structured programming is a paradigm that aims to make programs easier to comprehend from a reader's point of view. It does this by line arising the flow of control through a program. In structured programming, execution follows the writing order of the code. Structured programming caught favor with programming languages for its iconic opposition to the keyword goto, aiming to reduce the prevalence of spaghetti code.

6 Summary

To make programs easier to understand this unit discussed structured and unstructured programming. The difference between structured and unstructured language were also discussed. The unit deliberated on different types and components of structured programming language. It also presented the advantages and disadvantages of structured programming.

7 References/Further Reading

https://deepsource.io/glossary/structured-programming

https://searchsoftwarequality.techtarget.com/definition/structured-programming-modular-programming

Module 2 Language Structure

Having refreshed our memory about programming language in previous module, this module handles language structure which is one of the major topic in organization of programming language. The module is divided into four units. Unit 1 discusses the different structural layers of programming language as well as the designing and constructs of these layers. Unit 2 presents general problem of describing syntax as well as formal methods of describing syntax. Also, the attribute grammars, operational semantics, denotational semantic and axiomatic semantic will be talk about in unit 2. Unit 3 deliberates extensively on lexical analysis with focus on lexical process and building lexical analyzer. Also, the unit discusses the parsing problem, recursive-decent parsing and bottom-up parsing. The last unit which is unit 4 introduces the implementation of language processing by discussing interpretation, translation, concept of interpretative language and binding.

Unit 1 Concept of Language Structure

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
    - 3.1. Structural Layers
        - 3.1.1. Lexical Structure
            - 3.1.1.1. Identifiers
            - 3.1.1.2. Keywords
            - 3.1.1.3. Operators
            - 3.1.1.4. Separators
            - 3.1.1.5. Literals
            - 3.1.1.6. Comments
            - 3.1.1.7. Layout and Spacing
        - 3.1.2. Syntactic Structure
            - 3.1.2.1. Assignments
            - 3.1.2.2. Conditional Statements
            - 3.1.2.3. Loop Statements
        - 3.1.3. Contextual Structure
        - 3.1.4. Semantic Structure
    - 3.2. Error Types at Different levels
        - 3.2.1. Lexical Errors
        - 3.2.2. Syntactic Errors
        - 3.2.3. Contextual Errors
        - 3.2.4. Semantic Errors
        - 3.2.5. Examples of Conceptual and Semantic Errors
    - 3.3. Application of BNF Notation and Syntax Graph
        - 3.3.1. BNF Notation
        - 3.3.2. Syntax Graph
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 1 Concept of Language Structure

1 Introduction

A structure is used to represent information about something more complicated than a single number, character, or Boolean. Thus, this unit presents the fundamental concepts of language structuring by discussing the structural layers of programming language. The designing and constructs of these layers are also discussed.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- discuss in details the structural layers of programming language
- Understand types of error that occur in each layers

3 Main Content

3.1 Structural Layers

The structures of programming languages are grouped into four structural layers which are lexical, syntactic, contextual, and semantic.

3.1.1 Lexical structure

Lexical structure defines the vocabulary of a language. Lexical units are considered the building blocks of programming languages. The lexical structures of all programming languages are similar and normally include the following kinds of units:

3.1.1.1 Identifiers

Names that can be chosen by programmers to represent objects like variables, labels, procedures, and functions. Most programming languages require that an identifier start with an alphabetical letter and can be optionally followed by letters, digits, and some special characters.

3.1.1.2 Keywords:

Names reserved by the language designer and used to form the syntactic structure of the language.

3.1.1.3 Operators

Symbols used to represent the operations. All general-purpose programming languages should provide certain minimum operators such as mathematical operators like +, −, *, /, relational operators like <, ?, ==, >, ?, and logic operators like AND, OR, NOT, etc.

3.1.1.4 Separators:

Symbols used to separate lexical or syntactic units of the language. Space, comma, colon, semicolon, and parentheses are used as separators.

3.1.1.5 Literals:

Values that can be assigned to variables of different types. For example, integer-type literals are integer numbers, character-type literals are any character from the character set of the language, and string-type literals are any string of characters.

3.1.1.6 Comments:

Any explanatory text embedded in the program. Comments start with a specific keyword or separator. When the compiler translates a program into machine code, all comments will be ignored.

3.1.1.7 Layout and spacing:

Some languages are of free format such as C, C++, and Java. They use braces and parentheses for defining code blocks and separations. Additional whitespace characters (spaces, newlines, carriage returns, and tabs) will be ignored. Some languages consider layout and whitespace characters as lexical symbols. For example, Python does not use braces for defining the block of code. It uses indentation instead. Different whitespace characters are considered different lexical symbols.

3.1.2 Syntactic structure

Syntactic structure defines the grammar of forming sentences or statements using the lexical units. An imperative programming language normally offers the following basic kinds of statements:

3.1.2.1 Assignments:

An assignment statement assigns a literal value or an expression to a variable.

3.1.2.2 Conditional statements:

A conditional statement tests a condition and branches to a certain statement based on the test result (true or false). Typical conditional statements are if-then, if-then- else, and switch (case).

3.1.2.3 Loop statements:

A loop statement tests a condition and enters the body of the loop or exits the loop based on the test result (true or false). Typical loop statements are for-loop and while-loop.

3.1.3 Contextual structure

Contextual structure (also called static semantics) defines the program semantics before dynamic execution. It includes variable declaration, initialization, and type checking. Some imperative languages require that all variables be initialized when they are declared at the contextual layer, while other languages do not require variables to be initialized when they are declared, as long as the variables are initialized before their values are used. This means that initialization can be done either at the contextual layer or at the semantic layer. Contextual structure starts to deal with the meaning of the program. A statement that is lexically correct may not be contextually correct. For example:

```
String str = "hello";
int i = 0, j;
j = i + str;
```

The declaration and the assignment statements are lexically and syntactically correct, but the assignment statement is contextually incorrect because it does not make sense to add an integer variable to a string variable.

3.1.4 Semantic structure

Semantic structure describes the meaning of a program, or what the program does during the execution. The semantics of a language are often very complex. In most imperative languages, there is no formal definition of semantic structure. Informal descriptions are normally used to explain what each statement does. The semantic structures of functional and logic programming languages are normally defined based on the mathematical and logical foundation on which the languages are based. For example, the meanings of Scheme procedures are the same as the meanings of the lambda expressions in lambda calculus on which Scheme is based, and the meanings of Prolog clauses are the same as the meanings of the clauses in Horn logic on which Prolog is based.

3.2 Error types at different levels

Programming errors can occur at all levels of a program. We call these errors lexical errors, syntactic errors, contextual errors, and semantic errors, respectively, depending on the levels where the errors occur.

3.2.1 Lexical errors:

Errors at the lexical level. Compiler can detect all the lexical errors. For example:

```
int if = 0, 3var = 3; double IsTrue? = 0;
```

These declarations will cause compilation errors in C, because "if" is a keyword, a variable cannot start with a number, and "?" cannot be used in variable definition.

3.2.2 Syntactic errors:

Errors at the syntactic level. Compiler can detect all of them. For example:

```
main(){
   int x = 0, y = 3; double z = 0;
   if x == 1, y++; // syntax error: condition must be quoted by parentheses
   z = x+y         // missing semicolon
}
```

There are a number of syntactic errors in C in this piece of code:

- The condition if-statement must be quoted by parentheses.
- No comma between the condition and the following statement.
- A semicolon is missing at the end of z = x+y statement.

3.2.3 Contextual errors:

Errors at the contextual level. Contextual errors are complex and compiler implementations may or may not detect all of the initialization errors, depending on whether they actually compute the initialization expression or not. They include all the errors (excluding the lexical errors) in

- variable declaration,
- variable initialization, and
- type inconsistent in assignment.

The following are examples of contextual errors:

```
int x = 5/(3+2); // contextual error that compiler may not detect
x = "hello";     // type inconsistent in assignment
```

3.2.4 Semantic errors:

Errors at the semantic level. They include all the errors in the statements that will be executed after passing compilation. The compiler normally does not detect semantic errors. For example:

```
int x, y = 5;
x = y/(3+2);    // semantic error
```

3.2.5 Examples of contextual errors and semantic errors

Figure 2 shows several contextual and semantic errors with similar but different types of errors that the compilers may handle differently.

- In Figure 2 (a), there is a clear semantic error. The code will pass all compilers but will cause an exception at execution.
- In Figure 2 (b), there is a contextual error in initialization. Since the initialization expression is quite complex, both GCC and Visual Studio will not detect the error because they choose to compile the initialization statement as an execution statement in the form shown in Figure 2 (c). Therefore, the contextual error in initialization will be delayed to the execution stage. We still call such errors contextual errors because the compiler's choice of implementation should not impact the definitions of error types.
- In Figure 2 (c), the initialization statement is written as an execution statement, and, thus, the error changes from contextual error to semantic error.
- Figure 2 (d) has a clearly semantic error that will not be detected by any compilers, even though the expression is simple and straightforward, showing a division zero situation. Now, we move the execution statement in Figure 2 (d) to the declaration part in Figure 2 (e). It now will be a contextual error. This example shows a situation where different compilers will handle it differently. Visual Studio will throw a compiler error, whereas GCC will pass the code. Although GCC gives a warning of division by zero, it still generates executable.

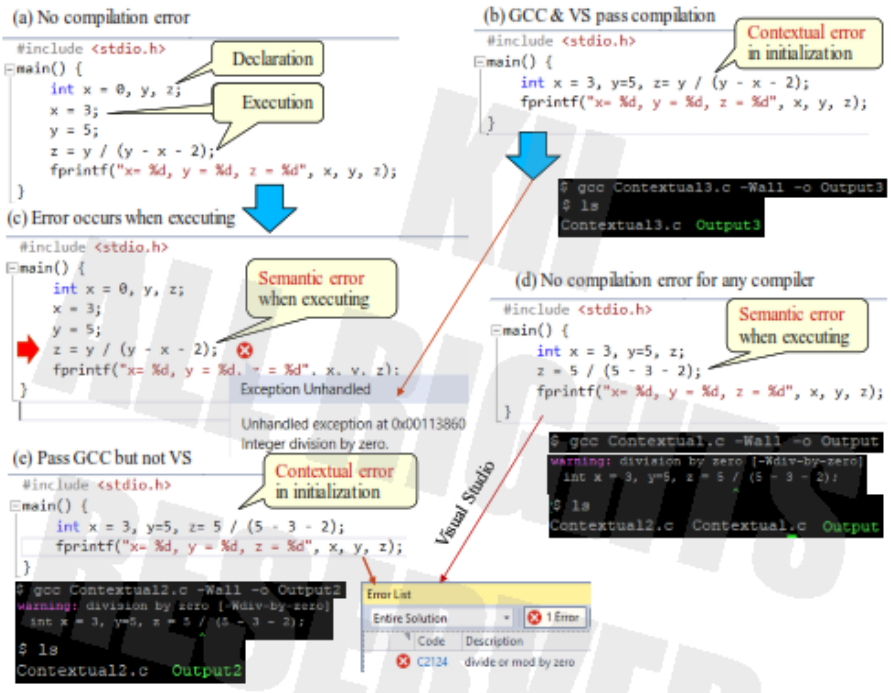


Figure 2

3.3 Application of BNF Notation and Syntax Graph

Lexical and syntactic structure of a language can be analyzed using BNF and syntax graph.

3.3.1 BNF Notation

BNF (Backus-Naur Form) is a meta language that can be used to define the lexical and syntactic structures of another language. For easy understanding of BFN, we will first use BNF to define a simplified English language that we are familiar with, and then we will learn BNF from the definition itself.

A simple English sentence consists of a subject, a verb, and an object. The subject, in turn, consists of possibly one or more adjectives followed by a noun. The object has the same grammatical structure. The verbs and adjectives must come from the vocabulary. Formally, we can define a simple English sentence as follows:

<sentence> ::= <subject><verb><object>
<subject> ::= <noun> | <article><noun> | <adjective><noun> | <article><adjective><noun>
<adjective> ::= <adjective> | <adjective><adjective>
<object> ::= <subject>
<noun> ::= table | horse | computer
<article> ::= the | a
<adjective> ::= big | fast | good | high
<verb> ::= is | makes

In the definitions, the symbol "::=" means that the name on the left-hand side is defined by the expression on the right-hand side. The name in a pair of angle brackets "<>" is nonterminal, which means that the name needs to be further defined. The vertical bar "|" represents an "or" relation. The boldfaced names are terminal, which means that the names need not be further defined. They form the vocabulary of the language. We can use the sentence definition to check whether the following sentences are syntactically correct.

fast high big computer is good table	1
the high table is a good table	2
a fast table makes the high horse	3
the fast big high computer is good	4
good table is high	5
a table is not a horse	6
is fast computer good	7

The first sentence is syntactically correct, although it does not make much sense. Three adjectives in the sentence are correct because the definition of an adjective recursively allows any number of adjectives to be used in the subject and the object of a sentence. The second and third sentences are also syntactically correct according to the definition. The fourth and fifth sentences are syntactically incorrect because a noun is missing in the object of the sentences. The sixth sentence is incorrect because "not" is not a terminal. The last sentence is incorrect because the definition does not allow a sentence to start with a verb.

After we have a basic understanding of BNF, we can use it to define a small programming language. The first five lines define the lexical structure, and the rest defines the syntactic structure of the language.

```
<letter>      ::=  a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>       ::=  0|1|2|3|4|5|6|7|8|9
<symbol>      ::=  _|@|.|~|?|#|$
<char>        ::=  <letter>|<digit>|<symbol>
<operator>    ::=  +|−|*|/|%|<|>|==|<=|>=|and|or|not
<identifier>  ::=  <letter>|<identifier><char>
<number>      ::=  <digit>|<number><digit>
<item>        ::=  <identifier>|<number>
<expression>  ::=  <item>|(<expression>)| <expression><operator><expression>
<branch>      ::=  if <expr>then {<block>} | if <expr>then {<block>}else {<block>}
<switch>      ::=  switch<expr>{<sbody>}
<sbody>       ::=  <cases> | <cases>; default :<block>
<cases>       ::=  case<value>:<block> | <cases> ; case<value>:<block>
<loop>        ::=  while <expr>do {<block>}
<assignment>  ::=  <identifier>=<expression>;
<statement>   ::=  <assignment>|<branch>|<loop>
<block>       ::=  <statement>|<block>;<statement>
```

Now we use the definition to check which of the following statements are syntactically correct.

```
sum1 = 0;                                                    1
while sum1 <= 100 do {                                       2
sum1 = sum1 + (a1 + a2) * (3b % 4*b); }                      3
if sum1 == 120 then 2sum − sum1 else sum2 + sum1;            4
p4#rd_2 = ((1a + a2) * (b3 % b4)) / (c7 − c8);               5
_foo.bar = (a1 + a2 − b3 − b4);                              6
(a1 / a2) = (c3 − c4);                                       7
```

According to the BNF definition of the language, statements 1 and 2 are correct. Statements 3 and 4 are incorrect because 3b and 2 sum are neither acceptable identifiers nor acceptable expressions. Statement 5 is incorrect. Statement 6 is incorrect because an identifier must start with a letter. Statement 7 is incorrect because the left-hand side of an assignment statement must be an identifier.

3.3.2 Syntax graph

BNF notation provides a concise way to define the lexical and syntactic structures of programming languages. However, BNF notations, especially the recursive definitions, are not always easy to understand. A graphic form, called a syntax graph, also known as railroad tracks, is often used to supplement the readability of BNF notation. For example, the identifier and the if-then-else statement corresponding to the BNF definitions can be defined using the syntax graphs in Figure 3. The syntax graph for the identifier requires that an identifier start with a letter, may exit with only one letter, or follow the loops to include any number of letters, digits, or symbols. In other words, to check the legitimacy of an identifier, we need to travel through the syntax graph following the arrows and see whether we can find a path that matches the given identifier. For instance, we can verify that len_23 is a legitimate identifier as follows. We travel through the first <letter> once, travel through the second <letter> on the back track twice, travel through the <symbol> once, and finally travel through the <digit> twice, and then we exit the definition. On the other hand, if you try to verify that 23_len is a legitimate identifier, you will not be able to find a path to travel through the syntax graph.

Using the if-then-else syntax graph in Figure 3, we can precisely verify whether a given statement is a legitimate if-then-else statement. The alternative route that bypasses the else branch signifies that the else branch is optional. Please note that the definition of the if-then-else statement here is not the same as the if- then-else statement in C language.

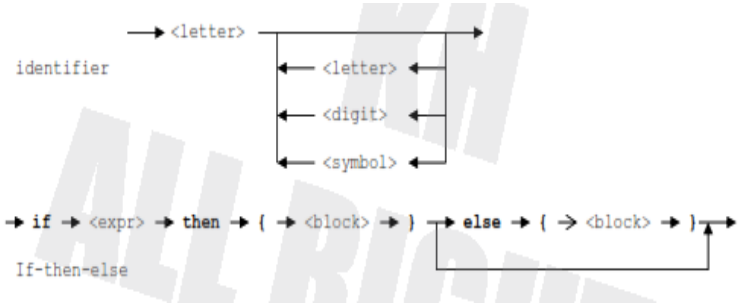


Figure 3. Definition of identifier and if-then-else statement.

As another example, Figure 4 shows the definitions of a set data structures, including the definitions of value, string, array, bool, number, and object. In syntax graphs, we use the same convention that terminals are in boldfaced text and nonterminals are enclosed in a pair of angle brackets.

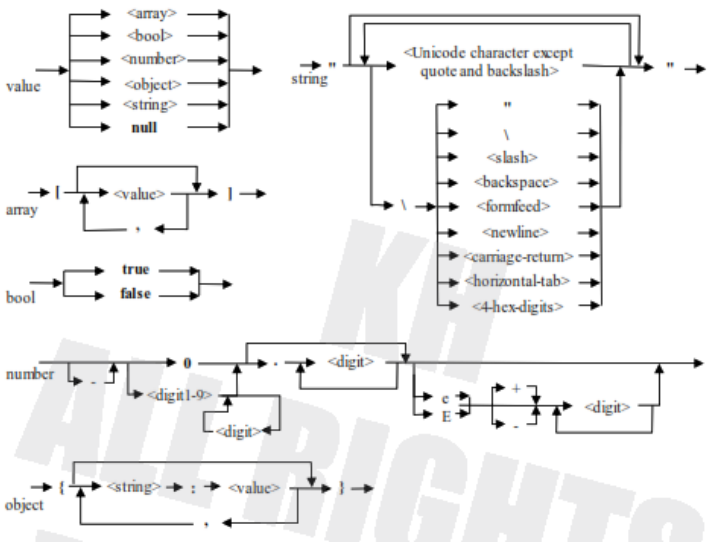


Figure 4 Definitions of different data structures. <digit>

4 Self-Assessment Exercises

- Compare and contrast the four structural layers: lexical, syntactic, contextual, and semantic structures
- Mention and explain error types that occur at each structural layer
- Explain the application of BNF and syntax graph in lexical and syntactic structure
- From the stated definitions below check if the following statements stated below are syntactically correct.

  Definitions

  <letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
  <digit> ::= 0|1|2|3|4|5|6|7|8|9

```
<symbol>      ::=   _|@|.|~|?|#|$
<char>        ::=   <letter>|<digit>|<symbol>
<operator>    ::=   +|−|*|/|%|<|>|==|<=|>=|and|or|not
<identifier>  ::=   <letter>|<identifier><char>
<number>      ::=   <digit>|<number><digit>
<item>        ::=   <identifier>|<number>
<expression>  ::=   <item>|(<expression>)| <expression><operator><expression>
<branch>      ::=   if <expr>then {<block>} | if <expr>then {<block>}else {<block>}
<switch>      ::=   switch<expr>{<sbody>}
<sbody>       ::=   <cases> | <cases>; default :<block>
<cases>       ::=   case<value>:<block> | <cases> ; case<value>:<block>
<loop>        ::=   while <expr>do {<block>}
<assignment>  ::=   <identifier>=<expression>;
<statement>   ::=   <assignment>|<branch>|<loop>
<block>       ::=   <statement>|<block>;<statement>
```

Statement

```
sum1 = 0;                                                    1
while sum1 <= 100 do {                                       2
sum1 = sum1 + (a1 + a2) * (3b % 4*b); }                      3
if sum1 == 120 then 2sum − sum1 else sum2 + sum1;            4
p4#rd_2 = ((1a + a2) * (b3 % b4)) / (c7 − c8);               5
_foo.bar = (a1 + a2 − b3 − b4);                              6
(a1 / a2) = (c3 − c4);                                       7
```

5 Conclusion

Defining the language vocabulary and grammar is very crucial in constructing a language. Thus, depth knowledge of the structural layers (lexical, syntactic, contextual, and semantic) will help in detecting error easily at each layer. Also, BNF and syntax graph can be used to define the lexical and syntactic structures of a language.

6 Summary

This unit presented the fundamental concepts of language structuring. The study discussed the structural layers of programming language which are lexical, syntactic, contextual, and semantic structure. Also the error type associated to each of mentioned structural layers were discussed.

7 References/Further Reading

Chen, Y. (2020). Chapter 1 Basic Principles of Programming Languages. In *Introduction to Programming Languages* (Sixth, pp. 1–40). Kendal Hunt Plublishing

Unit 2 Syntax and Semantics

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Syntax
   - 3.2. Semantics
   - 3.3. The General Problem of Describing Syntax
     - 3.3.1. Language Recognizers
     - 3.3.2. Language Generators
   - 3.4. Formal Methods of Describing Syntax
     - 3.4.1. Backus-Naur Form and Content-Free Grammars
     - 3.4.2. Extended BNF
     - 3.4.3. Grammars and Recognizers
   - 3.5. Attribute Grammars
     - 3.5.1. Static Semantics
     - 3.5.2. Basic Concepts
     - 3.5.3. Attribute Grammars Defined
     - 3.5.4. Intrinsic Attributes
     - 3.5.5. Examples of Attribute Grammar
     - 3.5.6. Computing Attribute Values
     - 3.5.7. Evaluation
   - 3.6. Describing the Meanings of Program: Dynamic Semantics
     - 3.6.1. Operational Semantics
       - 3.6.1.1. The Basic Process
       - 3.6.1.2. Evaluation
     - 3.6.2. Denotational Semantics
       - 3.6.2.1. Two Simple Examples
       - 3.6.2.2. The State of a Program
       - 3.6.2.3. Expressions
       - 3.6.2.4. Assignment Statements
       - 3.6.2.5. Logical Pretest Loops
       - 3.6.2.6. Evaluation
     - 3.6.3. Axiomatic Semantics
       - 3.6.3.1. Assertions
       - 3.6.3.2. Weakest Preconditions
       - 3.6.3.3. Assignment Statements
       - 3.6.3.4. Sequences
       - 3.6.3.5. Selection
       - 3.6.3.6. Logical Pretest Loops
       - 3.6.3.7. Program Proofs
       - 3.6.3.8. Evaluation
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 2 Syntax and Semantics

1 Introduction

Programming language like natural language can be into syntax and semantics. Thus, unit introduces the fundamental concept of syntax and semantics. The syntax of a programming language is the form of its expressions, statements, and program units while Its semantics is the meaning of those expressions, statements, and program units. The unit presents a discussion on general problem of describing syntax and formal methods of describing syntax. Also attribute grammars, which can be used to describe both the syntax and static semantics of programming languages, are discussed. By describing the meanings of program the following will briefly discuss the operational semantics, denotational semantic and axiomatic semantic.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- have full understanding of language description
- know how the expressions, statements, and program units of a language are formed and also their intended effect when executed
- determine how to encode software solutions by referring to a language reference manual. by referring to a language reference manual.

3 Main Content

3.1 Syntax

Syntax is described by a set of rules that define the form of a language: they define how sentences may be formed as sequences of basic constituents called words. Using these rules we can tell whether a sentence is legal or not. The syntax does not tell us anything about the content (or meaning) of the sentence–the semantic rules tell us that. As an example, C keywords (such as while, do, if, else,...), identifiers, numbers, operators, ... are words of the language. The C syntax tells us how to combine such words to construct wellformed statements and programs.

Words are not elementary. They are constructed out of characters belonging to an alphabet. Thus the syntax of a language is defined by two sets of rules: lexical rules and syntactic rules. Lexical rules specify the set of characters that constitute the alphabet of the language and the way such characters can for example, Pascal considers lowercase and uppercase characters to be identical, but C and Ada consider them to be distinct. Thus, according to the lexical rules, “Memory” and “memory” refer to the same variable in Pascal, but to distinct variables in C and Ada. The lexical

rules also tell us that <> (or ¦) is a valid operator in Pascal but not in C, where the same operator is represented by !=. Ada differs from both, since "not equal" is represented as /=; delimiter <> (called "box") stands for an undefined range of an array index.

3.2 Semantics

Syntax defines well-formed programs of a language. Semantics defines the meaning of syntactically correct programs in that language. For example, the semantics of C help us determine that the declaration int vector [10]; causes ten integer elements to be reserved for a variable named vector. The first element of the vector may be referenced by vector [0]; all other elements may be referenced by an index i, 0 □□i □□9.

As another example, the semantics of C states that the instruction if (a > b) max = a; else max = b; means that the expression a > b must be evaluated, and depending on its value, one of the two given assignment statements is executed. Note that the syntax rules tell us how to form this statement–for example, where to put a ";"–and the semantic rules tell us what the effect of the statement is.

3.3 The General Problem of Describing Syntax

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. Formal descriptions of the syntax of programming languages, for simplicity's sake, often do not include descriptions of the lowest-level syntactic units. These small units are called lexemes. The description of lexemes can be given by a lexical specification, which is usually separate from the syntactic description of the language. The lexemes of a programming language include its numeric literals, operators, and special words, among others. One can think of programs as strings of lexemes rather than of characters.

Lexemes are partitioned into groups—for example, the names of variables, methods, classes, and so forth in a programming language form a group called *identifiers*. Each lexeme group is represented by a name, or token. So, a token of a language is a category of its lexemes. For

example, an identifier is a token that can have lexemes, or instances, such as sum and total. In some cases, a token has only a single possible lexeme. For example, the token for the arithmetic operator symbol + has just one possible lexeme. Consider the following Java statement:

index = 2 * count + 17;

The lexemes and tokens of this statement are

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

3.3.1 Language Recognizers

In general, languages can be formally defined in two distinct ways: by recognition and by generation (although neither provides a definition that is practical by itself for people trying to learn or use a programming language). Suppose we have a language L that uses an alphabet Σ of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet Σ. R would indicate whether a given input string was or was not in L. In effect, R would either accept or reject the given string. Such devices are like filters, separating legal sentences from those that are incorrectly formed. If R, when fed any string of characters over Σ, accepts it only if it is in L, then R is a description of L. Because most useful languages are, for all practical purposes, infinite, this might seem like a lengthy and ineffective process. Recognition devices, however, are not used to enumerate all of the sentences of a language— they have a different purpose.

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.

3.3.2 Language Generators

A language generator is a device that can be used to generate the sentences of a language. We can think of the generator as having a button that produces a sentence of the language every time it is pushed. Because the particular sentence that is produced by a generator when its button is pushed is unpredictable, 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.

3.4 Formal Methods of Describing Syntax

This section discusses the formal language-generation mechanisms, usually called grammars, that are commonly used to describe the syntax of programming languages.

3.4.1 Backus-Naur Form and Context-Free Grammars

In the middle to late 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism, which subsequently became the most widely used method for programming language syntax.

3.4.1.1 Context-Free Grammars

In the mid-1950s, Noam Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (Chomsky, 1956, 1959). 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. Because Chomsky was a linguist, his primary interest was the theoretical nature of natural languages. He had no interest at

the time in the artificial languages used to communicate with computers. So it was not until later that his work was applied to programming languages.

3.4.1.2 Origins of Backus-Naur Form

Shortly after Chomsky's work on language classes, the ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959 (Backus, 1959). This paper introduced a new formal notation for specifying programming language syntax. The new notation was later modified slightly by Peter Naur for the description of ALGOL 60 (Naur, 1960). This revised method of syntax description became known as Backus-Naur Form, or simply BNF. BNF is a natural notation for describing syntax. In fact, something similar to BNF was used by Panini to describe the syntax of Sanskrit several hundred years before Christ (Ingerman, 1967). Although the use of BNF in the ALGOL 60 report was not immediately accepted by computer users, it soon became and is still the most popular method of concisely describing programming language syntax. It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars. In the remainder of the chapter, we refer to context-free grammars simply as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

3.4.1.3 Fundamentals

A metalanguage is a language that is used to describe another language. BNF is a metalanguage for programming languages. BNF uses abstractions for syntactic structures. A simple Java assignment statement, for example, might be represented by the abstraction <assign> (pointed brackets are often used to delimit names of abstractions). The actual definition of <assign> can be given by <assign> → <var> = <expression>

The text on the left side of the arrow, which is aptly called the left-hand side (LHS), is the abstraction being defined. The text to the right of the arrow is the definition of the LHS. It is called the right-hand side (RHS) and consists of some mixture of tokens, lexemes, and references to other abstractions. (Actually, tokens are also abstractions.) Altogether, the definition is called a rule, or production. In the example rule just given, the abstractions <var> and <expression> obviously must be defined for the <assign> definition to be useful.

This particular rule specifies that the abstraction <assign> is defined as an instance of the abstraction <var>, followed by the lexeme =, followed by an instance of the abstraction <expression>. One example sentence whose syntactic structure is described by the rule is

total = subtotal1 + subtotal2

The abstractions in a BNF description, or grammar, are often called nonterminal symbols, or simply nonterminals, and the lexemes and tokens of the rules are called terminal symbols, or simply terminals. A BNF description, or grammar, is a collection of rules. Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language. Multiple definitions can be written as a single rule, with the different definitions separated described with the rules

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>
or with the rule
<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

In these rules, <stmt> represents either a single statement or a compound statement.

Although BNF is simple, it is sufficiently powerful to describe nearly all of the syntax of programming languages. In particular, it can describe lists of similar constructs, the order in which different constructs must appear, and nested structures to any depth, and even imply operator precedence and operator associativity.

3.4.1.4 Describing Lists

Variable-length lists in mathematics are often written using an ellipsis (. . .); 1, 2, . . . is an example. BNF does not include the ellipsis, so an alternative method is required for describing lists of syntactic elements in programming languages (for example, a list of identifiers appearing on a data declaration statement). For BNF, the alternative is recursion. A rule is **recursive** if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

<ident_list> → identifier
| identifier, <ident_list>

This defines <ident_list> as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>.

3.4.1.5 Grammars and Derivations

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**. This sequence of rule applications is called a **derivation**. In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>. The simple grammar shown in Example 1 is used to illustrate derivations.

Example 1: A Grammar for a Small Language

<program> → begin <stmt_list> end
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

The language described by the grammar of Example 3.1 has only one statement form: assignment. A program consists of the special word **begin**, followed by a list of statements separated by semicolons, followed by the special word **end**. An expression is either a single variable or two variables separated by either a + or - operator. The only variable names in this language are A, B, and C. A derivation of a program in this language follows:

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**

=> **begin** A = B + C ; B = <expression> **end**

=> **begin** A = B + C ; B = <var> **end**

=> **begin** A = B + C ; B = C **end**

This derivation, like all derivations, begins with the start symbol, in this case <program>. The symbol => is read "derives." Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions. Each of the strings in the derivation, including <program>, is called a **sentential form**.

In this derivation, the replaced nonterminal is always the leftmost nonterminal in the previous sentential form. Derivations that use this order of replacement are called **leftmost derivations**. The derivation continues until the sentential form contains no nonterminals. That sentential form, consisting of only terminals, or lexemes, is the generated sentence. In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost. Derivation order has no effect on the language generated by a grammar. By choosing alternative RHSs of rules with which to replace nonterminals in the derivation, different sentences in the language can be generated. By exhaustively choosing all combinations of choices, the entire language can be generated. This language, like most others, is infinite, so one cannot generate *all* the sentences in the language in finite time. Example 2 is another example of a grammar for part of a typical programming language.

Example 2: A Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A| B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

The grammar of Example 3.2 describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses. For example, the statement

A = B * (A + C)

is generated by the leftmost derivation:

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = B * <expr>

=> A = B * (<expr>)

=> A = B * (<id> + <expr>)

=> A = B * (A + <expr>)

=> A = B * (A + <id>)

=> A = B * (A + C)

3.4.1.6 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 4 shows the structure of the assignment statement derived previously

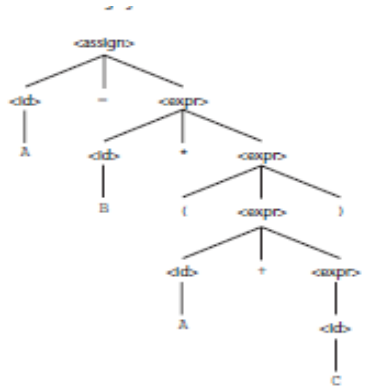


Figure 4

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.

3.4.1.7 Ambiguity

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**. Consider the grammar shown in Example 3, which is a minor variation of the grammar shown in Example 3.

Example 3: An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

The grammar of Example 3 is ambiguous because the sentence

A = B + C * A

has two distinct parse trees, as shown in Figure 5 and 6. The ambiguity occurs because the grammar specifies slightly less syntactic structure than does the grammar of

Figure 5

Figure 6

Example 2. Rather than allowing the parse tree of an expression to grow only on the right, this grammar allows growth on both the left and the right. Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form. Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree. If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely. This problem is discussed in two specific examples in the following

subsections. There are several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous. They include the following: (1) if the grammar generates a sentence with more than one leftmost derivation and (2) if the grammar generates a sentence with more than one rightmost derivation.

Some parsing algorithms can be based on ambiguous grammars. When such a parser encounters an ambiguous construct, it uses nongrammatical information provided by the designer to construct the correct parse tree. In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.

3.4.1.8 Operator Precedence

When an expression includes two different operators, for example, x + y * z, one obvious semantic issue is the order of evaluation of the two operators (for example, in this expression is it add and then multiply, or vice versa?). This semantic question can be answered by assigning different precedence levels to operators. For example, if * has been assigned higher precedence than + (by the language designer), multiplication will be done first, regardless of the order of appearance of the two operators in the expression.

A grammar can be written for the simple expressions we have been discuss- ing that is both unambiguous and specifies a consistent precedence of the + and * operators, regardless of the order in which the operators appear in an expression. The correct ordering is specified by using separate nonterminal symbols to represent the operands of the operators that have different precedence. This requires additional nonterminals and some new rules. Instead of using <expr> for both operands of both + and *, we could use three nonterminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree. If <expr> is the root symbol for expressions, + can be forced to the top of the parse tree by having <expr> directly generate only + operators, using the new nonterminal, <term>, as the right operand of +. Next, we can define <term> to generate * operators, using <term> as the left operand and a new nonterminal, <factor>, as its right operand. Now, * will always be lower in the parse tree, simply because it is farther from the start symbol than + in in every derivation. The grammar of Example 4 is such a grammar.

Example 4: An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>

<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

The grammar in Example 4 generates the same language as the grammars of Examples 2 and 3, but it is unambiguous and it specifies the usual precedence order of multiplication and addition operators. The following derivation of the sentence A = B + C * A uses the grammar

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

The parse tree for this sentence, as defined with the grammar of Example 4, is shown in Figure 7.

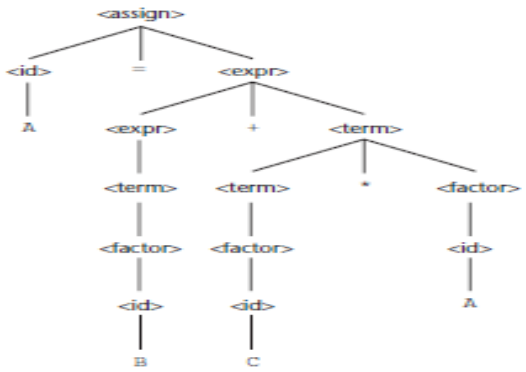


Figure 7

3.4.2 Extended BNF

Because of a few minor inconveniences in BNF, it has been extended in several ways. Most extended versions are called Extended BNF, or simply EBNF, even though they are not all exactly the same. The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability. Three extensions are commonly included in the various versions of EBNF. The first of these denotes an optional part of an RHS, which is delimited by brackets. For example, a C **if-else** statement can be described as

<if_stmt> → **if** (<expression>) <statement> [**else** <statement>]

Without the use of the brackets, the syntactic description of this statement would require the following two rules:

<if_stmt> → **if** (<expression>) <statement>

|**if** (<expression>) <statement> **else** <statement>

The second extension is the use of braces in an RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether. This extension allows lists to be built with a single rule, instead of using recursion and two rules. For example, lists of identifiers separated by commas can be described by the following rule:

<ident_list> → <identifier> {, <identifier>}

This is a replacement of the recursion by a form of implied iteration; the part enclosed within braces can be iterated any number of times. The third common extension deals with multiple-choice options. When a single element must be chosen from a group, the options are placed in parentheses and separated by the OR operator, |. For example,

<term> → <term> (* | / | %) <factor>

In BNF, a description of this <term> would require the following three rules:

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>

The brackets, braces, and parentheses in the EBNF extensions are **metasymbols**, which means they are notational tools and not terminal symbols in the syntactic entities they help describe. In cases where these metasymbols are also terminal symbols in the language being described, the instances that are terminal symbols can be underlined or quoted. Example 5 illustrates the use of braces and multiple choices in an EBNF grammar.

Example 5: BNF and EBNF Versions of an Expression Grammar

BNF: <expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF: <expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

The BNF rule

<expr> → <expr> + <term>

clearly specifies—in fact forces—the + operator to be left associative. However, the EBNF version,

<expr> → <term> {+ <term>}

does not imply the direction of associativity. This problem is overcome in a syntax analyzer based on an EBNF grammar for expressions by designing the syntax analysis process to enforce the correct associativity. Some versions of EBNF allow a numeric superscript to be attached to the right brace to indicate an upper limit to the number of times the enclosed part can be repeated. Also, some versions use a plus (+) superscript to indicate one or more repetitions. For example,

<compound> → **begin** <stmt> {<stmt>} **end**

and

<compound> → **begin** {<stmt>}+ **end**

are equivalent.

In recent years, some variations on BNF and EBNF have appeared. Among these are the following:

- In place of the arrow, a colon is used and the RHS is placed on the next line.
- Instead of a vertical bar to separate alternative RHSs, they are simply placed on separate lines.
- In place of square brackets to indicate something being optional, the subscript opt is used. For example,
- Constructor Declarator → SimpleName (FormalParameterListopt)
- Rather than using the | symbol in a parenthesized list of elements to indicate a choice, the words “one of” are used. For example,

Assignment Operator → one of = *= /= %= += -= <<= >>= &= ^= |=

There is a standard for EBNF, ISO/IEC 14977:1996(1996), but it is rarely used. The standard uses the equal sign (=) instead of an arrow in rules, terminates each RHS with a semicolon, and requires quotes on all terminal symbols. It also specifies a host of other notational rules.

3.4.3 Grammars and Recognizers

Earlier in this chapter, we suggested that there is a close relationship between generation and recognition devices for a given language. In fact, given a context-free grammar, a recognizer for the language generated by the grammar can be algorithmically constructed. A number of software systems have been developed that perform this construction. Such systems allow the quick creation of the syntax analysis part of a compiler for a new language and are therefore quite valuable. One of the first of these syntax analyzer generators is named yacc (*y*et *a*nother *c*ompiler *c*ompiler) (Johnson, 1975). There are now many such systems available.

3.5 Attribute Grammars

An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility. Before we formally define the form of attribute grammars, we must clarify the concept of static semantics.

3.5.1 Static Semantics

There are some characteristics of programming languages that are difficult to describe with BNF, and some that are impossible. As an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules. In Java, for example, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal. Although this restriction can be specified in BNF, it requires additional nonterminal symbols and rules. If all of the typing rules of Java were specified in BNF, the grammar would become too large to be useful, because the size of the grammar determines the size of the syntax analyzer. As an example of a syntax rule that cannot be specified in BNF, consider the common rule that all variables must be declared before they are referenced. It has been proven that this rule cannot be specified in BNF. These problems exemplify the categories of language rules called static semantics rules. The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics). Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required to check these specifications can be done at compile time. Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task. One such

mechanism, attribute grammars, was designed by Knuth (1968a) to describe both the syntax and the static semantics of programs. Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program. Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler (see Aho et al., 1986).

3.5.2 Basic Concepts

Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate functions. Attributes, which are associated with grammar symbols (the terminal and nonterminal symbols), are similar to variables in the sense that they can have values assigned to them. Attribute computation functions, sometimes called semantic functions, are associated with grammar rules. They are used to specify how attribute values are computed. Predicate functions, which state the static semantic rules of the language, are associated with grammar rules. These concepts will become clearer after we formally define attribute grammars and provide an example.

3.5.3 Attribute Grammars Defined

An attribute grammar is a grammar with the following additional features:

• Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets S(X) and I(X), called synthesized and inherited attributes, respectively. Synthesized attributes are used to pass semantic information up a parse tree, while inherited attributes pass semantic information down and across a tree.

- Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule. For a rule X0 SX1 c X*n*, the synthesized attributes of X0 are computed with semantic functions of the form S(X0) = f(A(X1), c , A(X*n*)). So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes. Inherited attributes of symbols X*j*, 1 … *j* … *n* (in the rule above), are computed with a semantic function of the form I(X*j*) = f(A(X0), c , A(X*n*)). So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes. Note that, to avoid circularity, inherited attributes are often restricted to functions of the form I(X*j*) =

f(A(X0), c, A(X(*j*-1))). This form prevents an inherited attribute from depending on itself or on attributes to the right in the parse tree.

- A predicate function has the form of a Boolean expression on the union of the attribute set 5A(X0), c, A(X*n*)6 and a set of literal attribute values. The only derivations allowed with an attribute grammar are those in which every predicate associated with every nonterminal is true. A false predicate function value indicates a violation of the syntax or static semantics rules of the language. A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node. If all the attribute values in a parse tree have been computed, the tree is said to be fully attributed. Although in practice it is not always done this way, it is convenient to think of attribute values as being computed after the complete unattributed parse tree has been constructed by the compiler.

3.5.4 Intrinsic Attributes

Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree. For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types. The contents of the symbol table are set based on earlier declaration statements. Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes. Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values.

3.5.5 Examples of Attribute Grammars

As a very simple example of how attribute grammars can be used to describe static semantics, consider the following fragment of an attribute grammar that describes the rule that the name on the end of an Ada procedure must match the procedure's name. (This rule cannot be stated in BNF.) The string attribute of <proc_name>, denoted by <proc_name>.string, is the actual string of characters that were found immediately following the reserved word procedure by the compiler. Notice that when there is more than one occurrence of a nonterminal in a syntax rule in an attribute grammar, the nonterminals are subscripted with brackets to distinguish them. Neither the subscripts nor the brackets are part of the described language.

Syntax rule: <proc_def> → **procedure** <proc_name>[1]

<proc_body> **end** <proc_name>[2];
Predicate: <proc_name>[1]string == <proc_name>[2].string

In this example, the predicate rule states that the name string attribute of the <proc_name> nonterminal in the subprogram header must match the name string attribute of the <proc_name> nonterminal following the end of the subprogram.

Next, we consider a larger example of an attribute grammar. In this case, the example illustrates how an attribute grammar can be used to check the type rules of a simple assignment statement. The syntax and static semantics of this assignment statement are as follows: The only variable names are A, B, and C. The right side of the assignments can be either a variable or an expression in the form of a variable added to another variable. The variables can be one of two types: int or real. When there are two variables on the right side of an assignment, they need not be the same type. The type of the expression when the operand types are not the same is always real. When they are the same, the expression type is that of the operands. The type of the left side of the assignment must match the type of the right side. So the types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type. The attribute grammar specifies these static semantic rules. The syntax portion of our example attribute grammar is

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

The attributes for the nonterminals in the example attribute grammar are described in the following paragraphs:

- *actual_type*—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr> nonterminal.
- *expected_type*—An inherited attribute associated with the nonterminal <expr>. It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

The complete attribute grammar follows in Example 6.

Example 6: An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicade: <expr>.actual_type == <expr>.expected_type:
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look- up(<var>.string)

The look- up function looks up a given variable name in the symbol table and returns the variable's type.

A parse tree of the sentence A = A + B generated by the grammar in Example 3.6 is shown in Figure 3.6. As in the grammar, bracketed numbers are added after the repeated node labels in the tree so they can be referenced unambiguously.

3.5.6 Evaluation

Checking the static semantic rules of a language is an essential part of all compilers. Even if a compiler writer has never heard of an attribute grammar, he or she would need to use their fundamental ideas to design the checks of static semantics rules for his or her compiler. One of the main difficulties in using an attribute grammar to describe all of the syntax and static semantics of a real contemporary programming language is the size and complexity of the attribute grammar. The large number of attributes and semantic rules required for a complete programming language make such grammars difficult to write and read. Furthermore, the attribute values on a large parse tree are costly to evaluate. On the other hand, less formal attribute grammars are a powerful and

commonly used tool for compiler writers, who are more interested in the process of producing a compiler than they are in formalism.

3.6 Describing the Meanings of Programs: Dynamic Semantics

We now turn to the difficult task of describing the dynamic semantics, or meaning, of the expressions, statements, and program units of a programming language. Because of the power and naturalness of the available notation, describing syntax is a relatively simple matter. On the other hand, no universally accepted notation or approach has been devised for dynamic semantics. In this section, we briefly describe several of the methods that have been developed. For the remainder of this section, when we use the term *semantics*, we mean dynamic semantics.

There are several different reasons underlying the need for a methodology and notation for describing semantics. Programmers obviously need to know precisely what the statements of a language do before they can use them effectively in their programs. Compiler writers must know exactly what language constructs mean to design implementations for them correctly. If there were a precise semantics specification of a programming language, programs written in the language potentially could be proven correct without testing. Also, compilers could be shown to produce programs that exhibited exactly the behavior given in the language definition; that is, their correctness could be verified. A complete specification of the syntax and semantics of a programming language could be used by a tool to generate a compiler for the language automatically.

Finally, language designers, who would develop the semantic descriptions of their languages, could in the process discover ambiguities and inconsistencies in their designs. Software developers and compiler designers typically determine the semantics of programming languages by reading English explanations in language manuals. Because such explanations are often imprecise and incomplete, this approach is clearly unsatisfactory. Due to the lack of complete semantics specifications of programming languages, programs are rarely proven correct without testing, and commercial compilers are never generated automatically from language descriptions.

3.6.1 Operational Semantics

The idea behind operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine. The effects on the machine are viewed as the

sequence of changes in its state, where the machine's state is the collection of the values in its storage. An obvious operational semantics description, then, is given by executing a compiled version of the program on a computer. Most programmers have, on at least one occasion, written a small test program to determine the meaning of some programming language construct, often while learning the language. Essentially, what such a programmer is doing is using operational semantics to determine the meaning of the construct.

There are several problems with using this approach for complete formal semantics descriptions. First, the individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous. Second, the storage of a real computer is too large and complex. There are usually several levels of memory devices, as well as connections to enumerable other computers and memory devices through networks. Therefore, machine languages and real computers are not used for formal operational semantics. Rather, intermediate-level languages and interpreters for idealized computers are designed specifically for the process. There are different levels of uses of operational semantics. At the highest level, the interest is in the final result of the execution of a complete program.

This is sometimes called natural operational semantics. At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed. This use is sometimes called structural operational semantics.

3.6.1.1 The Basic Process

The first step in creating an operational semantics description of a language is to design an appropriate intermediate language, where the primary desired characteristic of the language is clarity. Every construct of the intermediate language must have an obvious and unambiguous meaning. This language is at the intermediate level, because machine language is too low-level to be easily understood and another high-level language is obviously not suitable. If the semantics description is to be used for natural operational semantics, a virtual machine (an interpreter) must be constructed for the intermediate language.

The virtual machine can be used to execute either single statements, code segments, or whole programs. The semantics description can be used without a virtual machine if the meaning of a

single statement is all that is required. In this use, which is structural operational semantics, the intermediate code can be visually inspected.

The basic process of operational semantics is not unusual. In fact, the concept is frequently used in programming textbooks and programming language reference manuals. For example, the semantics of the C **for** construct can be described in terms of simpler statements, as in

C Statement	*Meaning*
for (expr1; expr2; expr3) . . . }	{ expr1; loop: **if** expr2 == 0 **goto** out . . . expr3; **goto** loop out: . . .

The human reader of such a description is the virtual computer and is assumed to be able to "execute" the instructions in the definition correctly and recognize the effects of the "execution." The intermediate language and its associated virtual machine used for formal operational semantics descriptions are often highly abstract. The intermediate language is meant to be convenient for the virtual machine, rather than for human readers. For our purposes, however, a more human-oriented intermediate language could be used. As such an example, consider the following list of statements, which would be adequate for describing the semantics of the simple control statements of a typical programming language:

ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var **goto** label

In these statements, relop is one of the relational operators from the set {=, <>, >, <, >=, <=}, ident is an identifier, and var is either an identifier or a constant. These statements are all simple and therefore easy to understand and implement.

A slight generalization of these three assignment statements allows more general arithmetic expressions and assignment statements to be described. The new statements are

ident = var bin_op var

ident = un_op var

where bin_op is a binary arithmetic operator and un_op is a unary operator. Multiple arithmetic data types and automatic type conversions, of course, complicate this generalization. Adding just a few more relatively simple instructions would allow the semantics of arrays, records, pointers, and subprograms to be described. using this intermediate language.

3.6.1.2 Evaluation

The first and most significant use of formal operational semantics was to describe the semantics of PL/I (Wegner, 1972). That particular abstract machine and the translation rules for PL/I were together named the Vienna Definition Language (VDL), after the city where IBM designed it. Operational semantics provides an effective means of describing semantics for language users and language implementors, as long as the descriptions are kept simple and informal. The VDL description of PL/I, unfortunately, is so complex that it serves no practical purpose.

Operational semantics depends on programming languages of lower levels, not mathematics. The statements of one programming language are described in terms of the statements of a lower-level programming language. This approach can lead to circularities, in which concepts are indirectly defined in terms of themselves. The methods described in the following two sections are much more formal, in the sense that they are based on mathematics and logic, not programming languages.

3.6.2 Denotational Semantics

Denotational semantics is the most rigorous and most widely known formal method for describing the meaning of programs. It is solidly based on recursive function theory. A thorough discussion of the use of denotational semantics to describe the semantics of programming languages is necessarily long and complex. It is our intent to provide the reader with an introduction to the central concepts of denotational semantics, along with a few simple examples that are relevant to programming language specifications.

The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object. Because the objects are rigorously defined, they model the exact meaning of their corresponding entities. The idea is

based on the fact that there are rigorous ways of manipulating mathematical objects but not programming language constructs. The difficulty with this method lies in creating the objects and the mapping functions. The method is named *denotational* because the mathematical objects denote the meaning of their corresponding syntactic entities.

The mapping functions of a denotational semantics programming language specification, like all functions in mathematics, have a domain and a range. The domain is the collection of values that are legitimate parameters to the function; the range is the collection of objects to which the parameters are mapped. In denotational semantics, the domain is called the syntactic domain, because it is syntactic structures that are mapped. The range is called the semantic domain.

Denotational semantics is related to operational semantics. In operational semantics, programming language constructs are translated into simpler programming language constructs, which become the basis of the meaning of the construct. In denotational semantics, programming language constructs are mapped to mathematical objects, either sets or, more often, functions. However, unlike operational semantics, denotational semantics does not model the step-by-step computational processing of programs.

3.6.2.1 Two Simple Examples

We use a very simple language construct, character string representations of binary numbers, to introduce the denotational method. The syntax of such binary numbers can be described by the following grammar rules:

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

A parse tree for the example binary number, 110, is shown in Figure 8. Notice that we put apostrophes around the syntactic digits to show they are not mathematical digits. This is similar to the relationship between ASCII coded digits and mathematical digits. When a program reads a number as a string, it must be converted to a mathematical number before it can be used as a value in the program.

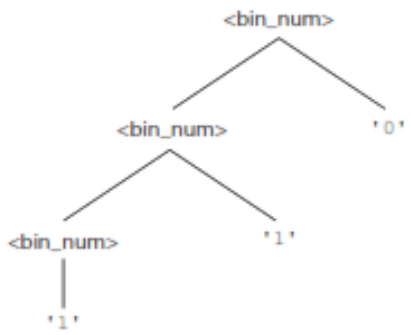


Figure 8

The syntactic domain of the mapping function for binary numbers is the set of all character string representations of binary numbers. The semantic domain is the set of nonnegative decimal numbers, symbolized by N. To describe the meaning of binary numbers using denotational semantics, we associate the actual meaning (a decimal number) with each rule that has a single terminal symbol as its RHS. In our example, decimal numbers must be associated with the first two grammar rules. The other two grammar rules are, in a sense, computational rules, because they combine a terminal symbol, to which an object can be associated, with a nonterminal, which can be expected to represent some construct. Presuming an evaluation that progresses upward in the parse tree, the nonterminal in the right side would already have its meaning attached. So, a syntax rule with a nonterminal as its RHS would require a function that computed the meaning of the LHS, which represents the meaning of the complete RHS. The semantic function, named Mbin, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers. The function Mbin is defined as follows:

Mbin('0') = 0

Mbin('1') = 1

Mbin(<bin_num> '0') = 2 * Mbin(<bin_num>)

Mbin(<bin_num> '1') = 2 * Mbin(<bin_num>) + 1

3.6.2.2 The State of a Program

The denotational semantics of a program could be defined in terms of state changes in an ideal computer. Operational semantics are defined in this way, and denotational semantics are defined

in nearly the same way. In a further simplification, however, denotational semantics is defined in terms of only the values of all of the program's variables. So, denotational semantics uses the state of the program to describe meaning, whereas operational semantics uses the state of a machine. The key difference between operational semantics and denotational semantics is that state changes in operational semantics are defined by coded algorithms, written in some programming language, whereas in denotational semantics, state changes are defined by mathematical functions. Let the state s of a program be represented as a set of ordered pairs, as follows:

s = {<i1, v1>, <i2, v2>, . . . , <i*n*, v*n*>}

Each i is the name of a variable, and the associated v's are the current values of those variables. Any of the v's can have the special value **undef**, which indicates that its associated variable is currently undefined. Let VARMAP be a function of two parameters: a variable name and the program state. The value of VARMAP (ij, s) is vj (the value paired with ij in state s). Most semantics mapping functions for programs and program constructs map states to states. These state changes are used to define the meanings of programs and program constructs. Some language constructs—for example, expressions—are mapped to values, not states.

3.6.2.3 Expressions

Expressions are fundamental to most programming languages. We assume here that expressions have no side effects. Furthermore, we deal with only very simple expressions: The only operators are + and *, and an expression can have at most one operator; the only operands are scalar integer variables and integer literals; there are no parentheses; and the value of an expression is an integer. Following is the BNF description of these expressions:

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

The only error we consider in expressions is a variable having an undefined value. Obviously, other errors can occur, but most of them are machine-dependent. Let Z be the set of integers, and let **error** be the error value. Then Z^U{**error**} is the semantic domain for the denotational specification for our expressions. The mapping function for a given expression E and state s

follows. To distinguish between mathematical function definitions and the assignment statements of programming languages, we use the symbol Δ = to define mathematical functions. The implication symbol, =>, used in this definition connects the form of an operand with its associated case (or switch) construct. Dot notation is used to refer to the child nodes of a node. For example, <binary_expr>.<left_expr> refers to the left child node of <binary_expr>.

```
Me(<expr>, s) Δ= case <expr> of
                    <dec_num>=>Mdec(<dec_num>, s)
                    <var> =>if VARMAP(<var>, s) == undef
                                then error
                                else VARMAP(<var>, s)
                      <binary_expr> =>
                      if(Me(<binary_expr>.<left_expr>,s) == undef OR
                        Me(<binary_expr>.<right_expr>, s) == undef)
                      then error
                      else if (<binary_expr>.<operator> == '+')
                                then Me(<binary_expr>.<left_expr>, s) +
                                   Me(<binary_expr>.<right_expr>, s)
                                else Me(<binary_expr>.<left_expr>, s) *
                                  Me(<binary_expr>.<right_expr>, s)
```

3.6.2.4 Assignment Statements

An assignment statement is an expression evaluation plus the setting of the target variable to the expression's value. In this case, the meaning function maps a state to a state. This function can be described with the following:

```
Ma(x = E, s) Δ= if Me(E, s) == error
                    then error
                    else s′ = {<i1, v1′ >, <i2, v2′>, . . . , <in, vn′>}, where
                            for j = 1, 2, . . . , n
                            if ij == x
                               then vj′ = Me(E, s)
                               else vj′ = VARMAP(ij, s)
```

Note that the comparison in the third last line above, ij == x, is of names, not values.

3.6.2.5 Logical Pretest Loops

The denotational semantics of a logical pretest loop is deceptively simple. To expedite the discussion, we assume that there are two other existing mapping functions, Msl and Mb, that map statement lists and states to states and Boolean expressions to Boolean values (or **error**), respectively. The function is

```
M1(while B do L, s) Δ= if Mb(B, s) == undef
                       then error
                       else if Mb(B, s) == false
                            then s
                            else if Msl(L, s) == error
                            then error
                            else M1(while B do L, Msl(L, s))
```

The meaning of the loop is simply the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors. In essence, the loop has been converted from iteration to recursion, where the recursion control is mathematically defined by other recursive state mapping functions. Recursion is easier to describe with mathematical rigor than iteration. One significant observation at this point is that this definition, like actual program loops, may compute nothing because of nontermination.

3.6.2.6 Evaluation

Objects and functions, such as those used in the earlier constructs, can be defined for the other syntactic entities of programming languages. When a complete system has been defined for a given language, it can be used to determine the meaning of complete programs in that language. This provides a framework for thinking about programming in a highly rigorous way. As stated previously, denotational semantics can be used as an aid to language design. For example, statements for which the denotational semantic description is complex and difficult may indicate to the designer that such statements may also be difficult for language users to understand and that an alternative design may be in order.

3.6.3 Axiomatic Semantics

Axiomatic semantics, thus named because it is based on mathematical logic, is the most abstract approach to semantics specification discussed in this chapter. Rather than directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program.

Recall that one of the possible uses of semantic specifications is to prove the correctness of programs.

In axiomatic semantics, there is no model of the state of a machine or program or model of state changes that take place when the program is executed. The meaning of a program is based on relationships among program variables and constants, which are the same for every execution of the program. Axiomatic semantics has two distinct applications: program verification and program semantics specification. This section focuses on program verification in its description of axiomatic semantics.

Axiomatic semantics was defined in conjunction with the development of an approach to proving the correctness of programs. Such correctness proofs, when they can be constructed, show that a program performs the computation described by its specification. In a proof, each statement of a program is both preceded and followed by a logical expression that specifies constraints on program variables. These, rather than the entire state of an abstract machine (as with operational semantics), are used to specify the meaning of the statement. The notation used to describe constraints—indeed, the language of axiomatic semantics—is predicate calculus. Although simple Boolean expressions are often adequate to express constraints, in some cases they are not. When axiomatic semantics is used to specify formally the meaning of a statement, the meaning is defined by the statement's effect on assertions about the data affected by the statement.

3.6.3.1 Assertions

The logical expressions used in axiomatic semantics are called predicates, or assertions. An assertion immediately preceding a program statement describes the constraints on the program variables at that point in the program. An assertion immediately following a statement describes the new constraints on those variables (and possibly others) after execution of the statement. These assertions are called the precondition and postcondition, respectively, of the statement. For two adjacent statements, the postcondition of the first serves as the precondition of the second. Developing an axiomatic description or proof of a given program requires that every statement in the program has both a precondition and a postcondition.

In the following sections, we examine assertions from the point of view that preconditions for statements are computed from given postconditions, although it is possible to consider these in the

opposite sense. We assume all variables are integer type. As a simple example, consider the following assignment statement and postcondition:

sum = 2 * x + 1 {sum > 1}

Precondition and postcondition assertions are presented in braces to distinguish them from parts of program statements. One possible precondition for this statement is {x > 10}. In axiomatic semantics, the meaning of a specific statement is defined by its precondition and its postcondition. In effect, the two assertions specify precisely the effect of executing the statement. The general concept of axiomatic semantics is to state precisely the meaning of statements and programs in terms of logic expressions. Program verification is one application of axiomatic descriptions of languages.

3.6.3.2 Weakest Preconditions

The weakest precondition is the least restrictive precondition that will guarantee the validity of the associated postcondition. For example, in the statement and postcondition given in Section 3.6.3.1, {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest of all preconditions in this case is {x > 0}.

If the weakest precondition can be computed from the most general postcondition for each of the statement types of a language, then the processes used to compute these preconditions provide a concise description of the semantics of that language. Furthermore, correctness proofs can be constructed for programs in that language. A program proof is begun by using the characteristics of the results of the program's execution as the postcondition of the last statement of the program. This postcondition, along with the last statement, is used to compute the weakest precondition for the last statement. This precondition is then used as the postcondition for the second last statement. This process continues until the beginning of the program is reached. At that point, the precondition of the first statement states the conditions under which the program will compute the desired results. If these conditions are implied by the input specification of the program, the program has been verified to be correct.

An inference rule is a method of inferring the truth of one assertion on the basis of the values of other assertions. The general form of an inference rule is as follows:

$$\frac{S1, S2, \ldots, Sn}{S}$$

This rule states that if S1, S2, . . . , and S*n* are true, then the truth of S can be inferred. The top part of an inference rule is called its antecedent; the bottom part is called its consequent. An axiom is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent. For some program statements, the computation of a weakest precondition from the statement and a postcondition is simple and can be specified by an axiom. In most cases, however, the weakest precondition can be specified only by an inference rule.

To use axiomatic semantics with a given programming language, whether for correctness proofs or for formal semantics specifications, either an axiom or an inference rule must exist for each kind of statement in the language. In the following subsections, we present an axiom for assignment statements and inference rules for statement sequences, selection statements, and logical pretest loop statements. Note that we assume that neither arithmetic nor Boolean expressions have side effects.

3.6.3.3 Assignment Statements

The precondition and postcondition of an assignment statement together define its meaning. To define the meaning of an assignment statement there must be a way to compute its precondition from its postcondition. Let x = E be a general assignment statement and Q be its postcondition. Then, its weakest precondition, P, is defined by the axiom

$P = Q_{x \rightarrow E}$

which means that P is computed as Q with all instances of x replaced by E. For example, if we have the assignment statement and postcondition

a = b / 2 - 1 {a < 10}

the weakest precondition is computed by substituting b / 2 - 1 for a in the postcondition {a < 10}, as follows:

b / 2 - 1 < 10

b < 22

Thus, the weakest precondition for the given assignment statement and postcondition is {b < 22}. Remember that the assignment axiom is guaranteed to be correct only in the absence of side effects. An assignment statement has a side effect if it changes some variable other than its target.

The usual notation for specifying the axiomatic semantics of a given statement form is

{P} S {Q}

where P is the precondition, Q is the postcondition, and S is the statement form. In the case of the assignment statement, the notation is

$\{Q_{x \rightarrow E}\}$ x = E{Q}

As another example of computing a precondition for an assignment statement, consider the following:

x = 2 * y - 3 {x > 25}

The precondition is computed as follows:

2 * y - 3 > 25

y > 14

So {y > 14} is the weakest precondition for this assignment statement and postcondition.

Note that the appearance of the left side of the assignment statement in its right side does not affect the process of computing the weakest precondition.

For example, for

x = x + y - 3 {x > 10}

the weakest precondition is

x + y - 3 > 10

y > 13 - x

Recall that axiomatic semantics was developed to prove the correctness of programs. In light of that, it is natural at this point to wonder how the axiom for assignment statements can be used to prove anything. Here is how: A given assignment statement with both a precondition and a postcondition can be considered a logical statement, or theorem. If the assignment axiom, when applied to the postcondition and the assignment statement, produces the given precondition, the theorem is proved. For example, consider the following logical statement:

{x > 3} x = x - 3 {x > 0}

Using the assignment axiom on the statement and its postcondition produces

{x > 3}, which is the given precondition. Therefore, we have proven the example logical statement.

3.6.3.4 Sequences

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule. Let S1 and S2 be adjacent program statements. If S1 and S2 have the following pre- and postconditions

{P} S1 {P2}

{P2} S2 {P3}

the inference rule for such a two-statement sequence is

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1, S2\ \{P3\}}$$

So, for our example, 5P16 S1; S2 5P36 describes the axiomatic semantics of the sequence S1; S2. The inference rule states that to get the sequence precondition, the precondition of the second statement is computed. This new assertion is then used as the postcondition of the first statement, which can then be used to compute the precondition of the first statement, which is also the precondition of the whole sequence. If S1 and S2 are the assignment statements

3.6.3.5 Selection

We next consider the inference rule for selection statements, the general form of which is

if B **then** S1 **else** S2

We consider only selections that include **else** clauses. The inference rule is

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S1\ \texttt{else}\ S2\ \{Q\}}$$

This rule specifies that selection statements must be proven both when the Boolean control expression is true and when it is false. The first logical statement above the line represents the **then**

clause; the second represents the **else** clause. According to the inference rule, we need a precondition P that can be used in the precondition of both the **then** and **else** clauses.

Consider the following example of the computation of the precondition using the selection inference rule. The example selection statement is

if x > 0 **then**

y = y - 1

else

y = y + 1

Suppose the postcondition, Q, for this selection statement is {y > 0}. We can use the axiom for assignment on the **then** clause

y = y - 1 {y > 0}

This produces {y - 1 > 0} or {y > 1}. It can be used as the P part of the precondition for the **then** clause. Now we apply the same axiom to the **else** clause

y = y + 1 {y > 0}

3.6.3.6 Logical Pretest Loops

Another essential construct of imperative programming languages is the logical pretest, or **while** loop. Computing the weakest precondition for a **while** loop is inherently more difficult than for a sequence, because the number of iterations cannot always be predetermined. In a case where the number of iterations is known, the loop can be unrolled and treated as a sequence.

The problem of computing the weakest precondition for loops is similar to the problem of proving a theorem about all positive integers. In the latter case, induction is normally used, and the same inductive method can be used for some loops. The principal step in induction is finding an inductive hypothesis. The corresponding step in the axiomatic semantics of a **while** loop is finding an assertion called a **loop invariant**, which is crucial to finding the weakest precondition. The inference rule for computing the precondition for a **while** loop is as follows:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{I \text{ and (not } B)\}}$$

In this rule, I is the loop invariant. This seems simple, but it is not. The complexity lies in finding an appropriate loop invariant. The axiomatic description of a **while** loop is written as
{P} **while** B **do** S **end** {Q}

The loop invariant must satisfy a number of requirements to be useful. First, the weakest precondition for the **while** loop must guarantee the truth of the loop invariant. In turn, the loop invariant must guarantee the truth of the postcondition upon loop termination. These constraints move us from the inference rule to the axiomatic description. During execution of the loop, the truth of the loop invariant must be unaffected by the evaluation of the loop-controlling Boolean expression and the loop body statements. Hence, the name *invariant*. Another complicating factor for **while** loops is the question of loop termination. A loop that does not terminate cannot be correct, and in fact computes nothing. If Q is the postcondition that holds immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates.

The complete axiomatic description of a **while** construct requires all of the following to be true, in which I is the loop invariant:
P => I
{I and B} S {I}
(I and (not B)) => Q
the loop terminates

Once again, the computed I can serve as P, and I passes the four requirements. Unlike our earlier example of finding a loop precondition, this one clearly is not a weakest precondition. Consider using the precondition {s > 1}. The logical statement
{s > 1} **while** s > 1 **do** s = s / 2 **end** {s = 1}

can easily be proven, and this precondition is significantly broader than the one computed earlier. The loop and precondition are satisfied for any positive value for s, not just powers of 2, as the process indicates. Because of the rule of consequence, using a precondition that is stronger than the weakest precondition does not invalidate a proof.

Finding loop invariants is not always easy. It is helpful to understand the nature of these invariants. First, a loop invariant is a weakened version of the loop postcondition and also a precondition for

the loop. So, I must be weak enough to be satisfied prior to the beginning of loop execution, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition. Because of the difficulty of proving loop termination, that requirement is often ignored. If loop termination can be shown, the axiomatic description of the loop is called total correctness. If the other conditions can be met but termination is not guaranteed, it is called partial correctness. In more complex loops, finding a suitable loop invariant, even for partial correctness, requires a good deal of ingenuity. Because computing the precondition for a **while** loop depends on finding a loop invariant, proving the correctness of programs with **while** loops using axiomatic semantics can be difficult.

3.6.3.7 Program Proofs

This section provides validations for two simple programs. The first example of a correctness proof is for a very short program, consisting of a sequence of three assignment statements that interchange the values of two variables.

{x = A AND y = B}

t = x;

x = y;

y = t;

{x = B AND y = A}

Because the program consists entirely of assignment statements in a sequence, the assignment axiom and the inference rule for sequences can be used to prove its correctness. The first step is to use the assignment axiom on the last statement and the postcondition for the whole program. This yields the precondition

{x = B AND t = A}

3.6.3.8 Evaluation

As stated previously, to define the semantics of a complete programming language using the axiomatic method, there must be an axiom or an inference rule for each statement type in the language. Defining axioms or inference rules for some of the statements of programming languages has proven to be a difficult task. An obvious solution to this problem is to design the language with the axiomatic method in mind, so that only statements for which axioms or inference

rules can be written are included. Unfortunately, such a language would necessarily leave out some useful and powerful parts.

Axiomatic semantics is a powerful tool for research into program correctness proofs, and it provides an excellent framework in which to reason about programs, both during their construction and later. Its usefulness in describing the meaning of programming languages to language users and compiler writers is, however, highly limited.

4 Self-Assessment Exercises

1. What is lexeme and token.
2. How are programming languages formally defined?
3. In which form is the programming language syntax commonly described?
4. What is an ambiguous grammar?
5. Explain the use of metasymbols in EBNFs.
6. What is the purpose of a predicate function?
7. How is the order of evaluation of attributes determined for the trees of a given attribute grammar?
8. What is the use of intrinsic attributes?
9. Describe the two levels of uses of operational semantics.
10. Explain the syntactic and semantic domains in denotational semantics?
11. What is an assertion in axiomatic semantics?
12. What two things must be defined for each language entity in order to construct a denotational description of the language?
13. Which part of an inference rule is the antecedent?
14. Write EBNF descriptions for the following:

a. A Java class definition header statement
b. A Java method call statement
c. A C switch statement
d. A C union definition
e. C float literals

15. Rewrite the BNF of Example 3.4 to give + precedence over * and force + to be right associative.

16. Using the grammar in Example 3.2, 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))

17. Using the grammar in Example 3.4, show a parse tree and a leftmost derivation for each of the following statements:

a. A = (A + B) * C

b. A = B + C + A

c. A = A * (B + C)

d. A = B * (C * (A + B))

18. Prove that the following grammar is ambiguous:

<S> → <A> <A> → <A> + <A> | <id> <id> → a | b | c

19. Describe, in English, the language defined by the following grammar:

<S> → <A> <B> <C> <A> → a <A> | a <B> → b <B> | b <C> → e

20. Consider the following grammar: <S> → <A> a <B> b <A> → <A> b | b <B> → b

Which of the following sentences are in the language generated by this grammar?

a. babb

b. bbbabb

c. bbaaaaabc

d. aaaaa

5 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 that describe static semantics rules.

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.

6 Summary

This unit discussed syntax of programming language and presented a discussion on general problem of describing syntax. Also, the formal methods of describing syntax such as Backus- Naur Form (BNF) and context- free grammars, Extended BNF, grammar and recognizers were deliberated on. The attribute grammars, which can be used to describe both the syntax and static semantics of programming languages were briefly discussed. Furthermore, the unit provided 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.

7 References/Further Reading

Sebesta, R. W. (2016). Concepts of Programming Languages (Eleventh Edition). Pearson Education Limited.

Sebesta, R. W. (2009). Concepts of Programming Languages (Tenth Edition). Pearson Education Limited.

Jaemin Hong and Sukyoung Ryu (2010) *Introduction to Programming Languages*

Ghezzi & Jazayeri (1996.) *Programming language concepts—Third edition* John Wiley & Sons New York Chichester Brisbane Toronto Singapore 1996.

Unit 3 Lexical Analysis and Parsing

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Lexical Analysis
   - 3.2. Building Lexical Analyzer
   - 3.3. The Parsing Problem
     - 3.3.1. Introduction to Parsing
     - 3.3.2. Top-Down Parsers
     - 3.3.3. Bottom-Up Parsers
     - 3.3.4. The Complexity of Parsing
   - 3.4. Recursive-Descent Parsing
     - 3.4.1. The Recursive-Decent Parsing Process
     - 3.4.2. The LL Grammar Class
   - 3.5. Bottom-Up Parsing
     - 3.5.1. The Parsing problem for Bottom-up Parsers
     - 3.5.2. Shift-Reduce Algorithm
     - 3.5.3. LR Parsers
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 3 Lexical Analysis and Parsing

1 Introduction

The syntax analyzer is the heart of a compiler, because several other important components, including the semantic analyzer and the intermediate code generator, are driven by the actions of the syntax analyzer. Syntax analyzers are based directly on the grammars as discussed in Module 2 unit 1 and 2 thus, it is necessary to discuss them as an application of grammars. Many applications, among them program listing formatters, programs that compute the complexity of programs, and programs that must analyze and react to the contents of a configuration file, all need to do lexical and syntax analyses. Therefore, lexical and syntax analyses are important topics for software developers, even if they never need to write a compiler. This unit discuss extensively on lexical analysis with focus on lexical process and building lexical analyzer. Also, the unit discusses the parsing problem, recursive-decent parsing and bottom-up parsing.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Explain lexical analysis
- Discuss parsing and parsing algorithm
- Understand the implementation process of recursive-decent parsing

3 Main Content

3.1 Lexical Analysis

A lexical analyzer is essentially a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern. Pattern matching is a traditional part of computing. One of the earliest uses of pattern matching was with text editors, such as the ed line editor, which was introduced in an early version of UNIX. Since then, pattern matching has found its way into some programming languages—for example, Perl and JavaScript. It is also available through the standard class libraries of Java, C++, and C#. A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis.

A lexical analyzer performs syntax analysis at the lowest level of program structure. An input program appears to a compiler as a single string of characters. The lexical analyzer collects

characters into logical groupings and assigns internal codes to the groupings according to their structure. In Chapter 3, these logical groupings are named **lexemes**, and the internal codes for categories of these groupings are named **tokens**. Lexemes are recognized by matching the input character string against character string patterns. Although tokens are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants.

Consider the following example of an assignment statement:

```
result = oldsum - value / 100;
```

Following are the tokens and lexemes of this statement:

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes. Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer. So, each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also, the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user.

3.2 Building Lexical Analyzer

There are three approaches to building a lexical analyzer:

- Write a formal description of the token patterns of the language using a descriptive language related to regular expressions.1 These descriptions are used as input to a software tool that automatically generates a lexical analyzer. There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.
- Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
- Design a state transition diagram that describes the token patterns of the language and hand-
- Construct a table-driven implementation of the state diagram.

A state transition diagram, or just state diagram, is a directed graph. The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states. An arc may also include actions the lexical analyzer must perform when the transition is taken.

State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognize members of a class of languages called regular languages. Regular grammars are generative devices for regular languages. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton. We now illustrate lexical-analyzer construction with a state diagram and the code that implements it. The state diagram could simply include states and transitions for each and every token pattern. However, that approach results in a very large and complex diagram, because every node in the state diagram would need a transition for every character in the character set of the language being analyzed. We therefore consider ways to simplify it.

Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands. Assume that the variable names consist of strings of uppercase letters, lowercase letters, and digits but must begin with a letter. Names have no length limitation. The first thing to observe is that there are 52 different characters (any uppercase or lowercase letter) that can begin a name, which would require 52 transitions from the transition

diagram's initial state. However, a lexical analyzer is interested only in determining that it is a name and is not concerned with which specific name it happens to be. Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name.

Another opportunity for simplifying the transition diagram is with the integer literal tokens. There are 10 different characters that could begin an integer literal lexeme. This would require 10 transitions from the start state of the state diagram. Because specific digits are not a concern of the lexical analyzer, we can build a much more compact state diagram if we define a character class named DIGIT for digits and use a single transition on any character in this character class to a state that collects integer literals.

Because our names can include digits, the transition from the node following the first character of a name can use a single transition on LETTER or DIGIT to continue collecting the characters of a name. Next, we define some utility subprograms for the common tasks inside the lexical analyzer. First, we need a subprogram, which we can name getChar, that has several duties. When called, getChar gets the next character of input from the input program and puts it in the global variable nextChar. getChar also must determine the character class of the input character and put it in the global variable charClass. The lexeme being built by the lexical analyzer, which could be implemented as a character string or an array, will be named lexeme.

We implement the process of putting the character in nextChar into the string array lexeme in a subprogram named addChar. This subprogram must be explicitly called because programs include some characters that need not be put in lexeme, for example the white-space characters between lexemes. In a more realistic lexical analyzer, comments also would not be placed in lexeme. When the lexical analyzer is called, it is convenient if the next character of input is the first character of the next lexeme. Because of this, a function named getNonBlank is used to skip white space every time the analyzer is called. Finally, a subprogram named lookup is needed to compute the token code for the single-character tokens. In our example, these are parentheses and the arithmetic operators. Token codes are numbers arbitrarily assigned to tokens by the compiler writer.

The state diagram in Figure 9 describes the patterns for our tokens. It includes the actions required on each transition of the state diagram. The following is a C implementation of a lexical analyzer specified in the state diagram of Figure 9, including a main driver function for testing.

purposes:

```
/* front.c - a lexical analyzer system for simple arithmetic expressions */
#include <stdio.h>
#include <ctype.h>
/* Global declarations */
/* Variables */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token; int nextToken;
FILE *in_fp, *fopen();
```

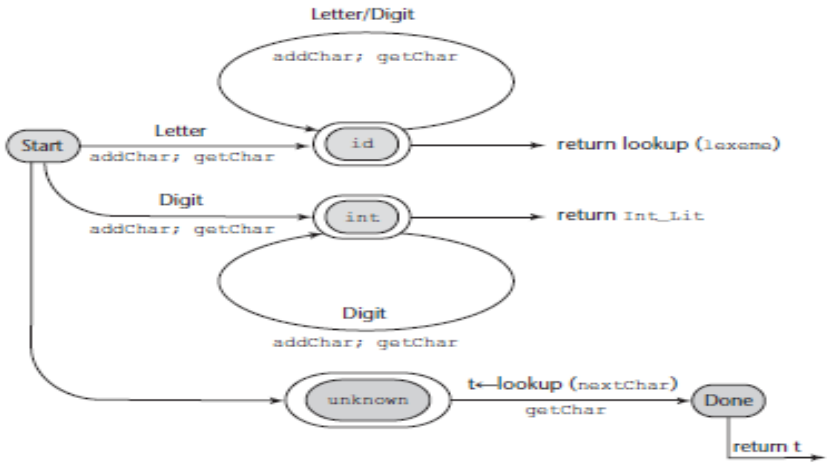


Figure 9: A state diagram to recognize names, parentheses and arithmetic operators

3.3 The Parsing Problem

The part of the process of analyzing syntax that is referred to as *syntax analysis* is often called parsing. We will use these two interchangeably. This section discusses the general parsing problem

and introduces the two main categories of parsing algorithms, top-down and bottom-up, as well as the complexity of the parsing process.

3.3.1 Introduction to Parsing

Parsers for programming languages construct parse trees for given programs. In some cases, the parse tree is only implicitly constructed, meaning that perhaps only a traversal of the tree is generated. But in all cases, the information required to build the parse tree is created during the parse. Both parse trees and derivations include all of the syntactic information needed by a language processor.

There are two distinct goals of syntax analysis: First, the syntax analyzer must check the input program to determine whether it is syntactically correct. When an error is found, the analyzer must produce a diagnostic message and recover. In this case, recovery means it must get back to a normal state and continue its analysis of the input program. This step is required so that the compiler finds as many errors as possible during a single analysis of the input program. If it is not done well, error recovery may create more errors, or at least more error messages. The second goal of syntax analysis is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input. The parse tree (or its trace) is used as the basis for translation.

Parsers are categorized according to the direction in which they build parse trees. The two broad classes of parsers are top-down, in which the tree is built from the root downward to the leaves, and bottom-up, in which the parse tree is built from the leaves upward to the root.

In this unit, we use a small set of notational conventions for grammar symbols and strings to make the discussion less cluttered. For formal languages, they are as follows:

- Terminal symbols—lowercase letters at the beginning of the alphabet (a, b, . . .)
- Nonterminal symbols—uppercase letters at the beginning of the alphabet (A, B, . . .)
- Terminals or nonterminals—uppercase letters at the end of the alphabet (W, X, Y, Z)
- Strings of terminals—lowercase letters at the end of the alphabet (w, x, y, z)
- Mixed strings (terminals and/or nonterminals)—lowercase Greek letters (a, b, d, g)

For programming languages, terminal symbols are the small-scale syntactic constructs of the language, what we have referred to as lexemes. The nonterminal symbols of programming

languages are usually connotative names or abbreviations, surrounded by angle brackets—for example, <while_statement>, <expr>, and <function_def>. The sentences of a language (programs, in the case of a programming language) are strings of terminals. Mixed strings describe right-hand sides (RHSs) of grammar rules and are used in parsing algorithms.

3.3.2 Top-Down Parsers

A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root. Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation.

In terms of the derivation, a top-down parser can be described as follows:
Given a sentential form that is part of a leftmost derivation, the parser's task is to find the next sentential form in that leftmost derivation. The general form of a left sentential form is xAa, whereby our notational conventions x is a string of terminal symbols, A is a nonterminal, and a is a mixed string. Because x contains only terminals, A is the leftmost nonterminal in the sentential form, so it is the one that must be expanded to get the next sentential form in a leftmost derivation. Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its LHS. For example, if the current sentential form is xAa and the A-rules are A→bB, A→cBb, and A→a, a top- down parser must choose among these three rules to get the next sentential form, which could be xbBa, xcBba, or xaa. This is the parsing decision problem for top-down parsers.

Different top-down parsing algorithms use different information to make parsing decisions. The most common top-down parsers choose the correct RHS for the leftmost nonterminal in the current sentential form by comparing the next token of input with the first symbols that can be generated by the RHSs of those rules. Whichever RHS has that token at the left end of the string it generates is the correct one. So, in the sentential form xAa, the parser would use whatever token would be the first generated by A to determine which A-rule should be used to get the next sentential form. In the example above, the three RHSs of the A-rules all begin with different terminal symbols. The parser can easily choose the correct RHS based on the next token of input, which must be a, b, or c in this example. In general, choosing the correct RHS is not so straightforward, because some of the RHSs of the leftmost nonterminal in the current sentential form may begin with a nonterminal.

The most common top-down parsing algorithms are closely related. A recursive-descent parser is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language. The most common alternative to recursive descent is to use a parsing table, rather than code, to implement the BNF rules. Both of these, which are called LL algorithms, are equally powerful, meaning they work on the same subset of all context-free grammars. The first L in LL specifies a left-to-right scan of the input; the second L specifies that a leftmost derivation is generated.

3.3.3 Bottom-Up Parsers

A bottom- up parser constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation. That is, the sentential forms of the derivation are produced in order of last to first. In terms of the derivation, a bottom-Up parser can be described as follows: Given a right sentential form α, the parser must determine what substring of α is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation. For example, the first step for a bottom-up parser is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS to get the second last sentential form in the derivation.

The process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS from the grammar of the language being parsed. The correct RHS is called the **handle**. A right sentential form is a sentential form that appears in a rightmost derivation. Consider the following grammar and derivation:

$S \rightarrow aAc$

$A \rightarrow aA \mid b$

$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$

A bottom-up parser of this sentence, aabc, starts with the sentence and must find the handle in it. In this example, this is an easy task, for the string contains only one RHS, b. When the parser replaces b with its LHS, A, it gets the second to last sentential form in the derivation, aaAc. In the general case, as stated previously, finding the handle is much more difficult, because a sentential form may include several different RHSs.

A bottom-up parser finds the handle of a given right sentential form by examining the symbols on one or both sides of a possible handle. Symbols to the right of the possible handle are usually tokens in the input that have not yet been analyzed. The most common bottom-up parsing algorithms are in the LR family, where the L specifies a left-to-right scan of the input and the R specifies that a rightmost derivation is generated.

3.3.4 The Complexity of Parsing

Parsing algorithms that work for any unambiguous grammar are complicated and inefficient. In fact, the complexity of such algorithms is O(n3), which means the amount of time they take is on the order of the cube of the length of the string to be parsed. This relatively large amount of time is required because these algorithms frequently must back up and reparse part of the sentence being analyzed. Reparsing is required when the parser has made a mistake in the parsing process. Backing up the parser also requires that part of the parse tree being constructed (or its trace) must be dismantled and rebuilt. O(n3) algorithms are normally not useful for practical processes, such as syntax analysis for a compiler, because they are far too slow. In situations such as this, computer scientists often search for algorithms that are faster, though less general. Generality is traded for efficiency. In terms of parsing, faster algorithms have been found that work for only a subset of the set of all possible grammars. These algorithms are acceptable as long as the subset includes grammars that describe programming languages. (Actually, as discussed in Chapter 3, the whole class of context-free grammars is not adequate to describe all of the syntax of most programming languages.) All algorithms used for the syntax analyzers of commercial compilers have complexity O(n), which means the time they take is linearly related to the length of the string to be parsed. This is vastly more efficient than O(n3) algorithms.

3.4 Recursive-Descent Parsing

This section introduces the recursive-descent top-down parser implementation process.

3.4.1 The Recursive-Descent Parsing Process

A recursive-descent parser is so named because it consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order. This recursion is a reflection of the nature of programming languages, which include several different kinds of nested structures. For example, statements are often nested in other statements. Also, parentheses in expressions

must be properly nested. The syntax of these structures is naturally described with recursive grammar rules.

EBNF is ideally suited for recursive-descent parsers. Recall from Chapter 3 that the primary EBNF extensions are braces, which specify that what they enclose can appear zero or more times, and brackets, which specify that what they enclose can appear once or not at all. Note that in both cases, the enclosedsymbols are optional. Consider the following examples:

<if_statement> → **if** <logic_expr> <statement> [**else** <statement>]
<ident_list> → ident {, ident}

In the first rule, the **else** clause of an **if** statement is optional. In the second, an <ident_list> is an identifier, followed by zero or more repetitions of a comma and an identifier.

A recursive-descent parser has a subprogram for each nonterminal in its associated grammar. The responsibility of the subprogram associated with a particular nonterminal is as follows: When given an input string, it traces out the parse tree that can be rooted at that nonterminal and whose leaves match the input string. In effect, a recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated nonterminal. Consider the following EBNF description of simple arithmetic expressions:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

Recall from Chapter 3 that an EBNF grammar for arithmetic expressions, such as this one, does not force any associativity rule. Therefore, when using such a grammar as the basis for a compiler, one must take care to ensure that the code generation process, which is normally driven by syntax analysis, produces code that adheres to the associativity rules of the language. This can be done easily when recursive-descent parsing is used.

In the following recursive-descent function, expr, the lexical analyzer is the function that is implemented in Section 4.2. It gets the next lexeme and puts its token code in the global variable nextToken. The token codes are defined as named constants, as in Section 4.2.

A recursive-descent subprogram for a rule with a single RHS is relatively simple. For each terminal symbol in the RHS, that terminal symbol is compared with nextToken. If they do not match, it is

a syntax error. If they match, the lexical analyzer is called to get the next input token. For each nonterminal, the parsing subprogram for that nonterminal is called.

The recursive-descent subprogram for the first rule in the previous example grammar, written in C, is

```
/*    expr
      Parses strings in the language generated by the rule:
      <expr> -> <term> {(+ | -) <term>}
      */
 void expr() {
      printf("Enter <expr>\n");

 /* Parse the first term */
     term();

 /*   As long as the next token is + or -, get
      the next token and parse the next term */
 while (nextToken == ADD_OP || nextToken == SUB_OP) {
      lex();
      term();
   }
   printf("Exit <expr>\n");
} /* End of function expr */
```

Recursive-descent parsing subprograms are written with the convention that each one leaves the next token of input in nextToken. So, whenever a parsing function begins, it assumes that nextToken has the code for the leftmost token of the input that has not yet been used in the parsing process.

The part of the language that the expr function parses consists of one or more terms, separated by either plus or minus operators. This is the language generated by the nonterminal <expr>. Therefore, first it calls the function that parses terms (term). Then it continues to call that function as long as it finds ADD_OP or SUB_OP tokens (which it passes over by calling lex). This recursive-descent function is simpler than most, because its associated rule has only one RHS. Furthermore, it does not include any code for syntax error detection or recovery, because there are no detectable errors associated with the grammar rule.

A recursive-descent parsing subprogram for a nonterminal whose rule has more than one RHS begins with code to determine which RHS is to be parsed. Each RHS is examined (at compiler construction time) to determine the set of terminal symbols that can appear at the beginning of sentences it can generate. By matching these sets against the next token of input, the parser can choose the correct RHS.

The parsing subprogram for <term> is similar to that for <expr>:

```
/*    term
      Parses strings in the language generated by the rule:
      <term> -> <factor> {(* | /) <factor>)
      */
void term() {
  printf("Enter <term>\n");

/* Parse the first factor */
    factor();

/* As long as the next token is * or /, get the
    next token and parse the next factor */
 while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
} /* End of function term */
```

The function for the <factor> nonterminal of our arithmetic expression grammar must choose between its two RHSs. It also includes error detection. In the function for <factor>, the reaction to detecting a syntax error is simply to call the error function. In a real parser, a diagnostic message must be produced when an error is detected. Furthermore, parsers must recover from the error so that the parsing process can continue.

```
/*    factor
       Parses strings in the language generated by the rule:
       <factor> -> id | int_constant | ( <expr )
       */
void factor() {
     printf("Enter <factor>\n");
```

```
/* Determine which RHS */
   if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
      lex();
/* If the RHS is ( <expr>), call lex to pass over the left parenthesis, call expr, and check for the
      right parenthesis */
else {
   if (nextToken == LEFT_PAREN) {
         lex();
         expr();
         if (nextToken == RIGHT_PAREN)
            lex();
         else
            error();
    } /* End of if (nextToken == ... */

/* It was not an id, an integer literal, or a left parenthesis */
         else
            error();
} /* End of else */ printf("Exit <factor>\n");;
} /* End of function factor */

 printf("Exit <factor>\n");;
} /* End of function factor */
```

Following is the trace of the parse of the example expression (sum + 47) / total, using the parsing functions expr, term, and factor, and the function lex from Section 4.2. Note that the parse begins by calling lex and the start symbol routine, in this case, expr.

```
Next token is: 25 Next lexeme is ( Enter <expr> Enter <term> Enter <factor>
Next token is: 11 Next lexeme is sum Enter <expr> Enter <term> Enter <factor>
Next token is: 21 Next lexeme is + Exit <factor> Exit <term>
Next token is: 10 Next lexeme is 47 Enter <term> Enter <factor>
Next token is: 26 Next lexeme is ) Exit <factor> Exit <term> Exit <expr>
Next token is: 24 Next lexeme is / Exit <factor>
Next token is: 11 Next lexeme is total Enter <factor>
Next token is: -1 Next lexeme is EOF Exit <factor> Exit <term> Exit <expr>
```

The parse tree traced by the parser for the preceding expression is shown in Figure 10

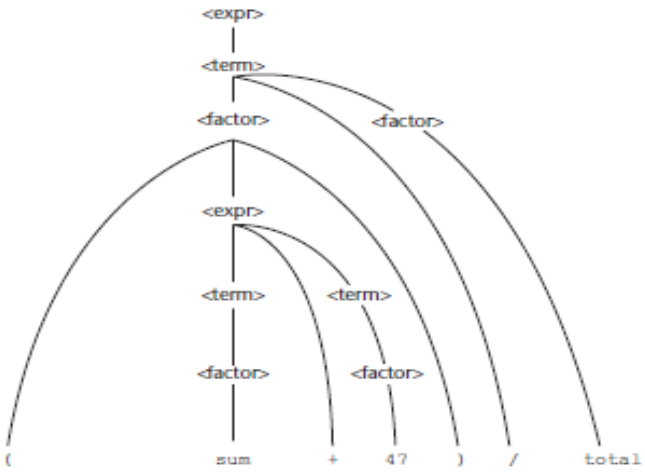


Figure 10: Parse tree for (Sum + 47)/total

3.4.2 The LL Grammar Class

Before choosing to use recursive descent as a parsing strategy for a compiler or other program analysis tool, one must consider the limitations of the approach, in terms of grammar restrictions. One simple grammar characteristic that causes a catastrophic problem for LL parsers is left recursion. For example, consider the following rule:

$A \rightarrow A + B$

A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS. That activation of the A parser subprogram then immediately calls itself again, and again, and so forth. It is easy to see that this leads nowhere (except to stack overflow). The left recursion in the rule $A \rightarrow A + B$ is called direct left recursion, because it occurs in one rule. Direct left recursion can be eliminated from a grammar by the following process:

For each nonterminal, A,

1. Group the A-rules as $A \rightarrow A\alpha_1, |\ldots| A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ where none of the β's begins with A

2. Replace the original A-rules with

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \varepsilon$

Note that e specifies the empty string. A rule that has e as its RHS is called an *erasure rule*, because its use in a derivation effectively erases its LHS from the sentential form. Consider the following example grammar and the application of the above process:

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

For the E-rules, we have α_1 = + T and β = T, so we replace the E-rules with

E →T E′
E′ → + T E′ | ε

For the T-rules, we have α_1 = *F and β = F, so we replace the T-rules with

T →F T′
T′ →* F T′ | ε

Because there is no left recursion in the F-rules, they remain the same, so the complete replacement grammar is

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

This grammar generates the same language as the original grammar but is not left recursive. As was the case with the expression grammar written using EBNF in Section 3.3.1, this grammar does not specify left associativity of operators. However, it is relatively easy to design the code generation based on this grammar so that the addition and multiplication operators will have left associativity. Indirect left recursion poses the same problem as direct left recursion.

3.5 Bottom-Up Parsing

3.5.1 The Parsing Problem for Bottom-Up Parsers

Consider the following grammar for arithmetic expressions:

E → E + T | T
T →T * F | F

F → (E) | id

Notice that this grammar generates the same arithmetic expressions as the example in Section 4.4. The difference is that this grammar is left recursive, which is acceptable to bottom-up parsers. Also note that grammars for bottom-up parsers normally do not include metasymbols such as those used to specify extensions to BNF. The following rightmost derivation illustrates this grammar:

E => E + T
=> E + T * F
=> E + T * id
=> E + F * id
=> E + id * id
=> T + id * id
=> F + id * id
=> id + id * id

The underlined part of each sentential form in this derivation is the RHS that is rewritten as its corresponding LHS to get the previous sentential form. The process of bottom-up parsing produces the reverse of a rightmost derivation. So, in the example derivation, a bottom-up parser starts with the last sentential form (the input sentence) and produces the sequence of sentential forms from there until all that remains is the start symbol, which in this grammar is E. In each step, the task of the bottom- up parser is to find the specific RHS, the handle, in the sentential form that must be rewritten to get the next (previous) sentential form. As mentioned earlier, a right sentential form may include more than one RHS. For example, the right sentential form

E + T * id

includes three RHSs, E + T, T, and id. Only one of these is the handle. For example, if the RHS E + T were chosen to be rewritten in this sentential form, the resulting sentential form would be E * id, but E * id is not a legal right sentential form for the given grammar.

The handle of a right sentential form is unique. The task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar. Formally, handle is defined as follows:

Definition: β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

In this definition, => rm specifies a rightmost derivation step, and => *rm specifies zero or more rightmost derivation steps. Although the definition of a handle is mathematically concise, it provides little help in finding the handle of a given right sentential form. In the following, we provide the definitions of several substrings of sentential forms that are related to handles. The purpose of these is to provide some intuition about handles.

Definition: β is a **phrase** of the right sentential form γ if and only if
$S => *\gamma = \alpha_1 A \alpha_2 => + \alpha_1 \beta \alpha_2$

In this definition, => + means one or more derivation steps.

Definition: β is a **simple phrase** of the right sentential form γ if and only if $S => *\gamma = \alpha_1 A \alpha_2 => + \alpha_1 \beta \alpha_2$

If these two definitions are compared carefully, it is clear that they differ only in the last derivation specification. The definition of phrase uses one or more steps, while the definition of simple phrase uses exactly one step.

The definitions of phrase and simple phrase may appear to have the same lack of practical value as that of a handle, but that is not true. Consider what a phrase is relative to a parse tree. It is the string of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree. A simple phrase is just a phrase that takes a single derivation step from its root nonterminal node. In terms of a parse tree, a phrase can be derived from a single nonterminal in one or more tree levels, but a simple phrase can be derived in just a single tree level. Consider the parse tree shown in Figure 11.

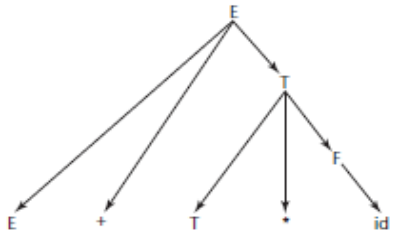


Figure 11: A parse tree fpr W + T * id

The leaves of the parse tree in Figure 4.3 comprise the sentential form E + T * id. Because there are three internal nodes, there are three phrases. Each internal node is the root of a subtree, whose leaves are a phrase. The root node of the whole parse tree, E, generates all of the resulting sentential

form, E + T * id, which is a phrase. The internal node, T, generates the leaves T * id, which is another phrase. Finally, the internal node, F, generates id, which is also a phrase. So, the phrases of the sentential form E + T * id are E + T * id, T * id, and id. Notice that phrases are not necessarily RHSs in the underlying grammar.

The simple phrases are a subset of the phrases. In the previous example, the only simple phrase is id. A simple phrase is always a RHS in the grammar. The reason for discussing phrases and simple phrases is this: The handle of any rightmost sentential form is its leftmost simple phrase. So now we have a highly intuitive way to find the handle of any right sentential form, assuming we have the grammar and can draw a parse tree. This approach to finding handles is of course not practical for a parser. (If you already have a parse tree, why do you need a parser?) Its only purpose is to provide the reader with some intuitive feel for what a handle is, relative to a parse tree, which is easier than trying to think about handles in terms of sentential forms.

We can now consider bottom-up parsing in terms of parse trees, although the purpose of a parser is to produce a parse tree. Given the parse tree for an entire sentence, you easily can find the handle, which is the first thing to rewrite in the sentence to get the previous sentential form. Then the handle can be pruned from the parse tree and the process repeated. Continuing to the root of the parse tree, the entire rightmost derivation can be constructed.

3.5.2 Shift-Reduce Algorithms

Bottom-up parsers are often called shift-reduce algorithms, because shift and reduce are the two most common actions they specify. An integral part of every bottom-up parser is a stack. As with other parsers, the input to a bottom-up parser is the stream of tokens of a program and the output is a sequence of grammar rules. The shift action moves the next input token onto the parser's stack. A reduce action replaces an RHS (the handle) on top of the parser's stack by its corresponding LHS. Every parser for a programming language is a pushdown automaton (PDA), because a PDA is a recognizer for a context-free language. You need not be intimate with PDAs to understand how a bottom-up parser works, although it helps. A PDA is a very simple mathematical machine

that scans strings of symbols from left to right. A PDA is so named because it uses a pushdown stack as its memory. PDAs can be used as recognizers for context-free languages. Given a string of symbols over the alphabet of a context-free language, a PDA that is designed for the purpose can determine whether the string is or is not a sentence in the language. In the process, the PDA can produce the information needed to construct a parse tree for the sentence.

With a PDA, the input string is examined, one symbol at a time, left to right. The input is treated very much as if it were stored in another stack, because the PDA never sees more than the leftmost symbol of the input. Note that a recursive-descent parser is also a PDA. In that case, the stack is that of the run-time system, which records subprogram calls (among other things), which correspond to the nonterminals of the grammar.

3.5.3 LR Parsers

Many different bottom-up parsing algorithms have been devised. Most of them are variations of a process called LR. LR parsers use a relatively small program and a parsing table that is built for a specific programming language. This algorithm, which is sometimes called canonical LR, was not used in the years immediately following its publication because producing the required parsing table required large amounts of computer time and memory. These are characterized by two properties: (1) They require far less computer resources to produce the required parsing table than the canonical LR algorithm, and (2) they work on smaller classes of grammars than the canonical LR algorithm.

There are three advantages to LR parsers:

- They can be built for all programming languages.
- They can detect syntax errors as soon as it is possible in a left-to-right scan.
- The LR class of grammars is a proper superset of the class parsable by LL parsers (for example, many left recursive grammars are LR, but none are LL).

The only disadvantage of LR parsing is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language.

Prior to the appearance of the LR parsing algorithm, there were a number of parsing algorithms that found handles of right sentential forms by looking both to the left and to the right of the substring of the sentential form that was suspected of being the handle.

4 Self-Assessment Exercises

i. What are the reasons why the use of BNF is advantageous over using an informal syntax description?
ii. How does a lexical analyzer serve as the front end of a syntax analyzer?
iii. Define *finite automata* and *regular grammar*.
iv. How can you construct a lexical analyzer with a state diagram?
v. Describe briefly the three approaches to building a lexical analyzer.
vi. What are the different grammar symbols for formal languages?
vii. Why are character classes used, rather than individual characters, for the letter and digit transitions of a state diagram for a lexical analyzer?
viii. What are the two distinct goals of syntax analysis?
ix. Describe the complexity of parsing algorithms.
x. Describe the recursive-descent parser.
xi. What do the two Ls in LL algorithm specify?
xii. State and explain the convention followed for writing a recursive-descent parsing subprogram.
xiii. State the advantages and disadvantage of LR parsing

5 Conclusion

Although there is terminology confusion between lexical analysis and syntax analysis but nearly all compilers separate the task of analyzing syntax into two parts, lexical analysis and syntax analysis. The lexical analyzer deals with small-Scale language constructs, such as names and numeric literals while the syntax analyzer deals with the large-scale constructs, such as expressions, statements, and program units. There are three reasons why lexical analysis is separated from syntax analysis because of its simplicity, efficiency and portability. However, the syntax analyzer can be platform independent and it is always good to isolate machine-dependent parts of any software system.

6 Summary

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. This unit discussed lexical process and how to build lexical analyzer. Also, discussed the parsing problem, recursive-decent parsing and bottom-up parsing.

7 References/Further Reading

Sebesta, R. W. (2016). Concepts of Programming Languages (Eleventh Edition). Pearson Education Limited.

Sebesta, R. W. (2009). Concepts of Programming Languages (Tenth Edition). Pearson Education Limited.

Unit 4 Language Processing

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Interpretation
   - 3.2. Translation
   - 3.3. Concept of Interpretative Language
   - 3.4. The Concept of Binding
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 4 Language Processing

1 Introduction

Machine languages are designed on the basis of speed of execution, cost of realization, and flexibility in building new software layers upon them. On the other hand, programming languages often are designed on the basis of the ease and reliability of programming. A basic problem, then, is how a higher level language eventually can be executed on a computer whose machine language is very different and at a much lower level. Thus, this unit focus on implementation of language processing by discussing interpretation, translation concept of interpretative language and binding.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Understand how constructs of the language are executed directly
- Understand how program are translated into an equivalent machine language before being executed
- Differentiate between compilers and interpreter
- Understand the concept of binding

3 Main Content

3.1 Interpretation

In this solution, the actions implied by the constructs of the language are executed directly (see Figures 12). Usually, for each possible action there exists a subprogram–written in machine language–to execute the action. Thus, interpretation of a program is accomplished by calling subprograms in the appropriate sequence.

More precisely, an interpreter is a program that repeatedly executes the following sequence.

- Get the next statement;
- Determine the actions to be executed;
- Perform the actions;

This sequence is very similar to the pattern of actions carried out by a traditional computer, that is:

- Fetch the next instruction (i.e., the instruction whose address is specified by the instruction pointer).

- Advance the instruction pointer (i.e., set the address of the instruction to be fetched next).
- Decode the fetched instruction.
- Execute the instruction.

This similarity shows that interpretation can be viewed as a simulation, on a host computer, of a special-purpose machine whose machine language is the higher level language.

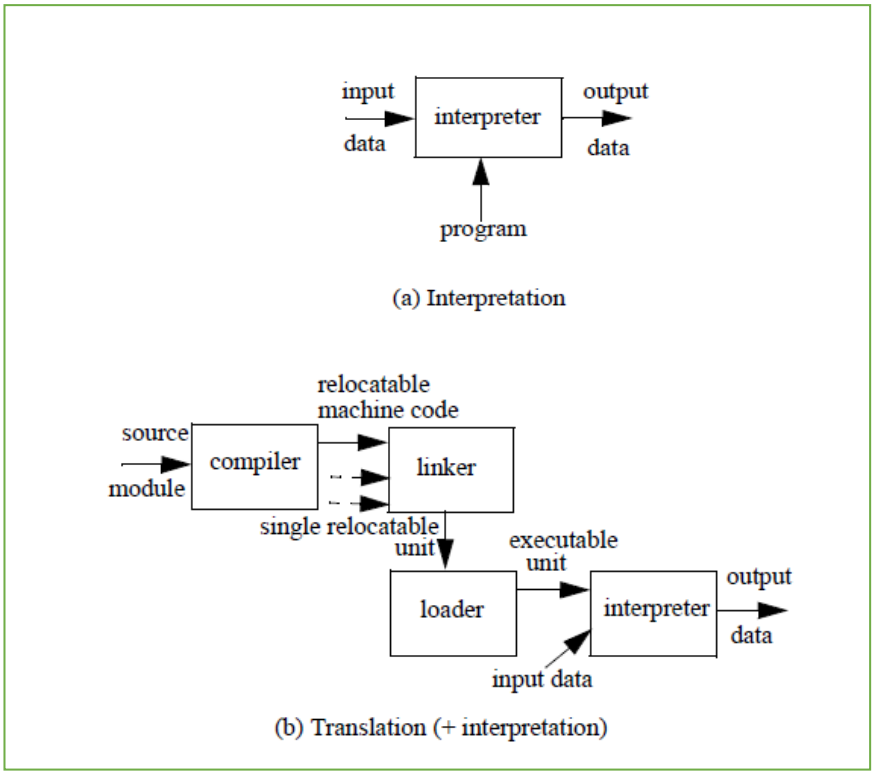


Figure 12: Language processing by interpretation (a) and translation (b)

3.2 Translation

In this solution, programs written in a high-level language are translated into an equivalent machine-language version before being executed. This translation is often performed in several steps (Figure 12). Program modules might first be separately translated into relocatable machine

code; modules of relocatable code are linked together into a single relocatable unit; finally, the entire program is loaded into the computer's memory as executable machine code. The translators used in each of these steps have specialized names: compiler, linker (or linkage editor), and loader, respectively.

In some cases, the machine on which the translation is performed (the host machine) is different from the machine that is to run the translated code (the target machine). This kind of translation is called cross-translation. Crosstranslators offer the only viable solution when the target machine is a specialpurpose processor rather than a general-purpose one that can support a translator.

3.3 Concept of Interpretative Language

Pure interpretation and pure translation are two ends of a continuous spectrum. In practice, many languages are implemented by a combination of the two techniques. A program may be translated into an intermediate code that is then interpreted. The intermediate code might be simply a formatted representation of the original program, with irrelevant information (e.g., comments and spaces) removed and the components of each statement stored in a fixed format to simplify the subsequent decoding of instructions. In this case, the solution is basically interpretive. Alternatively, the intermediate code might be the (low-level) machine code for a virtual machine that is to be later interpreted by software. This solution, which relies more heavily on translation, can be adopted for generating portable code, that is, code that is more easily, transferable to different machines than machine language code. For example, for portability purposes, one of the best known initial implementations of a Pascal compiler was written in Pascal and generated an intermediate code, called Pcode. The availability of a portable implementation of the language contributed to the rapid diffusion of Pascal in many educational environments. More recently, with the widespread use of Internet, code portability became a primary concern for network application developers. A number of language efforts have recently been undertaken with the goal of supporting code mobility over a network. Language Java is perhaps the best known and most promising example. Java is first translated to an intermediate code, called Java bytecode, which is interpreted in the client machine.

In a purely interpretive solution, executing a statement may require a fairly complicated decoding process to determine the operations to be executed and their operands. In most cases, this process is identical each time the statement is encountered. Consequently, if the statement appears in a

frequently-executed part of a program (e.g., an inner loop), the speed of execution is strongly affected by this decoding process. On the other hand, pure translation generates machine code for each high-level statement. In so doing, the translator decodes each high-level statement only once. Frequently-used parts are then decoded many times in their machine language representation; because this is done efficiently by hardware, pure translation can save processing time over pure interpretation. On the other hand, pure interpretation may save storage. In pure translation, each high-level language statement may expand into tens or hundreds of machine instructions. In a purely interpretive solution, high-level statements are left in the original form and the instructions necessary to execute them are stored in a subprogram of the interpreter. The storage saving is evident if the program is large and uses most of the language's statements. On the other hand, if all of the interpreter's subprograms are kept in main memory during execution, the interpreter may waste space for small programs that use only a few of the language's statements.

Compilers and interpreters differ in the way they can report on run-time errors. Typically, with compilation, any reference to the source code is lost in the generated object code. If an error is generated at run-time, it may be impossible to relate it to the source language construct being executed. This is why run-time error messages are often obscure and almost meaningless to the programmer. On the opposite, the interpreter processes source statements, and can relate a run-time error to the source statement being executed. For these reasons, certain programming environments contain both an interpreter and a compiler for a given programming language. The interpreter is used while the program is being developed, because of its improved diagnostic facilities. The compiler is then used to generate efficient code, after the program has been fully validated.

Macro processing is a special kind of translation that may occur as the first step in the translation of a program. A macro is a named source text fragment, called the macro body. Through macro processing, macro names in a text are replaced by the corresponding bodies. In C, one can write macros, handled by a preprocessor, which generates source C code through macro expansion. For example, one can use a macro to provide a symbolic name for a constant value, as in the following fragment.

```
#define upper_limit 100
. . .
```

```
sum = 0;
for (index = 0; index < upper_lmit; index++)
{
sum += a [index];
}
```

3.4 The Concept of Binding

Programs deal with entities, such as variables, routines, statements, and so on. Program entities have certain properties called attributes. For example, a variable has a name, a type, a storage area where its value is stored; a routine has a name, formal parameters of a certain type, certain parameter-passing conventions; a statement has associated actions. Attributes must be specified before an entity is elaborated. Specifying the exact nature of an attribute is known as *binding*. For each entity, attribute information is contained in a repository called a *descriptor*.

Binding is a central concept in the definition of programming language semantics. Programming languages differ in the number of entities with which they can deal, in the number of attributes to be bound to entities, in the time at which such bindings occur (*binding time*), and in the *stability* of the binding (i.e., whether an established binding is fixed or modifiable). A binding that cannot be modified is called *static*. A modifiable binding is called *dynamic*.

Some attributes may be bound at language definition time, others at program translation time (or compile time), and others at program execution time (or run time). The following is a (nonexhaustive) list of binding examples:

- Language definition time binding. In most languages (including FORTRAN, Ada, C, and C++) the type "integer" is bound at language definition time to its well-known mathematical counterpart, i.e., to a set of algebraic operations that produce and manipulate integers;
- Language implementation time binding. In most languages (including FORTRAN, Ada, C, and C++) a set of values is bound to the integer type at language implementation time. That is, the language definition states that type "integer" must be supported and the language implementation binds it to a memory representation, which–in turn–determines the set of values that are contained in the type.

- Compile time (or translation time) binding. Pascal provides a predefined definition of type integer, but allows the programmer to redefine it. Thus type integer is bound a representation at language implementation time, but the binding can be modified at
- translation time.
- Execution time (or run time) binding. In most programming languages variables are bound to a value at execution time, and the binding can be modified repeatedly during execution.
- In the first two examples, the binding is established before run time and cannot be changed thereafter. This kind of binding regime is often called *static*. The term static denotes both the binding time (which occurs before the program is executed) and the stability (the binding is fixed). Conversely, a binding established at run time is usually modifiable during execution. The fourth example illustrates this case. This kind of binding regime is often called *dynamic*. There are cases, however, where the binding is established at run time, and cannot be changed after being established. An example is a language providing (read only) constant variables that are initialized with an expression to be evaluated at run time.

In the first two examples, the binding is established before run time and cannot be changed thereafter. This kind of binding regime is often called *static*. The term static denotes both the binding time (which occurs before the program is executed) and the stability (the binding is fixed). Conversely, a binding established at run time is usually modifiable during execution. The fourth example illustrates this case. This kind of binding regime is often called *dynamic*. There are cases, however, where the binding is established at run time, and cannot be changed after being established. An example is a language providing (read only) constant variables that are initialized with an expression to be evaluated at run time.

4 Self-Assessment Exercises

- Discuss different between Compiler and interpreter
- With an aid the of diagram show the language processing by interpretation and translation
- List the sequence of executing an interpreter
- In what does sequence of an interpreter similar to the pattern carried out by a traditional computer?
- What is binding
- What is descriptor

5 Conclusion

In this unit, you have been introduced to the how language processing can be implemented through interpretation and translation. Also, the concepts of binding, binding time, and stability help clarify many semantic aspects of programming languages.

6 Summary

The unit focused on implementation of language processing through interpretation and translation. Interpretation is achieved via calling subprograms in the appropriate sequence. Also the unit explain the concept of binding which is regarded as a central concept in the definition of programming language semantics.

7 References/Further Reading

Ghezzi and Jazayeri (1996). *Programming language concepts* Third edition John Wiley & Sons. New York Chichester Brisbane Toronto Singapore

Module 3 Structuring Data

Each programming language contains constructs and mechanisms for structuring data. Instead of just the simple sequences of bits in the physical machine, a high level language provides complex, structured data which more easily lends itself to describing the structure of the problems that are to be solved. Unit 1 elaborates on data type and structure. Unit 2 explains the constructs used in programming languages for specification of sequence control. The unit 3 which is the last unit, discusses overview of run-time, identifies common errors of runtime and shows how to fix run-time errors. Also, it presents the difference between runtime and compile time.

Unit 1 Data Types and Data Structure

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Data Type
   - 3.2. Classes of Data Type
     - 3.2.1. Primitive Data Type
     - 3.2.2. Composite/Derived Data Type
     - 3.2.3. Enumerated Data Type
     - 3.2.4. Abstract Data Type
     - 3.2.5. Utility Data Type
     - 3.2.6. Other Data Type
   - 3.3. Data Structure
     - 3.3.1. Array
     - 3.3.2. Linked List
     - 3.3.3. Tree
     - 3.3.4. Hash table
     - 3.3.5. Graph
     - 3.3.6. Stack
     - 3.3.7. Queue
   - 3.4. Difference between data type and data structure
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 1 Data Types and Structure

1 Introduction

A data type defines a collection of data values and a set of predefined operations on those values. Computer programs produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types available in the language being used match the objects in the real world of the problem being addressed. Therefore, it is crucial that a language supports an appropriate collection of data types and structures.

2 Intended Learning Outcomes (ILOs)

- At the end of the unit, students should able to
- Understand meaning and different types of data type
- Understand different categories of data structure
- Know the difference between data types and data structure

3 Main Content

3.1 Data Types

A data type is the most basic and the most common classification of data. A data type is an attribute of data which tells the compiler (or interpreter) how the programmer intends to use the data. So basically data type is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory. Data type can be grouped into three namely;

- Scalar: basic building block (boolean, integer, float, char etc.)
- Composite: any data type (struct, array, string etc.) composed of scalars or composite types (also referred to as a 'compound' type).
- Abstract: data type that is defined by its behaviour (tuple, set, stack, queue, graph etc).

If we consider a composite type, such as a 'string', it *describes* a data structure which contains a sequence of char scalars (characters), and as such is referred to as being a 'composite' type. Whereas the underlying *implementation* of the string composite type is typically implemented using an array data structure. An abstract data type (ADT) describes the expected *behaviour*

associated with a concrete data structure. For example, a 'list' is an abstract data type which represents a countable number of ordered values, but again the *implementation* of such a data type could be implemented using a variety of different data structures, one being a 'linked list'.

Some basic examples are int, string etc. It is the type of any variable used in the code.

```
#include <iostream.h>
using namespace std;

void main()
{
  int a;
  a = 5;

  float b;
  b = 5.0;

  char c;
  c = 'A';

  char d[10];
  d = "example";
}
```

As seen from the theory explained above we come to know that in the above code, the variable 'a' is of data type integer which is denoted by int a. So the variable 'a' will be used as an integer type variable throughout the process of the code. And, in the same way, the variables 'b', 'c' and 'd' are of type float, character and string respectively. And all these are kinds of data types.

3.1.1 Primitive data types

All data in computers based on digital electronics is represented as bits (alternatives 0 and 1) on the lowest level. The smallest addressable unit of data is usually a group of bits called a byte (usually an octet, which is 8 bits). The unit processed by machine code instructions is called a word (as of 2011, typically 32 or 64 bits). Most instructions interpret the word as a binary number, such that a 32-bit word can represent unsigned integer values from 0 to or signed integer values from to. Because of two's complement, the machine language and machine doesn't need to distinguish between these unsigned and signed data types for the most part.

There is a specific set of arithmetic instructions that use a different interpretation of the bits in word as a floating-point number. Machine data types need to be *exposed* or made available in systems or low-level programming languages, allowing fine-grained control over hardware. The C programming language, for instance, supplies integer types of various widths, such as short and long. If a corresponding native type does not exist on the target platform, the compiler will break them down into code using types that do exist. For instance, if a 32-bit integer is requested on a 16-bit platform, the compiler will tacitly treat it as an array of two 16 bit integers. Several languages allow binary and hexadecimal literals, for convenient manipulation of machine data.

In higher level programming, machine data types are often hidden or abstracted as an implementation detail that would render code less portable if exposed. For instance, a generic numeric type might be supplied instead of integers of some specific bit-width. The following are primitive data type

3.1.1.1 Boolean type

The Boolean type represents the values true and false. Although only two values are possible, they are rarely implemented as a single binary digit for efficiency reasons. Many programming languages do not have an explicit boolean type, instead interpreting (for instance) 0 as false and other values as true.

3.1.1.2 Numeric types

- The integer data types, or "whole numbers". May be subtyped according to their ability to contain negative values (e.g. unsigned in C and C++). May also have a small number of predefined subtypes (such as short and long in C/C++); or allow users to freely define subranges such as 1..12 (e.g. Pascal/Ada).
- Floating point data types, sometimes misleadingly called reals, contain fractional values. They usually have predefined limits on both their maximum values and their precision. These are often represented as decimal numbers.
- Fixed point data types are convenient for representing monetary values. They are often implemented internally as integers, leading to predefined limits.
- Bignum or arbitrary precision numeric types lack predefined limits. They are not primitive types, and are used sparingly for efficiency reasons.

3.1.2 Composite / Derived data types

Composite types are derived from more than one primitive type. This can be done in a number of ways. The ways they are combined are called data structures. Composing a primitive type into a compound type generally results in a new type, e.g. *array-of-integer* is a different type to *integer*.

- An array stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed length or expandable.
- Record (also called tuple or struct) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.
- Union. A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time.
- A tagged union (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
- A set is an abstract data structure that can store certain values, without any particular order, and no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a boolean "in" or "not in".
- An object contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

3.1.3 Enumerated Type

This has values which are different from each other, and which can be compared and assigned, but which do not necessarily have any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily. For example, the four suits in a deck of playing cards may be four enumerators named *CLUB*, *DIAMOND*, *HEART*, *SPADE*, belonging to an enumerated type named *suit*. If a variable *V* is declared having *suit* as its data type, one can assign any of those four values to it. Some implementations allow programmers to assign integer values to the enumeration values, or even treat them as type-equivalent to integers.

3.1.3.1 String and text types

- Alphanumeric character. A letter of the alphabet, digit, blank space, punctuation mark, etc.
- Alphanumeric strings, a sequence of characters. They are typically used to represent words and text.

3.1.3.2 Character and string

Character and string types can store sequences of characters from a character set such as ASCII. Since most character sets include the digits, it is possible to have a numeric string, such as "1234". However, many languages would still treat these as belonging to a different type to the numeric value 1234. Character and string types can have different subtypes according to the required character "width". The original 7-bit wide ASCII was found to be limited and superseded by 8 and 16-bit sets.

3.1.4 Abstract data types

Any type that does not specify an implementation is an abstract data type. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set of non-contiguous memory blocks linked by pointers). Abstract types can be handled by code that does not know or "care" what underlying types are contained in them. Arrays and records can also contain underlying types, but are considered concrete because they specify how their contents or elements are laid out in memory.

In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

For example, an abstract stack could be defined by three operations:

- push, that inserts some data item onto the structure,
- pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and
- peek, that allows data on top of the structure to be examined without removal.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. Some common ADTs, which have proved useful in a great variety of programming applications, are – Container, Deque, List, Map, Multimap, Multiset Priority queue, Queue, Set, Stack, Tree, Graph

3.1.5 Utility data types

For convenience, high-level languages may supply ready-made "real world" data types, for instance *times*, *dates* and *monetary values* and *memory*, even where the language allows them to be built from primitive types.

3.2 Data Structure

A data structure is a collection of data type 'values' which are stored and organized in such a way that it allows for efficient access and modification. In some cases, a data structure can become the underlying implementation for a particular data type.

Data structures perform some special operations only like insertion, deletion and traversal. For example, you have to store data for many employees where each employee has his name, employee id and a mobile number. So this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So data structures are one of the most important aspects when implementing coding concepts in real-world applications. Data structures can be grouped into four forms:

- Linear: arrays, lists
- Tree: binary, heaps, space partitioning etc.
- Hash: distributed hash table, hash tree etc.
- Graphs: decision, directed, acyclic etc

The following are the common types of data structures frequently used in computer programming.

3.2.1 Array

An array is a finite group of data, which is allocated contiguous (i.e. sharing a common border) memory locations, and each element within the array is accessed via an index key (typically numerical, and zero based). The name assigned to an array is typically a pointer to the first item in

the array. Meaning that given an array identifier of arr which was assigned the value ["a", "b", "c"], in order to access the "b" element you would use the index 1 to lookup the value: arr[1].

Arrays are traditionally 'finite' in size, meaning you define their length/size (i.e. memory capacity) up front, but there is a concept known as 'dynamic arrays' (and of which you're likely more familiar with when dealing with certain high-level programmings languages) which supports the *growing* (or resizing) of an array to allow for more elements to be added to it.

In order to resize an array you first need to allocate a new slot of memory (in order to copy the original array element values over to), and because this type of operation is quite 'expensive' (in terms of computation and performance) you need to be sure you increase the memory capacity just the right amount (typically double the original size) to allow for more elements to be added at a later time without causing the CPU to have to resize the array over and over again unnecessarily.

One consideration that needs to be given is that you don't want the resized memory space to be *too* large, otherwise finding an appropriate slot of memory becomes more tricky.

When dealing with modifying arrays you also need to be careful because this requires significant overhead due to the way arrays are allocated memory slots.

So if you imagine you have an array and you want to remove an element from the middle of the array, try to think about that in terms of memory allocation: an array needs its indexes to be contiguous, and so we have to re-allocate a new chunk of memory and copy over the elements that were placed *around* the deleted element.

These types of operations, when done at scale, are the foundation behind why it's important to have an understanding of how data structures are implemented. The reason being, when you're writing an algorithm you will hopefully be able to recognize when you're about to do something (let's say modify an array many times within a loop construct) that could ultimately end up being quite a memory intensive set of operations.

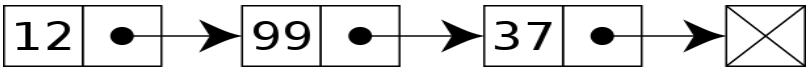
3.2.2 Linked List

A linked list is different to an array in that the order of the elements within the list are not determined by a contiguous memory allocation. Instead the elements of the linked list can be

sporadically placed in memory due to its design, which is that each element of the list (also referred to as a 'node') consists of two parts:

- the data
- a pointer

The data is what you've assigned to that element/node, whereas the pointer is a memory address reference to the next node in the list.



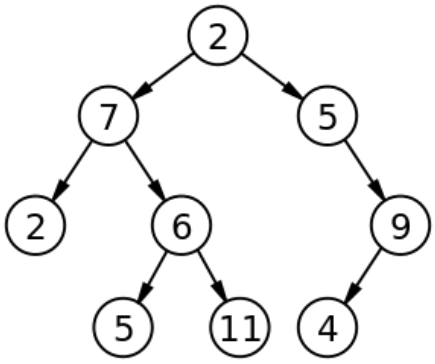
Also unlike an array, there is no index access. So in order to locate a specific piece of data you'll need to traverse the entire list until you find the data you're looking for.

This is one of the key performance characteristics of a linked list, and is why (for most implementations of this data structure) you're not able to *append* data to the list (because if you think about the performance of such an operation it would require you to traverse the entire list to find the end/last node). Instead linked lists generally will only allow *prepending* to a list as it's much quicker. The newly added node will then have its pointer set to the original 'head' of the list.

There is also a modified version of this data structure referred to as a 'doubly linked list' which is essentially the same concept but with the exception of a third attribute for each node: a pointer to the *previous* node (whereas a normal linked list would only have a pointer to the *following* node).

3.2.3 Tree

The concept of a 'tree' in its simplest terms is to represent a hierarchical tree structure, with a root value and subtrees of children (with a parent node), represented as a set of linked nodes.



A tree contains "nodes" (a node has a value associated with it) and each node is connected by a line called an "edge". These lines represent the *relationship* between the nodes.

The top level node is known as the "root" and a node with no children is a "leaf". If a node is connected to other nodes, then the preceeding node is referred to as the "parent", and nodes following it are "child" nodes.

There are various incarnations of the basic tree structure, each with their own unique characteristics and performance considerations: Binary Tree, Binary Search Tree, Red-Black Tree, B-tree, Weight-balanced Tree, Heap, Abstract Syntax Tree.

3.2.3.1 Binary Tree

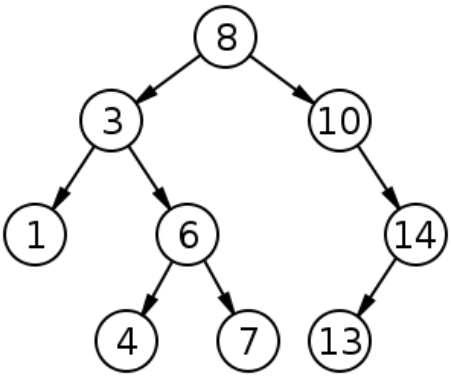
A binary tree is a 'rooted tree' and consists of nodes which have, at most, two children. This is as the name suggests (i.e. 'binary': 0 or 1), so *two* potential values/directions.

Rooted trees suggest a notion of *distance* (i.e. distance from the 'root' node)

Binary trees are the building blocks of *other* tree data structures (see also: this reference for more details), and so when it comes to the performance of certain operations (insertion, deletion etc) consideration needs to be given to the number of 'hops' that need to be made as well as the re-balancing of the tree (much the same way as the pointers for a linked list need to be updated).

3.2.3.2 Binary Search Tree

A binary search tree is a 'sorted' tree, and is named as such because it helps to support the use of a 'binary search' algorithm for searching more efficiently for a particular node (more on that later).



To understand the idea of the nodes being 'sorted' (or 'ordered') we need to compare the left node with the right node. The left node should always be a lesser number than the right node, and the parent node should be the decider as to whether a child node is placed to the left or the right.

Consider the example image above, where we can see the root node is 8. Let's imagine we're going to construct this tree.

We start with 8 as the root node and then we're given the number 3 to insert into the tree. At this point the underlying logic for constructing the tree will know that the number 3 is *less* than 8 and so it'll first check to see if there is already a left node (there isn't), so in this scenario the logic will determine that the tree should have a new left node under 8 and assign it the value of 3.

Now if we give the number 6 to be inserted, the logic will find that again it is less than 8 and so it'll check for a left node. There is a left node (it has a value of 3) and so the value 6 is *greater* than 3. This means the logic will now check to see if there is a right node (there isn't) and subsequently creates a new right node and assigns it the value 6.

This process continues on and on until the tree has been provided all of the relevant numbers to be sorted. In essence what this sorted tree design facilitates is the means for an operation (such as lookup, insertion, deletion) to only take, on average, time proportional to the logarithm of the number of items stored in the tree.

So if there were 1000 nodes in the tree, and we wanted to find a specific node, then the average case number of comparisons (i.e. comparing left/right nodes) would be 10.

By using the logarithm to calculate this we get: log 2(10) = 1024 which is the inverse of the exponentiation 2^10 ("2 raised to the power of 10"), so this says we'll execute 10 comparisons before finding the node we were after.

To break that down a bit further: the exponentiation calculation is 1024 = 2 × 2 × 2 x 2 x 2 x 2 × 2 × 2 x 2 x 2 = 2^10, so the "logarithm to base 2" of 10 is 1024.

The logarithm (i.e. the inverse function of exponentiation) of 1000 to base 2, in this case abstracted to n, is denoted as log 2 (n), but typically the base 2 is omitted to just log(n).

When determining the 'time complexity' for operations on this type of data structure we typically use 'Big O' notation and thus the Big O complexity would be defined as O(log n) for the average search case (which is good), but the *worst case* for searching would still be O(n) linear time (which is bad – and I'll explain why in the next section on red-black trees).

Similarly when considering complexity for a particular algorithm, we should take into account both 'time' and 'space' complexity. The latter is the amount of memory necessary for the algorithm to execute and is similar to time complexity in that we're interested in how that resource (time vs space) will change and affect the performance depending on the size of the input.

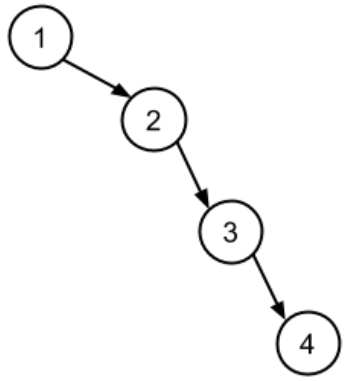
3.2.3.3 Red-Black Tree

The performance of a binary search tree is dependant on the height of the tree. Meaning we should aim to keep the tree as 'balanced' as possible, otherwise the logarithm performance is lost in favor of linear time.

To understand why that is, consider the following data stored in an array:

[1, 2, 3, 4]

If we construct a binary search tree from this data, what we would ultimately end up with is a very 'unbalanced' tree in the sense that all the nodes would be to the right, and none to the left.

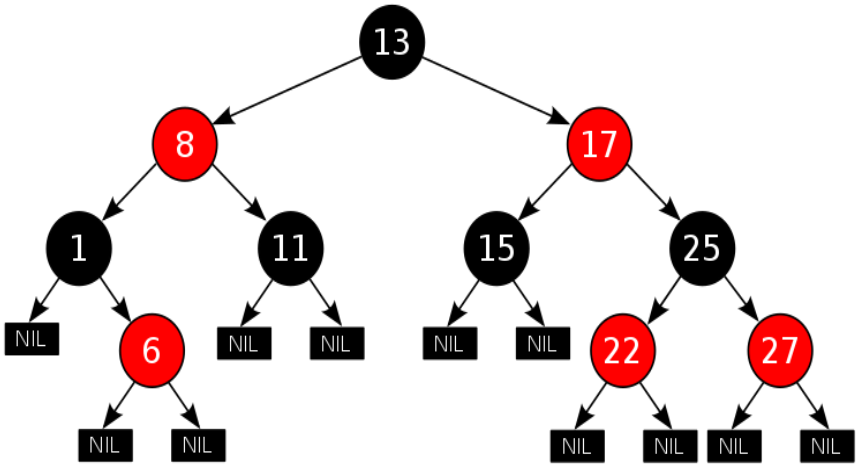


When we search this type of tree (which for all purposes is effectively a linked list) we would, worst case, end up with linear time complexity: O(n). To resolve that problem we need a way to balance the nodes in the tree.

This is where the concept of a red-black tree comes in to help us. With a red-black tree (due to it being consistently balanced) we get O(log n) for search/insert/delete operations (which is great).

Let's consider the properties of a red-black tree:

- Each node is either red or black.
- The root node is always black.
- All leaves are 'NIL' and should also be black.
- All red nodes should have two black child nodes.
- All paths from given node to NIL must have same num of black nodes.
- New nodes should be red by default (we'll clarify below).



The height of the tree is referred to as its 'black-height', which is the number of black nodes (not including the root) to the furthest leaf, and should be no longer than twice as long as the length of the shortest path (the nearest NIL).

These properties are what enable the red-black tree to provide the performance characteristics it has (i.e. O(log n)), and so whenever changes are made to the tree we want to aim to keep the tree height as short as possible.

On every node insertion, or deletion, we need to ensure we have not violated the red-black properties. If we do, then there are two possible steps that we have to consider in order to keep the tree appropriately balanced (which we'll check in this order):

- Recolour the node in the case of a red node no longer having two black child nodes.
- Make a rotation (left/right)in the case where recolouring then requires a structural change.

The goal of a rotation is to decrease the height of the tree. The way we do this is by moving larger subtrees up the tree, and smaller subtrees down the tree. We rotate in the direction of the smaller subtree, so if the smaller side is the right side we'll do a right rotation.

Note: there is an inconsistency between what node/subtree is affected by a rotation. Does the subtree being moved into the parent position indicate the direction or does the target node affected

by the newly moved subtree indicate the direction (I've opted for the latter, as we'll see below, but be aware of this when reading research material).

In essence there are three steps that need to be applied to the target node (T) being rotated, and this is the same for either a left rotation or a right rotation. Let's quickly look at both of these rotation movements:

- Left Rotation:
  i. T's right node (R) is unset & becomes T's parent †
  ii. R's *original* left node L is now orphaned.
  iii. T's right node is now set to L.

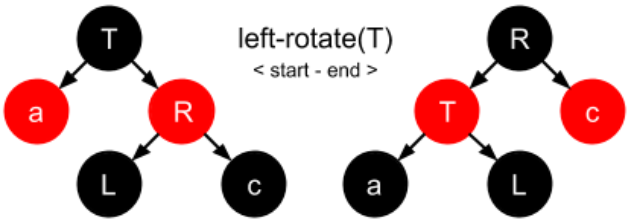
† we now find R's left pointer has to be set to T (in order for it to become the parent node), meaning R's original left pointer is orphaned.

- Right Rotation:
  i. T's left node (L) is unset & becomes T's parent †
  ii. L's *original* right node R is now orphaned.
  iii. T's left node is now set to R.

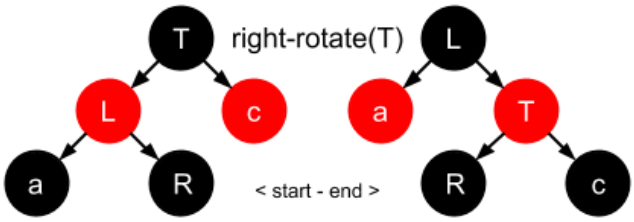
† we now find L's right pointer has to be set to T (in order for it to become the parent node), meaning L's original right pointer is orphaned.

Let's now visualize the movements for both rotations:

Left Rotation



Right Rotation



3.2.3.4 B-tree

A B-tree is a sorted tree that is very similar in essence to a red-black tree in that it is self-balancing and as such can guarantee logarithmic time for search/insert/delete operations.

A B-tree is useful for large read/writes of data and is commonly used in the design of databases and file systems, but it's important to note that a B-tree is *not* a binary search tree because it allows more than two child nodes.

The reasoning for allowing multiple children for a node is to ensure the height of the tree is kept as small as possible. The rationale is that B-trees are designed for handling huge amounts of data which itself cannot exist in-memory, and so that data is pulled (in chunks) from external sources.

This type of I/O is expensive and so keeping the tree 'fat' (i.e. to have a very short height instead of lots of node subtrees creating extra length) helps to reduce the amount of disk access.

The design of a B-tree means that all nodes allow a set range for its children but not all nodes will need the full range, meaning that there is a potential for wasted space.

Note: there are also variants of the B-tree, such as B+ trees and B* trees (which we'll leave as a research exercise for the reader).

3.2.3.5 Weight-balanced Tree

A weight-balanced tree is a form of binary search tree and is similar in spirit to a weighted graph, in that individual nodes are 'weighted' to indicate the more likely successful route with regards to searching for a particular value.

The search performance is the driving motivation for using this data structure, and typically used for implementing sets and dynamic dictionaries.

3.2.3.6 Binary Heap

A binary heap tree is a binary tree, not a binary search tree, and so it's not a sorted tree. It has some additional properties that we'll look at in a moment, but in essence the purpose of this data structure is primarily to be used as the underlying implementation for a priority queue.

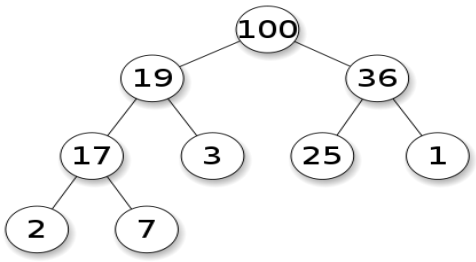
The additional properties associated with a binary heap are:

- heap property: the node value is either greater (or lesser depending on the direction of the heap) or equal to the value of its parent.
- shape property: if the last level of the tree is incomplete, the missing nodes are filled.

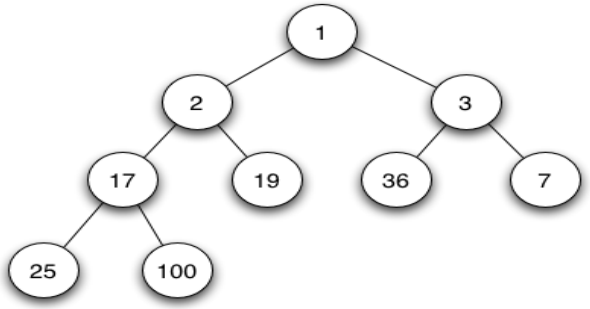
The insertion and deletion operations yield a time complexity of O(log n).

Below are some examples of a max and min binary heap tree structure.

Max Heap:



Min Heap:



3.2.4 Hash Table

A hash table is a data structure which is capable of maping 'keys' to 'values', and you'll typically find this is abstracted and enhanced with additional behaviours by many high-level programming languages such that they behave like an 'associative array' abstract data type.

In Python it's called a 'dictionary' and has the following structure (on top of which are functions such as del, get and pop etc that can manipulate the underlying data):

```
table = {'name': 'foobar',
    'number': 123}
```

The keys for the hash table are determined by way of a hash function but implementors need to be mindful of hash 'collisions' which can occur if the hash function isn't able to create a distinct or unique key for the table.

The better the hash generation, the more *distributed* the keys will be, and thus less likely to collide. Also the size of the underlying array data structure needs to accommodate the type of hash function used for the key generation.

For example, if using modular arithmetic you might find the array needs to be sized to a prime number.

There are many techniques for resolving hashing collisions, but here are two that I've encountered:

- Separate Chaining
- Linear Probing

3.2.4.1 Separate Chaining

With this option our keys will contain a nested data structure, and we'll use a technique for storing our conflicting values into this nested structure, allowing us to store the same hashed value key in the top level of the array.

3.2.4.2 Linear Probing

With this option when a collision is found, the hash table will check to see if the next available index is empty, and if so it'll place the data into that next index.

The rationale behind this technique is that because the hash table keys are typically quite distributed (e.g. they're rarely sequential 0, 1, 2, 3, 4), then it's likely that you'll have many empty empty elements and you can use that empty space to store your colliding data.

Personally I don't like the idea of the Linear Probing technique as it feels like it'll introduce more complexity and bugs. Also, there is a problem with this technique which is that it relies on the top level data structure being an array. Which is fine if the key we're constructing is numerical, but if you want to have strings for your keys then that wont work very well and so you'll need to be clever with how you implement this.

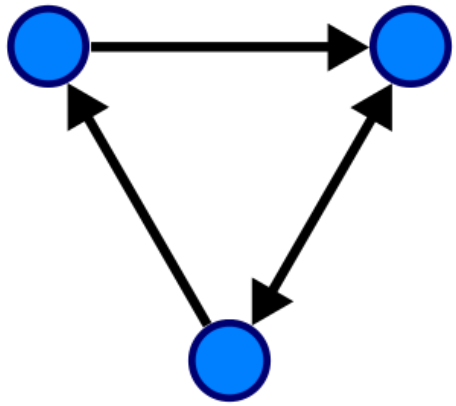
3.2.5 Graph

A graph is an abstract data type intended to guide the implementation of a data structure following the principles of graph theory.

The data struture itself is non-linear and it consists of:

- nodes: points on the graph (also known as 'vertices').
- edges: lines connecting each node.

The following image demonstrates a 'directed' graph (notice the edges have arrows indicating the direction and flow):



Note: an ‘undirected’ graph simply has no arrow heads, so the flow between nodes can go in either direction.

Some graphs are ‘weighted’ which means each ‘edge’ has a numerical attribute assigned to them. These weights can indicate a stronger preference for a particular flow of direction.

Graphs are used for representing networks (both real and electronic), such as streets on a map or friends on Facebook.

When it comes to searching a graph, there are two methods:

- Breadth First Search: look at siblings.
- Depth First Search: look at children.

Which approach you choose depends on the type of values you’re searching for. For example, relationship across fields would lend itself to BFS, whereas hierarchical tree searches would be better suited to DFS.

3.2.6 Stack

A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a ―Last In First Out‖ structure; however there are different variations of stack implementations.

There are basically three operations that can be performed on stacks. They are:

- inserting (―pushing‖) an item into a stack
- deleting (―popping‖) an item from the stack
- displaying the contents of the top item of the stack (―peeking‖)

3.2.7 Queue

A queue is an abstract data type or a linear data structure, in which the first element is inserted from one end (the ―tail‖), and the deletion of existing element takes place from the other end (the―head‖). A queue is a ―First In First Out‖ structure. The process of adding an element to a queue is called ―enqueuing‖ and the process of removing an element from a queue is called ―dequeuing‖.

3.3 Difference between data type and data structure:

The table 15 presents the differences between data types and data structures

Data Types	**Data Structures**
Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

4 Self-Assessment Exercises

5 Conclusion

The data types of a language are a large part of what determines that language's style and usefulness. Along with control structures, they form the heart of a language. The discussion of data structures in computing is a massive topic thus every aspect cannot be covered.

6 Summary

The unit discussed extensively on different data types such as primitive data types, composite data types, enumerated data types, abstract data types and utility data types. Also, justice was done in describing different types of data structure such as array, linked list, tree, hash table, graph, stack and queue. The unit presented the differences between data type and data structure.

7 References/Further Reading

- Gabbriell M. & Martini S. (2010). *Programming Languages: Principles and Paradigms*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-914-5_1, © Springer-Verlag London Limited 2010
- Archana M. *Principles of Programming Languages*
- https://www.integralist.co.uk/posts/data-types-and-data-structures/
- https://www.geeksforgeeks.org/
- www.sctevtservices.nic.in/docs/website/pdf/140338.pdf

Unit 2 Control Structure and Data Flow

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Expressions
     - 3.1.1. Expression Syntax
     - 3.1.2. Semantics of Expressions
     - 3.1.3. Evaluation of Expressions
     - 3.1.4. Subexpression Evaluation Order
   - 3.2. The Concept of Command
     - 3.2.1. The Variable
     - 3.2.2. Assignment
   - 3.3. Sequence Control Commands
     - 3.3.1. Sequential Command
     - 3.3.2. Composite Command
     - 3.3.3. Conditional Commands
     - 3.3.4. Iterative Commands
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 2 Control Structure and Data Flow

1 Introduction

This unit tackles the problem of managing sequence control, an important part in defining the execution of program instructions in a generic abstract machine's interpreter. In low-level languages, sequence control is implemented in a very simple way, just by updating the value of the PC (Program Counter) register. In high-level languages, however, there are special language-specific constructs which permit the structuring of control and the implementation of mechanisms that are much more abstract than those available on the physical machine. Also the unit discusses the constructs used in programming languages for the explicit or implicit specification of sequence control.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Manage sequence control implementation
- Understand sequence control command
- Understand the construct used for specification of sequence control

3 Main Content

3.1 Expressions

Expressions, together with commands and declarations, are one of the basic components of every programming language. We can say that expressions are the essential component of every language because, although there exist declarative languages in language. First, let us try to clarify what sorts of object we are talking about.

An expression defined as a syntactic entity whose evaluation either produces a value or fails to terminate, in which case the expression is undefined. The essential characteristic of an expression, that which differentiates it from a command, is therefore that its evaluation produces a value. Examples of numerical expressions are familiar to all: 4+3*2, for example, is an expression whose evaluation is obvious. Moreover, it can be seen that, even in such a simple case, in order to obtain the correct result, we have made an implicit assumption (derived from the mathematical convention) about operator precedence. This assumption, which tells us that * has precedence over + (and that, therefore, the result of the evaluation is 10 and not 14), specifies a control aspect for

evaluation of expressions. We will see below other more subtle aspects that can contribute to modify the result of the evaluation of an expression.

Expressions can be non-numeric, for example in LISP, we can write (cons a b) to denote an expression which, if it is evaluated, returns the so-called pair formed by a and b.

3.1.1 Expression Syntax

In general, an expression is composed of a single entity (constant, variable, etc.) or even of an operator (such as +, cons, etc.), applied to a number of arguments (or operands) which are also expressions. Expression syntax can be precisely described by a context-free grammar and that an expression can be represented by a derivation tree in which, in addition to syntax, there is also semantic information relating to the evaluation of the expression. Tree structures are also often used to represent an expression internally inside the computer. However, if we want to use expressions in a conventional way in the text of a program, linear notations allow us to write an expression as a sequence of symbols. Fundamentally, the various notations differ from each other by how they represent the application of an operator to its operands. We can distinguish three main types of notation.

3.1.1.1 Infix Notation

In this notation, a binary operation symbol is placed between the expressions representing its two operands. For example, we write x+y to denote than the addition of x and y, or (x+y)*z to denote the multiplication by z of the result of the addition of x and y. It can be seen that, in order to avoid ambiguity in the application of operator to operands, brackets and precedence rules are required. For operators other than binary ones, we must basically fall back on their representation in terms of binary symbols, even if, in this case, this representation is not the most natural. A programming language which insists on infix notation even for user-defined functions is Smalltalk, an object oriented language.

Infix notation is the one most commonly used in mathematics, and, as a consequence is the one used by most programming languages, at least for binary operators and for user syntax. Often, in fact, this notation is only an abbreviation or, as we say, a *syntactic sugar* used to make code more

readable. For example, in Ada, a + b is an abbreviation for +(a, b), while in C++ the same expression is an abbreviation for a.operator+(b).

3.1.1.2 Prefix Notation

Prefix notation is another type of notation. It is also called *prefix Polish notation*.1 The symbol which represents the operation precedes the symbols representing the operands (written from left to right, in the same way as text). Thus, to write the sum of x and y, we can write +(x,y), or, without using parentheses, + x y, while if we want to write the application of the function f to the operands a and b, we write f(a b) or fab.

It is important to note that when using this kind of notation, parentheses and operator precedence rules are of no relevance, provided that the arity (that is the number of operands) of every operator is already known. In fact, there is no ambiguity about which operator to apply to any operands, because it is always the one immediately preceding the operands. For example, if we write:

*(+(a b)+(c d))

or even

* + a b + c d

we mean the expression represented by (a+b)*(c+d) in normal infix notation. The majority of regular languages use prefix notation for unary operators (often using parentheses to group arguments) and for user-defined functions. Some programming languages even use prefix notation for binary operators. LISP represents functions using a particular notation known as Cambridge Polish, which places operators inside parentheses. In this notation, for example the last expression becomes:

(*(+ a b)(+ c d)).

3.1.1.3 Postfix Notation

Postfix notation is also called *Reverse Polish*. It is similar to the last notation but differs by placing the operator symbol after the operands. For example, the last expression above when written in postfix notation is: a b + c d + *.

Prefix notation is used in the intermediate code generated by some compilers. It is also used in programming languages (for example Postscript). In general, an advantage of Polish notation

(prefix or otherwise) over infix is that the former can be used in a uniform fashion to represent operators with any number of operands. In infix notation, on the other had, representing operators with more than two operands means that we have to introduce auxiliary operators. A second advantage, already stated, is that there is the possibility of completely omitting parentheses even if, for reasons of readability, both mathematical prefix notation f(a b) and Cambridge Polish (f a b) use parentheses. A final advantage of Polish notation, as we will see in the next subsection is that it makes the evaluation of an expression extremely simple. For this reason, this notation became rather successful during the 1970s and 80s when it was used for the first pocket-sized calculators.

3.1.2 Semantics of Expressions

According to the way in which an expression is represented, the way in which its semantics is determined changes and so, consequently, does its method of evaluation. In particular, in infix representation the absence of parentheses can cause ambiguity problems if the precedence rules for different operators and the associativity of every binary operator are not defined clearly. When considering the most common programming languages, it is also necessary to consider the fact that expressions are often represented internally in the form of a tree. In this section we will discuss these problems, starting with the evaluation of expressions in each of the three notations that we saw above.

3.1.2.1 Infix Notation: Precedence and Associativity

When using infix notation, we pay for the facility and naturalness of use with major complication in the evaluation mechanism for expressions. First of all, if parentheses are not used systematically, it is necessary to clarify the precedence of each operator. If we write4 + 3 * 5, for example, clearly we intend the value of 19 as the result of the expression and not 35: mathematical convention, in fact, tells us that we have to perform the multiplication first, and the addition next; that is, the expression is to be read as4 + (5 * 3) and not as(4 + 3) * 5. In the case of less familiar operators, present in programming languages, matters are considerably more complex. If, for example, in Pascal one writes:

x=4 and y=5

where the and is the logical operator, contrary to what many will probably expect, we will obtain an error (a static type error) because, according to Pascal's precedence rules, this expression can be interpreted as

x=(4 and y)=5

and not as

(x=4) and (y=5).

In order to avoid excessive use of parentheses (which, when in doubt it is good to use), programming languages employ *precedence rules* to specify a hierarchy between the operators used in a language based upon the relative evaluation order. Various languages differ considerably in their definition of such rules and the conventions of mathematical notation are not always respected to the letter. A second problem in expression evaluation concerns operator associativity. If we write 15-5-3, we could intend it to be read as either (15-5)-3 or as 15-(5-3), with clearly different results. In this case, too, mathematical convention says that the usual interpretation is the first. In more formal terms, the operator "–" associates from left to right.2 In fact, the majority of arithmetic operators in programming languages associate *from left to right* but there are exceptions. The exponentiation operator, for example, often associates from right to left, as in mathematical notation. If we write 5^{3^2} or, using a notation more familiar to programmers, 5 ** 3 ** 2, we mean $5^{(3^2)}$, or 5 ** (3 ** 2), and not $(5^3)^2$, or ((5 ** 3) ** 2). Thus, when an operator is used, it is useful to include parentheses when in doubt about precedence and associativity. In fact, there is no lack of special languages that in this respect have rather counter-intuitive behaviour.

In APL, for example, the expression 15-5-3 is interpreted as 15 - (5 - 3) rather than what we would ordinarily expect. The reason for this apparent strangeness is that in APL there are many new operators (defined to operate on matrices) that do not have an immediate equivalent in other formalisms. Hence, it was decided to abandon operator precedence and to evaluate all expressions from right to left. Even if there is no difficulty in conceiving of a direct algorithm to evaluate an expression in infix notation, the implicit use of precedence and associativity rules, together with the explicit presence of parentheses, complicates matters significantly. In fact, it is not possible to evaluate an expression in a single left-to-right scan (or one from right to left), given that in some cases we must first evaluate the rest of the expression and then return to a sub-expression of

interest. For example, in the case of 5+3*2, when the scan from left to right arrives at +, we have to suspend the evaluation of this operator but divert to the evaluation of 3*2 and then go back to the evaluation of +.

3.1.2.2 Prefix Notation

Expressions written in prefix Polish notation lend themselves to a simple evaluation strategy which proceeds by simply walking the expression from left to right using a stack to hold its components. It can be assumed that the sequence of symbols that forms the expression is syntactically correct and initially not empty. The evaluation algorithm is described by the following steps, where we use an ordinary stack (with the push and pop operations) and a counter *C* to store the number of operands requested by the last operator that was read:

a. Read in a symbol from the expression and push it on the stack;

b. If the symbol just read is an operator, initialise the counter *C* with the number of arguments of the operator and go to step 1.

c. If the symbol just read it is an operand, decrement *C*.

d. If *C* _= 0, go to 1.

e. If *C* = 0, execute the following operations:

- Apply the last operator stored on the stack to the operands just pushed onto the stack, storing the results in *R*, eliminate operator and operands from the stack and store the value of *R* on the stack.
- If there is no operator symbol in the stack go to 6.
- Initialise the counter *C* to $n - m$, where *n* is the number of the argument of the topmost operator on the stack, and *m* is number of operands present on the stack above this operator.
- Go to 4.

f. If the sequence remaining to be read is not empty, go to 1.

The result of the evaluation is located on the stack when the algorithm finishes. It should be noted that the evaluation of an expression using this algorithm assumes that we know in advance the number of operands required by each operator. This requires that we syntactically distinguish unary from binary operators. Furthermore, it is generally necessary to check that the stack contains enough operands for the application of the operator (Step 5.(c) in the algorithm above). This check is not required when using postfix notation, as we see below.

3.1.2.3 Postfix Notation

The evaluation of expression in Polish notation is even simpler. In fact, we do not need to check that all the operands for the last operator have been pushed onto the stack, since the operands are read (from left to right) before the operators. The evaluation algorithm is then the following (as usual, we assume that the symbol sequence is syntactically correct and is not empty):

a. Read the next symbol in the expression and push it on the stack.
b. If the symbol just read is an operator apply it to the operands immediately below it on the stack, store the result in *R*, pop operator and operands from the stack and push the value in *R* onto the stack.
c. If the sequence remaining to be read is not empty, go to 1.
d. If the symbol just read is an operand, go to 1.

This algorithm also requires us to know in advance the number of operands required by each operator.

3.1.3 Evaluation of Expressions

Expressions, like the other programming language constructs, can be conveniently represented by trees. In particular, can be represented by a tree (called the expression's *syntax tree*) in which:

- Every non-leaf node is labelled with an operator.
- Every subtree that has as root a child of a node *N* constitutes an operand for the operator associated with *N*.
- Every leaf node is labelled with a constant, variable or other elementary operand.

Trees like this can be directly obtained from the derivation trees of an (unambiguous) grammar for expressions by eliminating non-terminal symbols and by appropriate rearrangement of the nodes. It can be seen also that, given the tree representation, the linear infix, prefix and postfix

representations can be obtained by traversing the tree in a symmetric, prefix or postfix order, respectively. The representation of expressions as trees clarifies (without needing parentheses) precedence and associativity of operators. The subtrees found lower in the tree constitute the operands and therefore operators at lower levels must be evaluated before those higher in the tree.

For example the tree shown in Figure 13 represents the expression: (a+f(b))*(c+f(b))

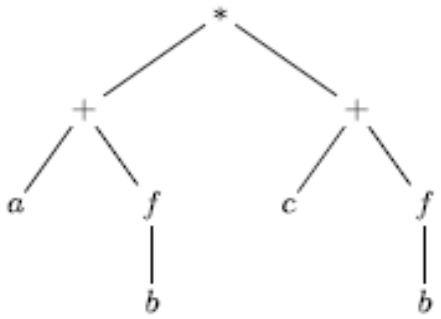


Figure 13: An expression

This expression can be obtained (parentheses apart) from the symmetric-order traversal of the tree (f is here an arbitrary unary operation).

For languages with a compilative implementation, as we have seen, the parser implements syntactic analysis by constructing a derivation tree. In the specific case of expressions then, infix representation in the source code is translated into a tree based representation. This representation is then used by successive phases of the compilation procedure to generate the object code implementing runtime expressions evaluation. This object code clearly depends on the type of machine for which the compiler is constructed. In the case in which we have a traditional physical machine, for example, code of a traditional kind (i.e. in the form opcode operand1 operand2) is generated which uses registers as well as a temporary memory location to store intermediate results of evaluation.

In some particular cases, on the other hand, object code can be represented using a prefix or postfix form which is subsequently evaluated by a stack architecture. This is the case for example in the executable code for many implementations of SNOBOL4 programs. In the case of languages with

an interpretative implementation, it is also convenient to translate expressions, normally represented in the source code in infix notation, into a tree representation which can then be directly evaluated using a tree traversal. This is the case, for example, in interpreted implementations of LISP, where the entire program is represented as a tree.

It is beyond the scope of the present text to go into details on mechanisms for generating code or for evaluating expression in an interpreter. However, it is important to clarify some difficult points which often cause ambiguity. For convenience, we will fix on the evaluation of expressions represented in infix form. We will see that what we have to say applies equally to the direct evaluation of expressions represented as a tree, as well as to code generation-mechanisms.

3.1.4 Subexpression Evaluation Order

Infix notation precedence and associativity rules (or the structure, when expressions are represented as trees) do not hint at the order to evaluate an operator's operands (i.e., nodes at the same level). For example, in the expression in Figure 13, nothing tells us that it is necessary first to evaluate either a+f(b) or c+f(b). There is also nothing explicit about whether the evaluation of operands or operator should come first; nor, in general, whether expressions which are mathematically equivalent can be inter-substituted without modifying the result (for example, (a-b+c) and(a+c-b) could be considered equivalent). While in mathematical terms these differences are unimportant (the result does not change), from our viewpoint these questions are extremely relevant and for the following five reasons.

3.1.4.1 Side effects:

In imperative programming languages, expression evaluation can modify the value of any variables through so-called side effects. A side effect is an action that influences the result (partial or final) of a computation without otherwise explicitly returning a value in the context in which it is found. The possibility of side effects renders the order of evaluation of operands relevant to the final result. In our example in Figure 13, if the evaluation of the function f were to modify the value of its operand through side effects, first executing a+f(b) rather than c+f(b), could change the value produced by the evaluation. As far as side effects are concerned, languages follow various approaches. On the one hand, pure declarative languages do not permit side effects at all, while languages which do allow them in some cases forbid the use in expressions of functions that

can cause side effects. In other, more common cases where the presence of side effects is permitted, the order with which expressions are evaluated is, though, clearly stated in the definition of the language. Java, for example, imposes left-to right evaluation of expressions (while C fixes no order at all).

3.1.4.2 Finite arithmetic

Given the set of numbers represented in a computer is finite (see also Sect. 8.3), reordering expressions can cause overflow problems. For example, if a has, as its value, the maximum integer representable and b and c are positive numbers such that b > c, right-to-left evaluation of (a-b+c) does not produce overflow, while we have an overflow resulting from the evaluation from left to right of (a+c-b). Moreover, when we do not have overflow, the limited precision of computer arithmetic implies that changing the order of the operands can lead to different results (this is particularly relevant in cases of floating point computation).

3.1.4.3 Undefined operands

When the application of operator to operands is considered, two evaluation strategies can be followed. The first, called *eager evaluation*, consists of first evaluating all the operands and then applying the operator to the values thus obtained. The strategy probably seems the most reasonable when reasoning in terms of normal arithmetic operators. The expressions that we use in programming languages, however, pose problems over and above those posed by arithmetic expressions, because some can be defined even when some of the operands are missing. Let us consider the example of a conditional expression of the form: a == o ? b : b/a

We can write this in C to denote the value of b/a when a is non-zero and b, otherwise. This expression results from the application of a single operator (expressed in infix notation using two binary operators ? and :) to three operands (the Boolean expression, a==0, and the two arithmetic expressions b and b/a). Clearly here we cannot use eager evaluation for such conditional expressions because the expression b/a would have to be evaluated even when a is equal to zero and this would produce an error.

In such a case, it is therefore better to use a *lazy evaluation* strategy which mainly consists of *not* evaluating operands before the application of the operator, but in passing the un-evaluated operands to the operator, which, when it is evaluated, will decide which operands are required, and

will only evaluate the ones it requires. The lazy evaluation strategy, used in some declarative languages, is much more expensive to implement than eager evaluation and for this reason, most languages use eager evaluation (with the significant exception of conditional expressions as we will see below). There are languages which use a mix of both the techniques (ALGOL, for example).

3.1.4.4 Short-circuit evaluation

The problem detailed in the previous point presents itself with particular clarity when evaluating Boolean expressions. For example, consider the following expression (in C syntax):

a == 0 || b/a > 2

If the value of a is zero and both operands of || are evaluated at the same time, it is clear that an error will result (in C, "||" denotes the logical operation of disjunction). To avoid this problem, and to improve the efficiency of the code, C, like other languages uses a form of lazy evaluation, also called *short-circuiting evaluation*, of boolean expressions. If the first operand of a disjunction has the value *true* then the second is not evaluated, given that the overall result will certainly have the value *true*. In such a case, the second expression is *short-circuited* in the sense that we arrive at the final value before knowing the value of all of the operands. Analogously, if the first operand of a conjunction has the value *false*, the second is not evaluated, given that the overall result can have nothing other than the value *false*. It is opportune to recall that not all languages use this strategy for boolean expressions. Counting on the presence of a short-circuited evaluation, without being certain that the language uses it, is dangerous. For example, we can write in Pascal

```
p := list;
while (p <> nil ) and (pˆ.value <> 3) do
p := pˆ.next;
```

The intention of this code is to traverse a list until we have arrived at the end or until we have encountered the value 3. This is badly written code that can produce a runtime error. Pascal, in fact, does not use short-circuit evaluation. In the case in which we have p = nil, the second operand of the conjunction (pˆ.value <>. 3) yields an error when it dereferences a null pointer. Similar code,

on the other hand, *mutatis mutandis*, can be written in C without causing problems. In order to avoid ambiguity, some languages (for example C and Ada), explicitly provide different boolean operators for short-circuit evaluation. Finally, it should be noted that this kind of evaluation can be simulated using a conditional command.

3.1.4.5 Optimisation

Frequently, the order evaluation of subexpressions influences the efficiency of the evaluation of an expression for reasons relating to the organization of the physical machine. For example, consider the following code:

```
a = vector[i];

b = a*a + c*d;
```

In the second expression, it is probably better first to evaluate c*d, given that the value of a has to be read from memory (with the first instruction) and might not be yet available; in such a case, the processor would have to wait before calculating a * a. In some cases, the compiler can change the order of operands expressions to obtain code that is more efficient but semantically equivalent.

The last point explains many of the semantic problems that appear while evaluating expressions. Given the importance of the efficiency of the object code produced by the compiler, it is given considerable liberty in the precise definition of its expression evaluation method, without it being specified at the level of semantic description of the language (as we have already said, Java is a rare exception). The result of this kind of approach is that, sometimes, different implementations of the same language produce different results for the same expression, or have errors at runtime whose source is hard to determine.

Wishing to capitalise in a pragmatic prescription, given what has been said so far, if we do not know the programming language well and the specific implementation we are using, if we want to write correct code, it is wise to use all possible means at our disposal to eliminate as many sources of ambiguity as possible in expression evaluation (such as brackets parentheses, specific boolean operations, auxiliary variables in expressions, etc.).

3.2 The Concept of Command

If, as we were saying above, expressions are present in all programming languages, the same is not true for commands. They are constructs that are typically present (but not entirely restricted to them) in so-called imperative languages. A command is a syntactic entity whose evaluation does not necessarily return a value but can have a side effect. A command, or more generally, any other construct, has a side-effect if it influences the result of the computation but its evaluation returns no value to the context in which it is located. This point is fairly delicate and merits clarification with an example. If the print command in a hypothetical programming language can print character strings supplied as an argument, when the command print "pippo" is evaluated, we will not obtain a value but only a side-effect which is composed of the characters "pippo" appearing on the output device.

The attentive reader will be aware that the definition of command, just as the previous definition of expression, it is not very precise, given that we have referred to an informal concept of evaluation (the one performed by the abstract machine of the language to which the command or the expression belongs). It is clear that we can always modify the interpreter so that we obtain some value as a result of the evaluation of the command. A precise definition and, equally, an exact distinction, between expressions and commands on the basis of their semantics is possible only in the setting of a formal definition of the semantics of language. In such a context, the difference between the two concepts derives from the fact that, once a starting state has been fixed, the result of the evaluation of an expression is a value (together with possible side effects). On the other hand, the result of evaluating a command is a new state which differs from the start state precisely in the modifications caused by the side-effects of the command itself (and which are due principally to assignments). Command is therefore a construct whose purpose is the modification of the *state*. The concept of state can be defined in various ways, we saw a simple version, one which took into account the value of all the variables present in the program. If the aim of a command is to modify the state, it is clear that the assignment command is the elementary construct in the computational mechanism for languages with commands. Before dealing with them, however, it is necessary to clarify the concept of variable.

3.2.1 The Variable

In mathematics, a variable is an unknown which can take on all the numerical values in a predetermined set. Even if we keep this in mind, in programming languages, it is necessary to specify this concept in more detail because, the imperative paradigm uses a model for variables which is substantially different from that employed the in logic and functional programming paradigms. The classical imperative paradigm uses *modifiable variables*. According to this model, the variable is seen as a sort of container, or location (clearly referring to physical memory), to which a name can be given and which contains values (usually of a homogeneous type, for example integers real, characters etc.). These values can be changed over time, by execution of assignment commands (whence comes the adjective "modifiable"). This terminology might seem tautological to the average computer person, who is almost always someone who knows an imperative language and is therefore used to modifiable variables. The attentive reader, though, will have noted that, in reality, variables are not always modifiable. In mathematics a variable represents a value that is unknown but when such a value is defined the link thus created cannot be modified later.

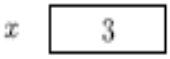


Figure 14: A modifiable variable

Modifiable variables are depicted in Figure 14. The small box which represents the variable with the name x can be re-filled with a value (in the figure, the value is 3). It can be seen that the variable (the box) is different from the name x which denotes it, even if it is common to say "the variable x" instead of "the variable with the name x".

Some imperative languages (particularly object-oriented ones) use a model that is different from this one. According to this alternative model, a variable is not a container for a value but is a reference to (that is a mechanism which allows access to) a value which is typically stored in the heap. This is a new concept analogous to that of the pointer (but does not permit the usual pointer-manipulation operations). We will see this in the next section after we have introduced assignment commands. This variable model is called, the "reference model", where it is discussed in the context of the language CLU, is called the "object model". Henceforth, we will refer to this as the *reference model* of variables. (Pure) functional languagesuse a concept of variable similar to the

mathematical one: a variable is nothing more than an identifier that stands for a value. Rather, it is often said that functional languages "do not have variables", meaning that (in their pure forms) they do not have any modifiable variables.

Logic languages also use identifiers associated with values as variables and, as with functional languages, once a link between a variable identifier and a value is created, it can never be eliminated. There is however a mode in which the value associated with a variable can be modified without altering the link.

3.2.2 Assignment

Assignment is the basic command that allows the modification of the values associated with modifiable variables. It also modifies the state in imperative languages. It is an apparently very simple command. However, as will be seen, in different programming languages, there are various subtleties to be taken into account. Let us first see the case that will probably be most familiar to the reader. This is the case of an imperative language which uses modifiable variables and in which assignment is considered only as a command (and not also as an expression). One example is Pascal, in which we can write X := 2 to indicate that the variable X is assigned the value 2. The effect of such a command is that, after its execution, the container associated with the variable (whose name is) X will contain the value 2 in place of the value that was there before. It should be noted that this is a side effect, given that the evaluation of the command does not on its own, return any kind of value. Furthermore, every access to X in the rest of the program will return the value 2 and not the one previously stored.

Consider now the following command: X := X+1

The effect of this assignment, as we know, is that of assigning to the variable X its previous value incremented by 1. Let us observe the different uses of the name, X, of the variable in the two operands of the assignment operator. The X appearing to the left of the := symbol is used to indicate the container (the location) inside which the variable's value can be found. The occurrence of the X on the right of the := denotes the value inside the container. This important distinction is formalised in programming languages using two different sets of values: *l-values* are those values that usually indicate locations and therefore are the values of expressions that can be on the left of

an assignment command. On the other hand, *r-values* are the values that can be stored in locations, and therefore are the values of expressions that can appear on the right of an assignment command. In general, therefore, the assignment command has the syntax of a binary operator in infix form:

exp1 OpAss exp2

where OpAss indicates the symbol used in the particular language to denote assignment (:= in Pascal, = in C, FORTRAN, SNOBOL and Java, ← in APL, etc.). The meaning of such a command (in the case of modifiable variables) is as follows: compute the l-value of exp1, determining, thereby, a container *loc*; compute the r-value of exp2 and modify the contents of *loc* by substituting the value just calculated for the one previously there. Which expressions denote (in the context on the left of an assignment) an l-value depends on the programming language: the usual cases are variables, array elements, record fields (note that, as a consequence, calculation of an l-value can be arbitrarily complex because it could involve function calls, for example when determining an array index). In some languages, for example C, assignment is considered to be an operator whose evaluation, in addition to producing a side effect, also returns the r-value thus computed.

Thus, if we write in C: x = 2;

the evaluation of such a command, in addition to assigning the value 2 to x, returns the value 2. Therefore, in C, we can also write: y = x = 2;
which should be interpreted as: (y = (x = 2));

This command assigns the value 2 to x as well as to y. In C, as in other languages, there are other assignment operators that can be used, either for increasing code legibility or avoiding unforeseen side effects. Let us take up the example of incrementing a variable. Once again we have: x = x+1;

This command, unless optimised by the compiler, requires, in principle, two accesses to the variable x: one to determine the l-value, and one to obtain the r-value. If, from the efficiency viewpoint, this is not serious (and can be easily optimised by the compiler), there is a question which is much more important and which is again related to side-effects. Let us then consider the code: b = 0;

a[f(3)] = a[f(3)]+1;

where a is a vector and f is a function defined as follows:

```
int f (int n){
if b == 0{
b=1;
return 1;
}
else return 2;
}
```

This function is defined in such a way that the non-local reference to b in the body of f refers to the same variable b that is cleared in the previous fragment. Given that f modifies the non-local variable b, it is clear that the assignment

a[f(3)] = a[f(3)]+1

does not have the effect of incrementing the value of the element a[f(3)] of the array, as perhaps we wanted it to do. Instead, it has the effect of assigning the value of a[1]+1 to a[2] whenever the evaluation of the left-hand component of the assignment precedes the evaluation of the right-hand one. It should be noted, on the other hand, that the compiler cannot optimise the computation of r-values, because the programmer might have wanted this apparently anomalous behaviour.

To avoid this problem, we can clearly use an auxiliary variable and write:

```
int j = f(3);
a[j] = a[j]+1;
```

Doing this obscures the code and introduces a variable which expresses very little. To avoid all of this, languages like C provide assignment operators which allow us to write:

a[f(3)] += 1;

This add to the r-value of the expression present on the left the quantity present on the right of the += operator, and then assigns the result to the location obtained as the l-value of the expression on

the left. There are many specific assignment commands that are similar to this one. The following is an incomplete list of the assignment commands in C, together with their descriptions:

- X = Y: assign the r-value of Y to the location obtained as the l-value of X and return the r-value of X;
- X += Y (or X -= Y): increment (decrement) X by the quantity given by the r-value of Y and return of the new r-value;
- ++X (or -X): increment (decrement) X by and return the new r-value of X;
- X++ (or X-): return the r-value of X and then increment (decrement) X.

We will now see how the reference model for variables differs from the traditional modifiable-variable one. In a language which uses the reference model (for example, CLU and, as we will see, in specific cases, Java) after an assignment of the form:

x=e

x becomes a reference to an object that is obtained from the evaluation of the expression e. Note that this does not copy the value of e into the location associated with x. This difference becomes clear if we consider an assignment between two variables using the reference model.

x=y

After such an assignment, x and y are two references to the same object. In the case in which this object is modifiable (for example, record or array), a modification performed using the variable x becomes visible through variable y and vice versa.

In this model, therefore, variables behave in a way similar to variables of a pointer type in languages which have that type of data. A value of a pointer type is no more than the location of some data item (or, equivalently, its address in some area of memory). In many languages which have pointer types, the values of such types can be explicitly manipulated. In the case of the reference model, however, these values can be manipulated only implicitly using assignments

3.3 Sequence Control Commands

Assignment is the basic command in imperative languages (and in "impure" declarative languages); it expresses the elementary computation step. The remaining commands serve to define sequence control, or rather serve to specify the order in which state modifications produced by assignments, are to be performed. These other commands can be divided into three categories:

- Commands for explicit sequence control These are the sequential command and goto. Let us consider, in addition, the composite command, which allows us to consider a group of commands as a single one, as being in this category.
- Conditional (or selection) commands These are the commands which allow the specification of alternative paths that the competition can take. They depend on the satisfaction of specific conditions.
- Iterative commands These allow the repetition of a given command for a predefined number of times, or until the satisfaction of specific conditions.

3.3.1 Sequential Command

The sequential command, indicated in many languages by a ";", allows us directly to specify the sequential execution of two commands. If we write:

C1 ; C2

the execution of C2 starts immediately after C1 terminates. In languages in which the evaluation of a command also returns a value, the value returned by the evaluation of the sequential command is that of the second argument.

Obviously we can write a sequence of commands such as:

C1 ; C2 ; ... ; Cn

with the implicit assumption that the operator ";" associates to the left.

3.3.2 Composite Command

In modern imperative languages, as we have already seen in Chap. 4, it is possible to group a sequence of commands into a *composite command* using appropriate delimiters such as those used by Algol:

```
begin
...
end
```
or those in C:
```
{
...
}
```

3.3.3 Conditional Commands

Conditional commands, or selection commands, express one alternative between two or more possible continuations of the computation based on appropriate logical conditions. We can divide conditional commands into two groups.

If The if command, originally introduced in the ALGOL60 language, is present in almost all imperative languages and also in some declarative languages, in various syntactic forms which, really, can be reduced to the form:

if Bexp **then** C1 **else** C2

where Bexp is a boolean expression, while C1 and C2 are commands. Informally, the semantics of such a command expresses an alternative in the execution of the computation, based on the evaluation of the expression Bexp. When this evaluation returns true, the command C1 is executed, otherwise the command C2 is executed. The command is often present in the form without the else branch:

if Bexp **then** C1

In this case, too, if the condition is false, the command C1 is not executed and control passes to the command immediately after the conditional. As we saw in Chap. 2, the presence of a branching if as in the command

if Bexp1 **if** Bexp2 **then** C1 **else** C2

causes problems of ambiguity, which can be resolved using a suitable grammar which formally describes the rules adopted by the language (for example, the else branch belongs to the innermost

if; this is the rule in Java and it is used in almost every language). To avoid problems of ambiguity, some languages use a " terminator" to indicate where the conditional command ends, as for example in:

if Bexp **then** C1 **else** C2 **endif**

Furthermore, in some cases, instead of using a list of nested if then elses, use is made of an if equipped with more branches, analogous to the following:

```
if Bexp1 then C1
   elseif Bexp2 then C2
   ...
   elseif Bexpn then Cn
   else Cn+1
endif
```

The implementation of the conditional command poses no problems, and makes use of instructions for test and jump that are found in the underlying physical machine. The evaluation of the boolean expression can use the shorter circuit technique that we saw above.

Case The command is a specialisation of the if command, just discussed, with more branches. In its simplest form it is written as follows:

```
case Exp of
      label1: C1;
      label2: C2;
      ...
      labeln: Cn;
else Cn+1
```

where Exp is an expression whose value is of a type compatible with that of the labels label1, ... , labeln, while C1, ... , Cn+1 are commands. Each label is represented by one or more constants and the constant used in different labels are different from each other. The type permitted for labels, as

well as their form, varies from language to language. In most cases, a discrete type is permitted, including enumerations and intervals. So, for example, we can use the constants 2 and 4 to denote a label, but in some languages we can also write 2,4 to indicate either the value 2 or the value 4, or 2 .. 4to indicate all values between 2 and 4 (inclusive).

Different languages exhibit significant differences in their case commands. In C, for example, the switch has the following syntax (also to be found in C++ and in Java):

switch (Exp) body

where body can be any command that all. In general, though, the body is formed from a block in which some commands can be labelled; that is they are of the form:

case label **:** command

while the last command of the block is of the form:

default : command

When the expression Exp is evaluated and control is to be transferred to the command whose label coincides with the resulting value, if there are no labels with such a value, control passes to the command with the label default. If there is no default command, control passes to the first command following the switch. It can be seen that, once a branch of the switch has been selected, control then flows into the immediately following branches. To obtain a construct with semantics analogous to that of the case we discussed above, it is necessary to insert an explicit control transfer at the end of the block, using a break:

```
switch (Exp){
    case label1: C1 break;
    case label2: C2 break;
    ...
    case labeln: Cn break;
    default: Cn+1 break;
}
```

It can be seen also that in a switch, the value returned by the evaluation of the expression might not appear in any label, in which case the entire command has no effect. Finally, lists or ranges of values are not permitted as labels. This however is no real limitation, given that lists of values can

be implemented using the fact that control passes from one branch to its successor when break is omitted. If, for example, we write:

```
switch (Exp){
    case 1:
    case 2: C2 break;
    case 3: C3 break;
    default: C4 break;
}
```

in the case in which the value of Exp is 1, given that the corresponding branch does not contain a break command, control passes from the case 1 branch immediately to the case 2 branch and therefore it is as if we had used a list of values 1,2 for the label of C2.

3.3.4 Iterative Commands

The commands that we have seen up to this point, excluding goto, only allow us to express finite computations, whose maximum length is determined statically by the length of the program text. A language which had only such commands would be of highly limited expressiveness. It would certainly not be Turing complete, in that it would not permit the expression of all possible algorithms (consider, for example, scanning a vector of *n* elements, where *n* is not known *a priori*).

In order to acquire the expressive power necessary to express all possible algorithms in low-level languages, jump instructions allowing the repetition of groups of instructions by jumping back to the start of the code are needed. In high-level languages, given that, as has been seen, it is desirable to avoid commands like goto, two basic mechanisms are employed to achieve the same effect: *structured iteration* and *recursion*. The first, which we consider in this section, is more familiar from imperative languages (and they almost always allow recursion as well). Suitable linguistic constructs (which we can regard as special versions of the jump command) allow us compactly to implement loops in which commands are repeated or iterated. At the linguistic level, it is possible to distinguish between unbounded iteration and bounded iteration. In bounded iteration, repetition is implemented by constructs that allow a determinate number of iterations. Unbounded iteration, on the other hand, is implemented by constructs which continue until some condition becomes true.

Recursion which we will consider in the next section, allows, instead, the expression of loops in an implicit fashion, including the possibility that a function (or procedure) can call itself, thereby repeating its own body an arbitrary number of times. The use of recursion is more common in declarative languages (in many functional and logic languages there does not, in fact, exist any iterative construct).

3.3.4.1 Unbounded iteration

Unbounded iteration is logically controlled iteration. It is implemented by linguistic constructs composed of two parts: a loop *condition* (or *guard*) and a *body*, which is composed of a (possibly compound) command. When executed, the body is repeatedly executed until the guard becomes false (or true, according to the construct). In its most common form, this type of iteration takes the form of the while command, originally introduced in ALGOL:while

while (Bexp) **do** C

The meaning of this command is as follows: (1) the boolean expression Bexp is evaluated; (2) if this evaluation returns the value *true*, execute the command C and return to (1); otherwise the while command terminates.

In some languages there are also commands that test the condition *after* execution of the command (which is therefore always executed at least once). This construct is for example present in Pascal in the following form:

repeat C **until** Bexp

This is no more than an abbreviation for:

C;
while not Bexp **do** C
(not Bexp here indicates the negation of the expression Bexp). In C an analogous construct is do:
do C **while** (Bexp)
which corresponds to:
C;
while Bexp **do** C

(note that the guard is not negated as in the case of repeat.)

The while construct is simple to implement, given that it corresponds directly to a loop that is implemented on the physical machine using a conditional jump instruction. This simplicity of implementation should not deceive us about the power of this construct. Its addition to a programming language which contains only assignment and conditional commands immediately makes the language Turing complete.

3.3.4.2 Bounded iteration

Bounded iteration (sometimes also called numerically controlled iteration) is implemented by linguistic constructs that are more complex than those used for unbounded iteration; their semantics is also more elaborate. These forms are very different and not always "pure" as we will see shortly. The model that we adopt in this discussion is that of ALGOL, which was then adopted by many other languages of the same family (but *not* by C or Java).

Bounded iteration is implemented using some variant of the for command. Without wishing to use any specific syntax, it can be described as:

```
for I = start to end by step do
    body
```

where I is a variable, called the *index*, or counter, or *control variable*; start and end are two expressions (for simplicity we can assume that they are of integer type and, in general, they must be of a discreet type); step is a (compile-time) non-zero integer constant; *body* is the command we want to repeat. This construct, in the "pure" form we are describing, is subject to the important static semantic constraint that the control variable can not be modified (either explicitly nor implicitly) during the execution of the body.

- *Semantics of bound iteration*

The semantics of the bounded iteration construct can be described informally as follows (assuming that step is positive):

1. The expression start is evaluated, as is end. The values are frozen and stored in dedicated variables (which cannot be updated by the programmer). We denote them, respectively, as start_save and end_save.
2. I is initialised with the value of start_save.
3. If the value of I is strictly greater than the value of end_save, execution of the for command is terminated.
4. Execute *body* and increment I by the value of step.
5. Go to 3.

In the case in which step is negative, the test in step (3) determines whether I is strictly less than end_save. It is worth emphasizing the importance of step (1) above and the constraint that the control variable cannot be modified in the body. Their combined effect is to *determine* the number of times and the body will be executed *before* the loop begins execution. This number is given by the quantity, *ic* (*iteration count*), which is defined as:

$$ic = \left\lfloor \frac{\mathtt{end} - \mathtt{start} + \mathtt{step}}{\mathtt{step}} \right\rfloor$$

if *ic* is positive, otherwise it is 0. It can be seen, finally, that there is no way of producing an infinite cycle with this construct.

- *Expressiveness of bounded iteration*

Using bounded iteration, we can express the repetition of a command for *n* times, where *n* it is an arbitrary value not known when the program is written, but is fixed at when the iteration starts. It is clear that this is something that cannot be expressed using only conditional commands and assignment, because it is possible to repeat a command only by repeating the command in the body of the program syntactically. Given that every program has a finite length, we have a limit on the maximum number of repetitions that we can include in a specific program.

4 Self-Assessment Exercises

i. Define expression

ii. Discuss in detail expression syntax

iii. Discuss in detail semantics of expression

iv. Define, in any programming language, a function, f , such that the evaluation of the expression $(a + f(b)) * (c + f(b))$ when performed from left-to-right has a result that differs from that obtained by evaluating right-to-left.

v. Show how the if then else construct can be used to simulate short-circuit evaluation of boolean expressions in a language which evaluates all operands before applying boolean operators.

vi. Consider the following case command:

```
Case    Exp of
  1:     C1;
  2,3:   C2;
  4..6:  C3;
  7:     C4
  else:  C5
```

Provide an efficient pseudocode assembly program that corresponds to the translation of this command.

5 Conclusion

The unit described and discussed a variety of statement-level in control structures and briefly evaluated the expression. A brief evaluation now seems to be in order. Also, the sequence control commands were deliberated on which led in grouping the commands to four categories namely sequential command, composite command, conditional (or selection) commands and Iterative commands.

6 Summary

The unit analyzed the components of high-level languages relating to the control of execution flow in programs. We first considered expressions and we have analyzed the types of syntax that most used for their description (as trees, or in prefix, infix and postfix linear form) and the related evaluation rules. Also, the precedence and associativity rules required for infix notation were

debated on. Furthermore, the unit discussed the problems generally related to the order of evaluation of the subexpressions of an expression.

7 References/Further Reading

Gabbriell M. & Martini S. (2010). *Programming Languages: Principles and Paradigms*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-914-5_1, © Springer-Verlag London Limited 2010

Unit 3 Run-time Consideration

1. Introduction
2. Intended Learning Outcomes (ILOs)
3. Main Content
   - 3.1. Overview of Run-time
   - 3.2. Run-time Errors
     - 3.2.1. Common Types of Run-time Error
     - 3.2.2. How to Fix a Run-time Error?
   - 3.3. Run-time Environment
   - 3.4. Run-time/Compiler time
4. Self-Assessment Exercises
5. Conclusion
6. Summary
7. References/Further Reading

Unit 3 Run-time Consideration

1 Introduction

In computer the period of time the program runs from the beginning to the end is regarded as running time. The execution time is very crucial in system evaluation. Thus, this unit presents the overview of run-time and deliberate on runtime error by discussing the common errors of runtime and how these errors can be fixed. Also, comparison between runtime and compile time is discussed.

2 Intended Learning Outcomes (ILOs)

At the end of the unit, students should able to

- Know different types of run-time error
- Fix run-time error
- Differentiate between run-time and compile time

3 Main Content

3.1 Overview of Run-time

Run time is a phase of a computer program in which the program is run or executed on a computer system. Run time is part of the program life cycle, and it describes the time between when the program begins running within the memory until it is terminated or closed by the user or the operating system. Run time is also known as execution time. Runtime is a system used primarily in software development to describe the period of time during which a program is running. Runtime is the final phase of the program lifecycle in which the machine executes the program's code.

When a user tries to start a program a loader runs that allocates memory and links the program with any necessary libraries, then the execution begins. Many people who use computer programs understand the runtime process; however, runtime is very important to software developers because if errors are found in the code the program will throw runtime errors.

3.2 Runtime errors

A runtime error is an error that occurs when a program you're using or writing crashes or produces a wrong output. At times, it may prevent you from using the application or even your personal computer. In some cases, users need only refresh their device or the program to resolve the runtime

error. However, sometimes, users may have to perform a particular action to fix the error. Before a runtime error shows up on your computer, you may have noticed its performance slowing down. When runtime errors occur, your computer will always display a prompt stating the specific type of error you've encountered. If a program experiences an error after it has been executed it will report back a runtime error. There are hundreds of different errors that programs can experience such as division by zero errors, domain errors, and arithmetic underflow errors.

Some programming languages have built-in exception handling which is designed to handle any runtime errors the code encounters. Exception handling can catch both predictable and unpredictable errors without excessive inline, manual error checking. Taking Java as an example, there are multiple ways to implement exception handling. Below we will cover try-catch blocks and throws. The following type of exception handling is called a try-catch block. It tells the program to try a block of code and, if it doesn't work, catch the exception and run another block of code:

```
public static String readFirstLine(String url) {
   try {
      Scanner scanner = new Scanner(new File(url));
      return scanner.nextLine();
   } catch(FileNotFoundException ex) {
      System.out.println("File not found.");
      return null;
   }
}
```

The next type of exception handling is called a throw. It tells the program to explicitly throw an exception object if specific criteria are met:

```
public class ThrowExample {
  static void checkEligibilty(int stuage, int stuweight){
    if(stuage<12 && stuweight<40) {
      throw new ArithmeticException("Student is not eligible for registration");
    } else {
```

```
        System.out.println("Student Entry is Valid!!");
      }
    }

    public static void main(String args[]){
      System.out.println("Welcome to the Registration process!!");
      checkEligibilty(10, 39);
      System.out.println("Have a nice day..");
   }
}
```

//If the student does not meet the necessary criteria,

//we will encounter the following error message.

Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration

3.2.1 **Common Types of Runtime Error**

To understand what constitutes a runtime error better, let's take a look at some of its common forms, which include:

3.2.1.1 ***Logic Error***

A logic error occurs when a developer enters the wrong statements into the application's source code. With if-then statements, for example, developers would sometimes make the mistake of leaving the logical values to revert to "true." Many runtime errors fall under this category.

3.2.1.2 ***Memory Leak***

Memory leaks happen when a program drains your computer's random access memory (RAM). It often arises from unpatched software, such as when you fail to update your operating system (OS) to the newest release.

3.2.1.3 ***Division by Zero Error***

Division by zero (DIV/0) is an error associated with Excel workbooks. When formula inputs in the spreadsheet are left blank, the total might display a DIV/0 error. The cell formulas need to be formatted in a precise manner to produce the correct output.

3.2.1.4 ***Undefined Object Error***

An undefined object error happens when a program attempts to call a function for a PHP or JavaScript object (or a C++ variable) that isn't defined or assigned a value. The error also occurs for deeply nested objects. In simpler terms, the code "cannot read" or find where a property is because it does not exist or is buried several levels deep within the code.

3.2.1.5 ***Input/Output Device Error***

Input/Output (I/O) device errors occur when issues arise with the read/write function of a device. Common causes include device malfunction, outdated drivers, OS incompatibility, and faulty universal serial bus (USB) ports. As a result, users would get a prompt saying that the device wasn't accessible, making it impossible to transfer or encode files into it. Usually, the memory drive or the computer only needs to be restarted to get rid of the issue.

3.2.1.6 ***Encoding Error***

Encoding errors happen when you're rendering a file, say a video file, to convert it into a usable or accessible file format. This is due to the resource-intensive nature of the encoding process. Error messages linked to this type of error include "encoding overloaded" or "encoding failed."

3.2.2 **How Do You Fix a Runtime Error?**

First off, you need to know that a runtime error occurs due to bugs that the software's programmers knew about but couldn't fix. More generally, though, a runtime error happens due to lack of memory or other system resources required for an application to run properly. The following listed are tips to fix a runtime error:

- Restart your computer. This is an age-old technique that most often than not fixes any problem, including runtime errors.
- Close other applications. It's possible for a runtime error to occur because another program conflicts with the one you're trying to run. In other cases, that other application is using

too many system resources, leaving not enough for the program you wish to load. Close applications that you don't need then try opening the program again.

- Run the application in safe mode. In safe mode, any program runs only the bare minimum so your computer can work. To do this, boot into safe mode then try running the program.
- Update the application. Sometimes, the problem stems from a bug or an error in the program's last release. If you can, update it or manually download its latest version using your browser.
- Reinstall the application. Your program may have been corrupted and needs to be reinstalled. Save important files from it then uninstall and reinstall it.

- Consult a forum or seek a tech expert's advice. If none of the above-mentioned tips work, look for users online facing the same problem. Forums like Reddit can be a valuable resource. You can also try contacting the program's support team.

3.3 Runtime vs Compile time

Runtime and compile time are programming terms that refer to different stages of software program development. Compile-time is the instance where the code you entered is converted to executable while Run-time is the instance where the executable is running. The terms "runtime" and "compile time" are often used by programmers to refer to different types of errors too.

Compile-time checking occurs during the compile time. Compile time errors are error occurred due to typing mistake, if we do not follow the proper syntax and semantics of any programming language then compile time errors are thrown by the compiler. They won't let your program to execute a single line until you remove all the syntax errors or until you debug the compile time errors. The following are usual compile time errors:

- Syntax errors
- Type checking errors
- Compiler crashes (Rarely)

Run-time type checking happens during run time of programs. Runtime errors are the errors that are generated when the program is in running state. These types of errors will cause your program

to behave unexpectedly or may even kill your program. They are often referred as Exceptions. The following are some usual runtime errors:

- Division by zero
- Dereferencing a null pointer
- Running out of memory

4 Self-Assessment Exercises

i. List and explain different type of run-time error
ii. What is run-time
iii. What is run-time error
iv. Explain how run-time error can be fixed
v. What is compile time
vi. Compare and contrast between run-time and compile time

5 Conclusion

Runtime is a technical term, used most often in software development. It is commonly seen in the context of a "runtime error," which is an error that occurs while a program is running. The term "runtime error" is used to distinguish from other types of errors, such as syntax errors and compilation errors, which occur before a program is run.

6 Summary

The unit analyzed the components of high-level languages relating to the control of execution flow in programs. We first considered expressions and we have analyzed the types of syntax that most used for their description (as trees, or in prefix, infix and postfix linear form) and the related evaluation rules. Also, the precedence and associativity rules required for infix notation were debated on. Furthermore, the unit discussed the problems generally related to the order of evaluation of the subexpressions of an expression.