



**NATIONAL OPEN UNIVERSITY OF NIGERIA**

**SCHOOL OF SCIENCE AND TECHNOLOGY**

**COURSE CODE: CIT 342**

**COURSE TITLE: Formal Languages and Automata Theory**



**CIT 342**  
**Formal Languages and Automata Theory**

Course Developer                      Afolorunso, A. A.  
National Open University of Nigeria  
Lagos.

Course Co-ordinator                      Afolorunso, A. A.  
National Open University of Nigeria  
Lagos.

Course Editor

Programme Leader                      Prof. Kehinde Obidairo



**NATIONAL OPEN UNIVERSITY OF NIGERIA**

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

Abuja Annex

245 Samuel Adesujo Ademulegun Street  
Central Business District  
Opposite Arewa Suites  
Abuja

e-mail: [centralinfo@nou.edu.ng](mailto:centralinfo@nou.edu.ng)

URL: [www.nou.edu.ng](http://www.nou.edu.ng)

National Open University of Nigeria 2011

First Printed 2011

ISBN

All Rights Reserved

Printed by .....

For

National Open University of Nigeria

## **TABLE OF CONTENTS**

## **PAGE**

Introduction.....	3
<i>What you will learn in this Course.....</i>	4
Course Aims.....	4

Course Objectives.....	4	
Related Courses	5	
Working through this Course.....	5	
Course Materials.....		5
Study Units .....	5-6	
Textbooks and References .....	6-9	
Assignment File.....	10	
Presentation Schedule.....	10	
<i>Assessment</i> .....	<i>10</i>	
Tutor Marked Assignments (TMAs) .....	11	
Examination and Grading.....	11	
Course Marking Scheme.....	12	
<i>Course Overview</i> .....	<i>12-13</i>	
<i>How to Get the Best from This Course</i> .....	<i>13-15</i>	
Tutors and Tutorials .....		15
<i>Summary</i> .....	<i>15</i>	

## Introduction

**CIT 342 – Formal Languages and Automata Theory** is a two (2) credit unit course of 16 units. The course will cover the important formal languages in the Chomsky hierarchy -- the regular sets, the context-free languages, and the recursively enumerable sets -- as well as the formalisms that generate these languages and the machines that recognize them. The course will also introduce the basic concepts of computability and complexity theory by focusing on the question, "What are the fundamental capabilities and limitations of computers?"

The concepts covered in this course will be amply illustrated by applications to current programming languages, algorithms, natural language processing, and hardware and software design.

Also, in this course we shall investigate whether it is possible at all for a given language to find out if a given word belongs to it or not, and if it is possible how hard it will be. These constitute the fields of *decidability theory* and *complexity theory*, respectively.

What we really want to do is to find out which **problems** can be solved in general, and for those problems that can be solved, how hard it is to solve them. In order to make these questions more precise, we encode problems as languages.

Although the idea of *automaton* is quite old (for example a simple pendulum), it was Post's work, contemporary with Turing, that made possible a general characterization of machines that has been so helpful in the development of ideas ranging from combinational circuits to finite-state languages. Although not as powerful as the machines associated with Chomsky and Turing automata remain a very important tool in the elucidation of the inner workings of machines and provide an excellent starting point in understanding the basic ideas underlying contemporary science.

It is a course for B. Sc. Computer science major students, and is normally taken in a student's third year. It should appeal to anyone who is interested in the design and implementation of programming languages. Anyone who does a substantial amount of programming should find the material valuable.

This course is divided into four modules. The first module deals with the general concepts of formal languages

The second module treats, extensively, regular languages and the class of automata, finite state automata, that recognises strings generated by regular grammar.

The third module deals with context-free languages and pushdown automata

The fourth module which is the concluding module of the course discusses Turing machines and the rest of the language classes

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

### **What you will learn in this course**

The main purpose of this course is to acquaint students with the fact that languages fall into various classes, according to their complexity. Some languages can be parsed, i.e. interpreted, by a very simple state machine. Others require the human brain, or something comparable.

Languages also have many representations: machines that recognize them, expressions that describe them and grammars that generate them.

Thus, we intend to achieve this through the following:

#### **Course Aims**

First, students will learn the key techniques in modern compiler construction, getting prepared for industry demands for compiler engineers.

Second, students will understand the rationale of various computational methods and analysis.

The third goal is to build the foundation for students to pursue the research in the areas of automata theory, formal languages, and computational power of machines

#### **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. Upon completing this course you should be able to:

- Discovering computational thinking
- Understanding the fundamental models of computation that underlie modern computer hardware, software, and programming languages.
- Discovering that there are problems no computer can solve.
- Discovering that there are limits as to how fast a computer can solve a problem.
- Learning the foundations of automata theory, computability theory, and complexity theory.
- Learning about applications of theory to other areas of computer science such as algorithms, programming languages, compilers, natural language translation, operating systems, and software verification.

#### **Related Courses**

Prerequisites: CIT 331; Computer Science students only

#### **Working through This Course**

In order to have a thorough understanding of the course units, you will need to read and understand the contents, practise the steps by designing a compiler of your own for a known language, and be committed to learning and implementing your knowledge.

This course is designed to cover approximately seventeen 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 16 study units in this course:

#### Module 1: General Concepts

- Unit 1: Alphabets, Strings, and Representations
- Unit 2: Formal Grammars
- Unit 3: Formal Languages
- Unit 4: Automata Theory

#### Module 2: Regular Languages

- Unit 1: Finite State Automata
- Unit 2: Regular Expressions
- Unit 3: Regular Grammars
- Unit 4: Closure Properties of Regular Languages
- Unit 5: The Pumping Lemma

#### Module 3: Context-Free Languages

- Unit 1: Context-Free Grammars
- Unit 2: Properties of Context-Free Languages
- Unit 3: Pushdown Automata
- Unit 4: CFGs and PDAs

#### Module 4: Turing Machines

- Unit 1: Turing Machines and the rest
- Unit 2: Turing Machines and Context-Sensitive Grammars
- Unit 3: Unrestricted Grammars

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

### **Textbooks and References**

1. Bryant, Randal E.; David, O'Hallaron (2003), *Computer Systems: A Programmer's Perspective* (2003 ed.), Upper Saddle River, NJ: Pearson Education
2. John E. Hopcroft and Jeffrey D. Ullman, [Introduction to Automata Theory, Languages, and Computation](#), Addison-Wesley Publishing, Reading Massachusetts, 1979
3. Chomsky, Noam (1956). "Three Models for the Description of Language". *IRE Transactions on Information Theory* **2** (2): 113–123.
4. Chomsky, Noam (1957). *Syntactic Structures*. The Hague: Mouton.
5. Ginsburg, Seymour (1975). *Algebraic and automata theoretic properties of formal languages*. North-Holland. pp. 8–9..
6. Harrison, Michael A. (1978). *Introduction to Formal Language Theory*. Reading, Mass.: Addison-Wesley Publishing Company.
7. Sentential Forms, Context-Free Grammars, David Matuszek
8. Grune, Dick & Jacobs, Criel H., *Parsing Techniques – A Practical Guide*, Ellis Horwood, England, 1990.
9. Earley, Jay, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, Vol. 13 No. 2, pp. 94-102, February 1970.
10. Joshi, Aravind K., *et al.*, "Tree Adjunct Grammars," *Journal of Computer Systems Science*, Vol. 10 No. 1, pp. 136-163, 1975.
11. Koster, Cornelis H. A., "Affix Grammars," in *ALGOL 68 Implementation*, North Holland Publishing Company, Amsterdam, p. 95-109, 1971.
12. Knuth, Donald E., "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Vol. 2 No. 2, pp. 127-145, 1968.
13. Knuth, Donald E., "Semantics of Context-Free Languages (correction)," *Mathematical Systems Theory*, Vol. 5 No. 1, pp 95-96, 1971.
14. Birman, Alexander, *The TMG Recognition Schema*, Doctoral thesis, Princeton University, Dept. of Electrical Engineering, February 1970.
15. Sleator, Daniel D. & Temperly, Davy, "Parsing English with a Link Grammar," Technical Report CMU-CS-91-196, Carnegie Mellon University Computer Science, 1991.
16. Sleator, Daniel D. & Temperly, Davy, "Parsing English with a Link Grammar," *Third International Workshop on Parsing Technologies*, 1993. (Revised version of above report.)
17. Ford, Bryan, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master's thesis, Massachusetts Institute of Technology, Sept. 2002.
18. John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
19. Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
20. James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**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 sixteen 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 Assignments (TMAS)**

There are twenty-two tutor marked assignments in this course. You need to submit all the assignments. The total marks for the best three (3) 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 learning 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

**Table 1: Course Marking Scheme**

### Course Overview

Unit	Title of Work	Weeks Activity	Assessment (End of Unit)
	Course Guide	Week 1	
	Module 1: General Concepts		
1	Unit 1: Alphabets, Strings, and Representations	Week 1	Assignment 1

2	Unit 2: Formal Grammars	Week 2	Assignment 2
3	Unit 3: Formal Languages	Week 2	Assignment 3
4	Unit 4: Automata Theory		
	Module 2: Regular Languages		
1	Unit 1: Finite State Automata	Week 3	Assignment 5
2	Unit 2: Regular Expressions	Week 3	Assignment 6
3	Unit 3: Regular Grammars	Week 4	Assignment 7
4	Unit 4: Closure Properties of Regular Languages	Week 4	
5	Unit 5: The Pumping Lemma		
	Module 3: Context-Free Languages		
1	Unit 1: Context-Free Grammars	Week 5	Assignment 8
2	Unit 2: Properties of Context-Free Languages	Week 6	Assignment 9
3	Unit 3: Pushdown Automata	Week 7 - 8	Assignment 10
4	Unit 4: CFGs and PDAs	Week 8 - 9	Assignment 11
	Module 4: Turing Machines		
1	Unit 1: Turing Machines and the rest	Week 12	Assignment 13
2	Unit 2: Turing Machines and Context-Sensitive Grammars	Week 13	Assignment 14
3	Unit 3: Unrestricted Grammars	Week 14	Assignment 15
	Revision	Week 16	
	Examination	Week 17	
Total		17 weeks	

### How to get the best from this course

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

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other

units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

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

1. Read this Course Guide thoroughly.
2. Organize a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.
7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.

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

## **Tutors and Tutorials**

There are 8 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**

Formal Languages and Automata Theory introduces you to the concepts associated languages, computation and machines. The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills, which you will find useful in a later course (CIT 445: Principles and Techniques of Compilers) in your fourth year and also for the appropriate situations later in life. Real-life situations have been created to enable you identify with and create some of your own. The essence is to help you in acquiring the necessary knowledge and competence by equipping you with the necessary tools to accomplish this.

We hope that by the end of this course you would have acquired the required knowledge to view computation, machines, and programming languages in a new way.

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

## Module 1: General Concepts

### Unit 1: Alphabets, Strings, and Representations

#### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Alphabets
  - 3.2 String
    - 3.2.1 Formal Theory
    - 3.2.2 Strings and Sets of Strings
    - 3.2.3 Alphabet of a string
    - 3.2.4 String substitution
    - 3.2.5 Concatenation and Substrings
    - 3.2.6 String length
    - 3.2.7 Character String Functions
  - 3.3 Representations
- 4.0 *Conclusion*
- 5.0 *Summary*
- 6.0 *Tutor-Marked Assignment*
- 7.0 References/Further Reading

#### 1.0 INTRODUCTION

The ability to represent information is crucial to communicating and processing information. Human societies created spoken languages to communicate on a basic level, and developed writing to reach a more sophisticated level.

The English language, for instance, in its spoken form relies on some finite set of basic sounds as a set of primitives. The words are defined in term of finite sequences of such sounds. Sentences are derived from finite sequences of words. Conversations are achieved from finite sequences of sentences, and so forth.

Written English uses some finite set of symbols as a set of primitives. The words are defined by finite sequences of symbols. Sentences are derived from finite sequences of words. Paragraphs are obtained from finite sequences of sentences, and so forth.

Similar approaches have been developed also for representing elements of other sets. For instance, the natural number can be represented by finite sequences of decimal digits.

Computations, like natural languages, are expected to deal with information in its most general form. Consequently, computations function as manipulators of integers,

graphs, programs, and many other kinds of entities. However, in reality computations only manipulate strings of symbols that represent the objects. The subsequent discussions in this course necessitate the following definitions.

In this introductory unit of this course, you will be taken through some of these definitions.

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

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

- define alphabet, words and strings
- state the basic relationship among these terms
- state and describe the various operations that can be carried out on these structures
- describe how they can be represented

## 3.0 MAIN CONTENT

### 3.1 Alphabets

In [computer science](#) and formal language, an **alphabet** or **vocabulary** is a finite set of [symbols](#) or *letters*, e.g. characters or digits. The most common alphabet is  $\{0,1\}$ , the **binary alphabet**. A finite [string](#) is a finite sequence of letters from an alphabet; for instance a binary string is a string drawn from the alphabet  $\{0,1\}$ . An infinite [sequence](#) of letters may be constructed from elements of an alphabet as well.

Given an alphabet  $\Sigma$ , we write  $\Sigma^*$  to denote the set of all finite strings over the alphabet  $\Sigma$ . Here, the  $*$  denotes the [Kleene star](#) operator. We write  $\Sigma^{\infty}$  (or occasionally,  $\Sigma^{\mathbb{N}}$  or  $\Sigma^{\omega}$ ) to denote the set of all infinite sequences over the alphabet  $\Sigma$ .

For example, if we use the binary alphabet  $\{0,1\}$ , the strings ( $\epsilon$ , 0, 1, 00, 01, 10, 11, 000, etc.) would all be in the Kleene closure of the alphabet (where  $\epsilon$  represents the [empty string](#))

Please note that alphabets are important in the use of formal languages, automata and semi-automata. In most cases, for defining instances of automata, such as deterministic finite automata (DFAs), it is required to specify an alphabet from which the input strings for the automaton are built.

### 3.2 String

In formal languages, which are used in mathematical logic and theoretical computer science, a **string** is a finite sequence of symbols that are chosen from a set or alphabet.

In computer programming, a string is, essentially, a sequence of characters. A string is generally understood as a data type storing a sequence of data values, usually bytes, in which elements usually stand for characters according to a character encoding, which differentiates it from the more general array data type. In this context, the terms **binary string** and **byte string** are used to suggest strings in which the stored data does not (necessarily) represent text.

A [variable](#) declared to have a string data type usually causes storage to be allocated in memory that is capable of holding some predetermined number of symbols. When a string appears literally in [source code](#), it is known as a [string literal](#) and has a representation that denotes it as such.

### 3.2.1 Formal Theory

Let  $\Sigma$  be an [alphabet](#), a [non-empty finite set](#). Elements of  $\Sigma$  are called *symbols* or *characters*. A **string** (or **word**) over  $\Sigma$  is any finite [sequence](#) of characters from  $\Sigma$ . For example, if  $\Sigma = \{0, 1\}$ , then *0101* is a string over  $\Sigma$ .

The [length](#) of a string is the number of characters in the string (the length of the sequence) and can be any [non-negative integer](#). The [empty string](#) is the unique string over  $\Sigma$  of length 0, and is denoted  $\epsilon$  or  $\lambda$ .

The set of all strings over  $\Sigma$  of length  $n$  is denoted  $\Sigma^n$ . For example, if  $\Sigma = \{0, 1\}$ , then  $\Sigma^2 = \{00, 01, 10, 11\}$ . Note that  $\Sigma^0 = \{\epsilon\}$  for any alphabet  $\Sigma$ .

The set of all strings over  $\Sigma$  of any length is the [Kleene closure](#) of  $\Sigma$  and is denoted  $\Sigma^*$ . In terms of  $\Sigma^n$ ,

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$$

For example, if  $\Sigma = \{0, 1\}$ ,  $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$ . Although  $\Sigma^*$  itself is [countably infinite](#), all elements of  $\Sigma^*$  have finite length.

A set of strings over  $\Sigma$  (i.e. any [subset](#) of  $\Sigma^*$ ) is called a [formal language](#) over  $\Sigma$ . For example, if  $\Sigma = \{0, 1\}$ , the set of strings with an even number of zeros ( $\{\epsilon, 1, 00, 11, 001, 010, 100, 111, 0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111, \dots\}$ ) is a formal language over  $\Sigma$ .

### 3.2.2 Strings and Sets of Strings

If  $V$  is a set, then  $V^*$  denotes the set of all finite strings of elements of  $V$  including the empty string which will be denoted by  $\epsilon$ . e.g.  $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

The set of all non empty strings of elements of  $V$  is denoted by  $V^+$ .

Usually,  $V^+ = V^* \setminus \{\epsilon\}$ , but when  $\epsilon \in V$ ,  $V^+ = V^*$ . e.g.

$$\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

but  $\{\epsilon, 0, 1\}^+ = \{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

If  $x \in V^*$  and  $y \in V^*$  then  $xy$  will denote their concatenation, that is, the string consisting of  $x$  followed by  $y$ .

If  $x \in V^*$  then  $x^n = \underbrace{xxx\dots x}_{n\text{-times}}$   $n \geq 0$

We assume  $x^0 = \epsilon$  the empty string.

e.g.  $\{a\}^* = \{\epsilon, a, a^2, a^3, \dots, a^n, \dots\} = \{a^n: n \geq 0\}$

$$\{a\}^+ = \{a, a^2, a^3, \dots, a^n, \dots\} = \{a^n: n \geq 1\}$$

Similarly, if  $X, Y$  are sets of strings, then their concatenation is also denoted

by  $XY$ . Of course  $XY = \{xy: x \in X \text{ and } y \in Y\}$ .

Also,  $X^n = \underbrace{XXX\dots X}_{n\text{-times}}$   $n \geq 0$ . Of course  $X^0 = \{\epsilon\}$ .

e.g.  $\{0, 1\} \{a, b, c\} = \{0a, 0b, 0c, 1a, 1b, 1c\}$   
 $\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

If  $x$  is a string, then  $|x|$  denotes the length of  $x$ , and this is the number of indivisible symbols in  $x$ . Of course  $|\epsilon| = 0$ .

### Self assessment Exercise

1) Determine the following sets.

(a)  $\{0,1\} \{ \epsilon, a, ba \}$       (b)  $\{b, aa\}^*$

2) Let  $V$  be a set of strings. Does  $V^+ = V V^*$  ?

### 3.2.3 Alphabet of a string

The **alphabet of a string** is a list of all of the letters that occur in a particular string. If  $s$  is a string, its alphabet is denoted by

$$\text{Alph}(s)$$

### 3.2.4 String substitution

Let  $L$  be a language, and let  $\Sigma$  be its alphabet. A **string substitution** or simply a **substitution** is a mapping  $f$  that maps letters in  $\Sigma$  to languages (possibly in a different alphabet). Thus, for example, given a letter  $a \in \Sigma$ , one has  $f(a) = L_a$  where  $L_a \subset \Delta^*$  is some language whose alphabet is  $\Delta$ . This mapping may be extended to strings as

$$f(\varepsilon) = \varepsilon$$

for the empty string  $\varepsilon$ , and

$$f(sa) = f(s)f(a)$$

for string  $a \in L$ . String substitution may be extended to the entire language as

$$f(L) = \bigcup_{s \in L} f(s)$$

An example of string substitution occurs in regular languages, which are closed under string substitution. That is, if the letters of a regular language are substituted by other regular languages, the result is still a regular language.

Another example is the conversion of an EBCDIC-encoded string to ASCII.

### 3.2.5 Concatenation and Substrings

**Concatenation** is an important **binary operation** on  $\Sigma^*$ . For any two strings  $s$  and  $t$  in  $\Sigma^*$ , their concatenation is defined as the sequence of characters in  $s$  followed by the sequence of characters in  $t$ , and is denoted  $st$ . For example, if  $\Sigma = \{a, b, \dots, z\}$ ,  $s = \text{bear}$ , and  $t = \text{hug}$ , then  $st = \text{bearhug}$  and  $ts = \text{hugbear}$ .

String concatenation is an **associative**, but non-**commutative** operation. The empty string serves as the **identity element**; for any string  $s$ ,  $\varepsilon s = s\varepsilon = s$ . Therefore, the set  $\Sigma^*$  and the concatenation operation form a **monoid**, the **free monoid** generated by  $\Sigma$ . In addition, the length function defines a **monoid homomorphism** from  $\Sigma^*$  to the non-negative integers.

A string  $s$  is said to be a **substring** or *factor* of  $t$  if there exist (possibly empty) strings  $u$  and  $v$  such that  $t = usv$ . The **relation** "is a substring of" defines a **partial order** on  $\Sigma^*$ , the **least element** of which is the empty string.

### 3.2.6 String length

Although formal strings can have an arbitrary (but finite) length, the length of strings in real languages is often constrained to an artificial maximum. In general, there are two types of string datatypes: *fixed length strings* which have a fixed maximum length and which use the same amount of memory whether this maximum is reached or not, and *variable length strings* whose length is not arbitrarily fixed and which use varying

amounts of memory depending on their actual size. Most strings in modern programming languages are variable length strings. Despite the name, even variable length strings are limited in length; although, generally, the limit depends only on the amount of memory available.

### 3.2.7 Character String Functions

String functions are used to manipulate a string or change or edit the contents of a string. They also are used to query information about a string. They are usually used within the context of a computer programming language.

The most basic example of a string function is the `length(string)` function, which returns the length of a string (not counting any terminator characters or any of the string's internal structural information) and does not modify the string. For example, `length("hello world")` returns 11.

There are many string functions which exist in other languages with similar or exactly the same syntax or parameters. For example in many languages the length function is usually represented as `len(string)`. Even though string functions are very useful to a computer programmer, a computer programmer using these functions should be mindful that a string function in one language could in another language behave differently or have a similar or completely different function name, parameters, syntax, and results.

## 3.3 Representations

Given the preceding definitions of alphabets and strings, representations of information can be viewed as the mapping of objects into strings in accordance with some rules. That is, formally speaking, a *representation* or *encoding* over an alphabet  $\Sigma$  of a set  $D$  is a function  $f$  from  $D$  to  $2^{\Sigma^*}$  that satisfies the following condition:  $f(e_1)$  and  $f(e_2)$  are disjoint nonempty sets for each pair of distinct elements  $e_1$  and  $e_2$  in  $D$ .

If  $\Sigma$  is a unary alphabet, then the representation is said to be a *unary representation*. If  $\Sigma$  is a binary alphabet, then the representation is said to be a *binary representation*.

In what follows each element in  $f(e)$  will be referred to as a representation, or encoding, of  $e$ .

### Example 1

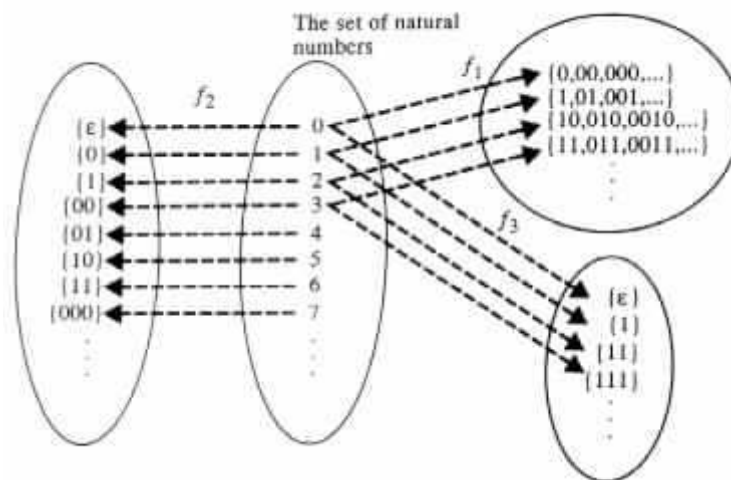
$f_1$  is a binary representation over  $\{0, 1\}$  of the natural numbers if  $f_1(0) = \{0, 00, 000, 0000, \dots\}$ ,  $f_1(1) = \{1, 01, 001, 0001, \dots\}$ ,  $f_1(2) = \{10, 010, 0010, 00010, \dots\}$ ,  $f_1(3) = \{11, 011, 0011, 00011, \dots\}$ , and  $f_1(4) = \{100, 0100, 00100, 000100, \dots\}$ , etc.

Similarly,  $f_2$  is a binary representation over  $\{0, 1\}$  of the natural numbers if it assigns to the  $i$ th natural number the set consisting of the  $i$ th canonically smallest binary

string. In such a case,  $f_2(0) = \{\epsilon\}$ ,  $f_2(1) = \{0\}$ ,  $f_2(2) = \{1\}$ ,  $f_2(3) = \{00\}$ ,  $f_2(4) = \{01\}$ ,  $f_2(5) = \{10\}$ ,  $f_2(6) = \{11\}$ ,  $f_2(7) = \{000\}$ ,  $f_2(8) = \{1000\}$ ,  $f_2(9) = \{1001\}$ ,  $\dots$

On the other hand,  $f_3$  is a unary representation over  $\{1\}$  of the natural numbers if it assigns to the  $i$ th natural number the set consisting of the  $i$ th alphabetically (= canonically) smallest unary string. In such a case,  $f_3(0) = \{\epsilon\}$ ,  $f_3(1) = \{1\}$ ,  $f_3(2) = \{11\}$ ,  $f_3(3) = \{111\}$ ,  $f_3(4) = \{1111\}$ ,  $\dots$ ,  $f_3(i) = \{1^i\}$ ,  $\dots$

The three representations  $f_1$ ,  $f_2$ , and  $f_3$  are illustrated in Figure 1



**Figure 1. Representations for the natural numbers.**

#### 4.0 CONCLUSION

In this unit you have been taken through some fundamental concepts in formal language. It is advised you master these concepts as a solid knowledge of these foundational concepts will aid your mastery and understanding of this course.

#### 5.0 SUMMARY

In this unit, you learnt that:

- an **alphabet** or **vocabulary** is a finite set of [symbols](#) or *letters*
- a **string** is a finite sequence of symbols that are chosen from a set or alphabet
- String functions are used to manipulate a string or change or edit the contents of a string
- For any two strings  $s$  and  $t$  in  $\Sigma^*$ , their concatenation is defined as the sequence of characters in  $s$  followed by the sequence of characters in  $t$ , and is denoted  $st$
- String concatenation is an [associative](#), but non-[commutative](#) operation
- A string  $s$  is said to be a [substring](#) or *factor* of  $t$  if there exist (possibly empty) strings  $u$  and  $v$  such that  $t = usv$

- a *representation* or *encoding* over an alphabet  $\Sigma$  of a set  $D$  is a function  $f$  from  $D$  to  $2^{\Sigma^*}$  that satisfies the following condition:  $f(e_1)$  and  $f(e_2)$  are disjoint nonempty sets for each pair of distinct elements  $e_1$  and  $e_2$  in  $D$ .

## 6.0 TUTOR-MARKED ASSIGNMENT

- 1) Define the following terms:
  - Strings
  - Alphabets
  - Vocabulary
- 2) If  $\Sigma = \{a, b, \dots, z\}$ ,  $s = \text{bear}$ , and  $t = \text{hug}$ , then find
  - i)  $st$
  - ii)  $ts$
- 3) Briefly describe the following:
  - i) unary representation
  - ii) binary representation

## 7.0 REFERENCES/FURTHER READING

- 1) Bryant, Randal E.; David, O'Hallaron (2003), *Computer Systems: A Programmer's Perspective* (2003 ed.), Upper Saddle River, NJ: Pearson Education
- 2) John E. Hopcroft and Jeffrey D. Ullman, [Introduction to Automata Theory, Languages, and Computation](#), Addison-Wesley Publishing, Reading Massachusetts, 1979

**Module 1: General Concepts****Unit 2: Formal Grammars****CONTENTS**

- 1.0 *Introduction*
- 2.0 *Objectives*
- 3.0 *Main Content*
  - 1.1 Formal Grammar
    - 3.1.1 Introductory Example
  - 3.2 Formal Definition
    - 3.2.1 The Syntax of Grammars
    - 3.2.2 The Semantics of Grammars
  - 3.3 The Chomsky Hierarchy
  - 3.4 Context-Free Grammars
  - 3.5 Regular Grammars
  - 3.6 Other Forms of Generative Grammars
  - 3.7 Analytic Grammars
- 4.0 *Conclusion*
- 5.0 *Summary*
- 6.0 *Tutor-Marked Assignment*
- 7.0 *References/Further Reading*

**1.0 INTRODUCTION**

Having learnt about strings and alphabets in the previous unit, you will be taken through another important concept in formal language and automata theory, which is grammar. This is because it is often convenient to specify languages in terms of grammars. The advantage in doing so arises mainly from the usage of a small number of rules for describing a language with a large number of sentences.

For instance, the possibility that an English sentence consists of a subject phrase followed by a predicate phrase can be expressed by a grammatical rule of the form:

$\langle \text{sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle$ . (The names in angular brackets are assumed to belong to the grammar metalanguage.)

Similarly, the possibility that the subject phrase consists of a noun phrase can be expressed by a grammatical rule of the form:

$\langle \text{subject} \rangle \rightarrow \langle \text{noun} \rangle$ .

You may, therefore, think of a grammar as a set of rules for your native language. Subject, predicate, prepositional phrase, past participle, and so on. This is a reasonably accurate, or at least helpful, description of a human language, but it is not entirely rigorous. Chomski formalized the concept of a grammar, and made important

observations regarding the complexity of the grammar, which in turn establishes the complexity of the language.

In this unit, you will be taken through some basic concepts of formal grammar

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

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

- define formal grammar
- define alphabet, words and strings
- state the types of formal grammars we have in the field of Computer Science
- describe the class of automata that can recognise strings generated by each grammar
- identify strings that are generated by a particular grammar
- describe the Chomsky hierarchy
- explain the relevance of formal grammar and language to computer programming

## 3.0 MAIN CONTENT

### 3.1 Formal Grammar

A **formal grammar** (sometimes simply called a **grammar**) is a set of rules for forming strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context – only their form.

Formal language theory, the discipline which studies formal grammars and languages, is a branch of applied mathematics. Its applications are found in theoretical computer science, theoretical linguistics, formal semantics, mathematical logic, and other areas.

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

Parsing is the process of recognizing an utterance (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language. Most languages have the meanings of their utterances structured according to their syntax—a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down part by part and look at its analyzed form (known as its parse tree in computer science, and as its deep structure in generative grammar).

### 3.1.1 Introductory Example

A grammar mainly consists of a set of rules for transforming strings. (If it *only* consisted of these rules, it would be a semi-Thue system.) To generate a string in the language, one begins with a string consisting of only a single *start symbol*. The *production rules* are then applied in any order, until a string that contains neither the start symbol nor designated *nonterminal symbols* is produced. The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.

For example, assume the alphabet consists of  $a$  and  $b$ , the start symbol is  $S$ , and we have the following production rules:

1.  $S \rightarrow aSb$
2.  $S \rightarrow ba$

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

## 3.2 Formal Definition

### 3.2.1 The Syntax of Grammars

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s, a grammar  $G$  consists of the following components:

- A finite set  $N$  of *nonterminal symbols*, none of which appear in strings formed from  $G$ .
- A finite set  $\Sigma$  of *terminal symbols* that is disjoint from  $N$ .
- A finite set  $P$  of *production rules*, each rule of the form

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

where  $*$  is the Kleene star operator and  $\cup$  denotes set union. That is, each production rule maps from one string of symbols to another, where the first string (the "head") contains an arbitrary number of symbols provided at least one of them is a nonterminal. In the case that the second string (the "body") consists solely of the empty string – i.e., that it contains no symbols at all – it may be denoted with a special notation (often  $\Lambda$ ,  $e$  or  $\varepsilon$ ) in order to avoid confusion.

- A distinguished symbol  $S \in N$  that is the *start symbol*.

A grammar is formally defined as the tuple  $(N, \Sigma, P, S)$ . Such a formal grammar is often called a rewriting system or a phrase structure grammar in the literature.

### 3.2.2 The Semantics of Grammars

The operation of a grammar can be defined in terms of relations on strings:

- Given a grammar  $G = (N, \Sigma, P, S)$ , the binary relation  $\Rightarrow_G$  (pronounced as "G derives in one step") on strings in  $(\Sigma \cup N)^*$  is defined by:

$$x \Rightarrow_G y \text{ iff } \exists u, v, p, q \in (\Sigma \cup N)^* : (x = upv) \wedge (p \rightarrow q \in P) \wedge (y = uqv)$$

- the relation  $\Rightarrow_G^*$  (pronounced as *G derives in zero or more steps*) is defined as the reflexive transitive closure of  $\Rightarrow_G$
- a *sentential form* is a member of  $(\Sigma \cup N)^*$  that can be derived in a finite number of steps from the start symbol  $S$ ; that is, a sentential form is a member of  $\{w \in (\Sigma \cup N)^* \mid S \Rightarrow_G^* w\}$ . A sentential form that contains no nonterminal symbols (i.e. is a member of  $\Sigma^*$ ) is called a *sentence*.
- the *language* of  $G$ , denoted as  $L(G)$ , is defined as all those sentences that can be derived in a finite number of steps from the start symbol  $S$ ; that is, the set  $\{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$ .

Note that the grammar  $G = (N, \Sigma, P, S)$  is effectively the semi-Thue system  $(N \cup \Sigma, P)$ , rewriting strings in exactly the same way; the only difference is in that we distinguish specific *nonterminal* symbols which must be rewritten in rewrite rules, and are only interested in rewritings from the designated start symbol  $S$  to strings without nonterminal symbols.

#### Example 1

*Please note that for these examples, formal languages are specified using set-builder notation.*

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

1.  $S \rightarrow aBSc$
2.  $S \rightarrow abc$
3.  $Ba \rightarrow aB$
4.  $Bb \rightarrow bb$

This grammar defines the language

$$L(G) = \{a^n b^n c^n | n \geq 1\}$$

where  $a^n$  denotes a string of  $n$  consecutive  $a$ 's. Thus, the language is the set of strings that consist of 1 or more  $a$ 's, followed by the same number of  $b$ 's, followed by the same number of  $c$ 's.

Some examples of the derivation of strings in  $L(G)$  are:

- $S \Rightarrow_2 abc$
- $S \Rightarrow_1 aBSc \Rightarrow_2 aBabcc \Rightarrow_3 aaBbcc \Rightarrow_4 aabbcc$
- $S \Rightarrow_1 aBSc \Rightarrow_1 aBaBSc \Rightarrow_2 aBaBabccc \Rightarrow_3 aaBBabccc \Rightarrow_3 aaBaBbcc \Rightarrow_3 aaaBBbcc \Rightarrow_4 aaaBbbccc \Rightarrow_4 aaabbbccc$

(Note on notation:  $P \Rightarrow_i Q$  reads "String  $P$  generates string  $Q$  by means of production  $i$ ", and the generated part is each time indicated in bold type.)

### 3.3 The Chomsky Hierarchy

When Noam Chomsky first formalized generative grammars in 1956, he classified them into types now known as the Chomsky hierarchy. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. Two important types are *context-free grammars* (Type 2) and *regular grammars* (Type 3). The languages that can be described with such a grammar are called *context-free languages* and *regular languages*, respectively. Although much less powerful than unrestricted grammars (Type 0), which can in fact express any language that can be accepted by a Turing machine, these two restricted types of grammars are most often used because parsers for them can be efficiently implemented. For example, all regular languages can be recognized by a finite state machine, and for useful subsets of context-free grammars there are well-known algorithms to generate efficient LL parsers and LR parsers to recognize the corresponding languages those grammars generate.

### 3.4 Context-Free Grammars

A *context-free grammar* is a grammar in which the left-hand side of each production rule consists of only a single nonterminal symbol. This restriction is non-trivial; not

all languages can be generated by context-free grammars. Those that can are called *context-free languages*.

The language defined above is not a context-free language, and this can be strictly proven using the pumping lemma for context-free languages, but for example the language  $L(G) = \{a^n b^n | n \geq 1\}$  (at least 1  $a$  followed by the same number of  $b$ 's) is context-free, as it can be defined by the grammar  $G_2$  with  $N = \{S\}$ ,  $\Sigma = \{a, b\}$ ,  $S$  the start symbol, and the following production rules:

1.  $S \rightarrow aSb$
2.  $S \rightarrow ab$

A context-free language can be recognized in  $O(n^3)$  time (see Big O notation) by an algorithm such as Earley's algorithm. That is, for every context-free language, a machine can be built that takes a string as input and determines in  $O(n^3)$  time whether the string is a member of the language, where  $n$  is the length of the string. Further, some important subsets of the context-free languages can be recognized in linear time using other algorithms.

### 3.5 Regular Grammars

In regular grammars, the left hand side is again only a single nonterminal symbol, but now the right-hand side is also restricted. The right side may be the empty string, or a single terminal symbol, or a single terminal symbol followed by a nonterminal symbol, but nothing else. (Sometimes a broader definition is used: one can allow longer strings of terminals or single nonterminals without anything else, making languages easier to denote while still defining the same class of languages.)

The language defined above is not regular, but the language  $\{a^n b^m | m, n \geq 1\}$  (at least 1  $a$  followed by at least 1  $b$ , where the numbers may be different) is, as it can be defined by the grammar  $G_3$  with  $N = \{S, A, B\}$ ,  $\Sigma = \{a, b\}$ ,  $S$  the start symbol, and the following production rules:

1.  $S \rightarrow aA$
2.  $A \rightarrow aA$
3.  $A \rightarrow bB$
4.  $B \rightarrow bB$
5.  $B \rightarrow \epsilon$

All languages generated by a regular grammar can be recognized in linear time by a finite state machine. Although, in practice, regular grammars are commonly expressed using regular expressions, some forms of regular expression used in practice do not strictly generate the regular languages and do not show linear recognitional performance due to those deviations.

### 3.6 Other Forms of Generative Grammars

Many extensions and variations on Chomsky's original hierarchy of formal grammars have been developed, both by linguists and by computer scientists, usually either in order to increase their expressive power or in order to make them easier to analyze or parse. Some forms of grammars developed include:

- Tree-adjoining grammars increase the expressiveness of conventional generative grammars by allowing rewrite rules to operate on parse trees instead of just strings.
- Affix grammars and attribute grammars allow rewrite rules to be augmented with semantic attributes and operations, useful both for increasing grammar expressiveness and for constructing practical language translation tools.

### 3.7 Analytic Grammars

Though there is a tremendous body of literature on parsing algorithms, most of these algorithms assume that the language to be parsed is initially *described* by means of a *generative* formal grammar, and that the goal is to transform this generative grammar into a working parser. Strictly speaking, a generative grammar does not in any way correspond to the algorithm used to parse a language, and various algorithms have different restrictions on the form of production rules that are considered well-formed.

An alternative approach is to formalize the language in terms of an analytic grammar in the first place, which more directly corresponds to the structure and semantics of a parser for the language. Examples of analytic grammar formalisms include the following:

- The Language Machine directly implements unrestricted analytic grammars. Substitution rules are used to transform an input to produce outputs and behaviour. The system can also produce the lm-diagram which shows what happens when the rules of an unrestricted analytic grammar are being applied.
- Top-down parsing language (TDPL): a highly minimalist analytic grammar formalism developed in the early 1970s to study the behavior of top-down parsers.
- Link grammars: a form of analytic grammar designed for linguistics, which derives syntactic structure by examining the positional relationships between pairs of words.
- Parsing expression grammars (PEGs): a more recent generalization of TDPL designed around the practical expressiveness needs of programming language and compiler writers.

## 4.0 CONCLUSION

In this unit you have been introduced to the concept of formal grammars. Grammars are very important in the field of automata theory since they are the building blocks of languages. In the next unit we will be discussing formal languages.

## 5.0 SUMMARY

In this unit, you learnt that a **formal grammar** is a set of rules of a specific kind, for forming strings in a formal language. It has four components that form its syntax and a set of operations that can be performed on it, which form its semantic.

Each type of grammars is recognised by a particular type of automata. For example, type-2 grammars are recognised by pushdown automata while type-3 grammars are recognised by finite state automata.

According to Chomsky hierarchy, there are four types of grammars. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. What you understand by grammars.
2. Briefly describe the following:
  - operations on a grammar
  - semantics of a grammar
3. Distinguish among the following grammar types:
  - Regular grammars
  - Context-free grammars
  - Analytical grammars
4. Briefly discuss the Chomsky hierarchy. What is the relationship among the various types of grammars described in the Chomsky hierarchy?

## 7.0 REFERENCES/FURTHER READING

1. Chomsky, Noam (1956). "Three Models for the Description of Language". *IRE Transactions on Information Theory* 2 (2): 113–123.
2. Chomsky, Noam (1957). *Syntactic Structures*. The Hague: Mouton.
3. Ginsburg, Seymour (1975). *Algebraic and automata theoretic properties of formal languages*. North-Holland. pp. 8–9..
4. Harrison, Michael A. (1978). *Introduction to Formal Language Theory*. Reading, Mass.: Addison-Wesley Publishing Company.
5. Sentential Forms, Context-Free Grammars, David Matuszek
6. Grune, Dick & Jacobs, Criel H., *Parsing Techniques – A Practical Guide*, Ellis Horwood, England, 1990.
7. Earley, Jay, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, Vol. 13 No. 2, pp. 94-102, February 1970.
8. Joshi, Aravind K., *et al.*, "Tree Adjunct Grammars," *Journal of Computer Systems Science*, Vol. 10 No. 1, pp. 136-163, 1975.

9. Koster , Cornelis H. A., "Affix Grammars," in *ALGOL 68 Implementation*, North Holland Publishing Company, Amsterdam, p. 95-109, 1971.
10. Knuth, Donald E., "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Vol. 2 No. 2, pp. 127-145, 1968.
11. Knuth, Donald E., "Semantics of Context-Free Languages (correction)," *Mathematical Systems Theory*, Vol. 5 No. 1, pp 95-96, 1971.
12. Birman, Alexander, *The TMG Recognition Schema*, Doctoral thesis, Princeton University, Dept. of Electrical Engineering, February 1970.
13. Sleator, Daniel D. & Temperly, Davy, "Parsing English with a Link Grammar," Technical Report CMU-CS-91-196, Carnegie Mellon University Computer Science, 1991.
14. Sleator, Daniel D. & Temperly, Davy, "Parsing English with a Link Grammar," *Third International Workshop on Parsing Technologies*, 1993. (Revised version of above report.)
15. Ford, Bryan, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master's thesis, Massachusetts Institute of Technology, Sept. 2002.

**Module 1: General Concepts****Unit 3: Formal Languages****CONTENTS**

- 1.0 *Introduction*
- 2.0 *Objectives*
- 3.0 *Main Content*
  - 3.1 Formal Language
  - 3.2 Words over an Alphabet
    - 3.2.1 Formal Definition
    - 3.2.2 Vocabulary and Language
  - 3.3 Language-Specification Formalisms
  - 3.4 Operations on Languages
  - 3.5 Other Operations on Languages
  - 3.6 Derivations and Language of a Grammar
- 4.0 *Conclusion*
- 5.0 *Summary*
- 6.0 *Tutor-Marked Assignment*
- 7.0 *References/Further Reading*

**1.0 INTRODUCTION**

You may think of a language as English or French, or perhaps perl or java, but there is a formal definition that is much more general. It encompasses these languages, and other, abstract languages such as the prime numbers, or the valid proofs of the 4 colour theorem.

Start with a finite set, which is called the alphabet. Consider all finite ordered strings, i.e. finite tuples, drawn from this alphabet. A language is any well defined subset of these strings. Each finite string in the language is called a word.

Since you have learnt about strings, alphabets, word and grammars in the preceding units, it will be easier for you to understand the topic of discussion in this unit, which is formal language.

You will see that languages fall into various classes, according to their complexity. Some languages can be parsed, i.e. interpreted, by a very simple state machine. Others require the human brain, or something comparable.

Languages also have many representations: machines that recognize them, expressions that describe them and grammars that generate them.

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

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

- define formal languages
- state the rules that define a formal language
- define word over an alphabet
- give examples of formal languages
- perform basic operations on languages
- explain the relevance of formal language to computer programming

## 3.0 MAIN CONTENT

### 3.1 Formal Language

A **formal language** is a set of *words*, i.e. finite strings of *letters*, or *symbols*. The inventory from which these letters are taken is called the *alphabet* over which the language is defined. A formal language is often defined by means of a formal grammar. Formal languages are a purely syntactical notion, so there is not necessarily any meaning associated with them. To distinguish the words that belong to a language from arbitrary words over its alphabet, the former are sometimes called *well-formed words* (or, in their application in logic, well-formed formulas).

Formal languages are studied in the fields of logic, computer science and linguistics. Their most important practical application is for the precise definition of syntactically correct programs for a programming language. The branch of mathematics and computer science that is concerned only with the purely syntactical aspects of such languages, i.e. their internal structural patterns, is known as **formal language theory**.

Although it is not formally part of the language, the words of a formal language often have a semantical dimension as well. In practice this is always tied very closely to the structure of the language, and a formal grammar (a set of formation rules that recursively defines the language) can help to deal with the meaning of (well-formed) words. Well-known examples for this are "Tarski's definition of truth" in terms of a T-schema for first-order logic, and compiler generators like lex and yacc.

### 3.2 Words over an Alphabet

An **alphabet**, in the context of formal languages can be any set, although it often makes sense to use an alphabet in the usual sense of the word, or more generally a character set such as ASCII. Alphabets can also be infinite; e.g. first-order logic is often expressed using an alphabet which, besides symbols such as  $\wedge$ ,  $\neg$ ,  $\forall$  and parentheses, contains infinitely many elements  $x_0, x_1, x_2, \dots$  that play the role of variables. The elements of an alphabet are called its **letters**.

A **word** over an alphabet can be any finite sequence, or string, of letters. The set of all words over an alphabet  $\Sigma$  is usually denoted by  $\Sigma^*$  (using the Kleene star). For any

alphabet there is only one word of length 0, the *empty word*, which is often denoted by  $\epsilon$ ,  $\varepsilon$  or  $\Lambda$ . By concatenation one can combine two words to form a new word, whose length is the sum of the lengths of the original words. The result of concatenating a word with the empty word is the original word.

As you learnt in the first unit of this course, in some applications, especially in logic, the alphabet is also known as the *vocabulary* and words are known as *formulas* or *sentences*; this breaks the letter/word metaphor and replaces it by a word/sentence metaphor.

### 3.2.1 Formal Definition

A **formal language**  $L$  over an alphabet  $\Sigma$  is just a subset of  $\Sigma^*$ , that is, a set of words over that alphabet.

In computer science and mathematics, which do not deal with natural languages, the adjective "formal" is usually omitted as redundant.

While formal language theory usually concerns itself with formal languages that are defined by some syntactical rules, the actual definition of a formal language is only as above: a (possibly infinite) set of finite-length strings, no more nor less. In practice, there are many languages that can be defined by rules, such as regular languages or context-free languages. The notion of a formal grammar may be closer to the intuitive concept of a "language," one defined by syntactic rules. By an abuse of the definition, a particular formal language is often thought of as being equipped with a formal grammar that defines it.

#### Example 1

The following rules define a formal language  $L$  over the alphabet  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}$ :

- Every nonempty string that does not contain  $+$  or  $=$  and does not start with  $0$  is in  $L$ .
- The string  $0$  is in  $L$ .
- A string containing  $=$  is in  $L$  if and only if there is exactly one  $=$ , and it separates two strings in  $L$ .
- A string containing  $+$  is in  $L$  if and only if every  $+$  in the string separates two valid strings in  $L$ .
- No string is in  $L$  other than those implied by the previous rules.

Under these rules, the string " $23+4=555$ " is in  $L$ , but the string " $=234=+$ " is not. This formal language expresses natural numbers, well-formed addition statements, and well-formed addition equalities, but it expresses only what they look like (their syntax), not what they mean (semantics). For instance, nowhere in these rules is there any indication that  $0$  means the number zero, or that  $+$  means addition.

For finite languages one can simply enumerate all well-formed words. For **example**, we can define a language  $L$  as just  $L = \{"a", "b", "ab", "cba"\}$ .

However, even over a finite (non-empty) alphabet such as  $\Sigma = \{a, b\}$  there are infinitely many words: "a", "abb", "ababba", "aaababbbbaab", .... Therefore formal languages are typically infinite, and defining an infinite formal language is not as simple as writing  $L = \{"a", "b", "ab", "cba"\}$ . Here are some examples of formal languages:

- $L = \Sigma^*$ , the set of *all* words over  $\Sigma$ ;
- $L = \{a^n\}^* = \{a^n\}$ , where  $n$  ranges over the natural numbers and  $a^n$  means "a" repeated  $n$  times (this is the set of words consisting only of the symbol "a");
- the set of syntactically correct programs in a given programming language (the syntax of which is usually defined by a context-free grammar);
- the set of inputs upon which a certain Turing machine halts; or
- the set of maximal strings of alphanumeric ASCII characters on this line, (i.e., the set {"the", "set", "of", "maximal", "strings", "alphanumeric", "ASCII", "characters", "on", "this", "line", "i", "e"}).

### 3.2.2 Vocabulary and Language

A vocabulary (or alphabet or character set or word list) is a finite nonempty set of indivisible symbols (letters, digits, punctuation marks, operators, etc.).

A language over a vocabulary  $V$  is any subset  $L$  of  $V^*$  which has a finite description. There are two approaches for making this mathematically precise. One is to use a grammar – a form of inductive definition of  $L$ . The other is to describe a method for recognizing whether an element  $x \in L$  is in the language  $L$  and automata theory is based on this approach.

### 3.3 Language-Specification Formalisms

Formal language theory rarely concerns itself with particular languages (except as examples), but is mainly concerned with the study of various types of formalisms to describe languages. For instance, a language can be given as

- those strings generated by some formal grammar (see Chomsky hierarchy);
- those strings described or matched by a particular regular expression;
- those strings accepted by some automaton, such as a Turing machine or finite state automaton;
- those strings for which some decision procedure (an algorithm that asks a sequence of related YES/NO questions) produces the answer YES.

Typical questions asked about such formalisms include:

- What is their expressive power? (Can formalism  $X$  describe every language that formalism  $Y$  can describe? Can it describe other languages?)
- What is their recognizability? (How difficult is it to decide whether a given word belongs to a language described by formalism  $X$ ?)
- What is their comparability? (How difficult is it to decide whether two languages, one described in formalism  $X$  and one in formalism  $Y$ , or in  $X$  again, are actually the same language?).

Surprisingly often, the answer to these decision problems is "it cannot be done at all", or "it is extremely expensive" (with a precise characterization of how expensive exactly). Therefore, formal language theory is a major application area of computability theory and complexity theory.

### 3.4 Operations on Languages

Certain operations on languages are common. This includes the standard set operations, such as union, intersection, and complementation. Another class of operation is the element-wise application of string operations.

#### Example 2:

Suppose  $L_1$  and  $L_2$  are languages over some common alphabet.

- The *concatenation*  $L_1L_2$  consists of all strings of the form  $vw$  where  $v$  is a string from  $L_1$  and  $w$  is a string from  $L_2$ .
- The *intersection*  $L_1 \cap L_2$  of  $L_1$  and  $L_2$  consists of all strings which are contained in both languages
- The *complement*  $\neg L$  of a language with respect to a given alphabet consists of all strings over the alphabet that are not in the language.

Such operations are used to investigate closure properties of classes of languages. A class of languages is closed under a particular operation when the operation, applied to languages in the class, always produces a language in the same class again. For instance, the context-free languages are known to be closed under union, concatenation, and intersection with regular languages, but not closed under intersection or complementation.

### 3.5 Other Operations on Languages

Some other operations frequently used in the study of formal languages are the following:

- The Kleene star: the language consisting of all words that are concatenations of 0 or more words in the original language;
- *Reversal*:
  - Let  $e$  be the empty word, then  $e^R = e$ , and

- for each non-empty word  $w = x_1 \dots x_n$  over some alphabet, let  $w^R = x_n \dots x_1$ ,
- then for a formal language  $L$ ,  $L^R = \{w^R \mid w \in L\}$ .
- String homomorphism.

### 3.6 Derivations and Language of a Grammar

Let  $G=(N,T,P,S)$  be any phrase structure grammar and let  $u, v \in (N \cup T)^*$ . We write  $u \Rightarrow v$  and say  $v$  is derived in one step from  $u$  by the rule  $x \rightarrow y$ , providing that  $u = pxq$  and  $v = pyq$ . (Here the rule  $x \rightarrow y$  is used to replace  $x$  by  $y$  in  $u$  to produce  $v$ . Note that  $p, q \in (N \cup T)^*$ .)

If  $u_1 \Rightarrow u_2 \Rightarrow u_3 \dots \Rightarrow u_n$  we say  $u_n$  is derived from  $u_1$  in  $G$  and write  $u_1 \Rightarrow^+ u_n$ .

Also if  $u_1 = u_n$  or  $u_1 \Rightarrow^+ u_n$  we write  $u_1 \Rightarrow^* u_n$

$L(G)$  the language of  $G$  is defined by:

$$L(G) = \{t \in T^* : Z \Rightarrow^* t \text{ for some } Z \in S\} = \{t \in T^* : t \in S \text{ or } Z \Rightarrow^+ t \text{ for some } Z \in S\}.$$

So, the elements of  $L(G)$  are those elements of  $T^*$  which are elements of  $S$  or which are derivable from elements of  $S$ .

## 4.0 CONCLUSION

In this unit you have been introduced to the concept of formal languages. Languages are very important in the field of automata theory since automata recognize languages, which is very important in computer programming. In the next unit we will be introducing you to automata.

## 5.0 SUMMARY

In this unit, you learnt that:

- a **formal language** is a set of *words*, i.e. finite strings of *letters*, or *symbols* and the inventory from which these letters are taken is called the *alphabet* over which the language is defined.
- a formal language is often defined by means of a formal grammar.
- a vocabulary (or alphabet or character set or word list) is a finite nonempty set of indivisible symbols
- common operations on languages are the standard set operations, such as union, intersection, and complementation.
- another class of operation that can be performed on languages is the element-wise application of string operations

## 6.0 TUTOR-MARKED ASSIGNMENT

1. What you understand by formal languages.
2. Is formal language finite or infinite? Discuss
3. Give any three examples of languages

## 7.0 REFERENCES/FURTHER READING

1. A. G. Hamilton, *Logic for Mathematicians*, Cambridge University Press, 1978.
2. Seymour Ginsburg, *Algebraic and automata theoretic properties of formal languages*, North-Holland, 1975.
3. Michael A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, 1978.
4. John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing, Reading Massachusetts, 1979.
5. Grzegorz Rozenberg, Arto Salomaa, *Handbook of Formal Languages: Volume I-III*, Springer, 1997
6. Patrick Suppes, *Introduction to Logic*, D. Van Nostrand, 1957,  
Source: [http://en.wikipedia.org/wiki/Formal\\_language](http://en.wikipedia.org/wiki/Formal_language)

**Module 1: General Concepts****Unit 4: Automata Theory****CONTENTS**1.0 *Introduction*2.0 *Objectives*3.0 *Main Content*

3.1 Automata Theory

3.2 Automata

3.2.1 Informal Description of Automaton

3.2.2 Formal Definitions

3.2.2.1 Automaton

3.2.2.2 Input Word

3.2.2.3 Run

3.2.2.4 Accepting Word

3.2.2.5 Recognized Language

3.2.2.6 Recognizable languages

3.2.3 Variations in Definition of Automata

3.2.3.1 Input

3.2.3.2 States

3.2.3.3 Transition Function

3.2.3.4 Acceptance Condition

3.3 Classes of automata

3.3.1 Discrete, Continuous, and Hybrid Automata

3.4 Applications of Automata Theory

4.0 *Conclusion*5.0 *Summary*6.0 *Tutor-Marked Assignment*7.0 *References/Further Reading***1.0 INTRODUCTION**

After learning about formal grammars and languages as you have done in the previous units, it is now time to introduce you to the concept of automata theory.

Now let us go through your study objectives for this unit.

**2.0 OBJECTIVES**

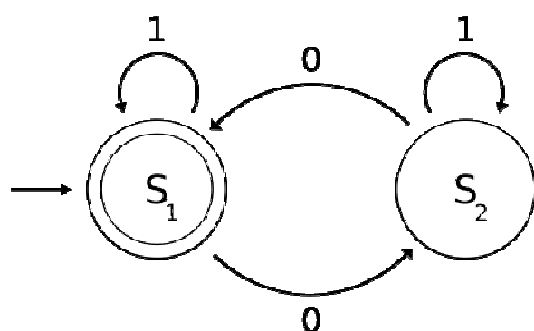
At the end of this unit, you should be able to:

- define an automaton
- explain automata theory
- state and describe types/classes of automata
- describe the operation of an automaton
- 

### 3.0 MAIN CONTENT

#### 3.1 Automata Theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.



**Figure 1: An Example of Automata**

Figure 1 above illustrates a finite state machine, which is one well-known variety of automata. This automaton consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its *transition function* (which takes the current state and the recent symbol as its inputs).

Automata theory is also closely related to formal language theory, as the automata are often classified by the class of formal languages they are able to recognize. An automaton can be a finite representation of a formal language that may be an infinite set.

In other words, automata theory is a subject matter which studies properties of various types of automata. For example, following questions are studied about a given type of automata.

- Which class of formal languages is recognizable by some type of automata? (Recognizable languages)

- Is certain automata *closed* under union, intersection, or complementation of formal languages? (Closure properties)
- How much is a type of automata expressive in terms of recognizing class of formal languages? And, their relative expressive power? (Language Hierarchy)

Automata theory also studies if there exist any effective algorithm or not to solve problems similar to the following list:

- Does an automaton accept any input word? (emptiness checking)
- Is it possible to transform a given non-deterministic automaton into deterministic automaton without changing the recognizing language? (Determinization)
- For a given formal language, what is the smallest automaton that recognizes it? (Minimization).

Automata play a major role in compiler design and parsing.

## 3.2 Automata

In the following sections you will be presented an introductory definition of one type of automata, which attempts to help one grasp the essential concepts involved in automata theory.

### 3.2.1 Informal Description of Automaton

An automaton is supposed to *run* on some given sequence or string of *inputs* in discrete time steps. At each time step, an automaton gets one input that is picked up from a set of *symbols* or *letters*, which is called an *alphabet*. At any time, the symbols so far fed to the automaton as input form a finite sequence of symbols, which is called a *word*. An automaton contains a finite set of states. At each instance in time of some run, automaton is *in* one of its states. At each time step when the automaton reads a symbol, it *jumps* or *transits* to next state depending on its current state and on the symbol currently read. This function in terms of the current state and input symbol is called *transition function*. The automaton *reads* the input word one symbol after another in the sequence and transits from state to state according to the transition function, until the word is read completely. Once the input word has been read, the automaton is said to have been *stopped* and the state at which automaton has stopped is called *final state*. Depending on the final state, it is said that the automaton either *accepts* or *rejects* an input word. There is a subset of states of the automaton, which is defined as the set of *accepting states*. If the final state is an accepting state, then the automaton *accepts* the word. Otherwise, the word is *rejected*.

The set of all the words accepted by an automaton is called the *language recognized by the automaton*.

### 3.2.2 Formal Definitions

### 3.2.2.1 Automaton

An **automaton** is represented formally by the 5-tuple  $\langle Q, \Sigma, \delta, q_0, F \rangle$ , where:

- $Q$  is a finite set of *states*.
- $\Sigma$  is a finite set of *symbols*, called the *alphabet* of the automaton.
- $\delta$  is the **transition function**, that is,  $\delta: Q \times \Sigma \rightarrow Q$ .
- $q_0$  is the *start state*, that is, the state which the automaton is *in* when no input has been processed yet, where  $q_0 \in Q$ .
- $F$  is a set of states of  $Q$  (i.e.  $F \subseteq Q$ ) called **accept states**.

### 3.2.2.2 Input Word

An automaton reads a finite string of symbols  $a_1, a_2, \dots, a_n$ , where  $a_i \in \Sigma$ , which is called a *input word*. Set of all words is denoted by  $\Sigma^*$ .

### 3.2.2.3 Run

A *run* of the automaton on an input word  $w = a_1, a_2, \dots, a_n \in \Sigma^*$ , is a sequence of states  $q_0, q_1, q_2, \dots, q_n$ , where  $q_i \in Q$  such that  $q_0$  is a start state and  $q_i = \delta(q_{i-1}, a_i)$  for  $0 < i \leq n$ . In words, at first the automaton is at the start state  $q_0$  and then automaton reads symbols of the input word in sequence. When automaton reads symbol  $a_i$  then it jumps to state  $q_i = \delta(q_{i-1}, a_i)$ .  $q_n$  said to be the *final state* of the run.

### 3.2.2.4 Accepting Word

A word  $w \in \Sigma^*$  is accepted by the automaton if  $q_n \in F$ .

### 3.2.2.5 Recognized Language

An automaton can recognize a formal language. The recognized language  $L \subset \Sigma^*$  by an automaton is the set of all the words that are accepted by the automaton.

### 3.2.2.6 Recognizable languages

The recognizable languages is the set of languages that are recognized by some automaton. For above definition of automata, the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

## 3.2.3 Variations in Definition of Automata

Automata are defined to study useful machines under mathematical formalism. So, the definition of an automaton is open to variations according to the "real world machine", which we want to model using the automaton. People have studied many variations of automata. Above, the most standard variant is described, which is called deterministic finite automaton. The following are some popular variations in the definition of different components of automata.

### 3.2.3.1 Input

- *Finite input*: An automaton that accepts only finite sequence of words. The above introductory definition only accepts finite words.
- *Infinite input*: An automaton that accepts infinite words ( $\omega$ -words). Such automata are called  *$\omega$ -automata*.
- *Tree word input*: The input may be a *tree of symbols* instead of sequence of symbols. In this case after reading each symbol, the automaton *reads* all the successor symbols in the input tree. It is said that the automaton *makes one copy* of itself for each successor and each such copy starts running on one of the successor symbol from the state according to the transition relation of the automaton. Such an automaton is called tree automaton.

### 3.2.3.2 States

- *Finite states*: An automaton that contains only a finite number of states. The above introductory definition describes automata with finite numbers of states.
- *Infinite states*: An automaton that may not have a finite number of states, or even a countable number of states. For example, the quantum finite automaton or topological automaton has uncountable infinity of states.
- *Stack memory*: An automaton may also contain some extra memory in the form of a stack in which symbols can be pushed and popped. This kind of automaton is called a *pushdown automaton*

### 3.2.3.3 Transition Function

- *Deterministic*: For a given current state and an input symbol, if an automaton can only jump to one and only one state then it is a *deterministic automaton*.
- *Nondeterministic*: An automaton that, after reading an input symbol, may jump into any of a number of states, as licensed by its transition relation. Notice that the term transition function is replaced by transition relation: The automaton *non-deterministically* decides to jump into one of the allowed choices. Such automaton are called *nondeterministic automaton*.
- *Alternation*: This idea is quite similar to tree automaton, but orthogonal. The automaton may run its *multiple copies* on the *same* next read symbol. Such automata are called *alternating automaton*. Acceptance condition must satisfy all runs of such *copies* to accept the input.

### 3.2.3.4 Acceptance Condition

- *Acceptance of finite words*: Same as described in the informal definition above.
- *Acceptance of infinite words*: an *omega automaton* cannot have final states, as infinite words never terminate. Rather, acceptance of the word is decided by looking at the infinite sequence of visited states during the run.
- *Probabilistic acceptance*: An automaton need not strictly accept or reject an input. It may accept the input with some probability between zero and one. For example, quantum finite automaton, geometric automaton and *metric automaton* has probabilistic acceptance.

Different combinations of the above variations produce many varieties of automata.

### 3.3 Classes of automata

In the following table is an incomplete list of some types of automata.

**Table 1: Types of Automata**

<b>Automata</b>	<b>Recognizable language</b>
Deterministic finite automata (DFA)	regular languages
Nondeterministic finite automata (NFA)	regular languages
Nondeterministic finite automata with $\epsilon$ transitions (FND- $\epsilon$ or $\epsilon$ -NFA)	regular languages
Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	Recursively enumerable languages
Timed automata	
Deterministic Büchi automata	omega limit languages
Nondeterministic Büchi automata	omega regular languages
Nondeterministic/Deterministic Rabin automata	omega regular languages
Nondeterministic/Deterministic Streett automata	omega regular languages
Nondeterministic/Deterministic parity automata	omega regular languages
Nondeterministic/Deterministic Muller automata	omega regular languages

#### 3.3.1 Discrete, Continuous, and Hybrid Automata

Normally automata theory describes the states of abstract machines but there are analog automata or continuous automata or hybrid discrete-continuous automata, using analog data, continuous time, or both. An automaton that computes a Boolean (yes-no) function is called an *acceptor*. Acceptors may be used as the membership criterion of a language. An automaton that produces more general output (typically a string) is called a *transducer*.

#### 3.4 Applications of Automata Theory

Each model in automata theory play varied roles in several applied areas. Finite automata are used in text processing, compilers, and hardware design. Context-free grammar (CFG) is used in programming languages and artificial intelligence. Originally, CFG were used in the study of the human languages. Cellular automata are

used in the field of biology, the most common example being John Conway's Game of Life. Some other examples which could be explained using automata theory in biology include mollusk and pine cones growth and pigmentation patterns. Going further, Stephen Wolfram claims that the entire universe could be explained by machines with a finite set of states and rules with a single initial condition. Other areas of interest which he has related to automata theory include: fluid flow, snowflake and crystal formation, chaos theory, cosmology, and financial analysis.

#### 4.0 CONCLUSION

In this unit you have been taken through the concept of automata theory. In the next unit you will be learning more specifically about each type of automata, their operations and the class of language they recognise.

#### 5.0 SUMMARY

In this unit, you learnt that:

- an *automaton* is a simple model of a computer.
- there is no formal definition for "automaton"--instead, there are various kinds of automata, each with its own formal definition.
- generally, an automaton
  - has some form of **input**,
  - has some form of **output**,
  - has internal **states**,
  - may or may not have some form of **storage**,
  - is **hard-wired** rather than programmable.
- an automaton can recognize a formal language
- the recognizable languages is the set of languages that are recognized by some automaton.

#### 6.0 TUTOR-MARKED ASSIGNMENT

1. What do you understand by automata theory?
2. State any four classes of automata
3. In the context of automata theory, briefly describe the following terms:
  - Recognised language
  - Recognizable languages
  - Run
  - Transducer
  - acceptor

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education. ISBN 0-201-44124-1.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**Module 2: Regular Languages****Unit 1: Finite State Automata****CONTENTS**

- 1.0 *Introduction*
- 2.0 *Objectives*
- 3.0 *Main Content*
  - 3.1 Finite State Automata
  - 3.2 Deterministic Finite Acceptors/Automata (DFA)
    - 3.2.1 Algorithm for the Operation of a DFA
    - 3.2.2 Implementing a DFA
      - 3.2.2.1 Using a GO TO Statement
    - 3.2.3 Formal Definition of a DFA
  - 3.3 Acceptor for Ada identifiers
    - 3.3.1 Abbreviated Acceptor for Ada Identifiers
  - 3.4 Nondeterministic Finite Automata/Acceptors (NFA)
    - 3.4.1 Implementing an NFA
      - 3.4.1.1 Recursive Implementation of NFAs
      - 3.4.1.2 State-Set Implementation of NFAs
      - 3.4.1.3 Formal Definition of NFAs
  - 3.5 Equivalence of FAs
- 4.0 *Conclusion*
- 5.0 *Summary*
- 6.0 *Tutor-Marked Assignment*
- 7.0 *References/Further Reading*
- 1.0 INTRODUCTION**

You have learnt the general concepts of automata theory in the previous module. In this introductory unit of this module, you will be learning about the most basic of all automata, which is the finite state automata (FSA).

The finite-state automata (FSA) or finite state machine (FSM) enjoy a special place in computer science. The FSA has proven to be a very useful model for many practical tasks and deserves to be among the tools of every practicing computer scientist. Many simple tasks, such as interpreting the commands typed into a keyboard or running a calculator, can be modelled by finite-state automata.

In this unit we examine the language recognition capability of FSA. We show that FSA recognize exactly the regular languages, languages defined by regular expressions and generated by regular grammars. We also provide an algorithm to find a FSA that is equivalent to a given FSA but has the fewest states. The two different types of FSA, deterministic and nondeterministic, are also discussed in this unit.

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

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

- describe FSA
- formally define DFA and NFA
- give an algorithm for the operations of a DFA
- describe ways of implementing DFAs and NFAs

## 3.0 MAIN CONTENT

### 3.1 Finite State Automata

Like grammars, finite state automata define languages. The finite state automata is often abbreviated FSA or FA (for finite automata); however, some texts use the term finite state machine, or FSM to correlate with Turing machines that will be discussed in module 4 of this course.

An FSA is a virtual device that manipulates a candidate string, one character at a time, and determines whether that string is in the language implemented by the machine. The simplest state machine reads the string exactly once, and has no memory, only registers. It is therefore a finite state automaton.

An FSA is defined by a set of states and a transition function that maps state/input pairs into states. In this state, reading this character, move to that state and advance to the next character.

Some states are designated "final" states, and strings that leave the FSA in one of these final states are, by definition, in the language.

One state is designated the start state. When the start state is also a final state,  $\epsilon$  is necessarily in the language.

#### Example 1:

The following 4-state machine defines binary strings with an even number of 1's and 0's. States are a, b, c, d, and input characters are 0 1. State a is both the start state and the final state

0 1  
a  $\rightarrow$  b c  
b  $\rightarrow$  a d  
c  $\rightarrow$  d a  
d  $\rightarrow$  c b

### 3.2 Deterministic Finite Acceptors/Automata (DFA)

DFA's are:

- Deterministic i.e. there is no element of choice
- Finite i.e. only a finite number of states and arcs
- Acceptors i.e. produce only a yes/no answer

A DFA is drawn as a graph, with each *state* represented by a circle.



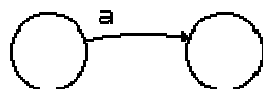
Start state

One designated state is the *start state*.



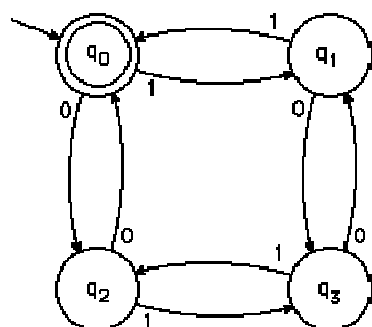
Final state

Some states (possibly including the start state) can be designated as *final states*.



State transition arc

Arcs between states represent *state transitions*. Each such arc is labelled with the symbol that triggers the transition.



*Figure 1: Example of DFA*

#### 3.2.1 Algorithm for the Operation of a DFA

- Start with the "current state" set to the start state and a "read head" at the beginning of the input string;
- while there are still characters in the string:
  - Read the next character and advance the read head;
  - From the current state, follow the arc that is labelled with the character just read; the state that the arc points to becomes the next current state;
- When all characters have been read, *accept* the string if the current state is a final state, otherwise *reject* the string.

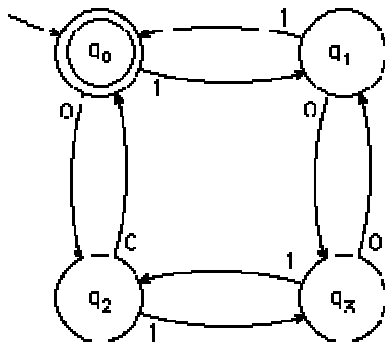
### Example 2

Consider the following input string: 1 0 0 1 1 1 0 0. Using the DFA in Figure 1 above, a sample trace will be as follows:

q0 1 q1 0 q3 0 q1 1 q0 1 q1 1 q0 0 q2 0 q0

Since q0 is a final state, the string is accepted.

### 3.2.2 Implementing a DFA



#### 3.2.2.1 Using a GO TO Statement

If you do not object to the **go to** statement, below is an easy way to implement a DFA:

```

q0 : read char;
    if eof then accept string;
    if char = 0 then go to q2;
    if char = 1 then go to q1;
  
```

```

q1 : read char;
    if eof then reject string;
    if char = 0 then go to q3;
    if char = 1 then go to q0;
  
```

```

q2 : read char;
  
```

```

    if eof then reject string;
    if char = 0 then go to q0;
    if char = 1 then go to q3;

```

```

q3 : read char;
    if eof then reject string;
    if char = 0 then go to q1;
    if char = 1 then go to q2;

```

### 3.2.2.2 Using a CASE Statement

If you are not allowed to use a **go to** statement, you can fake it with a combination of a loop and a case statement:

```

state := q0;
loop
  case state of
    q0 : read char;
        if eof then accept string;
        if char = 0 then state := q2;
        if char = 1 then state := q1;

    q1 : read char;
        if eof then reject string;
        if char = 0 then state := q3;
        if char = 1 then state := q0;

    q2 : read char;
        if eof then reject string;
        if char = 0 then state := q0;
        if char = 1 then state := q3;

    q3 : read char;
        if eof then reject string;
        if char = 0 then state := q1;
        if char = 1 then state := q2;
  end case;
end loop;

```

### 3.2.3 Formal Definition of a DFA

A *deterministic finite acceptor/automaton* or *DFA* is a quintuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q$  is a finite set of *states*,
- $\Sigma$  is a finite set of symbols, the *input alphabet*,
- $\delta: Q \times \Sigma \rightarrow Q$  is a *transition function*,
- $q_0 \in Q$  is the *initial state*,
- $F \subseteq Q$  is a set of *final states*.

Note: The fact that  $\delta$  is a function implies that every vertex has an outgoing arc for each member of  $\Sigma$ .

We can also define an *extended transition function*  $\delta^*$  as

$$\delta^*: Q \times \Sigma^* \rightarrow Q.$$

If a DFA  $M = (Q, \Sigma, \delta, q_0, F)$  is used as a membership criterion, then the set of strings accepted by  $M$  is a language. That is,

$$L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \in F\}.$$

Languages that can be defined by DFAs are called *regular languages*.

### 3.3 Acceptor for Ada identifiers

In Ada, an identifier consists of a letter followed by any number of letters, digits, and underlines. However, the identifier may not end in an underline or have two underlines in a row.

Here is an automaton to recognize Ada identifiers.

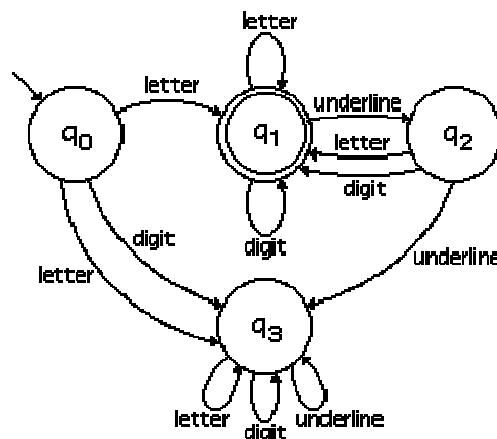


Figure 2: Formal acceptor for Ada identifiers.

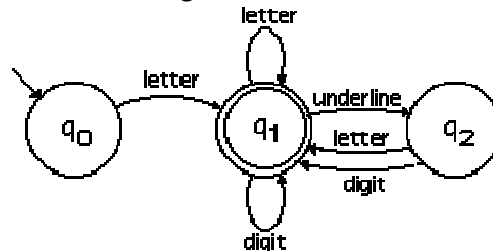
$M = (Q, \Sigma, \delta, q_0, F)$ , where

- $Q$  is  $\{q_0, q_1, q_2, q_3\}$ ,
- $\Sigma$  is  $\{\text{letter, digit, underline}\}$ ,
- $\delta$  is given by

- $\delta(q_0, \text{letter}) = q_1$
  - $\delta(q_0, \text{digit}) = q_3$
  - $\delta(q_0, \text{underline}) = q_3$
  - $\delta(q_2, \text{letter}) = q_1$
  - $\delta(q_2, \text{digit}) = q_1$
  - $\delta(q_2, \text{underline}) = q_3$
  - $q_0 \in Q$  is the *initial state*,
  - $\{q_1\} \subseteq Q$  is a set of *final states*.
- $\delta(q_1, \text{letter}) = q_1$
  - $\delta(q_1, \text{digit}) = q_1$
  - $\delta(q_1, \text{underline}) = q_2$
  - $\delta(q_3, \text{letter}) = q_3$
  - $\delta(q_3, \text{digit}) = q_3$
  - $\delta(q_3, \text{underline}) = q_3$

### 3.3.1 Abbreviated Acceptor for Ada Identifiers

The following is an *abbreviated* automaton (my terminology) to recognize Ada identifiers. You might use something like this in a course on compiler construction.



**Figure 3:** Informal acceptor for Ada identifiers.

The difference is that, in this automaton,  $\delta$  does not appear to be a function. It looks like a *partial function*, that is, it is not defined for all values of  $Q \times \Sigma$ .

We can complete the definition of  $\delta$  by assuming the existence of an "invisible" state and some "invisible" arcs. Specifically,

- There is exactly one implicit *error state*;
- If there is no path shown from a state for a given symbol in  $\Sigma$ , there is an implicit path for that symbol to the error state;
- The error state is a *trap state*: once you get into it, all arcs (one for each symbol in  $\Sigma$ ) lead back to it; and
- The error state is not a final state.

The automaton represented in Figure 3 above is really exactly the same as the automaton in Figure 2; we just have not bothered to draw one state and a whole bunch of arcs that we know must be there.

I do not think you will find abbreviated automata in the textbook. They are not usually allowed in a formal course. However, if you ever use an automaton to design a lexical scanner, putting in an explicit error state just clutters up the diagram.

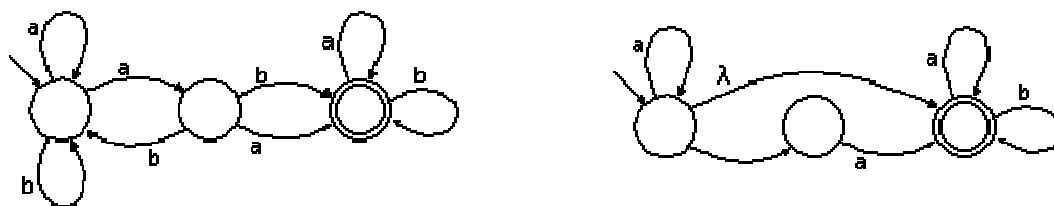
### 3.4 Nondeterministic Finite Automata/Acceptors (NFA)

An FSA is nondeterministic if it is confronted with several choices when processing each character. Thus the transition function of a nondeterministic FSA maps state/input pairs into sets of states. The machine somehow traverses all possible paths in parallel. In addition,  $\epsilon$  transitions are permitted, allowing the machine to change states without reading an input character. A string is accepted by an NFA if one of its parallel transition sequences leads to a final state.

This seems to add a great deal of power, but in fact it does not. Any NFA can be emulated by an FSA with more states. Start with an NFA containing  $n$  states  $x_1 x_2 x_3$  etc, and construct a deterministic FSA with  $2^n$  states as follows. Each state in the new FSA corresponds to a unique combination of states in the original NFA. The initial state  $y_0$  corresponds to the union of the initial state  $x_0$  and all other  $x_j$  states that are accessible from  $x_0$  via  $\epsilon$  transitions. The state  $y_i$  in the FSA is a final state if any of the corresponding  $x_j$  states, represented by  $y_i$ , is a final state in the original NFA. To determine the transition function  $f(y_i, c)$ , apply  $c$  to each corresponding  $x_j$  state, and bring in any new states that are accessible via  $\epsilon$  transitions. The combination of all these states determines a particular  $y_k$ . Thus state  $y_i$ , reading character  $c$ , moves to state  $y_k$ .

By induction on string length, any string that leaves the constructed FSA in state  $y_i$  also leaves the original FSA in any of the corresponding states  $x_j$ . One machine says yes to the input word if and only if the other one does. Therefore nondeterministic FSAs are no more powerful than their deterministic counterparts.

A finite-state automaton can be *nondeterministic* in either or both of two ways:



**Figure 4: Nondeterministic Finite Acceptor**

A state may have two or more arcs emanating from it labelled with the same symbol. When the symbol occurs in the input, either arc may be followed. A state may have one or more arcs emanating from it labelled with  $\lambda$  (the empty string). These arcs may optionally be followed without looking at the input or consuming an input symbol.

Due to nondeterminism, the same string may cause an NFA to end up in one of several different states, some of which may be final while others are not. The string is accepted if **any** possible ending state is a final state.

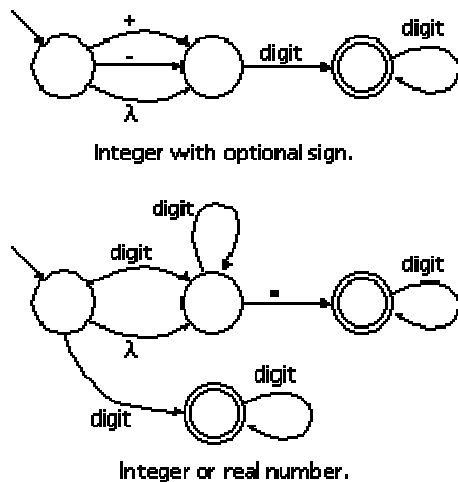


Figure 5: Examples of NFAs

### 3.4.1 Implementing an NFA

If you think of an automaton as a computer, how does it handle nondeterminism?

There are two ways that this could, in theory, be done:

1. When the automaton is faced with a choice, it always (magically) chooses correctly. We sometimes think of the automaton as consulting an *oracle* which advises it as to the correct choice.
2. When the automaton is faced with a choice, it spawns a new process, so that all possible paths are followed simultaneously.

The first of these alternatives, using an oracle, is sometimes attractive mathematically. But if we want to write a program to implement an NFA, that is not feasible.

There are three ways, two feasible and one not yet feasible, to simulate the second alternative:

1. Use a recursive backtracking algorithm. Whenever the automaton has to make a choice, cycle through all the alternatives and make a recursive call to determine whether any of the alternatives leads to a solution (final state).
2. Maintain a state set or a state vector, keeping track of *all* the states that the NFA could be in at any given point in the string.
3. Use a *quantum computer*. Quantum computers explore literally all possibilities simultaneously. They are theoretically possible, but are at the cutting edge of physics. It may (or may not) be feasible to build such a device.

#### 3.4.1.1 Recursive Implementation of NFAs

An NFA can be implemented by means of a recursive search from the start state for a path (directed by the symbols of the input string) to a final state.

Here is a rough outline of such an implementation:

```
function NFA (state A) returns Boolean:
  local state B, symbol x;
  for each  $\lambda$  transition from state A to some state B do
    if NFA (B) then return True;
  if there is a next symbol then
    { read next symbol (x);
      for each x transition from state A to
        some state B do
          if NFA (B) then
            return True;
        return False;
    }
  else
    { if A is a final state then return True;
      else return False;
    }
```

One problem with this implementation is that it could get into an infinite loop if there is a cycle of  $\lambda$  transitions. This could be prevented by maintaining a simple counter.

### 3.4.1.2 State-Set Implementation of NFAs

Another way to implement an NFA is to keep either a *state set* or a *bit vector* of all the states that the NFA could be in at any given time. Implementation is easier if you use a bit-vector approach ( $v[i]$  is True iff state  $i$  is a possible state), since most languages provide vectors, but not sets, as a built-in datatype. However, it is a bit easier to describe the algorithm if you use a state-set approach, so that is what we will do. The logic is the same in either case.

```
function NFA (state set A) returns Boolean:
  local state set B, state a, state b, state c, symbol
  x;

  for each a in A do
    for each  $\lambda$  transition from a
      to some state b do
        add b to B;
  while there is a next symbol do
    { read next symbol (x);
      B :=  $\emptyset$ ;
      for each a in A do
```

```

do
    { for each  $\lambda$  transition from a to some state b
    do
        add b to B;
        for each x transition from a to some state b
    do
        add b to B;
    }
    for each  $\lambda$  transition from
        some state b in B to some state c not in B do
        add c to B;
    A := B;
}
if any element of A is a final state then
    return True;
else
    return False;

```

### 3.4.1.3 Formal Definition of NFAs

The extension of our notation to NFAs is somewhat strained.

A *nondeterministic finite acceptor/automaton* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q$  is a finite set of *states*,
- $\Sigma$  is a finite set of symbols, the *input alphabet*,
- $\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$  is a *transition function*,
- $q_0 \in Q$  is the *initial state*,
- $F \subseteq Q$  is a set of *final states*.

These are all the same as for a DFA except for the definition of  $\delta$ :

- Transitions on  $\lambda$  are allowed in addition to transitions on elements of  $\Sigma$ , and
- The range of  $\delta$  is  $2^Q$  rather than  $Q$ . This means that the values of  $\delta$  are not elements of  $Q$ , but rather are sets of elements of  $Q$ .

The language defined by NFA  $M$  is defined as

$$L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \cap F \neq \emptyset\}$$

## 3.5 Equivalence of FAs

Two acceptors are *equivalent* if they accept the same language.

A DFA is just a special case of an NFA that happens not to have any null transitions or multiple transitions on the same symbol. So DFAs are not more powerful than NFAs.

For any NFA, we can construct an equivalent DFA (see below). So NFAs are not more powerful than DFAs. DFAs and NFAs define the same class of languages – the *regular* languages.

To translate an NFA into a DFA, the trick is to label each state in the DFA with a *set of states* from the NFA. Each state in the DFA summarizes all the states that the NFA might be in. If the NFA contains  $|Q|$  states, the resultant DFA could contain as many as  $|2^Q|$  states. (Usually far fewer states will be needed.)

#### 4.0 CONCLUSION

In this unit you have been taken through a class of automata called finite automata, its various types and ways of implementing each type.

#### 5.0 SUMMARY

In this unit you learnt that:

- finite state automata define languages
- An FSA is nondeterministic if it is confronted with several choices when processing each character
- A finite-state automaton can be *nondeterministic* in either or both of two ways
- Two acceptors are *equivalent* if they accept the same language
- A DFA is just a special case of an NFA that happens not to have any null transitions or multiple transitions on the same symbol
- For any NFA, we can construct an equivalent DFA
- In Ada, an identifier consists of a letter followed by any number of letters, digits, and underlines

#### 6.0 TUTOR-MARKED ASSIGNMENT

- 1) Give the formal definition of the following:
  - FSA
  - DFA
  - NFA
- 2) How is a DFA different from an NFA? How are they similar?
- 3) Is an NFA more powerful than a DFA? Discuss.
- 4) Briefly describe the various ways that an NFA can be implemented.
- 5) Describe an algorithm for the Operation of a DFA

#### 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education. ISBN 0-201-44124-1.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Part One: Automata and Languages, chapters 1–2, pp.29–122. Section 4.1: Decidable Languages, pp.152–159. Section 5.1: Undecidable Problems from Language Theory, pp.172–183.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**Module 2: Regular Languages****Unit 2: Regular Expressions****CONTENTS**

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Primitive Regular Expressions
3.2	Regular Expressions
3.3	Languages Defined by Regular Expressions
3.4	Building Regular Expressions
3.4.1	Example Regular Expressions
3.5	Regular Expressions and Automata
3.5.1	From Primitive Regular Expressions to NFAs
3.5.2	From Regular Expressions to NFAs
3.5.3	From NFAs to Regular Expressions
3.6	Three Ways of Defining a Language
3.6.1	Definition by Grammar
3.6.2	Definition by NFA
3.6.3	Definition by Regular Expression
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

**1.0 INTRODUCTION**

In the previous unit, you learnt about finite state automata (which is a way of characterising regular languages), the different types and the various ways of implementing them. In this unit we will be discussing regular expressions, which is another way characterising regular languages.

Now let us go through your study objectives for this unit.

**2.0 OBJECTIVES**

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

- define regular expressions
- state the rules that can be applied to primitive regular expressions to create more regular expressions
- state the precedence of the rules

- describe the three ways of defining a language
- demonstrate how to convert regular expressions to DFAs and NFAs and vice versa

### 3.0 MAIN CONTENT

#### 3.1 Primitive Regular Expressions

A *regular expression* can be used to define a language. A regular expression represents a "pattern;" strings that match the pattern that are in the language, strings that do not match the pattern are not in the language.

As usual, the strings are over some alphabet  $\Sigma$ .

The following are *primitive regular expressions*:

- $x$ , for each  $x \in \Sigma$ ,
- $\lambda$ , the empty string, and
- $\emptyset$ , indicating no strings at all.

Thus, if  $|\Sigma| = n$ , then there are  $n+2$  primitive regular expressions defined over  $\Sigma$ .

Here are the languages defined by the primitive regular expressions:

- For each  $x \in \Sigma$ , the primitive regular expression  $x$  denotes the language  $\{x\}$ . That is, the only string in the language is the string "x".
- The primitive regular expression  $\lambda$  denotes the language  $\{\lambda\}$ . The only string in this language is the empty string.
- The primitive regular expression  $\emptyset$  denotes the language  $\{\}$ . There are *no* strings in this language.

#### 3.2 Regular Expressions

Every primitive regular expression is a regular expression.

We can compose additional regular expressions by applying the following rules a *finite* number of times:

- If  $r_1$  is a regular expression, then so is  $(r_1)$ .
- If  $r_1$  is a regular expression, then so is  $r_1^*$ .
- If  $r_1$  and  $r_2$  are regular expressions, then so is  $r_1r_2$  (Concatenation)
- If  $r_1$  and  $r_2$  are regular expressions, then so is  $r_1+r_2$  or  $r_1/r_2$  (Union)

Here is what the above notation means:

- Parentheses are just used for grouping.

- The postfix star (Kleene closure) indicates zero or more repetitions of the preceding regular expression. Thus, if  $x \in \Sigma$ , then the regular expression  $x^*$  denotes the language  $\{\lambda, x, xx, xxx, \dots\}$ .
- Juxtaposition/concatenation of  $r_1$  and  $r_2$  indicates any string described by  $r_1$  immediately followed by any string described by  $r_2$ . For example, if  $x, y \in \Sigma$ , then the regular expression  $xy$  describes the language  $\{xy\}$ .
- The plus (+) or | sign, read as "or," denotes the language containing strings described by either of the component regular expressions i.e. the union of the component regular expressions. For example, if  $x, y \in \Sigma$ , then the regular expression  $x+y$  or  $x|y$  describes the language  $\{x, y\}$ .

### Precedence

- 1) The unary operator \* (Kleene closure) has the highest precedence and is left associative. For example,  $a+bc^*$  or  $a|bc^*$  denotes the language  $\{a, b, bc, bcc, bccc, bcccc, \dots\}$ .
- 2) Concatenation has a second highest precedence and is left associative.
- 3) Union has lowest precedence and is left associative.
- 4) Parentheses override operator precedence as usual. For example,  $(0|1)^*$  stands for all possible binary strings,  $0|1^*$  stands for either a 0 or an arbitrarily long string of 1's, and  $01^*$  stands for 0 followed by an arbitrarily long string of 1's.

The symbol  $\epsilon$  represents the null string, and can be used like any other alphabetic character. Thus,  $(0|\epsilon)(1(0|\epsilon))^*$  stands for all binary strings without adjacent zeros.

Computer languages such as `ed`, `sed`, `grep`, and `perl` employ regular expressions, but there are many more features for your convenience. For instance,  $s+ = ss^*$ ,  $s? = (s|\epsilon)$ ,  $s\{7,\} = ssssss+$ , and so on. Check out ``man perlre'` for more details.

### 3.3 Languages Defined by Regular Expressions

There is a simple correspondence between regular expressions and the languages they denote:

Regular expression	L(regular expression)
$x$ , for each $x \in \Sigma$	$\{x\}$
$\lambda$	$\{\lambda\}$
$\emptyset$	$\{\}$
$(r_1)$	$L(r_1)$
$r_1^*$	$(L(r_1))^*$
$r_1 r_2$	$L(r_1) L(r_2)$
$r_1 + r_2$	$L(r_1) \cup L(r_2)$

### 3.4 Building Regular Expressions

Here are some hints on building regular expressions. We will assume  $\Sigma = \{a, b, c\}$ .

#### Zero or more.

$a^*$  means "zero or more a's." To say "zero or more ab's," that is,  $\{\lambda, ab, abab, ababab, \dots\}$ , you need to say  $(ab)^*$ . Don't say  $ab^*$ , because that denotes the language  $\{a, ab, abb, abbb, abbbb, \dots\}$ .

#### One or more.

Since  $a^*$  means "zero or more a's", you can use  $aa^*$  (or equivalently,  $a^*a$ ) to mean "one or more a's." Similarly, to describe "one or more ab's," that is,  $\{ab, abab, ababab, \dots\}$ , you can use  $ab(ab)^*$ .

#### Zero or one.

You can describe an optional a with  $(a+\lambda)$ .

#### Any string at all.

To describe any string at all (with  $\Sigma = \{a, b, c\}$ ), you can use  $(a+b+c)^*$ .

#### Any nonempty string.

This can be written as any character from  $\Sigma$  followed by any string at all:  $(a+b+c)(a+b+c)^*$ .

#### Any string not containing...

To describe any string at all that does not contain an a (with  $\Sigma = \{a, b, c\}$ ), you can use  $(b+c)^*$ .

#### Any string containing exactly one...

To describe any string that contains exactly one a, put "any string not containing an a," on either side of the a, like this:  $(b+c)^*a(b+c)^*$ .

#### 3.4.1 Example Regular Expressions

Give regular expressions for the following languages on  $\Sigma = \{a, b, c\}$ .

##### All strings containing exactly one a.

$(b+c)^*a(b+c)^*$

##### All strings containing no more than three a's.

We can describe the string containing zero, one, two, or three a's (and nothing else) as

$(\lambda+a)(\lambda+a)(\lambda+a)$

Now we want to allow arbitrary strings not containing a's at the places marked by X's:

$X(\lambda+a)X(\lambda+a)X(\lambda+a)X$

so we put in  $(b+c)^*$  for each X:

$(b+c)^*(\lambda+a)(b+c)^*(\lambda+a)(b+c)^*(\lambda+a)(b+c)^*$

**All strings which contain at least one occurrence of each symbol in  $\Sigma$ .**

The problem here is that we cannot assume the symbols are in any particular order. We have no way of saying "in any order", so we have to list the possible orders:

$abc+acb+bac+bca+cab+cba$

To make it easier to see what's happening, let's put an X in every place we want to allow an arbitrary string:

$XaXbXcX + XaXcXbX + XbXaXcX + XbXcXaX + XcXaXbX + XcXbXaX$

Finally, replacing the X's with  $(a+b+c)^*$  gives the final (unwieldy) answer:

$(a+b+c)^*a(a+b+c)^*b(a+b+c)^*c(a+b+c)^* +$   
 $(a+b+c)^*a(a+b+c)^*c(a+b+c)^*b(a+b+c)^* +$   
 $(a+b+c)^*b(a+b+c)^*a(a+b+c)^*c(a+b+c)^* +$   
 $(a+b+c)^*b(a+b+c)^*c(a+b+c)^*a(a+b+c)^* +$   
 $(a+b+c)^*c(a+b+c)^*a(a+b+c)^*b(a+b+c)^* +$   
 $(a+b+c)^*c(a+b+c)^*b(a+b+c)^*a(a+b+c)^*$

**All strings which contain no runs of a's of length greater than two.**

We can fairly easily build an expression containing no a, one a, or one aa:

$(b+c)^*(\lambda+a+aa)(b+c)^*$

but if we want to repeat this, we need to be sure to have at least one non-a between repetitions:

$(b+c)^*(\lambda+a+aa)(b+c)^*((b+c)(b+c)^*(\lambda+a+aa)(b+c)^*)^*$

**All strings in which all runs of a's have lengths that are multiples of three.**

$(aaa+b+c)^*$

**3.5 Regular Expressions and Automata**

Languages described by deterministic finite acceptors (DFAs) are called *regular languages*.

For any nondeterministic finite acceptor (NFA) we can find an equivalent DFA. Thus NFAs also describe regular languages.

Regular expressions also describe regular languages. We will show that regular expressions are equivalent to NFAs by doing two things:

1. For any given regular expression, we will show how to build an NFA that accepts the same language. (This is the easy part.)
2. For any given NFA, we will show how to construct a regular expression that describes the same language. (This is the hard part.)

### 3.5.1 From Primitive Regular Expressions to NFAs

Every NFA we construct will have a single start state and a single final state. We will build more complex NFAs out of simpler NFAs, each with a single start state and a single final state. The simplest NFAs will be those for the primitive regular expressions.

For any  $x$  in  $\Sigma$ , the regular expression  $x$  denotes the language  $\{x\}$ . This NFA represents exactly that language.

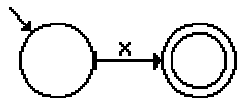


Figure 1: nfa for  $x$

Note that if this were a NFA, we would have to include arcs for all the other elements of  $\Sigma$ .



Figure 2: nfa for  $\lambda$

The regular expression  $\lambda$  denotes the language  $\{\lambda\}$ , that is, the language containing only the empty string.

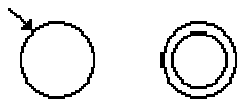


Figure 3: nfa for  $\emptyset$

The regular expression  $\emptyset$  denotes the language  $\emptyset$ ; no strings belong to this language, not even the empty string.

Since the final state is unreachable, why bother to have it at all? The answer is that it simplifies the construction if every NFA has exactly one start state and one final state. We could do without this final state, but we would have more special cases to consider, and it does not hurt anything to include it.

### 3.5.2 From Regular Expressions to NFAs

We will build more complex NFAs out of simpler NFAs, each with a single start state and a single final state. Since we have NFAs for primitive regular expressions, we

need to compose them for the operations of grouping, juxtaposition, union, and Kleene star (\*).

For grouping (parentheses), we don't really need to do anything. The NFA that represents the regular expression ( $r_1$ ) is the same as the NFA that represents  $r_1$ .

For juxtaposition (strings in  $L(r_1)$  followed by strings in  $L(r_2)$ ), we simply chain the NFAs together, as shown. The initial and final states of the original NFAs (boxed) stop being initial and final states; we include new initial and final states. (We could make do with fewer states and fewer  $\lambda$  transitions here, but we aren't trying for the best construction; we're just trying to show that a construction is possible.)

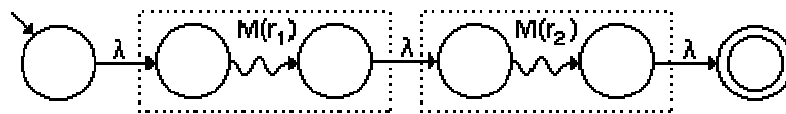


Figure 4: nfa for  $r_1 r_2$

The + denotes "or" in a regular expression, so it makes sense that we would use an NFA with a choice of paths. (This is one of the reasons that it's easier to build an NFA than a DFA.)

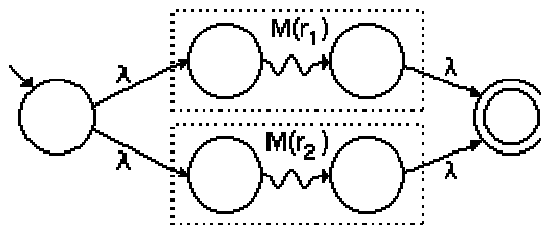


Figure 5: nfa for  $r_1 + r_2$

The star denotes zero or more applications of the regular expression, so we need to set up a loop in the NFA. We can do this with a backward-pointing  $\lambda$  arc. Since we might want to traverse the regular expression zero times (thus matching the null string), we also need a forward-pointing  $\lambda$  arc to bypass the NFA entirely.

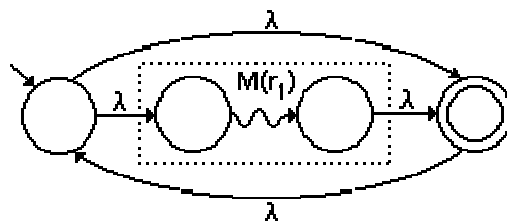


Figure 6: nfa for  $r_1^*$

### 3.5.3 From NFAs to Regular Expressions

Creating a regular expression to recognize the same strings as an NFA is trickier than you might expect, because the NFA may have arbitrary loops and cycles. Here's the basic approach (details supplied later):

1. If the NFA has more than one final state, convert it to an NFA with only one final state. Make the original final states nonfinal, and add a  $\lambda$  transition from each to the new (single) final state.
2. Consider the NFA to be a *generalized transition graph*, which is just like an NFA except that the edges may be labeled with arbitrary regular expressions. Since the labels on the edges of an NFA may be either  $\lambda$  or members of  $\Sigma$ , each of these can be considered to be a regular expression.
3. Remove states one by one from the NFA, relabeling edges as you go, until only the initial and the final state remain.
4. Read the final regular expression from the two-state automaton that results.

The regular expression derived in the final step accepts the same language as the original NFA.

Since we can convert an NFA to a regular expression, and we can convert a regular expression to an NFA, the two are equivalent formalisms--that is, they both describe the same class of languages, the regular languages.

There are two complicated parts to extracting a regular expression from an NFA: removing states, and reading the regular expression off the resultant two-state generalized transition graph.

Here I show to delete a state:

To delete state  $Q$ , where  $Q$  is neither the initial state nor the final state,



Figure 7: Deleting a State

You should convince yourself that this transformation is "correct", in the sense that paths which leave you in  $Q_i$  in the original will leave you in  $Q_i$  in the replacement, and similarly for  $Q_j$ .

- What if state  $Q$  has connections to more than two other states, say,  $Q_i$ ,  $Q_j$ , and  $Q_k$ ? Then you have to consider these states pairwise:  $Q_i$  with  $Q_j$ ,  $Q_j$  with  $Q_k$ , and  $Q_i$  with  $Q_k$ .
- What if some of the arcs in the original state are missing? There are too many cases to work this out in detail, but you should be able to figure it out for any specific case, using the above as a model.

You will end up with an NFA that looks like this, where  $r_1$ ,  $r_2$ ,  $r_3$ , and  $r_4$  are (probably very complex) regular expressions. The resultant NFA in figure 8 below represents the regular expression  $r_1^*r_2(r_4 + r_3r_1^*r_2)^*$

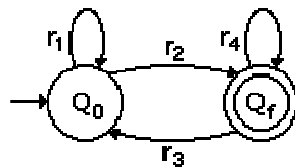


Figure 8: NFA for  $r_1^*r_2(r_4 + r_3r_1^*r_2)^*$

(you should verify that this is indeed the correct regular expression). All you have to do is plug in the correct values for  $r_1$ ,  $r_2$ ,  $r_3$ , and  $r_4$ .

### 3.6 Three Ways of Defining a Language

The following presents an example solved three different ways. No new information is presented.

**Problem:** Define a language containing all strings over  $\Sigma = \{a, b, c\}$  where no symbol ever follows itself; that is, no string contains any of the substrings  $aa$ ,  $bb$ , or  $cc$ .

#### 3.6.1 Definition by Grammar

Define the grammar  $G = (V, T, S, P)$  where

- $V = \{S, \dots \text{some other variables} \dots\}$ .
- $T = \Sigma = \{a, b, c\}$ .
- The start symbol is  $S$ .
- $P$  is given below.

These should be pretty obvious except for the set  $V$ , which we generally make up as we construct  $P$ .

Since the empty string belongs to the language, we need the production

$$S \rightarrow \lambda$$

Some strings belonging to the language begin with the symbol  $a$ . The  $a$  can be followed by any other string in the language, so long as this other string does not begin with  $a$ . So we make up a variable, call it  $NOTA$ , to produce these other strings, and add the production

$$S \rightarrow a \text{ NOTA}$$

By similar logic, we add the variables  $NOTB$  and  $NOTC$  and the productions

$$S \rightarrow b \text{ NOTB}$$

$$S \rightarrow c \text{ NOTC}$$

Now,  $NOTA$  is either the empty string, or some string that begins with  $b$ , or some string that begins with  $c$ . If it begins with  $b$ , then it must be followed by  $a$  (possibly empty) string that does not begin with  $b$ --and we already have a variable for that case,  $NOTB$ . Similarly, if  $NOTA$  is some string beginning with  $c$ , the  $c$  must be followed by  $NOTC$ . This gives the productions

$$\text{NOTA} \rightarrow \lambda$$

$$\text{NOTA} \rightarrow b \text{ NOTB}$$

$$\text{NOTA} \rightarrow c \text{ NOTC}$$

Similar logic gives the following productions for  $NOTB$  and  $NOTC$ :

$$\text{NOTB} \rightarrow \lambda$$

$$\text{NOTB} \rightarrow a \text{ NOTA}$$

$$\text{NOTB} \rightarrow c \text{ NOTC}$$

$$\text{NOTC} \rightarrow \lambda$$

$$\text{NOTC} \rightarrow a \text{ NOTA}$$

$$\text{NOTC} \rightarrow b \text{ NOTB}$$

We add  $NOTA$ ,  $NOTB$ , and  $NOTC$  to set  $V$ , and we're done.

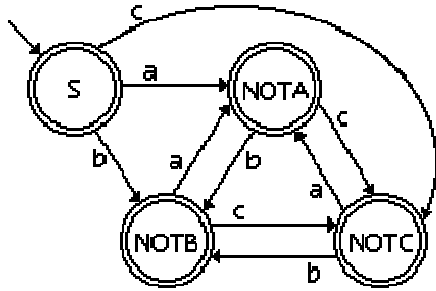
### Example derivation:

$$S \Rightarrow_a \text{NOTA} \Rightarrow_a b \text{NOTB} \Rightarrow_a b a \text{NOTA} \Rightarrow_a b a c \text{NOTC} \Rightarrow_a b a c.$$

### 3.6.2 Definition by NFA

Defining the language by an NFA follows almost exactly the same logic as defining the language by a grammar. Whenever an input symbol is read, go to a state that will accept any symbol other than the one read. To emphasize the similarity with the

preceding grammar, we will name our states to correspond to variables in the grammar.



**Figure 9: Definition of Language by NFA**

### 3.6.3 Definition by Regular Expression

As usual, it is more difficult to find a suitable regular expression to define this language, and the regular expression we do find bears little resemblance to the grammar or to the NFA.

The key insight is that strings of the language can be viewed as consisting of zero or more repetitions of the symbol  $a$ , and between them must be strings of the form  $bc$  or  $cb$ . So we can start with

$$X a Y a Y a Y a \dots Y a Z$$

where we have to find suitable expressions for  $X$ ,  $Y$ , and  $Z$ . But first, let's get the above expression in a proper form, by getting rid of the "...". This gives

$$X a (Y a)^* Z$$

and, since we might not have any  $a$ 's at all,

$$(X a (Y a)^* Z) + X$$

Now  $X$  can be empty, a single  $b$ , a single  $c$ , or can consist of an alternating sequence of  $bc$  and  $cb$ . This gives

$$X = (\lambda + b + c + (bc)^* + (cb)^*)$$

This isn't quite right, because it does not allow  $(bc)^*b$  or  $(cb)^*c$ . When we include these, we get

$$X = (\lambda + b + c + (bc)^* + (cb)^* + (bc)^*b + (cb)^*c)$$

This is now correct, but could be simplified. The last four terms include the  $\lambda + b + c$  cases, so we can drop those three terms. Then we can combine the last four terms into

$$X = (bc)^*(b + \lambda) + (cb)^*(c + \lambda)$$

Now, what about  $Z$ ? As it happens, there isn't any difference between what we need for  $Z$  and what we need for  $X$ , so we can also use the above expression for  $Z$ .

Finally, what about  $Y$ ? This is just like the others, except that  $Y$  cannot be empty. Luckily, it's easy to adjust the above expression for  $X$  and  $Z$  so that it can't be empty:

$$Y = ((bc)^*b + (cb)^*c)$$

Substituting into  $(X a (Y a)^* Z) + X$ , we get

$$((bc)^*(b + \lambda) + (cb)^*(c + \lambda) a ((bc)^*b + (cb)^*c) a)^* (bc)^*(b + \lambda) + (cb)^*(c + \lambda) + (bc)^*(b + \lambda) + (cb)^*(c + \lambda)$$

#### 4.0 CONCLUSION

In this unit you have been taken through regular expressions and the important role it plays in the definition of languages. In the next unit you will be learning about regular grammars

#### 5.0 SUMMARY

In this unit, you learnt that:

- A regular expression can be used to define a language.
- A regular expression represents a "pattern;" strings that match the pattern that are in the language, strings that do not match the pattern are not in the language
- There is a simple correspondence between regular expressions and the languages they denote
- Languages described by deterministic finite acceptors (DFAs) are called regular languages
- Regular expressions also describe regular languages

#### 6.0 TUTOR-MARKED ASSIGNMENT

1. Define primitive regular expressions
2. State the rules for creating addition regular expressions from any given regular expression(s)
3. How do regular expressions relate to automata?
4. Describe how to convert regular expressions to DFA. Is the reverse possible? Explain
5. With the aid of illustrative examples, briefly describe the three ways of defining a language

#### 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 2: Regular Languages

### Unit 3: Regular Grammars

#### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Grammars for Regular Languages
  - 3.2 Classifying Grammars
    - 3.2.1 Right-Linear Grammars
      - 3.2.1.1 Right-Linear Grammars and NFAs
    - 3.2.2 Left-Linear Grammars
  - 3.3 Regular Grammars
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Reading

#### 1.0 INTRODUCTION

In the preceding unit you learnt about regular expressions and how they can be used to define a language. In this unit, you will be learning about regular grammars, which is another way of defining languages.

Now let us go through your study objectives for this unit.

#### 2.0 OBJECTIVES

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

- Define regular grammars
- Classify grammars
- Show the connection between right-linear grammars and NFAs
- Construct a right-linear grammar from a left-linear grammar
- Distinguish between right-linear grammars from left-linear grammars
- With the aid of illustrative examples, state the relationship between regular grammars and each of the following:
  - DFAs
  - NFAs
  - Regular expressions

#### 3.0 MAIN CONTENT

### 3.1 Grammars for Regular Languages

From the previous unit, you already know that:

- A language defined by a DFA is a regular language.
- Any DFA can be regarded as a special case of an NFA.
- Any NFA can be converted to an equivalent DFA; thus, a language defined by an NFA is a regular language.
- A regular expression can be converted to an equivalent NFA; thus, a language defined by a regular expression is a regular language.
- An NFA can (with some effort!) be converted to a regular expression.

So DFAs, NFAs, and regular expressions are all "equivalent," in the sense that any language you define with one of these could be defined by the others as well.

We also know that languages can be defined by grammars. Now we will begin to classify grammars; and the first kinds of grammars we will look at are the *regular grammars*. As you might expect, regular grammars will turn out to be equivalent to DFAs, NFAs, and regular expressions.

### 3.2 Classifying Grammars

Recall that a *grammar*  $G$  is a quadruple  $G = (V, T, S, P)$  where:

- $V$  is a finite set of (meta)symbols, or *variables*.
- $T$  is a finite set of *terminal symbols*.
- $S \in V$  is a distinguished element of  $V$  called the *start symbol*.
- $P$  is a finite set of *productions*.

The above is true for *all* grammars. We will distinguish among different kinds of grammars based on the *form of the productions*. If the productions of a grammar all follow a certain pattern, we have one kind of grammar. If the productions all fit a different pattern, we have a different kind of grammar.

Productions have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*$$

Different types of grammars can be defined by putting additional restrictions on the left-hand side of productions, the right-hand side of productions, or both.

#### 3.2.1 Right-Linear Grammars

In general, productions have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*$$

In a *right-linear grammar*, all productions have one of the two forms:

$$V \rightarrow T^*V$$

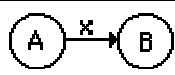


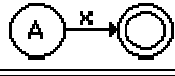
or

$$V \rightarrow T^*$$

That is, the left-hand side must consist of a single variable, and the right-hand side consists of any number of terminals (members of  $\Sigma$ ) optionally followed by a single variable. (The "right" in "right-linear grammar" refers to the fact that, following the arrow, a variable can occur only as the rightmost symbol of the production.)

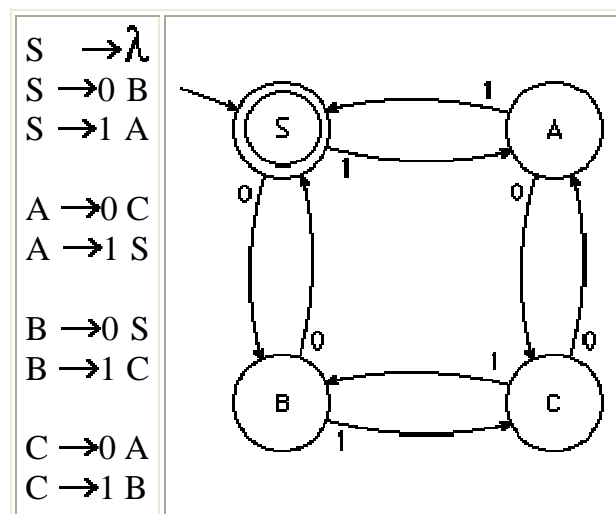
### 3.2.1.1 Right-Linear Grammars and NFAs

There is a simple connection between right-linear grammars and NFAs, as suggested by the following diagrams:

$A \rightarrow_x B$	
$A \rightarrow_x y z B$	
$A \rightarrow B$	
$A \rightarrow_x$	

*Figure 1: Connection between right-linear grammars and NFAs*

As an example of the correspondence between an NFA and a right-linear grammar, the following automaton and grammar both recognize the set of strings consisting of an even number of 0's and an even number of 1's.



**Figure 2: automaton and grammar for the set of strings consisting of an even number of 0's and an even number of 1's**

### 3.2.2 Left-Linear Grammars

In a *left-linear grammar*, all productions have one of the two forms:

$$V \rightarrow VT^*$$

or

$$V \rightarrow T^*$$

That is, the left-hand side must consist of a single variable, and the right-hand side consists of an optional single variable followed by any number of terminals. This is just like a right-linear grammar except that, following the arrow, a variable can occur only on the left of the terminals, rather than only on the right.

We will not pay much attention to left-linear grammars, because they turn out to be equivalent to right-linear grammars. Given a left-linear grammar for language  $L$ , we can construct a right-linear grammar for the same language, as follows:

**Table 1: Construction of right-linear grammar for any given left-linear grammar**

Step	Method
Construct a right-linear grammar for the (different) language $L^R$ .	Replace each production $A \rightarrow x$ of $L$ with a production $A \rightarrow x^R$ , and replace each production $A \rightarrow Bx$ with a production $A \rightarrow x^R B$ .
Construct an NFA for $L^R$ from the right-linear grammar. This NFA should have just one final state.	We talked about deriving an NFA from a right-linear grammar in section 3.2.1.1. If the NFA has more than one final state, we can make those states nonfinal, add a new final state, and put $\lambda$ transitions from each previously final state to the new final state.
Reverse the NFA for $L^R$ to obtain an NFA for $L$ .	<ol style="list-style-type: none"> <li>1. Construct an NFA to recognize language <math>L</math>.</li> <li>2. Ensure the NFA has only a single final state.</li> <li>3. Reverse the direction of the arcs.</li> <li>4. Make the initial state final and the final state initial.</li> </ol>
Construct a right-linear grammar for $L$ from the NFA for $L$ .	This is the technique we just talked about on in section 3.2.1.1

### 3.3 Regular Grammars

You have learned three ways of characterising regular languages: regular expressions, finite automata and construction from simple languages using simple operations. There is yet another way of characterizing them; that is by something called grammar.

A grammar is a set of rewrite rules which are used to generate strings by successively rewriting symbols. For example consider the language represented by  $a^+$ , which is  $\{a, aa, aaa, \dots\}$ . One can generate the strings of this language by the following procedure: Let  $S$  be a symbol to start the process with. Rewrite  $S$  using one of the following two rules:  $S \rightarrow a$ , and  $S \rightarrow aS$ . These rules mean that  $S$  is rewritten as  $a$  or as  $aS$ . To generate the string  $aa$  for example, start with  $S$  and apply the second rule to replace  $S$  with the right hand side of the rule, i.e.  $aS$ , to obtain  $aS$ . Then apply the first rule to  $aS$  to rewrite  $S$  as  $a$ . That gives us  $aa$ . We write  $S \Rightarrow aS$  to express that  $aS$  is obtained from  $S$  by applying a single production. Thus the process of obtaining  $aa$  from  $S$  is written as  $S \Rightarrow aS \Rightarrow aa$ . If we are not interested in the intermediate steps, the fact that  $aa$  is obtained from  $S$  is written as  $S \Rightarrow^* aa$ . In general if a string  $\beta$  is obtained from a string  $\alpha$  by applying productions of a grammar  $G$ , we write  $\alpha \Rightarrow_G^* \beta$  and say that  $\beta$  is derived from  $\alpha$ . If there is no ambiguity about the grammar  $G$  that is referred to, then we simply write  $\alpha \Rightarrow^* \beta$ .

Formally, a grammar consists of a set of nonterminals (or variables)  $V$ , a set of terminals  $\Sigma$  (the alphabet of the language), a start symbol  $S$ , which is a nonterminal, and a set of rewrite rules (productions)  $P$ . A production has in general the form  $\gamma \rightarrow \alpha$ , where  $\gamma$  is a string of terminals and nonterminals with at least one nonterminal in it and  $\alpha$  is a string of terminals and nonterminals. A grammar is regular if and only if  $\gamma$  is a single nonterminal and  $\alpha$  is a single terminal or a single terminal followed by a single nonterminal, that is a production is of the form  $X \rightarrow a$  or  $X \rightarrow aY$ , where  $X$  and  $Y$  are nonterminals and  $a$  is a terminal.

For example,  $\Sigma = \{a, b\}$ ,  $V = \{S\}$  and  $P = \{S \rightarrow aS, S \rightarrow bS, S \rightarrow \Lambda\}$  is a regular grammar and it generates all the strings consisting of  $a$ 's and  $b$ 's including the empty string.

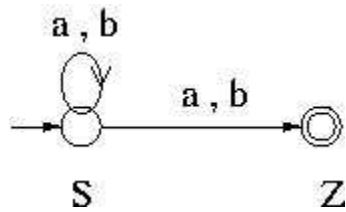
The following theorem holds for regular grammars.

**Theorem 1:** A language  $L$  is accepted by an FSA i.e. regular, if  $L - \{\Lambda\}$  can be generated by a regular grammar.

This can be proven by constructing an FSA for the given grammar as follows: For each nonterminal create a state.  $S$  corresponds to the initial state. Add another state as the accepting state  $Z$ . Then for every production  $X \rightarrow aY$ , add the transition  $\delta(X, a) = Y$  and for every production  $X \rightarrow a$  add the transition  $\delta(X, a) = Z$ .

For example  $\Sigma = \{a, b\}$ ,  $V = \{S\}$  and  $P = \{S \rightarrow aS, S \rightarrow bS, S \rightarrow a, S \rightarrow b\}$  form a regular grammar which generates the language  $(a + b)^+$ . An NFA that recognizes this language can be obtained by creating two states  $S$  and  $Z$ , and adding transitions  $\delta(S, a) = \{S, Z\}$  and  $\delta(S, b) = \{S, Z\}$ , where  $S$  is the initial state and  $Z$  is the accepting state of the NFA.

The NFA thus obtained is shown below.



Thus  $L - \{ \Lambda \}$  is regular. If  $L$  contains  $\Lambda$  as its member, then since  $\{ \Lambda \}$  is regular,  $L = (L - \{ \Lambda \}) \cup \{ \Lambda \}$  is also regular.

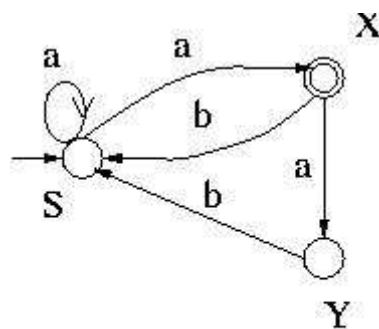
Conversely, from any NFA  $\langle Q, \Sigma, \delta, q_0, A \rangle$  a regular grammar  $\langle Q, \Sigma, P, q_0 \rangle$  is obtained as follows:

for any  $a$  in  $\Sigma$ , and nonterminals  $X$  and  $Y$ ,  $X \rightarrow aY$  is in  $P$  if and only if  $\delta(X, a) = Y$ , and for any  $a$  in  $\Sigma$  and any nonterminal  $X$ ,  $X \rightarrow a$  is in  $P$  if and only if  $\delta(X, a) = Y$  for some accepting state  $Y$ .

Thus the following converse of Theorem 1 is obtained.

**Theorem 2:** If  $L$  is regular i.e. accepted by an NFA, then  $L - \{ \Lambda \}$  is generated by a regular grammar.

For example, a regular grammar corresponding to the NFA given below is  $\langle Q, \{ a, b \}, P, S \rangle$ , where  $Q = \{ S, X, Y \}$ ,  $P = \{ S \rightarrow aS, S \rightarrow aX, X \rightarrow bS, X \rightarrow aY, Y \rightarrow bS, S \rightarrow a \}$ .



NFA

As you have learnt in the previous module, in addition to regular languages there are

three other types of languages in Chomsky hierarchy: context-free languages, context-sensitive languages and phrase structure languages. They are characterized by context-free grammars, context-sensitive grammars and phrase structure grammars, respectively.

These grammars are distinguished by the kind of productions they have but they also form a hierarchy, that is the set of regular languages is a subset of the set of context-free languages which is in turn a subset of the set of context-sensitive languages and the set of context-sensitive languages is a subset of the set of phrase structure languages.

A *regular grammar* is either a right-linear grammar or a left-linear grammar.

To be a right-linear grammar, *every* production of the grammar must have one of the two forms  $V \rightarrow T^*V$  or  $V \rightarrow T^*$ .

To be a left-linear grammar, *every* production of the grammar must have one of the two forms  $V \rightarrow VT^*$  or  $V \rightarrow T^*$ .

You do *not* get to mix the two. For example, consider a grammar with the following productions:

$$\begin{aligned} S &\rightarrow \lambda \\ S &\rightarrow a X \\ X &\rightarrow S b \end{aligned}$$

This grammar is neither right-linear nor left-linear, hence it is not a regular grammar. We have no reason to suppose that the language it generates is a regular language (one that is generated by a DFA).

In fact, the grammar generates a language whose strings are of the form  $a^n b^n$ . This language cannot be recognized by a DFA.

#### 4.0 CONCLUSION

In this unit you have been taken through regular grammars, which is another way of defining regular languages. You have also learnt that any regular grammar can be classified as either a right-linear or left-linear grammar.

In the next unit you will be learning about some of the properties of regular languages.

#### 5.0 SUMMARY

In this unit, you learnt that:

- You can distinguish among different kinds of grammars based on the *form of the productions*
- In a *right-linear grammar*, all productions have one of the two forms viz:  $V \rightarrow T^*V$  or  $V \rightarrow T^*$
- In a *left-linear grammar*, all productions have one of the two forms viz:  $V \rightarrow VT^*$  or  $V \rightarrow T^*$
- A grammar is a set of rewrite rules which are used to generate strings by successively rewriting symbols
- A language  $L$  is accepted by an FSA can be generated by a regular grammar
- If  $L$  is regular then  $L - \{ \Lambda \}$  is generated by a regular grammar.
- A *regular grammar* is either a right-linear grammar or a left-linear grammar

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Define regular grammars
2. With an illustrative example, show that if  $L$  is regular then  $L - \{ \Lambda \}$  is generated by a regular grammar
3. Distinguish between right-linear grammar and left-linear grammar
4. Outline the steps involved in constructing a right-linear grammar from a left-linear grammar. Hence or otherwise, given the left-linear grammar below, construct an equivalent right-linear grammar:

$$S \rightarrow Xab$$

$$X \rightarrow c$$

5. Construct an NFA for the right-linear grammar derived from the grammar in question (3) above

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.



**Module 2: Regular Languages****Unit 4: Closure Properties of Regular Languages****CONTENTS**

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Closure I
3.2	Closure II: Union, Concatenation, Negation, Kleene Star, Reverse
3.2.1	General Approach
3.3	Closure III: Intersection and Set Difference
3.3.1	Intersection
3.3.2	Set difference
3.4	Closure IV: Homomorphism
3.5	Closure V: Right Quotient
3.6	Standard Representations
3.7	Membership.
3.8	Finiteness.
3.9	Equivalence.
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

**INTRODUCTION**

In the last three units you started learning about regular languages and how they can be defined. In this unit, you will learn about some of the useful properties of regular languages and how each of these properties can be used to show that a language is regular.

Now let us go through your study objectives for this unit.

**2.0 OBJECTIVES**

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

- Enumerate the closure properties of regular languages
- Describe the steps to follow in applying each of these properties
- State the standard ways in which regular languages can be represented
- Prove the finiteness or otherwise of a language  $L$
- Prove that a string belong to a language

- Prove the equivalence of two languages

### 3.0 MAIN CONTENT

#### 3.1 Closure I

A set is *closed* under an operation if, whenever the operation is applied to members of the set, the result is also a member of the set.

For example, the set of integers is closed under addition, because  $x+y$  is an integer whenever  $x$  and  $y$  are integers. However, integers are not closed under division: if  $x$  and  $y$  are integers,  $x/y$  may or may not be an integer.

We have defined several operations on languages:

$L_1 \cup L_2$  Strings in either  $L_1$  or  $L_2$

$L_1 \cap L_2$  Strings in both  $L_1$  and  $L_2$

$L_1 L_2$  Strings composed of one string from  $L_1$  followed by one string from  $L_2$

$\neg L_1$  All strings (over the same alphabet) not in  $L_1$

$L_1^*$  Zero or more strings from  $L_1$  concatenated together

$L_1 - L_2$  Strings in  $L_1$  that are not in  $L_2$

$L_1^R$  Strings in  $L_1$ , reversed

In mathematical notations (especially set theory), the above can be written as:

$$\bar{L} = \{x \mid x \text{ is in } \Sigma^* \text{ and } x \text{ is not in } L\}$$

$$L_1 \cup L_2 = \{x \mid x \text{ is in } L_1 \text{ or } L_2\}$$

$$L_1 \cap L_2 = \{x \mid x \text{ is in } L_1 \text{ and } L_2\}$$

$$L_1 - L_2 = \{x \mid x \text{ is in } L_1 \text{ but not in } L_2\}$$

$$L_1 L_2 = \{xy \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$$

$$L^* = L^i = L^0 \cup L^1 \cup L^2 \cup \dots$$

$$L^+ = L^i = L^1 \cup L^2 \cup \dots$$

$$L_1^R = \{x \mid x \text{ is reversed}\}$$

We will show that the set of regular languages is closed under each of these operations. We will also define the operations of "homomorphism" and "right

quotient" and show that the set of regular languages is also closed under these operations.

### 3.2 Closure II: Union, Concatenation, Negation, Kleene Star, Reverse

#### 3.2.1 General Approach

- 1) Build automata (DFAs or NFAs) for each of the languages involved.
- 2) Show how to combine the automata to create a new automaton that recognizes the desired language.
- 3) Since the language is represented by an NFA or DFA, conclude that the language is regular.

#### 3.2.2 Union of $L_1$ and $L_2$

- 1) Create a new start state.
- 2) Make a  $\lambda$  transition from the new start state to each of the original start states.

#### 3.2.3 Concatenation of $L_1$ and $L_2$

- 1) Put a  $\lambda$  transition from each final state of  $L_1$  to the initial state of  $L_2$
- 2) Make the original final states of  $L_1$  nonfinal

#### 3.2.4 Negation of $L_1$

- 1) Start with a (complete) DFA, not with an NFA.
- 2) Make every final state nonfinal and every nonfinal state final.

#### 3.2.5 Kleene Star of $L_1$

- 1) Make a new start state; connect it to the original start state with a  $\lambda$  transition.
- 2) Make a new final state; connect the original final states (which become nonfinal) to it with  $\lambda$  transitions.
- 3) Connect the new start state and new final state with a pair of  $\lambda$  transitions.

#### 3.2.6 Reverse of $L_1$

- 1) Start with an automaton with just one final state.
- 2) Make the initial state final and the final state initial.
- 3) Reverse the direction of every arc.

### 3.3 Closure III: Intersection and Set Difference

Just as with the other operations, you prove that regular languages are closed under intersection and set difference by starting with automata for the initial languages, and constructing a new automaton that represents the operation applied to the initial languages. However, the constructions are somewhat trickier.

In these constructions you form a completely new machine, whose states are each labelled with an ordered pair of state names: the first element of each pair is a state from  $L_1$ , and the second element of each pair is a state from  $L_2$ . (Usually you will not need a state for every such pair, just some of them.)

1. Begin by creating a start state whose label is (start state of  $L_1$ , start state of  $L_2$ ).
2. Repeat the following until no new arcs can be added:
  - i. Find a state  $(A, B)$  that lacks a transition for some  $x$  in  $\Sigma$ .
  - ii. Add a transition on  $x$  from state  $(A, B)$  to state  $(\delta(A, x), \delta(B, x))$ . (If this state doesn't already exist, create it.)

The same construction is used for both intersection and set difference. The distinction is in how the final states are selected.

### 3.3.1 Intersection

Mark a state  $(A, B)$  as final if *both*

- (i)  $A$  is a final state in  $L_1$ , and
- (ii)  $B$  is a final state in  $L_2$ .

### 3.3.2 Set difference

Mark a state  $(A, B)$  as final if  $A$  is a final state in  $L_1$ , but  $B$  is *not* a final state in  $L_2$ .

## 3.4 Closure IV: Homomorphism

You should note that "homomorphism" is a term borrowed from group theory. What we refer to as a "homomorphism" is really a special case.

Suppose  $\Sigma$  and  $\Gamma$  are alphabets (not necessarily distinct). Then a *homomorphism*  $h$  is a function from  $\Sigma^*$  to  $\Gamma^*$ .

If  $w$  is a string in  $\Sigma^*$ , then we define  $h(w)$  to be the string obtained by replacing each symbol  $x \in \Sigma$  by the corresponding string  $h(x) \in \Gamma^*$ .

If  $L$  is a language on  $\Sigma$ , then its *homomorphic image* is a language on  $\Gamma$ . Formally,

$$h(L) = \{h(w) : w \in L\}$$

**Theorem.** If  $L$  is a regular language on  $\Sigma$ , then its homomorphic image  $h(L)$  is a regular language on  $\Gamma$ . That is, if you replaced every string  $w$  in  $L$  with  $h(w)$ , the resultant set of strings would be a regular language on  $\Gamma$ .

**Proof.**

- 1) Construct a DFA representing  $L$ . This is possible because  $L$  is regular.

- 2) For each arc in the DFA, replace its label  $x \in \Sigma$  with  $h(x) \in \Gamma$ .
- 3) If an arc is labelled with a string  $w$  of length greater than one, replace the arc with a series of arcs and (new) states, so that each arc is labeled with a single element of  $\Gamma$ . The result is an NFA that recognizes exactly the language  $h(L)$ .
- 4) Since the language  $h(L)$  can be specified by an NFA, the language is regular.  
Q.E.D.

### 3.5 Closure V: Right Quotient

Let  $L_1$  and  $L_2$  be languages on the same alphabet. The *right quotient* of  $L_1$  with  $L_2$  is  $L_1/L_2 = \{w: wx \in L_1 \text{ and } x \in L_2\}$

That is, the strings in  $L_1/L_2$  are strings from  $L_1$  "with their tails cut off." If some string of  $L_1$  can be broken into two parts,  $w$  and  $x$ , where  $x$  is in language  $L_2$ , then  $w$  is in language  $L_1/L_2$ .

**Theorem.** If  $L_1$  and  $L_2$  are both regular languages, then  $L_1/L_2$  is a regular language.

**Proof:** Again, the proof is by construction. We start with a DFA  $M(L_1)$  for  $L_1$ ; the DFA we construct is exactly like the DFA for  $L_1$ , except that (in general) different states will be marked as final.

For each state  $Q_i$  in  $M(L_1)$ , determine if it should be final in  $M(L_1/L_2)$  as follows:

- Starting in state  $Q_i$  as if it were the initial state, determine if any of the strings in language  $L_2$  are accepted by  $M(L_1)$ . If there are any, then state  $Q_i$  should be marked as final in  $M(L_1/L_2)$ . (Why?)

That is the basic algorithm. However, one of the steps in it is problematical: since language  $L_2$  may have an infinite number of strings, how do we determine whether some unknown string in the language is accepted by  $M(L_1)$  when starting at  $Q_i$ ? We *cannot* try all the strings, because we insist on a finite algorithm.

The trick is to construct a new DFA that recognizes the *intersection* of two languages: (1)  $L_2$ , and (2) the language that would be accepted by DFA  $M(L_1)$  if  $Q_i$  were its initial state. We already know we can build this machine. Now, if this machine recognizes *any string whatever* (we can check this easily), then the two machines have a nonempty intersection, and  $Q_i$  should be a final state.

We have to go through this same process for every state  $Q_i$  in  $M(L_1)$ , so the algorithm is too lengthy to step through by hand. However, it is enough for our purposes that the algorithm exists.

Finally, since we can construct a DFA that recognizes  $L_1/L_2$ , this language is therefore regular, and we have shown that the regular languages are closed under right quotient.

### 3.6 Standard Representations

A regular language is given in a *standard representation* if it is specified by one of:

- A finite automaton (DFA or NFA).
- A regular expression.
- A regular grammar.

(The importance of these particular representations is simply that they are precise and unambiguous; thus, we can prove things about languages when they are expressed in a standard representation.)

### 3.7 Membership.

If  $L$  is a language on alphabet  $\Sigma$ ,  $L$  is in a standard representation, and  $w \in \Sigma^*$ , then there is an algorithm for determining whether  $w \in L$ .

**Proof.** Build the automaton and use it to test  $w$ .

### 3.8 Finiteness.

If language  $L$  is specified by a standard representation, there is an algorithm to determine whether the set  $L$  is empty, finite, or infinite.

**Proof.** Build the automaton.

- If there is no path from the initial state to a final state, then the language is empty (and finite).
- If there is a path containing a cycle from the initial state to some final state, then the language is infinite.
- If no path from the initial state to a final state contains a cycle, then the language is finite.

### 3.9 Equivalence.

If languages  $L_1$  and  $L_2$  are each given in a standard representation, then there is an algorithm to determine whether the languages are identical.

**Proof.** Construct the language

$$(L_1 \cap \neg L_2) \cup (\neg L_1 \cap L_2)$$

If this language is empty, then  $L_1 = L_2$ .

## 4.0 CONCLUSION

In this unit you have been taken through the closure properties of regular languages and how they can be useful in generating regular languages and also showing that a

language is regular. In the next unit, you will be learning about the pumping lemma for regular languages.

## 5.0 SUMMARY

In this unit, you learnt that:

- A set is *closed* under an operation if, whenever the operation is applied to members of the set, the result is also a member of the set
- you prove that regular languages are closed under the various operations by starting with automata for the initial languages
- A regular language is given in a *standard representation* if it is specified by one of the following:
  - A finite automaton (DFA or NFA).
  - A regular expression.
  - A regular grammar.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. When is a regular language given in a *standard representation*?
2. Describe an algorithm to show the equivalence of two languages
3. Describe how you will show that the set of regular languages is closed under each of the following operations:
  - Set difference
  - Union
  - Negation
  - Intersection
4. Show that the language consisting of all strings of balanced parentheses is not regular.
5. Prove that the two regular expressions  $(a+b)^*$  and  $(a^*b^*)^*$  generate the same language.
6. Consider the function on languages  $noprefix(L) = \{ w \text{ in } L \mid \text{no proper prefix of } w \text{ is a member of } L \}$ . Show that the regular languages are closed under the *noprefix* function.
7. Consider the function on languages  $remove\_middle\_third(L) = \{ xz \mid \text{for some } y, xyz \text{ is in } L \text{ where } |x| = |y| = |z| \}$ . Show that the regular languages are not closed under the *remove\\_middle\\_third* function.
8. An equivalence relation  $R$  on a language  $L$  contained in  $\Sigma^*$  is *right invariant* if  $xRy$  implies  $xzRyz$  for all  $z$  in  $\Sigma^*$ .  $R$  is of *finite index* if it partitions  $L$  into a finite number of equivalence classes. Show that  $L$  is regular if and only if it is

the union of some of the equivalence classes of a right-invariant equivalence relation on  $L$  of finite index.

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 2: Regular Languages

### Unit 5: The Pumping Lemma

#### CONTENTS

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	The Pigeonhole Principle
3.1.1	Pigeonhole
3.1.2	Pigeonhole Principle
3.2	The Pumping Lemma
3.2.1	Applying the Pumping Lemma
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	<i>References/Further Reading</i>

#### 1.0 INTRODUCTION

In this concluding unit of module 2, you will be taken through the pumping lemma for regular languages. You will also learn about how to apply the pumping lemma.

Now let us go through your study objectives for this unit.

#### 2.0 OBJECTIVES

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

- Define pigeon hole
- Explain the pigeon hole principle
- State the pumping lemma
- State the use of the pumping lemma
- Apply the pumping lemma to regular languages

#### 3.0 MAIN CONTENT

##### 3.1 The Pigeonhole Principle

###### 3.1.1 Pigeonhole

1. a hole or small recess for pigeons to nest

2. a small open compartment (as in a desk or cabinet) for keeping letters or documents
3. a neat category which usually fails to reflect actual complexities.

### 3.1.2 Pigeonhole Principle

If  $n$  objects are put into  $m$  containers, where  $n > m$ , then at least one container must hold more than one object.

The pigeonhole can be used to prove that certain infinite languages are not regular. (Remember, any finite language *is* regular.)

As we have informally observed, DFAs "can't count." This can be shown formally by using the pigeonhole principle. As an example, we show that  $L = \{a^n b^n : n > 0\}$  is not regular. The proof is by contradiction.

Suppose  $L$  is regular. There are an infinite number of values of  $n$  but  $M(L)$  has only a finite number of states. By the pigeonhole principle, there must be distinct values of  $i$  and  $j$  such that  $a^i$  and  $a^j$  end in the same state. From this state,

- $b^i$  must end in a final state, because  $a^i b^i$  is in  $L$ ; and
- $b^i$  must end in a nonfinal state, because  $a^j b^i$  is not in  $L$ .

Since the state reached cannot be both final and nonfinal, we have a contradiction. Thus our assumption, that  $L$  is regular, must be incorrect. Q.E.D.

### 3.2 The Pumping Lemma

Pumping Lemma relates the size of string accepted with the number of states in a DFA. Here is what the *pumping lemma* says:

- If an infinite language is regular, it can be defined by a DFA.
- The DFA has some finite number of states (say,  $n$ ).
- Since the language is infinite, some strings of the language must have length  $> n$ .
- For a string of length  $> n$  accepted by the DFA, the walk through the DFA must contain a cycle.
- Repeating the cycle an arbitrary number of times must yield another string accepted by the DFA.

The *pumping lemma* for regular languages is another way of proving that a given (infinite) language is not regular. (The pumping lemma *cannot* be used to prove that a given language *is* regular.)

The proof is always by contradiction. A brief outline of the technique is as follows:

- Assume the language  $L$  is regular.

- By the pigeonhole principle, any sufficiently long string in  $L$  must repeat some state in the DFA; thus, the walk contains a cycle.
- Show that repeating the cycle some number of times ("pumping" the cycle) yields a string that is not in  $L$ .
- Conclude that  $L$  is not regular.

This is hard because:

- We do not know the DFA (if we did, the language would be regular!). Thus, we have to do the proof for an arbitrary DFA that accepts  $L$ .
- Since we do not know the DFA, we certainly do not know the cycle.

But we can sometimes pull it off for the following reasons:

- We get to choose the string (but it must be in  $L$ ).
- We get to choose the number of times to "pump."

### 3.2.1 Applying the Pumping Lemma

Here is a more formal definition of the pumping lemma:

If  $L$  is an infinite regular language, then there exists some positive integer  $m$  such that any string  $w \in L$  whose length is  $m$  or greater can be decomposed into three parts,  $xyz$ , where

- $|xy|$  is less than or equal to  $m$ ,
- $|y| > 0$ ,
- $w_i = xy^iz$  is also in  $L$  for all  $i = 0, 1, 2, 3, \dots$

Here is what it all means:

- $m$  is a (finite) number chosen so that strings of length  $m$  or greater *must* contain a cycle. Hence,  $m$  must be equal to or greater than the number of states in the DFA. Remember that we *do not know the DFA*, so we can't actually choose  $m$ ; we just know that such an  $m$  must exist.
- Since string  $w$  has length greater than or equal to  $m$ , we can break it into two parts,  $xy$  and  $z$ , such that  $xy$  must contain a cycle. We do not know the DFA, so we do not know exactly where to make this break, but we know that  $|xy|$  can be less than or equal to  $m$ .
- We let  $x$  be the part before the cycle,  $y$  be the cycle, and  $z$  the part after the cycle. (It is possible that  $x$  and  $z$  contain cycles, but we do not care about that.) Again, we do not know exactly where to make this break.
- Since  $y$  is the cycle we are interested in, we must have  $|y| > 0$ , otherwise it is not a cycle.
- By repeating  $y$  an arbitrary number of times,  $xy^iz$ , we must get other strings in  $L$ .

- If, despite all the above uncertainties, we can show that the DFA has to accept some string that we know is not in the language, then we can conclude that the language is not regular.

To use this lemma, we need to show:

1. For *any* choice of  $m$ ,
2. for some  $w \in L$  that we get to choose (and we will choose one of length at least  $m$ ),
3. for *any* way of decomposing  $w$  into  $xyz$ , so long as  $|xy|$  is not greater than  $m$  and  $y$  is not  $\lambda$ ,
4. we can choose an  $i$  such that  $xy^iz$  is not in  $L$ .

We can view this as a game wherein our opponent makes moves 1 and 3 (choosing  $m$  and choosing  $xyz$ ) and we make moves 2 and 4 (choosing  $w$  and choosing  $i$ ). Our goal is to show that we can *always* beat our opponent. If we can show this, we have proved that  $L$  is not regular.

### Example 1

Prove that  $L = \{a^n b^n : n \geq 0\}$  is not regular.

1. We do not know  $m$ , but assume there is one.
2. Choose a string  $w = a^n b^n$  where  $n > m$ , so that any prefix of length  $m$  consists entirely of a's.
3. We do not know the decomposition of  $w$  into  $xyz$ , but since  $|xy| \leq m$ ,  $xy$  must consist entirely of a's. Moreover,  $y$  cannot be empty.
4. Choose  $i = 0$ . This has the effect of dropping  $|y|$  a's out of the string, without affecting the number of b's. The resultant string has fewer a's than b's, hence does not belong to  $L$ . Therefore  $L$  is not regular.

### Example 2

Prove that  $L = \{a^n b^k : n > k \text{ and } n \geq 0\}$  is not regular.

1. We do not know  $m$ , but assume there is one.
2. Choose a string  $w = a^n b^k$  where  $n > m$ , so that any prefix of length  $m$  consists entirely of a's, and  $k = n - 1$ , so that there is just one more a than b.
3. We do not know the decomposition of  $w$  into  $xyz$ , but since  $|xy| \leq m$ ,  $xy$  must consist entirely of a's. Moreover,  $y$  cannot be empty.
4. Choose  $i = 0$ . This has the effect of dropping  $|y|$  a's out of the string, without affecting the number of b's. The resultant string has fewer a's than before, so it has either fewer a's than b's, or the same number of each. Either way, the string does not belong to  $L$ , so  $L$  is not regular.

### Example 3

Prove that  $L = \{a^n : n \text{ is a prime number}\}$  is not regular.

1. We do not know  $m$ , but assume there is one.
2. Choose a string  $w = a^n$  where  $n$  is a prime number and  $|xyz| = n > m+1$ . (This can always be done because there is no largest prime number.) Any prefix of  $w$  consists entirely of  $a$ 's.
3. We do not know the decomposition of  $w$  into  $xyz$ , but since  $|xy| \leq m$ , it follows that  $|z| > 1$ . As usual,  $|y| > 0$ ,
4. Since  $|z| > 1$ ,  $|xz| > 1$ . Choose  $i = |xz|$ . Then  $|xy^iz| = |xz| + |y||xz| = (1 + |y|)|xz|$ . Since  $(1 + |y|)$  and  $|xz|$  are each greater than 1, the product must be a composite number. Thus  $|xy^iz|$  is a composite number.

### Self Assessment Exercise

1. Construct a PDA that accepts  $\{wcw^R \mid w \text{ is any string of } a\text{'s and } b\text{'s}\}$  by final state.
2. Construct a PDA that accepts  $\{wcw^R \mid w \text{ is any string of } a\text{'s and } b\text{'s}\}$  by empty stack.
3. Construct a PDA that accepts  $\{ww^R \mid w \text{ is any string of } a\text{'s and } b\text{'s}\}$  by final state.
4. Construct a PDA that accepts  $\{ww^R \mid w \text{ is any string of } a\text{'s and } b\text{'s}\}$  by empty stack.
5. Construct a PDA  $P$  such that  $N(P) = L(G)$  where  $G$  is  $S \rightarrow (S)S \mid \epsilon$ .

## 4.0 CONCLUSION

In this unit you have learnt about the pumping lemma for regular languages. The pumping lemma is based on the pigeon hole principle and it can be used to prove that an infinite language is not regular. It can never be used to show that a language is regular.

In the next module, you will be learning about another type of languages that is next to regular languages in the Chomsky hierarchy.

## 5.0 SUMMARY

In this unit, you learnt:

- The pigeonhole can be used to prove that certain infinite languages are not regular
- the pigeonhole principle can be used to formally show that DFAs "cannot count."
- Pumping Lemma relates the size of string accepted with the number of states in a DFA

- The *pumping lemma* for regular languages is another way of proving that a given (infinite) language is not regular
- The pumping lemma *cannot* be used to prove that a given language *is* regular

## 6.0 TUTOR-MARKED ASSIGNMENT

1. What is a pigeon hole?
2. Briefly describe the pigeon hole principle. How is it related to the pumping lemma for regular languages?
3. What does the pumping lemma say?
4. What do we need to do to use the pumping lemma?

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 3: Context-Free Languages

### Unit 1: Context-Free Grammars

#### CONTENTS

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Context-Free Grammars (CFG)
3.2	Regular Grammars are Context Free
3.2.1	Notes on Terminology
3.3	Languages and Grammars
3.4	Sentential Forms
3.5	Leftmost and Rightmost Derivations:
3.6	Derivation Trees
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

#### 1.0 INTRODUCTION

In the previous module, you learnt about regular languages. In this module you will be learning about context-free languages and the automata that accepts strings generated by context-free languages. But in this introductory unit of the module, let's take you through the basic definitions of context-free grammar.

Now let us go through your study objectives for this unit.

#### 2.0 OBJECTIVES

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

- Define context-free grammars
- Distinguish between regular grammars and context-free grammars
- Determine strings generated by a context-free grammar

#### 3.0 MAIN CONTENT

##### 3.1 Context-Free Grammars (CFG)

A **context-free grammar** is a grammar  $G = (X, T, S, P)$  for which all rules, or productions, in  $P$  have the special form  $A \Rightarrow \alpha$ , for  $A \in X - T$  and  $\alpha \in X^*$ .

Additionally, for any two strings  $u, v \in X^*$  write  $u \Rightarrow v$  ( $u$  **directly produces**  $v$ ) if and only if

$$(1) u = u_1 A u_2 \text{ for } u_1, u_2 \in X^* \text{ and } A \in X - T \text{ and}$$

$$(2) v = v_1 \alpha v_2 \text{ and } A \Rightarrow \alpha, \alpha \in X^*,$$

is a production from  $P$ .

The reduction  $u \Rightarrow v$  is also called a **direct production**. Finally, write  $u \Rightarrow^* v$  for two strings  $u, v \in X^*$  ( $u$  **derives**  $v$ ) if there is a sequence  $u = u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_n = v$  of direct productions  $u_i \Rightarrow u_{i+1}$  from  $R$ . The **length** of the derivation is  $n$ . The **language generated by  $G$**  is  $\{x \in T^* \mid S \Rightarrow^* x\}$ .

Thus, the definition just articulates the reduction of  $A$  to  $\alpha$  in *any context* in which  $A$  occurs. The productions for a context-free grammar are a restricted form of the productions allowed for a general grammar. Thus, a context-free grammar is a grammar.

It is trivial that every regular language is context-free. The reverse, as will be seen presently, is not true. Before proving the central theorem for this section two typical examples are given.

### Example 1

Consider  $G = (X, T, P, S)$  with  $T = \{a, b\}$  and  $X = \{S, a, b, \lambda\}$ . The productions, or grammar rules, are:  $S \Rightarrow aSb \mid \lambda$ . Then it is clear that  $L(G) = \{a^n b^n \mid n \geq 0\}$ . From the previous module it is known that this language is not regular.

### Example 2: A Grammar for Arithmetic Expressions

Let  $X = \{E, T, F, id, +, -, *, /, (, ), a, b, c\}$  and  $T = \{a, b, c, +, -, *, /, (, )\}$ . The start symbol  $S$  is  $E$  and the productions are as follows:

$$E \Rightarrow E + T \mid E - T \mid T$$

$$T \Rightarrow T * F \mid T / F \mid F$$

$$F \Rightarrow (E) \mid id$$

$$id \Rightarrow a \mid b \mid c$$

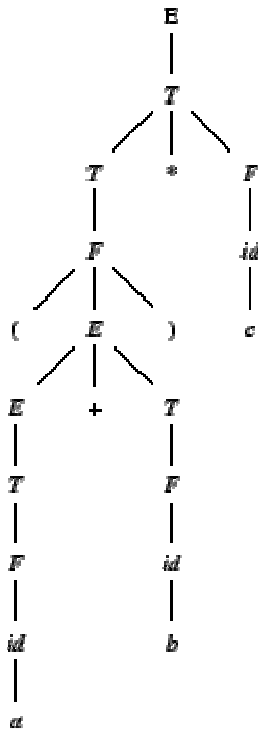
Then the string  $(a + b)^*c$  belongs to  $L(G)$ . Indeed, it is easy to write down a derivation of this string:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow (E + T) * F$$

$$\Rightarrow (T + T) * F \Rightarrow (F + T) * F \Rightarrow (id + T) * F \Rightarrow (a + T) * F$$

$$\Rightarrow (a + F) * F \Rightarrow (a + id) * F \Rightarrow (a + b) * F \Rightarrow (a + b) * id \Rightarrow (a + b)^*c$$

The derivation just adduced is *leftmost* in the sense that the leftmost nonterminal was always substituted. Although derivations are in general by no means unique, the leftmost one is. The entire derivation can also be nicely represented in a tree form, as Figure. 1 suggests.



**Figure 1:** Derivation Tree for the Expression  $(a + b)^*c$

The internal nodes of the derivation, or syntax, tree are nonterminal symbols and the frontier of the tree consists of terminal symbols. The start symbol is the *root* and the derived symbols are *nodes*. The *order* of the tree is the maximal number of successor nodes for any given node. In this case, the tree has order 3. Finally, the *height* of the tree is the length of the longest path from the root to a leaf node, *i.e.* a node that has no successor. The string  $(a + b)^*c$  obtained from the concatenation of the leaf nodes together from left to right is called the *yield* of the tree.

### 3.2 Regular Grammars are Context Free

Recall that productions of a right-linear grammar must have one of the two forms  
 $A \rightarrow x$

or  $A \rightarrow xB$

where  $A, B \in X$ , and  $x \in T^*$ .

Since  $T^* \subseteq (X \cup T)^*$  and  $T^*X \subseteq (V \cup T)^*$ , it follows that *every right-linear grammar is also a context-free grammar*.

Similarly, right-linear grammars and linear grammars are also context-free grammars.

A *context-free language (CFL)* is a language that can be defined by a context-free grammar.

### 3.2.1 Notes on Terminology

Every regular grammar is a context-free grammar, in the same way that every dog is an animal.

In normal speech we try to be as specific as possible. If we know that, say, Fido is a dog; we generally refer to Fido as a dog. We do not refer to Fido as an animal (unless we are trying to be deliberately vague). But if asked whether Fido is an animal, the correct answer is certainly "yes."

In the same way, if language  $L$  is a regular language, we generally refer to  $L$  as a regular language. We do not refer to  $L$  as a context-free language unless we are being deliberately vague. But if asked whether  $L$  is a context-free language, the correct answer is "yes."

The usual convention of being as specific as possible sometimes leads to confusion. If I say language  $L$  is a context-free language, I probably mean either (a)  $L$  is *not* regular, or (b) I *do not know* whether  $L$  is regular. If I do know that  $L$  is a regular language, I should call it a regular language, not a context-free language.

### 3.3 Languages and Grammars

A *regular language* is a language that can be defined by a regular grammar.

A *context-free language* is a language that can be defined by a context-free grammar.

If grammar  $G$  is context free but not regular, we know the language  $L(G)$  is context free. We *do not know* that  $L(G)$  is not regular. It might be possible to find a regular grammar  $G_2$  that also defines  $L$ .

#### Example 3

Consider the following grammar:

$$G = (\{S, A, B\}, \{a, b\}, S, \{S \rightarrow AB, A \rightarrow aA, A \rightarrow \lambda, B \rightarrow Bb, B \rightarrow \lambda\})$$

Is  $G$  a context-free grammar? Yes.

Is  $G$  a regular grammar? No.

Is  $L(G)$  a context-free language? Yes.

Is  $L(G)$  a regular language? *Yes* - the language  $L(G)$  is regular because it can be defined by the regular grammar:

$$G = (\{S, A, B\}, \{a, b\}, S, \{S \rightarrow A, A \rightarrow aA, A \rightarrow B, B \rightarrow bB, B \rightarrow \lambda\})$$

**Example 4**

We have shown that  $L = \{a^n b^n : n \geq 0\}$  is not regular. Here is a context-free grammar for this language.

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow \lambda\})$$

**Example 5**

We have shown that  $L = \{a^n b^k : k > n \geq 0\}$  is not regular. Here is a context-free grammar for this language.

$$G = (\{S, B\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow B, B \rightarrow bB, B \rightarrow b\}).$$

**Example 6**

The language  $L = \{ww^R : w \in \{a, b\}^*\}$ , where each string in  $L$  is a palindrome, is not regular. Here is a context-free grammar for this language.

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \lambda\}).$$

**Example 7**

The language  $L = \{w : w \in \{a, b\}^*, n_a(w) = n_b(w)\}$ , where each string in  $L$  has an equal number of a's and b's, is not regular. Consider the following grammar:

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow bSa, S \rightarrow SS, S \rightarrow \lambda\}).$$

1. Does every string recognized by this grammar have an equal number of a's and b's?
2. Is every string consisting of an equal number of a's and b's recognized by this grammar?

**Example 8**

The language  $L$ , consisting of balanced strings of parentheses, is context-free but not regular. The grammar is simple, but we have to be careful to keep our symbols '(' and ')' separate from our metasymbols ( and ).

$$G = (\{S\}, \{ (, ) \}, S, \{S \rightarrow (S), S \rightarrow SS, S \rightarrow \lambda\}).$$

**3.4 Sentential Forms**

A *sentential form* is the start symbol  $S$  of a grammar or any string in  $(X \cup T)^*$  that can be derived from  $S$ .

Consider the linear grammar

$(\{S, B\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow B, B \rightarrow bB, B \rightarrow \lambda\})$ .

A derivation using this grammar might look like this:

$S \Rightarrow aS \Rightarrow aB \Rightarrow abB \Rightarrow abbB \Rightarrow abb$

Each of  $\{S, aS, aB, abB, abbB, abb\}$  is a sentential form.

Because this grammar is linear, each sentential form has at most one variable. Hence there is never any choice about which variable to expand next.

### 3.5 Leftmost and Rightmost Derivations:

Now consider the grammar

$G = (\{S, A, B, C\}, \{a, b, c\}, S, P)$

where

$P = \{S \rightarrow ABC, A \rightarrow aA, A \rightarrow \lambda, B \rightarrow bB, B \rightarrow \lambda, C \rightarrow cC, C \rightarrow \lambda\}$ .

With this grammar, there is a choice of variables to expand. Here is a sample derivation:

$S \Rightarrow ABC \Rightarrow aABC \Rightarrow aABcC \Rightarrow aBcC \Rightarrow abBcC \Rightarrow abBc \Rightarrow abbBc \Rightarrow abbc$

If we always expanded the leftmost variable first, we would have a *leftmost derivation*:

$S \Rightarrow ABC \Rightarrow aABC \Rightarrow aBC \Rightarrow abBC \Rightarrow abbBC \Rightarrow abbC \Rightarrow abbcC \Rightarrow abbc$

Conversely, if we always expanded the rightmost variable first, we would have a *rightmost derivation*:

$S \Rightarrow ABC \Rightarrow ABcC \Rightarrow ABc \Rightarrow AbBc \Rightarrow AbbBc \Rightarrow Abbc \Rightarrow aAbbc \Rightarrow abbc$

There are two things to notice here:

1. Different derivations result in quite different sentential forms, but
2. For a context-free grammar, it really does not make much difference in what order we expand the variables.

### Self Assessment Exercise

1. Give a CFG for all strings of a's and b's with twice as many a's as b's. Show leftmost derivations for the four shortest strings generated by your grammar.

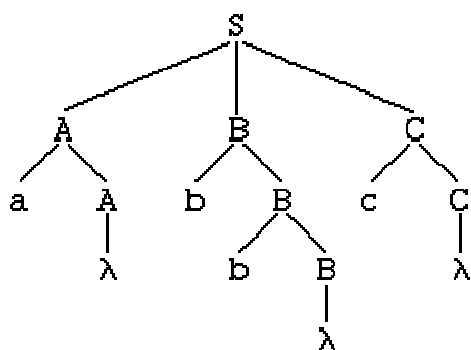
### 3.6 Derivation Trees

Since the order in which we expand the variables in a sentential form does not seem to make any difference, it would be nice to show a derivation in some way that is independent of the order. A *derivation tree* is a way of presenting a derivation in an order-independent fashion.

For example, for the following derivation:

$$S \Rightarrow ABC \Rightarrow aABC \Rightarrow aABcC \Rightarrow aBcC \Rightarrow abBcC \Rightarrow abBc \Rightarrow abbBc \Rightarrow abbc$$

we would have the derivation tree:



This tree represents not just the given derivation, but all the different orders in which the same productions could be applied to produce the string *abbc*.

A *partial derivation tree* is any subtree of a derivation tree such that, for any node of the subtree, either all of its children are also in the subtree, or none of them are.

The *yield* of the tree is the final string obtained by reading the leaves of the tree from left to right, ignoring the  $\lambda$ s (unless *all* the leaves are  $\lambda$ , in which case the yield is  $\lambda$ ). The yield of the above tree is the string *abbc*, as expected.

The yield of a partial derivation tree that contains the root is a sentential form.

### 4.0 CONCLUSION

In this unit you have been taken through context-free grammars and their relationship to regular grammars. You were also introduced to the concept of derivations and parse

tree. In the next unit, you will be learning more about context-free languages by learning about some properties of context-free grammars.

## 5.0 SUMMARY

In this unit, you learnt that:

- every regular language is context-free but the reverse is not true
- A *context-free language* is a language that can be defined by a context-free grammar
- A *sentential form* is the start symbol  $S$  of a grammar or any string in  $(X \cup T)^*$  that can be derived from  $S$
- If we always expanded the leftmost variable of a sentential form first, we would have a *leftmost derivation* and if we always expanded the rightmost variable first, we would have a *rightmost derivation*
- A *derivation tree* is a way of presenting a derivation in an order-independent fashion
- A *partial derivation tree* is any subtree of a derivation tree such that, for any node of the subtree, either all of its children are also in the subtree, or none of them are.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Define context-free grammars
2. Given the grammar  $G$ :

$$\begin{aligned} E &\Rightarrow E + T \mid E - T \mid T \\ T &\Rightarrow T^*F \mid T/F \mid F \\ F &\Rightarrow (E) \mid id \\ id &\Rightarrow a \mid b \mid c \end{aligned}$$

find the

- i. rightmost derivation
- ii. leftmost derivation

for the following strings

- a)  $a+a+a$
- b)  $a(c/b)$

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**Module 3: Context-Free Languages****Unit 2: Properties of Context-Free Languages***CONTENTS*

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Syntax Trees
3.1.1	Definition of Syntax Tree
3.1.2	Ambiguity
3.1.2.1	Ambiguous Grammars, Ambiguous Languages
3.2	Chomsky Normal Form
3.2.1	Definition of Chomsky Normal Form
3.2.2	Putting a CFG into Chomsky Normal Form
3.3	Non Context-Free Languages: Ogden's Lemma (The Pumping Lemma for CFL)
3.3.1	Using the Pumping Lemma
3.4	Closure Properties of Context Free Languages
3.5	Parsing
3.5.1	Exhaustive Search Parsing
3.5.2	Grammars for Exhaustive Parsing
3.5.3	Efficient Parsing
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	<i>References/Further Reading</i>

**1.0 INTRODUCTION**

In the previous unit you learnt about context-free grammars and the type of language that is generated by them. In this unit you will be taken through the properties of context-free languages.

It is desirable not only to classify languages by the architecture of machines that recognize them but also to have tests to show that a language is not of a particular type. For this reason we establish so-called pumping lemmas whose purpose is to show how strings in one language can be elongated or "pumped up." Pumping up may reveal that a language does not fall into a presumed language category. We also develop other properties of languages that provide mechanisms for distinguishing among language types. Because of the importance of context-free languages, we examine how they are parsed, a key step in programming language translation.

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

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

- State the properties of CFL
- State the pumping lemma for CFL
- Use the pumping lemma for CFL
- Determine when a grammar is ambiguous
- Define syntax tree

## 3.0 MAIN CONTENT

### 3.1 Syntax Trees

Tree representations of derivations, also known as syntax trees, were briefly introduced in the preceding unit to promote intuition of derivations. Since these are such important tools for the investigation of context-free languages, they will be dealt with a little more systematically here.

#### 3.1.1 Definition of Syntax Tree

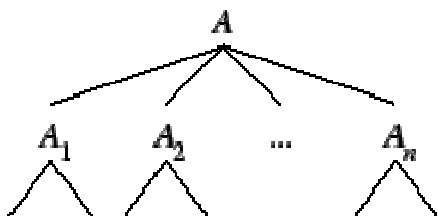
Let  $G = (X, T, P, S)$  be a context-free grammar. A **syntax tree** for this grammar consists of one of the following:

- 1) A single node  $x$  for an  $x \in T$ . This  $x$  is both root and leaf node.
- 2) An edge



corresponding to a production  $A \Rightarrow \alpha \in P$ .

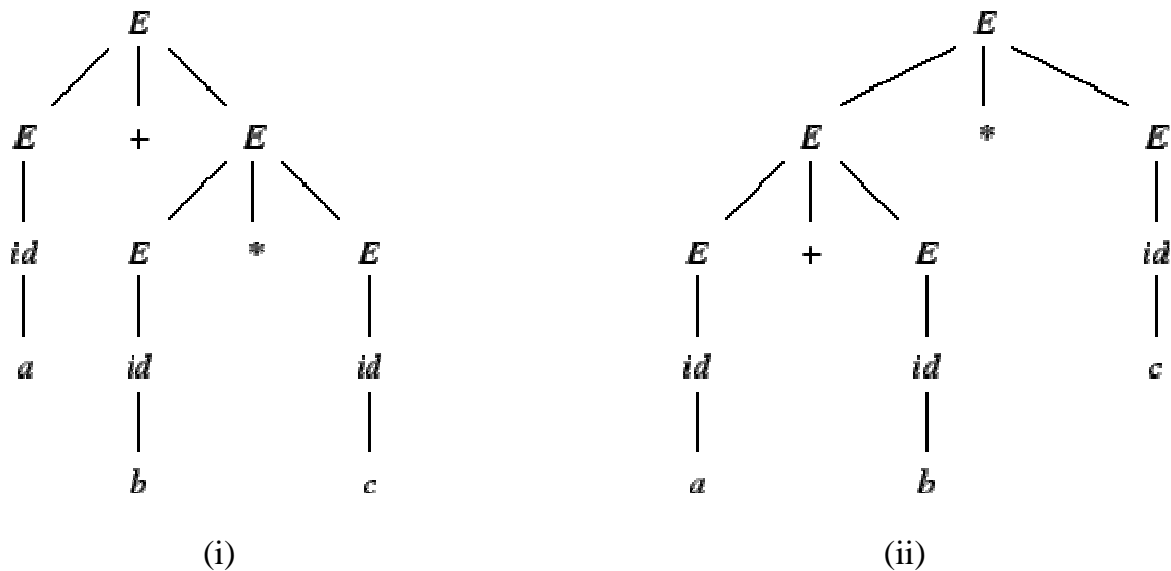
- 3) A tree



where the  $A_1, A_2, \dots, A_n$  are the root nodes of syntax trees. Their yields are read from left to right.

#### 3.1.2 Ambiguity

Until now the syntax trees were uniquely determined – even if the sequence of direct derivations were not. Separating the productions corresponding to the operator hierarchy, from weakest to strongest, in the expression grammar  $+, -, *, /, ()$  preserves this natural hierarchy. If this is not done, then syntax trees with a false evaluation sequence are often the result. Suppose, for instance, that the rules of the expression grammar were written  $E \Rightarrow E + E \mid E * E \mid id$ , then two *different* syntax trees as in Figure 2 below are the result. If the first production  $E \Rightarrow E + E$  were chosen then the result would be the tree in Figure 1(i).



**Figure 1: Syntax trees for string  $a + b * c$ .**

On the other hand, choosing the production  $E \Rightarrow E * E$  first results in a syntax tree of an entirely different ilk (Figure 1(ii)).

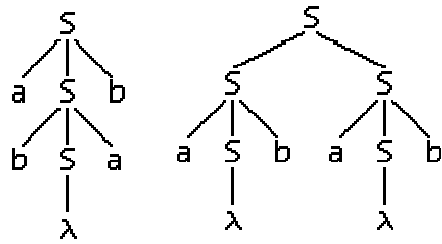
Thus this grammar is *ambiguous*, because it is possible to generate two different syntax trees for the expression  $a + b * c$ .

### Example 1:

The following grammar generates strings having an equal number of a's and b's.

$$G = (\{S\}, \{a, b\}, S, S \rightarrow aSb \mid bSa \mid SS \mid \lambda)$$

The string "abab" can be generated from this grammar in two distinct ways, as shown by the following derivation trees:



Similarly, abab has two distinct leftmost derivations:

$$S \Rightarrow aSb \Rightarrow abSb \Rightarrow abab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

Likewise, abab has two distinct rightmost derivations:

$$S \Rightarrow aSb \Rightarrow abSb \Rightarrow abab$$

$$S \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \Rightarrow abab$$

Each derivation tree can be turned into a unique rightmost derivation, or into a unique leftmost derivation. Each leftmost or rightmost derivation can be turned into a unique derivation tree. So these representations are largely interchangeable.

### 3.1.2.1 Ambiguous Grammars, Ambiguous Languages

Since derivation trees, leftmost derivations, and rightmost derivations are equivalent notations, the following definitions are equivalent:

A grammar  $G$  is *ambiguous* if there exists some string  $w \in L(G)$  for which

- there are two or more distinct derivation trees, or
- there are two or more distinct leftmost derivations, or
- there are two or more distinct rightmost derivations.

Grammars are used in compiler construction. Ambiguous grammars are undesirable because the derivation tree provides considerable information about the semantics of a program; conflicting derivation trees provide conflicting information.

Ambiguity is a property of a grammar, and it is usually (but not always) possible to find an equivalent unambiguous grammar.

An *inherently ambiguous language* is a language for which no unambiguous grammar exists.

### Self Assessment Exercise I

2. Describe in words the language generated by  $G$  below.

$$\begin{aligned}
S &\rightarrow SS \mid AB \mid AC \\
A &\rightarrow a \\
B &\rightarrow b \\
C &\rightarrow SB
\end{aligned}$$

b) Is  $G$  ambiguous?

## 3.2 Chomsky Normal Form

Work with a given context-free grammar is greatly facilitated by putting it into a so-called *normal form*. This provides some kind of regularity in the appearance of the right-hand sides of grammar rules. One of the most important normal forms is the *Chomsky normal form*.

### 3.2.1 Definition of Chomsky Normal Form

The context-free Grammar  $G = (X, T, P, S)$  is said to be in **Chomsky normal form** if all grammar rules have the form

$$A \Rightarrow a \mid BC, \tag{1}$$

for  $a \in T$  and  $B, C \in X - T$ . There is one exception. If  $\lambda \in L(G)$ , then the single extra rule

$$S \Rightarrow \lambda \tag{2}$$

is permitted. If  $\lambda \notin L(G)$  then production rule 2 is not allowed.

**Theorem 1:** Any context-free grammar  $G = (X, T, P, S)$  can be rewritten in Chomsky normal form.

**Proof:**

To facilitate rewriting the grammar rules it is first a good idea to eliminate unnecessary pathology in the original grammar. Although a CFG always defines its grammar rules with single nonterminal symbols on the left-hand side of every production, it can easily happen that there are nonterminal symbols that *never* appear on the left-hand side of any production. It is then seen that they cannot generate any sequence of terminal symbols and, moreover, may never appear in the right-hand side of any production to help produce a sentence in the associated language. It can even happen that there are nonterminal symbols that appear in no sentential form 1 derivable from the start symbol. These symbols are called *useless* and are to be

expurgated at the outset. It should be abundantly clear that no part of any rule containing one of these symbols has the least influence of  $L(G)$ . Consider, for example, the grammar

$$S \Rightarrow AB \mid CA \mid AD$$

$$B \Rightarrow BC \mid AB$$

$$A \Rightarrow aA \mid a$$

$$C \Rightarrow b \mid aB \mid bC$$

No terminal symbol is derivable from  $B$  and  $D$  never appears in the right-hand side of any rule. Thus, the grammar simplifies to:

$$S \Rightarrow CA$$

$$A \Rightarrow aA \mid a$$

$$C \Rightarrow b \mid bC$$

Now that the grammar has been stripped of extraneous elements, the grammatical transformation can begin. The rules for  $G$  can be rewritten as follows:

- Purge  $P$  of rules of the form  $A \Rightarrow \lambda$ . If there is another rule with  $A$  occupying the left-hand side, then proceed as follows. For every rule in which  $A$  appears on the right-hand side, add another rule to  $P$  without this occurrence of  $A$ . If  $A$  occurs more than once, then add rules with each individual occurrence of  $A$  elided, while retaining the other occurrences of that nonterminal symbol. Finally, add rules with pairs of individual occurrences of  $A$  eliminated, *etc.* until all combinations have been expunged. For example, the rules  $A \Rightarrow a \mid \lambda$  and  $A \Rightarrow ABAC$  could be replaced by  $A \Rightarrow a$  and  $A \Rightarrow BAC \mid ABC \mid BC \mid ABAC$ .
- Replace any rule of the form  $A \Rightarrow \alpha_1, \alpha_2, \dots, \alpha_n$ , by the  $n - 1$  rules  $A_1 \Rightarrow \alpha_1 A_2$ ,  $A_2 \Rightarrow \alpha_2 A_3, \dots, A_{n-1} \Rightarrow \alpha_{n-1} \alpha_n$ .
- Eliminate "useless" rules  $A \Rightarrow B$ , where  $A$  and  $B$  are nonterminal symbols. Indeed, if there is a rule  $B \Rightarrow^* \alpha$ , where  $\alpha$  consists of more than one symbol, then reduce to  $A \Rightarrow \alpha$ .

### 3.2.2 Putting a CFG into Chomsky Normal Form

Recall that a grammar  $G$  is in Chomsky Normal Form if each production in  $G$  is one of two forms:

1.  $A \rightarrow BC$  where  $A, B,$  and  $C$  are nonterminals, or
2.  $A \rightarrow a$  where  $a$  is a terminal.

We will further assume  $G$  has no useless symbols. Every context-free language without  $\varepsilon$  can be generated by a Chomsky Normal Form grammar.

Let us assume we have a CFG  $G$  with no useless symbols,  $\varepsilon$ -productions, or unit productions. We can transform  $G$  into an equivalent Chomsky Normal Form grammar as follows:

- Arrange that all bodies of length two or more consist only of nonterminals.
- Replace bodies of length three or more with a cascade of productions, each with a body of two nonterminals.

Applying these two transformations to the grammar  $H$  in 3.2.3 above, we get:

$$\begin{aligned} E &\rightarrow EA \mid TB \mid LC \mid a \\ A &\rightarrow PT \\ P &\rightarrow + \\ B &\rightarrow MF \\ M &\rightarrow * \\ L &\rightarrow ( \\ C &\rightarrow ER \\ R &\rightarrow ) \\ T &\rightarrow TB \mid LC \mid a \\ F &\rightarrow LC \mid a \end{aligned}$$

### Example 2:

Consider again the expression grammar  $G$ :

$$\begin{aligned} E &\Rightarrow E + T \mid E - T \mid T \\ T &\Rightarrow T * F \mid T / F \mid F \\ F &\Rightarrow (E) \mid id \\ id &\Rightarrow a \mid b \mid c \end{aligned}$$

Then the rule  $E \Rightarrow E + T \mid E - T$ , is replaced by  $E \Rightarrow EE^{\#}$ ,  $E^{\#} \Rightarrow POPT$ ,  $POP \Rightarrow + \mid -$ . Similar replacements hold for  $T \Rightarrow T * F \mid T / F$  and  $F \Rightarrow (E)$ . Finally the productions  $E \Rightarrow T$ ,  $T \Rightarrow F$  and  $F \Rightarrow id$  are reduced to  $E \Rightarrow a \mid b \mid c$ ,  $T \Rightarrow a \mid b \mid c$  and  $F \Rightarrow a \mid b \mid c$  respectively.

### Self Assessment Exercise 1:

1) Put the following grammar into Chomsky Normal Form:

$$\begin{array}{l} S \rightarrow ASB \mid \varepsilon \\ A \rightarrow aAS \mid a \\ B \rightarrow BbS \mid A \mid bb \\ C \rightarrow aB \mid b \end{array}$$

### 3.3 Non Context-Free Languages: Ogden's Lemma (The Pumping Lemma for CFL)

A *pumping lemma* is a theorem used to show that, if certain strings belong to a language, then certain other strings must also belong to the language. In this section we discuss a pumping lemma for context-free languages

As with finite automata there is a version of the pumping lemma that demonstrates certain languages are not context free. This theorem is often called Ogden's lemma after its discoverer.

**Theorem 2:** Let  $G = (X, T, R, S)$  be a context-free language. Then there is an integer  $n = n(G)$  for which every string  $x \in L(G)$  having a length  $|x|$  greater than  $n$  can be written

$$x = uvwyz, \tag{3}$$

and

1.  $vy \neq \lambda$  (that is,  $v \neq \lambda$  or  $y \neq \lambda$ ).
2. The length of  $vwyz$  satisfies  $|vwyz| \leq n$ .
3. For each integer  $k \geq 0$ , it follows that  $uv^kwy^kz \in L(G)$ .

**Proof:**

Assume that  $G$  is in Chomsky normal form. For  $x \in L(G)$  consider the (binary) syntax tree for the derivation of  $x$ . Assume the height of this tree is  $h$  as illustrated in Figure 2.

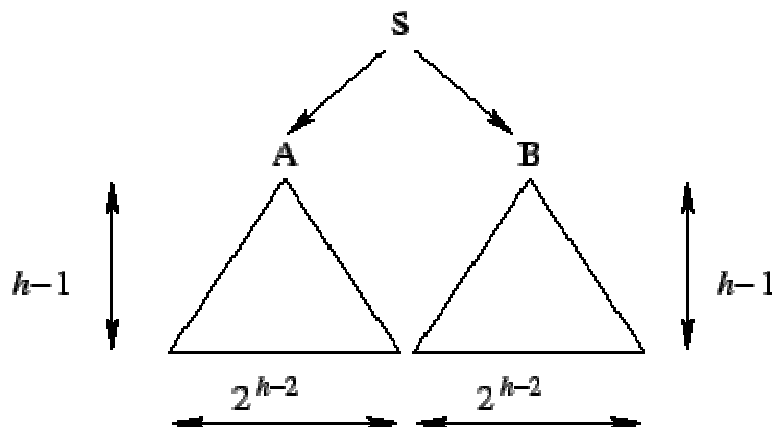


Figure 2: Derivation Tree for the string  $x \in VL(G)$

Then it follows that  $|x| \leq 2^{h-2} + 2^{h-2} = 2^{h-1}$ , *i.e.* the yield of the tree with height  $h$  is at most  $2^{h-1}$ . If  $G$  has  $k$  nonterminal symbols, let  $n = 2^k$ . Then let  $x \in VL(G)$  be a string with  $|x| \geq n$ . Thus the syntax tree for  $x$  has height at least  $k + 1$ , thus on the path from the root downwards that defines the height of the tree there are at least  $k + 2$  nodes, *i.e.* at least  $k + 1$  nonterminal symbols. It then follows that there is some nonterminal symbol  $A$  that appears at least twice. Consulting Figure 3, it is seen that the partial derivation  $S \Rightarrow^* uAz \Rightarrow^* uvAyz$  obtains.

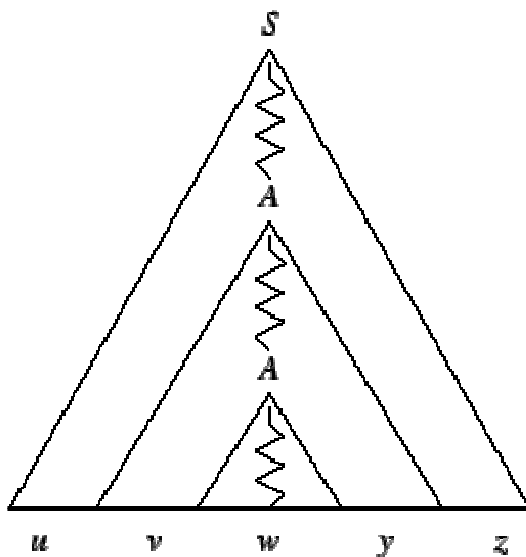


Figure 3: Nonterminal  $A$  appears twice in the derivation of  $x$

If, now, both  $u$  and  $z$  were empty, then derivations of the form  $S \Rightarrow uAz \Rightarrow A$  would be possible, contrary to the assumption of Chomsky normal form. For the same reason either  $v$  or  $y$  are nonempty. If  $|vwy| > n$  then apply the procedure anew until the condition  $|vwy| \leq n$  holds. Finally, since the derivation  $A \Rightarrow vAy$  can be repeated as often as one pleases, it follows that  $S \Rightarrow uAz \Rightarrow^* uvAyz \Rightarrow^* uv^2Ay^2z \Rightarrow^* uv^2wy^2zi$ , *etc.* can be generated. This completes the proof.

### 3.3.1 Using the Pumping Lemma

As mentioned earlier, the pumping lemma can be used to show that certain languages are *not* context free. As an example, we will show that the language  $L = \{a^i b^i c^i : i > 0\}$  is not context-free.

**Example 3:**

The language  $L = \{a^i b^i c^i \mid i \geq 1\}$  is not context free.

**Proof:**

Assume  $L$  were context-free. Then let  $n$  be the  $n$  from the preceding theorem and put  $x = a^n b^n c^n$ . Ogden's lemma then provides the decomposition  $x = uvwyz$  with the stated properties. There are several cases to consider.

**Case 1:** The string  $vy$  contains only a's. But then the string  $uwz \in L$ , which is impossible, because it contains fewer a's than b's and c's.

**Case 2,3:**  $vy$  contains only b's or c's. This case is similar to case 1.

**Case 4,5:**  $vy$  contains only a's and b's or only b's or c's. Then it follows that  $uwz$  contains more c's than a's and b's or more a's than b's and c's. This is again a contradiction.

Since  $|vwy| \leq n$  it is not possible that  $vy$  contain a's and c's.

**Example 4:**

In this example the power of Ogden's lemma will be extended. The language  $L = \{a^i b^j c^k \mid i < j < k\}$  is not context-free.

**Proof:**

The pumping properties of Ogden's lemma are of little use here. Hence we return to the syntax tree in the proof of the theorem. If, now,  $L$  were context-free then consider the path in its syntax tree from the root to the leaf node containing the rightmost a. For sufficiently large  $i$  there is likewise a nonterminal symbol  $A$  appearing twice on this path. Then  $x = ua^m wyz$  for some  $m \geq 0$ . There are 2 cases to be considered here.

**Case 1:**  $m = 0$ . Then the matching substring  $y$  cannot be empty. On the other hand the theorem guarantees that  $|wy| \leq i$ , hence all c's must be contained in  $z$ . But then  $y$  contains at least one b, so for sufficiently large  $n$  the string  $uw y^n z$  belongs to  $L$ , which is impossible, because for large enough  $n$ , this string will contain more b's than c's – contrary to assumption.

**Case 2:**  $m \geq 1$ . It then follows that

$$S \Rightarrow^* uAz \Rightarrow^* ua^m Ayz \Rightarrow^* ua^{2m} Ay^2 z \Rightarrow^* \dots \Rightarrow^* ua^{nm} wy^n z.$$

Now obviously  $y$  cannot contain more than one letter kind and this letter is either b or c. If  $y$  contained only b's then the string  $ua^{nm} wy^n z$  would contain more a's than c's

for  $n$  sufficiently large. As similar argument holds if  $y$  contained only  $c$ 's. In any case, a contradiction is derived and thus the assumption that  $L$  is context-free is false.

### 3.4 Closure Properties of Context Free Languages

Proceeding by analogy, one would expect the closure theorems for regular languages to generalize to CFLs. Surprisingly, not everything carries over. The following theorem, however, articulates a property of context-free languages analogous to finite automata.

**Theorem 3:** The context-free languages are closed under the formation of:

- Union
- Concatenation
- Kleene star.

**Proof:**

Let  $G_1 = (X_1, T_1, P_1, S_1)$  and  $G_2 = (X_2, T_2, P_2, S_2)$  be context-free grammars. Without loss of generality, it can be supposed that  $(X_1 - T_1) \cap (X_2 - T_2) = \emptyset$ . If not, then rewrite the grammar rules of one with new nonterminal symbols. Let  $S$  be another nonterminal symbol. Then set  $P = P_1 \cup P_2 \cup \{S \Rightarrow S_1, S \Rightarrow S_2\}$  and  $G = (X_1 \cup X_2 \cup \{S\}, T_1 \cup T_2, P, S)$ . It then follows that  $L(G) = L(G_1) \cup L(G_2)$ . Verifying the closure under concatenation and Kleene star in a similar manner is left to you as an exercise.

The closure under intersection property is not quite exact.

**Theorem 4:** The intersection of a context-free language  $L_1$  and a regular language  $L_2$  is context-free.

**Proof:**

If  $L_1$  is context-free then it is recognized by some pushdown automaton  $P = (X, Z_1, S_1, R_1, F_1)$ . If  $L_2$  is a regular language, then it is recognized by a *deterministic* automaton  $A = (X, Z_2, f, S_2, F_2)$ . Define a new PDA  $P' = (X, Z, S, R, S_A, F)$  as follows:  $Z = Z_1 \times Z_2$ ,  $S = S_1$ ,  $S_A = (S_1, S_2)$  and, finally,  $F = F_1 \times F_2$ . The transition relation  $R$  is obtained directly from the transition relation of  $P$  and the transition function of  $A$ , viz. for every transition of  $P$  of the form  $((a_1, z_1, S'), (z_1', S_1')) \in R_1$  and for each state  $z_2 \in Z_2$ , put

$$((a_1, (z_1, z_2), S_1^{\#}), ((z_1', f(a, z_2), S_1^{\#\#})) \in R$$

and for each  $\lambda$ -move of  $P$  of the form  $((\lambda, z_1, S_1^{\#}), (z_1^{\#}, S_1^{\#})) \in R$  and  $z_2 \in Z_2$  put  $((\lambda, (z_1, z_2), S_1^{\#}), ((z_1^{\#}, z_2), S_1^{\#})) \in R$ ,  
 or, stated in words,  $P'$  passes from state  $(z_1, z_2)$  into state  $(z_1^{\#}, z_2^{\#})$  if and only if  $P$  passes from  $z_1$  to  $z_2$  and  $A$  passes from  $z_2$  to  $z_2^{\#}$ , i.e.  $x \in L(P')$  if and only if  $x \in L(P) \cap L(A)$ .

**Theorem 5:** The class of context-free languages is not closed under intersection and complement.

**Proof:**

It is easy to see that the two languages

$$L_1 = \{a^n b^n c^m \mid m, n \geq 0\}$$

and

$$L_2 = \{a^m b^n c^n \mid m, n \geq 0\}$$

are context free. The intersection, however,  $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$  is not context-free. From the complement identity

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}},$$

it is seen that the complements  $\overline{L_1}$  and  $\overline{L_2}$  are not in general context-free.

### Self Assessment Exercise 2:

1. Show that  $\{a^n b^n c^n \mid n \geq 0\}$  is not context free.
2. Show that  $\{a^n b^n c^i \mid i \leq n\}$  is not context free.
3. Show that  $\{ss^R s \mid s \text{ is a string of } a\text{'s and } b\text{'s}\}$  is not context free.

### 3.5 Parsing

There are two ways to use a grammar:

- Use the grammar to *generate* strings of the language. This is easy -- start with the start symbol, and apply derivation steps until you get a string composed entirely of terminals.
- Use the grammar to *recognize* strings; that is, test whether they belong to the language. For CFGs, this is usually much harder.

A language is a set of strings, and any well-defined set must have a membership criterion. A context-free grammar can be used as a membership criterion – **if** we can find a general algorithm for using the grammar to recognize strings.

*Parsing* a string is finding a derivation (or a derivation tree) for that string.

Parsing a string is like recognizing a string. An algorithm to recognize a string will give us only a yes/no answer; an algorithm to parse a string will give us additional information about how the string can be formed from the grammar.

Generally speaking, the only realistic way to recognize a string of a context-free grammar is to parse it.

### 3.5.1 Exhaustive Search Parsing

The basic idea of *exhaustive search parsing* is this: to parse a string  $w$ , generate all strings in  $L$  and see if  $w$  is among them.

**Problem:**  $L$  may be an infinite language.

We need two things:

1. A systematic approach, so that we know we have not overlooked any strings, and
2. A way to stop after generating only a *finite* number of strings – knowing that, if we have not generated  $w$  by now, we never will.

Systematic approaches are easy to find. Almost any exhaustive search technique will do.

We can (almost) make the search finite by terminating every search path at the point that it generates a sentential form containing more than  $|w|$  terminals.

### 3.5.2 Grammars for Exhaustive Parsing

The idea of exhaustive search parsing for a string  $w$  is to generate all strings of length not greater than  $|w|$ , and see whether  $w$  is among them. To ensure that the search is finite, we need to make sure that we cannot get into an infinite loop applying productions that don't increase the length of the generated string.

Note: for the time being, we will ignore the possibility that  $\lambda$  is in the language.

Suppose we make the following restrictions on the grammar:

- Every variable expands to at least one terminal. We can enforce this by disallowing productions of the form  $A \rightarrow \lambda$ .
- Every production either has at least one terminal on its right hand side (thus directly increasing the number of terminals), or it has at least two variables (thus indirectly increasing the number of terminals). In other words, we disallow productions of the form  $A \rightarrow B$ , where  $A$  and  $B$  are both variables.

With these restrictions,

- A sentential form of length  $n$  yields a sentence of length at least  $n$ .
- Every derivation step increases either the length of the sentential form or the number of terminals in it.
- Hence, any string  $w \in L$  can be generated in at most  $2|w|-1$  derivation steps.
- We have shown that exhaustive search parsing is a finite process, provided that there are no productions of the form  $A \rightarrow \lambda$  or  $A \rightarrow B$  in the grammar. As discussed in section 3.2 such productions can be removed from a grammar without altering the language recognized by the grammar. There is, however, one special case we need to consider.
- If  $\lambda$  belongs to the language, we need to keep the production  $S \rightarrow \lambda$ . This creates a problem if  $S$  occurs on the right hand side of some production, because then we have a way of decreasing the length of a sentential form. All we need to do in this case is to add a new start symbol, say  $S_0$ , and to replace the production  $S \rightarrow \lambda$  with the pair of productions

$$\begin{array}{l} S_0 \rightarrow \lambda \\ S_0 \rightarrow S \end{array}$$

### 3.5.3 Efficient Parsing

Exhaustive search parsing is, of course, extremely inefficient. It requires time exponential in  $|w|$ .

For any context-free grammar  $G$ , there are algorithms for parsing strings  $w \in L(G)$  in time proportional to the cube of  $|w|$ . This is still unsatisfactory for practical purposes.

There are ways to further restrict context-free grammars so that strings may be parsed in linear or near-linear time. These restricted grammars are covered in courses in compiler construction, but will not be considered here. All such methods *do* reduce the power of the grammar, thus limiting the languages that can be recognized. There is no known linear or near-linear algorithm for parsing strings of a general context-free grammar.

## 4.0 CONCLUSION

In this unit you have been taken through some of the properties of CFL. It is expected that with the knowledge you have gained in this unit, you will be able to determine if a grammar is context-free or not and generate other strings that might belong to the grammar. In the next unit, you will be learning about the class of automata that recognises this class of grammars.

## 5.0 SUMMARY

In this unit, you learnt that:

- syntax trees are tree representations of derivations

- A grammar  $G$  is *ambiguous* if there exists some string  $w \in L(G)$  for which there are two or more distinct derivation trees
- An *inherently ambiguous language* is a language for which no unambiguous grammar exists
- The context-free languages are closed under the formation of union, concatenation, Kleene star.
- The class of context-free languages is not closed under intersection and complement
- *Parsing* a string is finding a derivation (or a derivation tree) for that string

## 6.0 TUTOR-MARKED ASSIGNMENT

- 1) State the pumping lemma for CFL
- b) With the aid of illustrative example, demonstrate how to use the pumping lemma to show that a certain grammar is not context-free
- 2) Briefly explain the concept of ambiguity in grammars
- 3) Construct a CFG that generates the language  $\{ a^n b^n \mid n \geq 0 \}$ .
- 4) Prove that the language generated by the grammar  $G$  below:

$$\begin{aligned}
 S &\Rightarrow S ( S ) \\
 &\Rightarrow S ( S ) ( S ) \\
 &\Rightarrow ( S ) ( S ) \\
 &\Rightarrow ( ) ( S ) \\
 &\Rightarrow ( ) ( )
 \end{aligned}$$

consists of all strings of balanced parentheses.

- 5) Construct a CFG that generates  $ELP = \{ ww^R \mid w \text{ is any string of } a\text{'s and } b\text{'s} \}$ . This is the language of even-length palindromes over the alphabet  $\{a, b\}$ . A palindrome is a string that reads the same in both directions.
- 6) Prove that  $ELP$  is not a regular language.
- 7) Construct a CFG for all regular expressions over the alphabet  $\{a, b\}$ .

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing..

- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 3: Context-Free Languages

### Unit 3: Pushdown Automata

#### CONTENTS

2.0	<i>Introduction</i>
3.0	<i>Objectives</i>
4.0	<i>Main Content</i>
3.1	Pushdown Automata
3.2.1	Nondeterministic Pushdown Automata (NPDA)
3.2.1.1	Transition Functions for NPDAs
3.2.1.2	Drawing NPDAs
3.2.1.3	NPDA Execution
3.2.1.4	Accepting Strings with an NPDA
3.2.1.5	Accepting Strings with an NPDA (Formal Version)
3.2.2	Deterministic Pushdown Automata
3.2.2.1	From a CFG to an equivalent PDA
3.2.2.2	From a PDA to an equivalent CFG
5.0	<i>Conclusion</i>
6.0	<i>Summary</i>
7.0	<i>Tutor-Marked Assignment</i>
8.0	References/Further Reading

## 2.0 INTRODUCTION

Having learnt about context-free languages and their properties in the previous units of this module, in this unit you will be studying the machine that accepts context-free languages, the *pushdown automaton* or PDA.

The finite-state automaton (FSA) and the pushdown automaton (PDA) enjoy a special place in computer science. The FSA has proven to be a very useful model for many practical tasks and deserves to be among the tools of every practicing computer scientist. Many simple tasks, such as interpreting the commands typed into a keyboard or running a calculator, can be modelled by finite-state machines. The PDA is a model to which one appeals when writing compilers because it captures the essential architectural features needed to parse context-free languages, languages whose structure most closely resembles that of many programming languages.

A DFA (or NFA) is not powerful enough to recognize many context-free languages because a DFA cannot count. But counting is not enough. Consider a language of

palindromes, containing strings of the form  $ww^R$ . Such a language requires more than an ability to count; it requires a stack.

A *pushdown automaton (PDA)* is basically an NFA with a stack added to it.

Now let us go through your study objectives for this unit.

## 2.0 OBJECTIVES

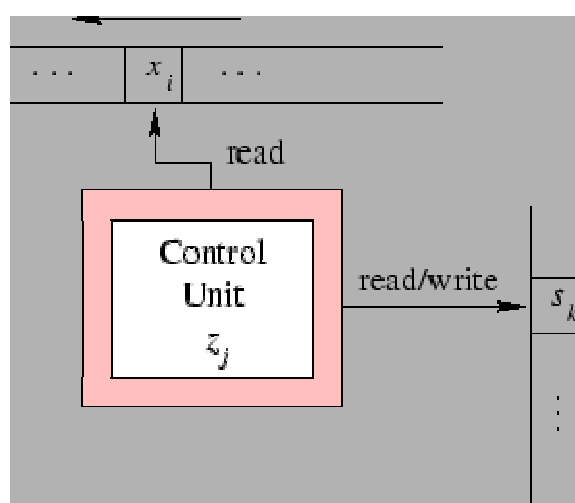
At the end of this unit, you should be able to:

- Describe a pushdown automata
- Distinguish PDAs from FSAs
- Formally define a PDA
- Compare a DPDA and an NPDA

## 3.0 MAIN CONTENT

### 3.1 Pushdown Automata

This machine is fed input just as a finite automaton is. Some texts speak of the input coming in on a read-only tape with a tape head that moves left to right until it comes to the end of the input. At that point it reads a special character that marks a blank tape cell. We will use the character  $\Delta$  or  $\lambda$  as a "blank". When the tape head reads a blank the machine halts, or begins the process of halting (which will be explained later.) The tape head may not reverse directions, nor may it be used to write to the tape. The input tape is infinitely long in the rightward direction, which allows a PDA to accept a finite input of any length. The cells of the tape are numbered, with the first input character occurring in cell 1 where the tape head is set initially.

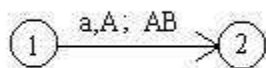


**Figure 1:** Conceptual Model of a Pushdown Automaton

In addition to the input tape, a PDA has an associated stack onto which it can push characters to remember them. This stack has no limit to its size so the PDA can push as many characters as it likes. The machine begins processing with an empty stack. Typically the first thing the machine does is push a "bottom-of-the-stack marker" onto the stack. We shall use the  $\Delta$  as that marker. Note that a PDA has two associated alphabets, one containing characters that may appear on the input tape, the other containing characters that may be pushed onto the stack. The two alphabets may be the same but they do not have to be. We usually name the input alphabet  $\Sigma$  and the stack alphabet  $\Gamma$ .

Basically, the input tape consists of a linear configuration of cells each of which contains a character from an alphabet. This tape can be moved one cell at a time to the left. The stack is also a sequential structure that has a first element and grows in either direction from the other end. Contrary to the tape head associated with the input tape, the head positioned over the current stack element can read and write special stack characters from that position. The current stack element is always the top element of the stack, hence the name "stack". The control unit contains both tape heads and finds itself at any moment in a particular state.

Initially, we will not draw PDAs the way they are drawn in our textbook (but later we will). Instead we will draw them much like finite automata, except for the labels we use on the transitions. Each label will consist of three parts: the input character, the character popped off of the top of the stack, and the characters that need to be pushed onto the stack. For example, suppose we find the following transition in a PDA:

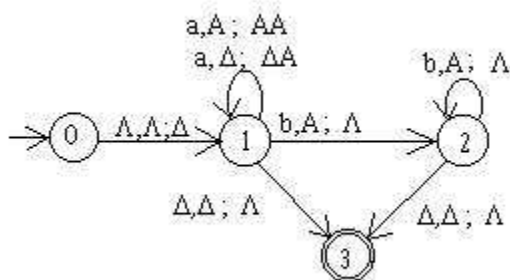


**Figure 2: Transition in a PDA**

This transition says that if we are in state 1 and there is an **a** in the current cell of the input tape and we pop an **A** off the top of the stack, then we may go to state 2 and must push the string **AB** onto the stack, pushing the **A** first and then the **B**. We may use a  $\Delta$  in any of the three parts of the transition label. It always means that we do not do the task that part of the label involves. In other words, a  $\Delta$  in place of an input character means that we do not read from the tape. A  $\Delta$  in place of the character popped off the stack means that we do not pop anything off the stack, and a  $\Delta$  in place of the string to be pushed means that we do not push anything. We will discuss when these  $\Delta$ 's make the machine nondeterministic later.

Here is a machine that accepts the language  $\{a^n b^n \mid n \geq 0\}$ . The machine begins with its input on the input tape and an empty stack. The first thing the machine does is go from

state 0 to state 1, pushing the bottom-of-the-stack marker onto the stack. In state 1, if the machine reads an **a** from the input and pops the blank off the stack, then that is the first **a** found in the input and the machine pushes the blank back onto the stack followed by an **A**. The machine counts the **a**'s in the input by pushing an **A** onto the stack for each **a** that it reads off the tape. From this point on, if the PDA is in state 1 and it finds an **a** in the input, there will be an **A** to pop off the stack. The machine then pushes two **A**'s back on the stack, one to make up for the **A** that was popped and one to count the **a** just found in the input. As soon as the machine sees a **b** in the input it changes to state 2 and pops an **A** off the stack. It continues popping an **A** for each **b** that it finds. If the input was a correctly formatted string, the machine will read a blank off the input tape at the same time that it pops a blank off the stack and go to state 3, an accept state. Since the language includes the empty string, we also have a transition from state 1 to state 3 that is used if the machine reads a blank from the input at the same time as it pops the marker off the stack at the very beginning of processing.



**Figure 3: Machine that accepts the language  $\{a^n b^n \mid n \geq 0\}$**

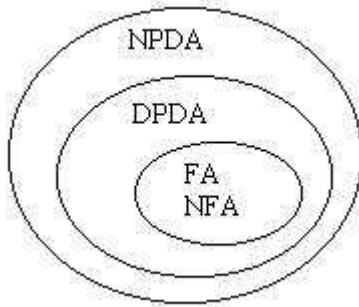
The previous machine shows that PDAs have more power than FSAs because that machine accepts a nonregular language, something that an FSA cannot do.

### 3.2 Types of PDAs

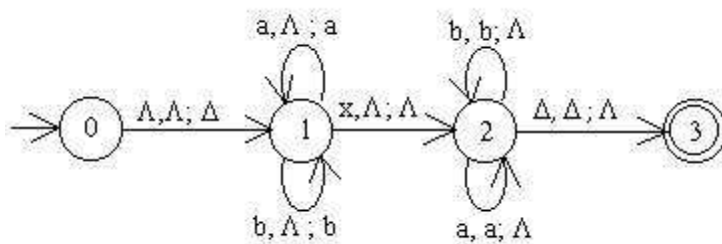
Like FSA, there are two types of PDAs:

- A *deterministic PDA* (DPDA) is one in which every input string has a unique path through the machine.
- A *nondeterministic PDA* (NPDA) is one in which we may have to choose among several paths for an input string.

We say that an input string is accepted if there is at least one path that leads to an accept state. We shall see that a nondeterministic PDA is more powerful than a deterministic one, unlike the situation with DFAs and NFAs. In other words, there are languages that can be accepted by an NPDA that cannot be accepted by a DPDA. We have this arrangement among languages accepted by the machines we have studied:



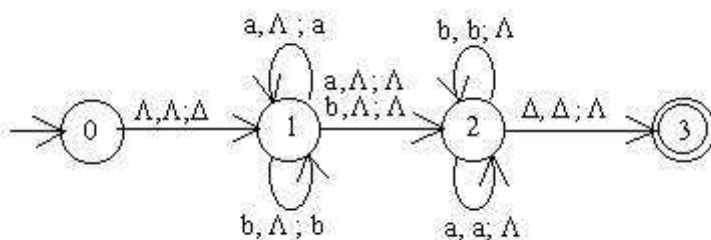
Here is a PDA that accepts the language PALINDROMEX over the alphabet  $\{a,b,x\}$ .  $PALINDROMEX = \{sxs^R\}$  where  $s$  is a string over  $\{a,b\}$  and  $s^R$  is the reverse of  $s$ .



**Figure 4: PDA that accepts the language  $\{sxs^R\}$  over the alphabet  $\{a,b,x\}$ .**

Note that the lambdas in the above machine do not make the machine nondeterministic. There is only one path through the machine for any string, although there is an implied trap state in the machine and a string's path may take it to that implied trap state. For example, the string  $abxab$  will cause the machine to enter the trap state when it reads the second  $a$  and pops a  $b$  off the stack. Some authors would say that the machine "crashes" when such an event occurs, but regardless of how we describe the situation, it results in the same thing, nonacceptance of the string.

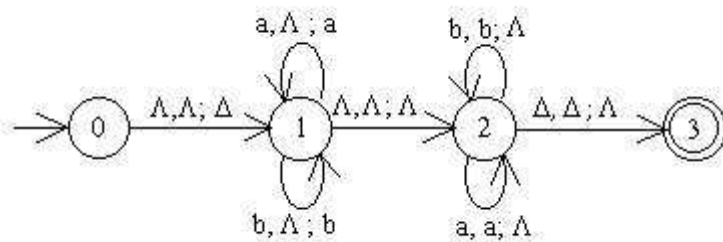
The  $x$ 's in the strings of  $PALINDROMEX$  are essential to our ability to recognize the language with a deterministic machine. Without the  $x$  we would not know when to change states. Consider the language  $ODDPALINDROME$  which contains all odd length palindromes over  $\{a,b\}$ . By changing the label  $(x, \Lambda; \Lambda)$  to the two labels  $(a, \Lambda; \Lambda)$  and  $(b, \Lambda; \Lambda)$ , we have an NPDA that recognizes  $ODDPALINDROME$ .



**Figure 5: NPDA that recognizes ODDPALINDROME**

This machine is nondeterministic because from state 1, when there is an **a** in the input, the machine can either stay in state 1 and not pop the stack or it can go to state 2 and not pop the stack. Similarly, the machine has two choices if it reads a **b** and it is in state 1.

Now consider a machine for  $\text{EVENPALINDROME} = \{ss^R\}$ . There is no middle character in an even-length palindrome, so the label on the transition from the first state to the second is labeled with  $(\Lambda, \Lambda; \Lambda)$ . This is like a  $\Lambda$ -transition in an FA, in that the machine does not read any input or pop the stack when it takes this transition.

**Figure 6: NPDA that recognizes EVENPALINDROME  $\{ss^R\}$** 

Formally, a PDA is defined as a collection of seven things:

- an alphabet  $\Sigma$  of input letters
- an input tape containing an input string followed by  $\Delta$
- an alphabet  $\Gamma$  of stack letters
- a pushdown stack, initially empty
- one start state
- a set of accept states
- a *transition function*

From now on, the default type of PDA is a nondeterministic PDA, so the acronym PDA implies that the machine may be nondeterministic. If we want to say that the machine definitely is nondeterministic we will affix the N and label it an NPDA.

Here is a theorem whose proof should seem rather obvious to you.

**Theorem 1:** For any regular language there is a DPDA that accepts it.

**Proof:** A finite state machine is simply a PDA that does not make use of its stack. Given a regular language we can create a DPDA to accept it as follows: First create a DFA for the regular language, then change its transition labels so that instead of

simply an input character, each transition is labelled with an input character and two lambdas in place of the pop and push characters. **QED**

You should note that henceforth in this course, PDAs will be drawn as they are drawn in the texts.

### 3.2.1 Nondeterministic Pushdown Automata (NPDA)

As stated earlier, a *nondeterministic pushdown automaton (NPDA)* is basically an NFA with a stack added to it.

We therefore start the formal definition of NPDA with the formal definition of an NFA, which is a 5-tuple, and add two things to it:

- $\Gamma$  is a finite set of symbols called the *stack alphabet*, and
- $z \in \Gamma$  is the *stack start symbol*.

We also need to modify  $\delta$ , the transition function, so that it manipulates the stack.

A *nondeterministic pushdown automaton* or *NPDA* is, therefore, a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

where

- $Q$  is a finite set of *states*,
- $\Sigma$  is a the *input alphabet*,
- $\Gamma$  is the *stack alphabet*,
- $\delta$  is a *transition function*,
- $q_0 \in Q$  is the *initial state*,
- $z \in \Gamma$  is the *stack start symbol*, and
- $F \subseteq Q$  is a set of *final states*.

#### 3.2.1.1 Transition Functions for NPDAs

The transition function for an NPDA has the form  
 $\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$

$\delta$  is now a function of three arguments. The first two are the same as before: the state, and either  $\lambda$  or a symbol from the input alphabet. The third argument is *the symbol on top of the stack*. Just as the input symbol is "consumed" when the function is applied, the stack symbol is also "consumed" (removed from the stack).

Note that while the second argument may be  $\lambda$  rather than a member of the input alphabet (so that no input symbol is consumed), there is no such option for the third argument.  $\delta$  always consumes a symbol from the stack; no move is possible if the stack is empty.

In the deterministic case, when the function  $\delta$  is applied, the automaton moves to a new state  $q \in Q$  and pushes a new string of symbols  $x \in \Gamma^*$  onto the stack. Since we are dealing with a nondeterministic pushdown automaton, the result of applying  $\delta$  is a finite set of  $(q, x)$  pairs. If we were to draw the automaton, each such pair would be represented by a single arc.

As with an NFA, we do not need to specify  $\delta$  for every possible combination of arguments. For any case where  $\delta$  is not specified, the transition is to  $\emptyset \subseteq Q$ , the empty set of states.

### 3.2.1.2 Drawing NPDAs

NPDAs are not usually drawn. However, with a few minor extensions, we can draw an NPDA similar to the way we draw an NFA.

Instead of labelling an arc with an element of  $\Sigma$ , we can label arcs with  $a/x,y$  where  $a \in \Sigma$ ,  $x \in \Gamma$ , and  $y \in \Gamma^*$ .

#### Example 1:

Consider the following NPDA

$$(Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, \Gamma = \{0, 1\}, \delta, q_0, z = 0, F = \{q_3\})$$

Where

$$\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}$$

$$\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}$$

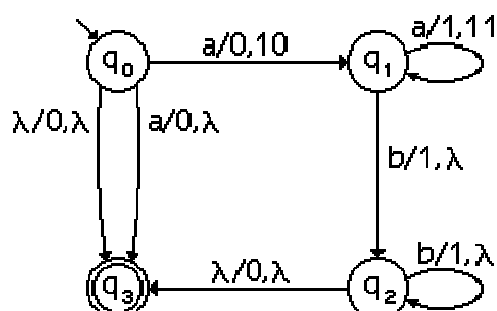
$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$

$$\delta(q_1, b, 1) = \{(q_2, \lambda)\}$$

$$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$$

$$\delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}$$

This NPDA can be drawn as



Note: the top of the stack is considered to be *to the left*, so that, for example, if we get an **a** from the starting position, the stack changes from  $\underline{\quad 0}$  to  $\underline{1 0}$ .

### 3.2.1.3 NPDA Execution

Suppose someone is in the middle of stepping through a string with a DFA, and we need to take over and finish the job. We will need to know two things:

- (1) the state the DFA is in, and
- (2) what the remaining input is.

But if the automaton is an NPDA instead of a DFA, we also need to know

- (3) the contents of the stack.

An *instantaneous description* of a pushdown automaton is a triplet  $(q, w, u)$ , where

- $q$  is the current state of the automaton,
- $w$  is the unread part of the input string, and
- $u$  is the stack contents (written as a string, with the leftmost symbol at the top of the stack).

Let the symbol " $\vdash$ " indicate a move of the NPDA, and suppose that  $\delta(q_1, a, x) = \{(q_2, y), \dots\}$ . Then the following move is possible:

$$(q_1, aW, xZ) \vdash (q_2, W, yZ)$$

where  $W$  indicates the rest of the string following the  $a$ , and  $Z$  indicates the rest of the stack contents underneath the  $x$ . This notation says that in moving from state  $q_1$  to state  $q_2$ , an  $a$  is consumed from the input string  $aW$ , and the  $x$  at the top (left) of the stack  $xZ$  is replaced with  $y$ , leaving  $yZ$  on the stack.

### 3.2.1.4 Accepting Strings with an NPDA

Suppose you have the NPDA  $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ . How do you use this NPDA to recognize strings?

To recognize string  $w$ , begin with the instantaneous description

$$(q_0, w, z)$$

where

- $q_0$  is the start state,
- $w$  is the entire string to be processed, and

- $z$  is the start stack symbol.

Starting with this instantaneous description, make zero or more moves, just as you would with an NFA. There are two kinds of moves that you can make:

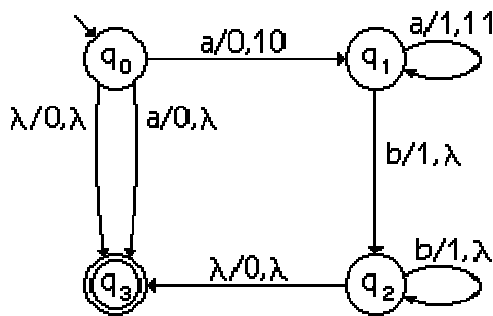
- $\lambda$ -transitions. If you are in state  $q_1$ ,  $x$  is the top (leftmost) symbol in the stack, and  $\delta(q_1, \lambda, x) = \{(q_2, w_2), \dots\}$ , then you can replace the symbol  $x$  with the string  $w_2$  and move to state  $q_2$ .
- Nonempty transitions. If you are in state  $q_1$ ,  $a$  is the next unconsumed input symbol,  $x$  is the top (leftmost) symbol in the stack, and  $\delta(q_1, a, x) = \{(q_2, w_2), \dots\}$ , then you can remove the  $a$  from the input string, replace the symbol  $x$  with the string  $w_2$ , and move to state  $q_2$ .

If you are in a final state when you reach the end of the string (and maybe make some  $\lambda$  transitions after reaching the end), then the string is accepted by the NPDA. It does not matter what is on the stack.

As usual with nondeterministic machines, the string is accepted if there is any way it could be accepted. If we take the "oracle" viewpoint, then every time we have to make a choice, we magically always make the right choice, so we will end in a final state if at all possible.

### Example 2: (NPDA Execution)

Consider the following NPDA:



$$\begin{aligned} \delta(q_0, a, 0) &= \{(q_1, 10), (q_3, \lambda)\} \\ \delta(q_0, \lambda, 0) &= \{(q_3, \lambda)\} \\ \delta(q_1, a, 1) &= \{(q_1, 11)\} \\ \delta(q_1, b, 1) &= \{(q_2, \lambda)\} \\ \delta(q_2, b, 1) &= \{(q_2, \lambda)\} \\ \delta(q_2, \lambda, 0) &= \{(q_3, \lambda)\} \end{aligned}$$

We can recognize the string **aaabbb** by the following sequence of moves:

$(q_0, aaabbb, 0)$   
 $\vdash (q_1, aabbb, 10)$   
 $\vdash (q_1, abbb, 110)$   
 $\vdash (q_1, bbb, 1110)$   
 $\vdash (q_2, bb, 110)$   
 $\vdash (q_2, b, 10)$   
 $\vdash (q_2, \lambda, 0)$   
 $\vdash (q_3, \lambda, \lambda).$

Since  $q_3 \in F$ , the string is accepted.

### 3.2.1.5 Accepting Strings with an NPDA (Formal Version)

We have the notation " $\vdash$ " to indicate a single move of an NPDA. We will also use " $\vdash^*$ " to indicate a sequence of zero or more moves, and we will use " $\vdash^+$ " to indicate a sequence of one or more moves.

If  $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$  is an NPDA, then the language accepted by  $M$ ,  $L(M)$ , is given by

$$L(M) = \{w \in \Sigma^*: (q_0, w, z) \vdash^*(p, \lambda, u), p \in F, u \in \Gamma^*\}.$$

You should understand this notation.

### 3.2.2 Deterministic Pushdown Automata

A nondeterministic finite acceptor differs from a deterministic finite acceptor in two ways:

- The transition function  $\delta$  is single-valued for a DFA, multi-valued for an NFA.
- An NFA may have  $\lambda$ -transitions.

A nondeterministic pushdown automaton differs from a *deterministic pushdown automaton (DPDA)* in *almost* the same ways:

- The transition function  $\delta$  is *at most* single-valued for a DPDA, multi-valued for an NPDA.

Formally:  $|\delta(q, a, b)| = 0$  or  $1$ ,  
for every  $q \in Q$ ,  $a \in \Sigma \cup \{\lambda\}$ , and  $b \in \Gamma$ .

- Both NPDAs and DPDAs may have  $\lambda$ -transitions; but a DPDA may have a  $\lambda$ -transition only if no other transition is possible.

Formally: If  $\delta(q, \lambda, b) \neq \emptyset$ ,  
then  $\delta(q, c, b) = \emptyset$  for every  $c \in \Sigma$ .

A *deterministic context-free language* is a language that can be recognized by a DPDA.

The deterministic context-free languages are a proper subset of the context-free languages.

### 3.2.2.1 From a CFG to an equivalent PDA

Given a CFG  $G$ , we can construct a PDA  $P$  such that  $N(P) = L(G)$ . The PDA will simulate leftmost derivations of  $G$ .

#### Algorithm to construct a PDA for a CFG

Input: a CFG  $G = (V, T, Q, S)$ .

Output: a PDA  $P$  such that  $N(P) = L(G)$ .

Method: Let  $P = (\{q\}, T, V \cup T, \delta, q, S)$  where:

1.  $\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is in } Q\}$  for each nonterminal  $A$  in  $V$ .
2.  $\delta(q, a, a) = \{(q, \varepsilon)\}$  for each terminal  $a$  in  $T$ .

For a given input string  $w$ , the PDA simulates a leftmost derivation for  $w$  in  $G$ .

We can prove that  $N(P) = L(G)$  by showing that  $w$  is in  $N(P)$  iff  $w$  is in  $L(G)$ :

- If part: If  $w$  is in  $L(G)$ , then there is a leftmost derivation
- $S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n = w$   
We show by induction on  $i$  that  $P$  simulates this leftmost derivation by the sequence of moves  
 $(q, w, S) \vdash^* (q, \gamma_i, \alpha_i)$   
such that if  $\gamma_i = x_i \alpha_i$ , then  $x_i \gamma_i = w$ .
- Only-if part: If  $(q, x, A) \vdash^* (q, \varepsilon, \varepsilon)$ , then  $A \Rightarrow^* x$ .

We can prove this statement by induction on the number of moves made by  $P$ .

### 3.2.2.2 From a PDA to an equivalent CFG

Given a PDA  $P$ , we can construct a CFG  $G$  such that  $L(G) = N(P)$ . The basic idea of the proof is to generate the strings that cause  $P$  to go from state  $q$  to state  $p$ , popping a symbol  $X$  off the stack, by a nonterminal of the form  $[qXp]$ .

#### Algorithm to construct a CFG for a PDA

Input: a PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ .

Output: a CFG  $G = (V, \Sigma, R, S)$  such that  $L(G) = N(P)$ .

Method:

1. Let the nonterminal  $S$  be the start symbol of  $G$ . The other nonterminals in  $V$  will be symbols of the form  $[pXq]$  where  $p$  and  $q$  are states in  $Q$ , and  $X$  is a stack symbol in  $\Gamma$ .
2. The set of productions  $R$  is constructed as follows:
  - For all states  $p$ ,  $R$  has the production  $S \rightarrow [q_0Z_0p]$ .
  - If  $\delta(q, a, X)$  contains  $(r, Y_1Y_2 \dots Y_k)$ , then  $R$  has the productions

$$[qXr_k] \rightarrow a[rY_1r_1] [r_1Y_2r_2] \dots [r_{k-1}Y_kr_k]$$

for all lists of states  $r_1, r_2, \dots, r_k$ .

We can prove that  $[qXp] \Rightarrow^* w$  iff  $(q, w, X) \vdash^* (p, \varepsilon, \varepsilon)$ .

From this, we have  $[q_0Z_0p] \Rightarrow^* w$  iff  $(q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon)$ , so we can conclude  $L(G) = N(P)$ .

### Self Assessment Exercise II

- 1) Construct a PDA  $P$  from  $G$  below such that  $N(P) = L(G)$ . Show how your PDA accepts  $ababab$ .

$$\begin{array}{l} G: \quad S \rightarrow SS \mid AB \mid AC \\ \quad A \rightarrow a \\ \quad B \rightarrow b \\ \quad C \rightarrow SB \end{array}$$

- 2) Let  $L$  be the set of palindromes over  $\{a, b\}$  containing an equal number  $a$ 's and  $b$ 's. Is  $L$  context free? If yes, give a CFG for  $L$ . If no, prove  $L$  is not context free.

## 4.0 CONCLUSION

In this unit you have been taken through PDAs, the class of automata that recognises context-free languages, the different types and how these types differ. In the next unit, which is the concluding unit of this module, you will be learning more about CFGs and NPDAs.

## 5.0 SUMMARY

In this unit, you learnt that:

- PDAs have more power than FSAs
- a *PDA* is basically an NFA with a stack added to it
- An *instantaneous description* of a pushdown automaton is a triplet
- there are two types of PDAs

- A *deterministic PDA* (DPDA) is one in which every input string has a unique path through the machine.
- A *nondeterministic PDA* (NPDA) is one in which we may have to choose among several paths for an input string.
- an input string is accepted if there is at least one path that leads to an accept state
- A nondeterministic finite acceptor differs from a deterministic finite acceptor in two ways
- A *deterministic context-free language* is a language that can be recognized by a DPDA.
- The deterministic context-free languages are a proper subset of the context-free languages

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Formally define a PDA.
2. Briefly describe the operations and features of a PDA. How is it different from an FSA?
3. Distinguish between DPDA and NPDA. Which is more powerful
4. Construct a PDA that accepts  $\{ w cw^R \mid w \text{ is any string of a's and b's } \}$  by final state.
5. Construct a PDA that accepts  $\{ w cw^R \mid w \text{ is any string of a's and b's } \}$  by empty stack.
6. Construct a PDA that accepts  $\{ ww^R \mid w \text{ is any string of a's and b's } \}$  by final state.
7. Construct a PDA that accepts  $\{ ww^R \mid w \text{ is any string of a's and b's } \}$  by empty stack.
8. Construct a PDA  $P$  such that  $N(P) = L(G)$  where  $G$  is  $S \rightarrow (S)S \mid \epsilon$ .

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education..
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**Module 3: Context-Free Languages****Unit 4: CFGs and PDAs***CONTENTS*

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Simplifying Context-Free Grammars
3.1.1	Eliminating $\epsilon$ -productions from a CFG (Empty Production Removal)
3.1.2	Eliminating Useless Symbols from a CFG
3.1.3	Eliminating Unit Productions from a CFG
3.1.4	Left Recursion Removal
3.2	Normal Forms of Context-Free Grammars
3.2.1	Chomsky Normal Form
3.2.2	Greibach Normal Form
3.3	From CFG to NPDA
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

**3.0 INTRODUCTION**

Having learnt about CFGs and PDAs in the preceding units of this module, it is now time for you to study the relationship between CFGs and PDAs and ways of simplifying CFGs.

Now let us go through your study objectives for this unit.

**4.0 OBJECTIVES**

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

- Describe ways of simplifying CFGs
- Describe the various normal forms for CFGs and the advantages of each
- Describe how to convert CFGs to NPDAs

**3.0 MAIN CONTENT****3.1 Simplifying Context-Free Grammars**

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammar.

Before we discuss Chomsky Normal Form, let us discuss some useful operations that may be carried out on CFGs that you will find useful in putting CFGs in Chomsky normal form:

### 3.1.1 Eliminating $\epsilon$ -productions from a CFG (Empty Production Removal)

If the empty string does not belong to a language, then there is a way to eliminate productions of the form  $A \rightarrow \lambda$  from the grammar.

If the empty string *does* belong to a language, then we can eliminate from all productions save for the single production  $S \rightarrow \lambda$ . In this case we can also eliminate any occurrences of S from the right-hand-side of productions.

In other words, if a language L has a CFG, then  $L - \{ \epsilon \}$  has a CFG without any  $\epsilon$ -productions. A nonterminal A in a grammar is *nullable* if  $A \Rightarrow^* \epsilon$ .

The nullable nonterminals can be determined iteratively. We can eliminate all  $\epsilon$ -productions in a grammar as follows:

- Eliminate all productions with  $\epsilon$  bodies.
- Suppose  $A \rightarrow X_1 X_2 \dots X_k$  is a production and  $m$  of the  $k$   $X_i$ 's are nullable. Then add the  $2^m$  versions of this production where the nullable  $X_i$ 's are present or absent. (But if all symbols are nullable, do not add an  $\epsilon$ -production.)

#### Example 1:

Let us eliminate the  $\epsilon$ -productions from the grammar G below:

$$\begin{array}{l} S \rightarrow AB \\ A \rightarrow aAA \mid \epsilon \\ B \rightarrow bBB \mid \epsilon \end{array}$$

S, A and B are nullable.

For the production  $S \rightarrow AB$  we add the productions  $S \rightarrow A \mid B$

For the production  $A \rightarrow aAA$  we add the productions  $A \rightarrow aA \mid a$

For the production  $B \rightarrow bBB$  we add the productions  $B \rightarrow bB \mid b$

The resulting grammar H with no  $\epsilon$ -productions is

$$\begin{array}{l} S \rightarrow AB \mid A \mid B \\ A \rightarrow aAA \mid aA \mid a \\ B \rightarrow bBB \mid bB \mid b \end{array}$$

We can prove that  $L(H) = L(G) - \{ \epsilon \}$ .

### 3.1.2 Eliminating Useless Symbols from a CFG

A symbol  $X$  is *useful* for a CFG if there is a derivation of the form  $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$  for some string of terminals  $w$ . If  $X$  is not useful, then we say  $X$  is *useless*. To be useful, a symbol  $X$  needs to be:

1. *generating*; that is,  $X$  needs to be able to derive some string of terminals.
2. *reachable*; that is, there needs to be a derivation of the form  $S \Rightarrow^* \alpha X \beta$  where  $\alpha$  and  $\beta$  are strings of nonterminals and terminals.

To eliminate useless symbols from a grammar, we

- i. identify the nongenerating symbols and eliminate all productions containing one or more of these symbols, and then
- ii. eliminate all productions containing symbols that are not reachable from the start symbol.

#### Example 2:

In the grammar below,

$$\begin{aligned} S &\rightarrow AB \mid a \\ A &\rightarrow b \end{aligned}$$

$S$ ,  $A$ ,  $a$ , and  $b$  are generating.  $B$  is not generating. Eliminating the productions containing the nongenerating symbols we get:

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow b \end{aligned}$$

Now we see  $B$  is not reachable from  $S$ , so we can eliminate the second production to get

$$S \rightarrow a$$

The generating symbols can be computed inductively bottom-up from the set of terminal symbols.

The reachable symbols can be computed inductively starting from  $S$ .

### 3.1.3 Eliminating Unit Productions from a CFG

We can eliminate productions of the form  $A \rightarrow B$  from a context-free grammar.

A *unit* production is one of the form  $A \rightarrow B$  where both  $A$  and  $B$  are nonterminals.

Let us assume we are given a grammar  $G$  with no  $\epsilon$ -productions. From  $G$  we can create an equivalent grammar  $H$  with no unit productions as follows:

- Define  $(A, B)$  to be a unit pair if  $A \Rightarrow^* B$  in  $G$ .
- We can inductively construct all unit pairs for  $G$ .
- For each unit pair  $(A, B)$  in  $G$ , we add to  $H$  the productions  $A \rightarrow \alpha$  where  $B \rightarrow \alpha$  is a nonunit production of  $G$ .

### Example 3:

Consider the standard grammar  $G$  for arithmetic expressions:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow ( E ) \mid a \end{array}$$

The unit pairs are  $(E, E)$ ,  $(E, T)$ ,  $(E, F)$ ,  $(T, T)$ ,  $(T, F)$ ,  $(F, F)$ .

The equivalent grammar  $H$  with no unit productions is:

$$\begin{array}{l} E \rightarrow E + T \mid T * F \mid ( E ) \mid a \\ T \rightarrow T * F \mid ( E ) \mid a \\ F \rightarrow ( E ) \mid a \end{array}$$

### 3.1.4 Left Recursion Removal

A variable  $A$  is *left-recursive* if it occurs in a production of the form:

$$A \rightarrow Ax$$

for any  $x \in (V \cup T)^*$ . A grammar is *left-recursive* if it contains at least one left-recursive variable.

Every context-free language can be represented by a grammar that is not left-recursive. You will learn more about this in CIT 445: Principles and Techniques of Compilers.

### Self-Assessment Exercise I

- 1) Eliminate useless symbols from the following grammar:

$$\begin{array}{l} S \rightarrow AB \mid CA \\ A \rightarrow a \\ B \rightarrow BC \mid AB \\ C \rightarrow aB \mid b \end{array}$$

- 2) Outline an algorithm to determine whether a CFG generates the empty string. What is the running time of your algorithm in terms of  $n$ , the size of the grammar?

## 3.2 Normal Forms of Context-Free Grammars

### 3.2.1 Chomsky Normal Form

A grammar is in *Chomsky Normal Form* if all productions are of the form:

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where  $A$ ,  $B$ , and  $C$  are variables and  $a$  is a terminal. Any context-free grammar that does not contain  $\lambda$  can be put into Chomsky Normal Form.

(Most textbook authors also allow the production  $S \rightarrow \lambda$  so long as  $S$  does not appear on the right hand side of any production.)

Chomsky Normal Form is particularly useful for programs that have to manipulate grammars.

### 3.2 Greibach Normal Form

A grammar is in *Greibach Normal Form* if all productions are of the form:

$$A \rightarrow ax$$

where  $a$  is a terminal and  $x \in V^*$ .

Grammars in Greibach Normal Form are typically ugly and much longer than the CFG from which they were derived. Greibach Normal Form is useful for proving the equivalence of CFGs and NPDAs. When we discuss converting a CFG to an NPDA, or vice versa, we will use Greibach Normal Form.

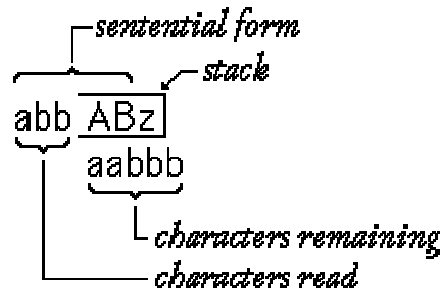
### 3.3 From CFG to NPDA

For any context-free grammar in Greibach Normal Form we can build an equivalent nondeterministic pushdown automaton. This establishes that an NPDA is at least as powerful as a CFG.

**Key idea:** Any string of a context-free language has a leftmost derivation. We set up the NPDA so that the stack contents "correspond" to this sentential form; every move of the NPDA represents one derivation step.

The sentential form is:

- the characters already read,
- PLUS the symbols on the stack
- MINUS the final  $z$  (the initial stack symbol).



**Figure 1: From CFG to NPDA**

In the NPDA we will construct, the states are hardly important at all. All the real work is done on the stack. In fact, we will use only the following three states, regardless of the complexity of the grammar:

- Start state  $q_0$  just gets things initialized. We use the transition from  $q_0$  to  $q_1$  to put the grammar's start symbol on the stack.

$$\delta(q_0, \lambda, z) \rightarrow \{(q_1, Sz)\}$$

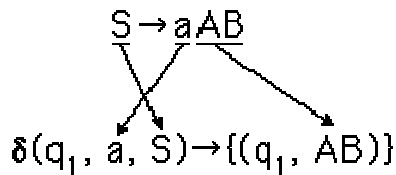
- State  $q_1$  does the bulk of the work. We represent every derivation step as a move from  $q_1$  to  $q_1$ .
- We use the transition from  $q_1$  to  $q_f$  to accept the string.

$$\delta(q_1, \lambda, z) \rightarrow \{(q_f, z)\}$$

### Example 1:

Consider the grammar  $G = (\{S, A, B\}, \{a, b\}, S, P)$ , where  $P = \{S \rightarrow a, S \rightarrow aAB, A \rightarrow aA, A \rightarrow a, B \rightarrow bB, B \rightarrow b\}$ .

These productions can be turned into transition functions by rearranging the components:



This yields the following table:

(start)	$\delta(q_0, \lambda, z) \rightarrow \{(q_1, Sz)\}$
$S \rightarrow a,$	$\delta(q_1, a, S) \rightarrow \{(q_1, \lambda)\}$
$S \rightarrow aAB,$	$\delta(q_1, a, S) \rightarrow \{(q_1, AB)\}$
$A \rightarrow aA,$	$\delta(q_1, a, A) \rightarrow \{(q_1, A)\}$
$A \rightarrow a,$	$\delta(q_1, a, A) \rightarrow \{(q_1, \lambda)\}$
$B \rightarrow bB,$	$\delta(q_1, b, B) \rightarrow \{(q_1, B)\}$
$B \rightarrow b$	$\delta(q_1, b, B) \rightarrow \{(q_1, \lambda)\}$
(finish)	$\delta(q_1, \lambda, z) \rightarrow \{(q_f, z)\}$

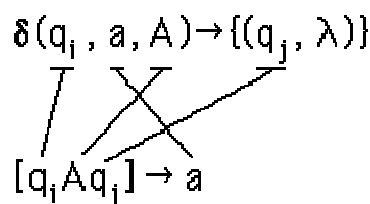
For example, the derivation

$$S \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aabb$$

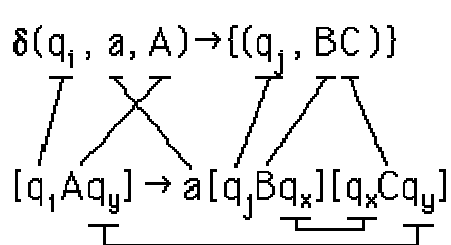
maps into the sequence of moves

$$(q_0, aabb, z) \vdash (q_1, aabb, Sz) \vdash (q_1, abb, ABz) \vdash (q_1, bb, Bz) \vdash (q_1, b, Bz) \vdash (q_1, \lambda, z) \vdash (q_2, \lambda, \lambda)$$

When we write a grammar, we can use any variable names we choose. As in programming languages, we like to use "meaningful" variable names. When we translate an NPDA into a CFG, we will use variable names that encode information about both the state of the NPDA and the stack contents. Variable names will have the form  $[q_i A q_j]$ , where  $q_i$  and  $q_j$  are states and  $A$  is a variable. The "meaning" of the variable  $[q_i A q_j]$  is that the NPDA can go from state  $q_i$  with  $Ax$  on the stack to state  $q_j$  with  $x$  on the stack.



Each transition of the form  $\delta(q_i, a, A) = (q_j, \lambda)$  results in a single grammar rule.



Each transition of the form  $\delta(q_i, a, A) = (q_j, BC)$  results in a multitude of grammar rules, one for each pair of states  $q_x$  and  $q_y$  in the NPDA.

This algorithm results in a lot of useless (unreachable) productions, but the useful productions define the context-free grammar recognized by the NPDA.

### Self Assessment Exercise II

- 3) Convert the following CFG into an equivalent grammar in Chomsky Normal Form. Show the shortest leftmost derivation for the string  $abb$  in both grammars.

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow BAB \mid B \\ B &\rightarrow bb \mid \varepsilon \end{aligned}$$

## 4.0 CONCLUSION

In this unit you have been briefly taken through the relationship between CFG and PDAs and how to simplify CFGs.

## 5.0 SUMMARY

In this unit, you learnt that:

- The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammar
- If the empty string does not belong to a language, then there is a way to eliminate productions of the form  $A \rightarrow \lambda$  from the grammar

- A grammar is *left-recursive* if it contains at least one left-recursive variable. E.g.  $A \rightarrow Ax$
- A grammar is in *Chomsky Normal Form* if all productions are of the form  $A \rightarrow BC$  or  $A \rightarrow a$
- A grammar is in *Greibach Normal Form* if all productions are of the form  $A \rightarrow ax$ , where  $a$  is a terminal and  $x \in V^*$
- Greibach Normal Form is useful for proving the equivalence of CFGs and NPDAs
- For any context-free grammar in Greibach Normal Form you can build an equivalent nondeterministic pushdown automaton

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Briefly describe the various ways that CFGs can be simplified.
2. What is the advantage of a CFG that in Greibach's Normal Form?
3. What are the disadvantage of Greibach's Normal Form?
4. Is an NPDA as powerful as a CFG? Discuss
5. Prove that  $L = \{ ww \mid w \text{ is any string of } a\text{'s and } b\text{'s} \}$  is not a CFL.
6. If  $L$  is context free, is  $LL^R$  context free? Justify your answer.
7. Let  $L$  be the complement of the language  $\{ ww \mid w \text{ is any string of } a\text{'s and } b\text{'s} \}$ . Show that  $L$  is context free.

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 4: Turing Machines

### Unit 1: Turing Machines and the rest

#### CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.0 Main Content
  - 3.1 Turing Machines and the rest
  - 3.2 What is a Turing machine?
  - 3.3 Universal Turing Machine
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor-Marked Assignment
- 7.0 References/Further Reading

#### 1.0 INTRODUCTION

This is the concluding module of this course and here you will be learning about the Turing machine. The machine for decidable languages.

A basic Turing machine is a model for studying computation. Turing machines can solve decision problems and compute results based on inputs. When studying computation we usually restrict our attention to integers. Since a real number has infinitely many fraction digits we cannot compute a real number in a finite time. Rational numbers are approximations to real numbers are equivalent and can be put in one-to-one correspondence with the integers.

Programming a Turing machine is tedious and thus much work at higher levels of abstraction make the reasonable assumption that any completely defined algorithm or computer program could be implemented by a Turing machine.

Now let us go through your study objectives for this unit.

#### 2.0 OBJECTIVES

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

- Define a Turing machine
- Distinguish between Turing machine and other classes of machines for language recognition discussed so far in this course

- Describe the best way to code a Turing machine

### 3.0 MAIN CONTENT

#### 3.1 Turing Machines and the Rest

A *Turing machine* (TM) is a generalization of a PDA which uses a tape instead of a stack. Turing machines are an abstract version of a computer - they have been used to define formally what is *computable*. There are a number of alternative approaches to formalize the concept of computability (e.g. called the  $\lambda$ -calculus, or  $\mu$ -recursive functions, ...) but they can all be shown to be equivalent. That this is the case for any reasonable notion of computation is called the *Church-Turing Thesis*.

On the other side there is a generalization of context-free grammars called phrase structure grammars or just grammars. Here we allow several symbols on the left hand side of a production, e.g. we may define the context in which a rule is applicable. Languages definable by grammars correspond precisely to the ones which may be accepted by a Turing machine and those are called *Type-0-languages* or the *recursively enumerable languages*.

Turing machines behave different from the previous machine classes we have seen/discussed: they may run forever, without stopping. To say that a language is accepted by a Turing machine means that the TM will stop in an accepting state for each word which is in the language. However, if the word is not in the language the Turing machine may stop in a non-accepting state or loop forever. In this case we can never be sure whether the given word is in the language – i.e. the Turing machine does not decide the word problem.

We say a language is *decidable*, if there is a TM which will always stop. There are *type-0-languages* which are not decidable – the most famous one is the *halting problem* (this will be fully discussed in the next unit). This is the language of encodings of Turing machines which will always stop.

There is no type of grammars which captures all decidable languages (and for theoretical reasons there cannot be one). However there is a subset of decidable languages which are called *context-sensitive languages* which are given by *context-sensitive grammars*, these are those grammars where the left hand side of a production is always shorter than the right hand side. Context-sensitive languages on the other hand correspond to linear bounded TMs, these are those TMs which use only a tape whose length can be given by a linear function over the length of the input.

#### 3.2 What is a Turing machine?

A Turing machine  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$  is given by the following data

- A finite set  $Q$  of states,
- A finite set  $\Sigma$  of symbols (the alphabet),

- A finite set  $\Gamma$  of tape symbols s.t.  $\Sigma \subseteq \Gamma$ . This is the case because we use the tape also for the input.
- A transition function:  $\delta \in Q \times \Gamma \rightarrow \{\text{stop}\} \cup Q \times \Gamma \times \{\text{L}, \text{R}\}$

The transition function defines how the function behaves if is in state  $q$  and the symbol on the tape is  $x$ . If  $\delta(q, x) = \text{stop}$  then the machine stops otherwise if  $\delta(q, x) = (q', y, d)$  the machines gets into state  $q'$ , writes  $y$  on the tape (replacing  $x$ ) and moves left if  $d = \text{L}$  or right, if  $d = \text{R}$ .

- An initial state  $q_0 \in Q$ ,
- The blank symbol  $B \in \Gamma$  but  $B \notin \Sigma$ . In the beginning only a finite section of the tape containing the input is not blank.
- A set of final states  $F \subseteq Q$ .

In most texts the transition function is defined without the stop option as

$$\delta \in Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}.$$

However they allow  $\delta$  to be undefined which correspond to our function returning stop.

This defines deterministic Turing machines, for non-deterministic TMs we change the transition function to

$$\delta \in Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{\text{L}, \text{R}\})$$

Here stop corresponds to  $\delta$  returning an empty set. As for finite automata (and unlike for PDAs) there is no difference in the strength of deterministic or non-deterministic TMs.

As for PDAs we define instantaneous descriptions ID for Turing machines. We have  $\text{ID} = \Gamma^* \times Q \times \Gamma^*$  where  $(\eta, q, \gamma_r)$  means that the TM is in state  $Q$  and left from the head the non-blank part of the tape is  $\eta$  and starting with the head itself and all the non-blank symbols to the right is  $\gamma_r$ .

We define the next state relation  $\vdash_M$  similar as for PDAs:

1.  $(\gamma_l, q, x\gamma_r) \vdash_M (\gamma_l y, q', \gamma_r)$  if  $\delta(q, x) = (q', y, R)$
2.  $(\gamma_l z, q, x\gamma_r) \vdash_M (\gamma_l, q', zy\gamma_r)$  if  $\delta(q, x) = (q', y, L)$
3.  $(\gamma_l, q, \epsilon) \vdash_M (\gamma_l y, q', \gamma_r)$  if  $\delta(q, B) = (q', y, R)$
4.  $(\epsilon, q, x\gamma_r) \vdash_M (\gamma_l, q', By\gamma_r)$  if  $\delta(q, x) = (q', y, L)$

The cases 3. and 4. are only needed to deal with the situation if we have reached the end of the (non-blank part of) the tape.

We say that a TM  $M$  accepts a word if it goes into an accepting state, i.e. the language of a TM is defined as

$$L(M) = \{w \in \Sigma^* \mid (\epsilon, q_0, w) \vdash_M^* (\gamma_l, q', \gamma_r) \wedge q' \in F\}$$

i.e. the TM stops automatically if it goes into an accepting state. However, it may also stop in a non-accepting state if  $\delta$  returns stop – in this case the word is rejected.

A TM  $M$  decides a language if it accepts it and it never loops (in the negative case).

To illustrate this we define a TM  $M$  which accepts the language  $L = a^n b^n c^n \mid n \in \mathbb{N}$  – this is a language which cannot be recognized by a PDA or be defined by a CFG.

We define  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$  by

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$$

$$\Sigma = \{a, b, c\}$$

$$\Gamma = \Sigma \cup \{X, Y, Z, \sqcup\}$$

$\delta$  is given by

$\delta(q_0, \sqcup)$	=	$(\sqcup, q_6, R)$
$\delta(q_0, a)$	=	$(X, q_1, R)$
$\delta(q_1, a)$	=	$(a, q_1, R)$
$\delta(q_1, X)$	=	$(X, q_1, R)$
$\delta(q_1, b)$	=	$(Y, q_2, R)$
$\delta(q_2, b)$	=	$(b, q_2, R)$
$\delta(q_2, Y)$	=	$(Y, q_2, R)$
$\delta(q_2, c)$	=	$(Z, q_3, R)$
$\delta(q_3, \sqcup)$	=	$(\sqcup, q_4, L)$
$\delta(q_3, c)$	=	$(c, q_5, L)$
$\delta(q_4, Z)$	=	$(Z, q_4, L)$
$\delta(q_4, b)$	=	$(b, q_4, L)$

$\delta(q_4, Y)$	=	$(Y, q_4, L)$
$\delta(q_4, a)$	=	$(a, q_4, L)$
$\delta(q_4, X)$	=	$(X, q_0, R)$
$\delta(q_5, Z)$	=	$(Z, q_5, L)$
$\delta(q_5, Y)$	=	$(Y, q_4, L)$
$\delta(q_5, X)$	=	$(X, q_6, R)$
$\delta(q, x)$	=	stop everywhere else

$$q_0 = q_0$$

$$B = \sqcup$$

$$F = \{q_6\}$$

The machine replaces an  $a$  by  $X(q_0)$  and then looks for the first  $b$  replaces it by  $Y(q_1)$  and looks for the first  $c$  and replaces it by a  $Z(q_2)$ . If there are more  $c$ s left it moves left to the next  $a$  ( $q_4$ ) and repeats the cycle. Otherwise it checks whether there are no  $a$ s and  $b$ s left ( $q_5$ ) and if so goes in an accepting state ( $q_6$ ).

E.g. consider the sequence of IDs on  $aabbcc$ :

$(\epsilon, q_0, aabbcc)$	$\vdash (X, q_1, abbcc)$
	$\vdash (Xa, q_1, bbcc)$
	$\vdash (XaY, q_2, bcc)$
	$\vdash (XaYb, q_2, cc)$
	$\vdash (XaYbZ, q_3, c)$
	$\vdash (XaYb, q_4, Zc)$
	$\vdash (XaY, q_4, bZc)$
	$\vdash (Xa, q_4, YbZc)$
	$\vdash (X, q_4, aYbZc)$
	$\vdash (\epsilon, q_4, XaYbZc)$
	$\vdash (X, q_0, aYbZc)$

$\vdash (XX, q_1, YbZc)$
$\vdash (XXY, q_1, bZc)$
$\vdash (XXYY, q_2, Zc)$
$\vdash (XXYYZ, q_2, c)$
$\vdash (XXYYZZ, q_2, \epsilon)$
$\vdash (XXYYZ, q_5, Z)$
$\vdash (XXYY, q_5, ZZ)$
$\vdash (XXY, q_5, YZZ)$
$\vdash (XX, q_5, YYZZ)$
$\vdash (X, q_5, XYYZZ)$
$\vdash (\epsilon, q_6, XXYYZZ)$

We see that  $M$  accepts  $aabbcc$ . Since  $M$  never loops it does actually decide  $L$ .

There are a lot of possible Turing machines and a useful technique is to code Turing machines as binary integers. A trivial coding is to use the 8 bit ASCII for each character in the written description of a Turing machine concatenated into one long bit stream.

Having encoded a specific Turing machine as a binary integer, we can talk about  $TM_i$  as the Turing machine encoded as the number "i".

It turns out that the set of all Turing machines is countable and enumerable.

### 3.3. Universal Turing Machine

Now we can construct a Universal Turing Machine (UTM) that takes an encoded Turing machine on its input tape followed by normal Turing machine input data on that same input tape. The Universal Turing Machine first reads the description of the Turing machine on the input tape and uses this description to simulate the Turing machines actions on the following input data. Of course a UTM is a TM and can thus be encoded as a binary integer, so a UTM can read a UTM from the input tape, read a TM from the input tape, then read the input data from the input tape and proceed to simulate the UTM that is simulating the TM. etc.

Since a UTM can be represented as an integer and can thus also be the input data on the input tape of itself or another Turing machine. This will be used in the next unit in the Halting Problem

#### 4.0 CONCLUSION

In this unit you have been introduced to Turing machines. This is the class of machines that can recognise any string from any language. They can recognise strings that the earlier machines/automata discussed in this course so far cannot recognise. In the next unit you will be learning more about the class of languages recognised by this machine.

#### 5.0 SUMMARY

In this unit, you learnt that:

- A TM  $M$  decides a language if it accepts it and it never loops
- A language is *decidable*, if there is a TM which will always stop
- A *Turing machine* (TM) is a generalization of a PDA which uses a tape instead of a stack.
- Turing machines are an abstract version of a computer
- A language is accepted by a Turing machine means that the TM will stop in an accepting state for each word which is in the language
- There is no type of grammars which captures all decidable languages
- the set of all Turing machines is countable and enumerable

#### 6.0 TUTOR-MARKED ASSIGNMENT

1. Define Turing machine
2. How is a Turing machine different from the other machines discussed so far in this course?
3. What does it mean to say a language is accepted by a Turing machine?
4. Design a Turing machine that accepts all strings of  $a$ 's and  $b$ 's with an equal number of  $a$ 's and  $b$ 's. Show the sequence of moves your Turing machine makes on the input  $aabb$

#### 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

## Module 4: Turing Machines

### Unit 2: Turing Machines and Context-Sensitive Grammars

#### CONTENTS

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Grammars and Context-Sensitivity
3.2	The Halting Problem
3.2.1	The Halting Problem for Turing machines.
3.3	Decision Problems
3.4	Godel Incompleteness Theorem
3.5	Unsolvable
3.6	Undecidable
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

#### 1.0 INTRODUCTION

In this unit you will be learning about the Turing machine. The machine for decidable languages.

A basic Turing machine is a model for studying computation. Turing machines can solve decision problems and compute results based on inputs. When studying computation we usually restrict our attention to integers. Since a real number has infinitely many fraction digits we cannot compute a real number in a finite time. Rational numbers are approximations to real numbers are equivalent and can be put in one-to-one correspondence with the integers.

Programming a Turing machine is tedious and thus much work at higher levels of abstraction make the reasonable assumption that any completely defined algorithm or computer program could be implemented by a Turing machine.

Now let us go through your study objectives for this unit.

#### 2.0 OBJECTIVES

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

- Define context-sensitive grammars

- Distinguish context-sensitive grammars from others that have been discussed so far in this course
- Briefly explain the halting problem
- State Godel's incompleteness theorem
- Define the following with respect to TM:
  - Unsolvability
  - Undecidability

### 3.0 MAIN CONTENT

#### 3.1 Grammars and Context-Sensitivity

Grammars  $G = (V, \Sigma, S, P)$  are defined as context-free grammars before with the only difference that there may be several symbols on the left-hand side of a production, i.e.

$P \subseteq (V \cup T)^+ \times (V \cup T)^*$ . Here  $(V \cup T)^+$  means that at least one symbol has to be present. The relation derives  $\Rightarrow_G$  (and  $\Rightarrow_G^*$ ) is defined as before :

$$\Rightarrow_G \subseteq (V \cup T)^* \times (V \cup T)^*$$

$$\alpha\beta\gamma \Rightarrow_G \alpha\beta'\gamma \quad :\Leftrightarrow \beta \rightarrow \beta' \in P$$

and as before the language of  $P$  is defined as:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

We say that a grammar is context-sensitive (or type 1) if the left hand side of a production is at least as long as the right hand side. That is for each  $\alpha \rightarrow \beta \in P$  we have  $|\alpha| \leq |\beta|$

Here is an example of a context-sensitive grammar  $G = (V, \Sigma, S, P)$ : with  $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N} \wedge n \geq 1\}$ . where

$$V = \{S, B, C\}$$

$$\Sigma = \{a, b, c\}$$

$$P = \{ \quad \quad \quad S \rightarrow aSBC$$

$$\quad \quad \quad S \rightarrow aBC$$

$$\quad \quad \quad aB \rightarrow ab$$

$$\quad \quad \quad CB \rightarrow BC$$

$$bB \rightarrow bB$$

$$bC \rightarrow bc$$

$$cC \rightarrow cc\}$$

We present without proof:

**Theorem 1:** For a language  $L \subseteq \Sigma^*$  the following is equivalent:

1.  $L$  is accepted by a Turing machine  $M$ , i.e.  $L = L(M)$
2.  $L$  is given by a grammar  $G$ , i.e.  $L = L(G)$

**Theorem 2:** For a language  $L \subseteq \Sigma^*$  the following is equivalent:

1.  $L$  is accepted by a Turing machine  $M$ , i.e.  $L = L(M)$  such that the length of the tape is bounded by a linear function in the length of the input, i.e.

$$|\eta| + |\gamma_r| \leq f(x) \text{ where } f(x) = ax + b \text{ with } a, b \in \mathbb{N}.$$

2.  $L$  is given by a context sensitive grammar  $G$ , i.e.  $L = L(G)$

### 3.2 The Halting Problem

Turing showed that there are languages which are accepted by a TM (i.e. type 0 languages) but which are undecidable. The technical details of this construction are quite involved but the basic idea is quite simple and is closely related to Russell's paradox, which we have seen in MCS.

The "Halting Problem" is a very strong, provably correct, statement that no one will ever be able to write a computer program or design a Turing machine that can determine if a arbitrary program will halt (stop, exit) for a given input.

This is NOT saying that some programs or some Turing machines cannot be analyzed to determine that they, for example, always halt.

The Halting Problem says that no computer program or Turing machine can determine if ALL computer programs or Turing machines will halt or not halt on ALL inputs. To prove the Halting Problem is unsolvable we will construct one program and one input for which there is no computer program or Turing machine.

We will use very powerful mathematical concepts and do the proofs for both a computer program and a Turing machine. The mathematical concepts we need are:

1. **Proof by contradiction:** Assume a statement is true, show that the assumption leads to a contradiction. Thus the statement is proven false.

2. **Self referral:** Have a computer program or a Turing machine operate on itself, well, a copy of itself, as input data. Specifically we will use diagonalization, taking the enumeration of Turing machines and using  $TM_i$  as input to  $TM_i$ .
3. **Logical negation:** Take a black box that accepts input and outputs true or false, put that black box in a bigger black box that switches the output so it is false or true respectively.

The simplest demonstration of how to use these mathematical concepts to get an unsolvable problem is to write on the front and back of a piece of paper "The statement on the back of this paper is false."

Starting on side 1, you could choose "True" and thus deduce side 2 is "False". But starting on side 2, which is exactly the same as side 1, you get that side 2 is "True" and side 1 is "False."

Since side 1, and side 2, can be both "True" and "False" there is a contradiction. The problem of determining if sides 1 and 2 are "True" or "False" is unsolvable.

The Halting Problem for a programming language. We will use the "C" programming language, yet any language will work.

Assumption: There exists a way to write a function named Halts such that:

```
int Halts(char * P, char * I)
{
    /* code that reads the source code for a "C"
program, P,
determines that P is a legal program, then
determines if P
eventually halts (or exits) when P reads the
input string I,
and finally sets a variable "halt" to 1 if P
halts on input I,
else sets "halt" to 0 */
    return halt;
}
```

Construct a program called Diagonal.c as follows:

```
int main()
{
    char I[100000000]; /* make as big as you
want or use malloc */
    read_a_C_program_into( I );
```

```

    if ( Halts(I,I) ) { while(1){} } /* loop
forever, means does not halt */
    else return 1;
}

```

Compile and link Diagonal.c into the executable program Diagonal.

Now execute

```
Diagonal < Diagonal.c
```

Consider two mutually exclusive cases:

**Case 1:** Halts(I,I) returns a value 1.

This means, by the definition of Halts, that Diagonal.c halts when given the input Diagonal.c.

BUT! we are running Diagonal.c (having been compiled and linked) and so we see that Halts(I,I) returns a value 1 causes the "if" statement to be true and the "while(1){}" statement to be executed, which never halts, thus our executing Diagonal.c does NOT halt.

This is a contradiction because this case says that Diagonal.c does halt when given input Diagonal.c. We will try the other case.

**Case 2:** Halts(I,I) returns a value 0.

This means, by the definition of halts, that Diagonal.c does NOT halt when given the input Diagonal.c.

BUT! we are running Diagonal.c (having been compiled and linked) and so we see that Halts(I,I) returns a value 0 causes the "else" to be executed and the main function halts (stops, exits).

This is a contradiction because this case says that Diagonal.c does NOT halt when given input Diagonal.c. There are no other cases, Halts can only return 1 or 0. Thus what must be wrong is our assumption "there exists a way to write a function named Halts..."

### 3.2.1 The Halting Problem for Turing machines.

**Assumption:** There exists a Turing machine,  $TM_h$ , such that: When the input tape contains the encoding of a Turing machine,  $TM_j$  followed by input data  $k$ ,  $TM_h$  accepts if  $TM_j$  halts with input  $k$  and  $TM_h$  rejects if  $TM_j$  is not a Turing machine or  $TM_j$  does not halt with input  $k$ .

Note that  $TM_h$  always halts and either accepts or rejects.

Pictorially  $TM_h$  is:





Thus we have proved that no Turing machine  $TM_h$  can ever be created that can be given the encoding of any Turing machine,  $TM_j$ , and any input,  $k$ , and always determine if  $TM_j$  halts on input  $k$ .

### 3.3 Decision Problems

Decision problems are stated as questions where the answer is binary, 0 or 1, False or True, No or yes, Reject or Accept and so forth.

Generally a decision problem states a problem and gives a candidate solution, asking if the solution solves the problem.

#### Examples:

Given the math expression  $2+2$  is the answer 4?

Given a formal language and a string, is the string in the language?

Given a grammar and a string, is the string accepted by the grammar?

### 3.4 Godel Incompleteness Theorem

Any formal system powerful enough to express arithmetic must have true theorems that cannot be proven within the formal system.

Basically Godel proved that when trying to add axioms to a formal system in order to prove all true theorems within the formal system, eventually the system will become inconsistent before it becomes complete.

A complete formal system is a formal system where all true theorems can be proved.

An inconsistent formal system is a formal system where at least one false statement can be proved within the formal system.

Due to the computational equivalence of formal systems to other computational capability, we get the Halting problem, the uncomputable numbers and other unsolvable problems.

### 3.5 Unsolvable

A formally stated problem is Unsolvable if no Turing machine exists to compute the solution.

A formally stated problem is provably unsolvable if it can be proved no Turing machine exists to compute the solution.

### 3.6 Undecidable

A formally stated problem is Undecidable if no total recursive function and thus, no Turing machine that always halts can be constructed to decide the problem, usually true or false.

Let us fix a simple alphabet  $\Sigma = \{0,1\}$ . As computer scientists we are well aware that everything can be coded up in bits and hence we accept that there is an encoding of TMs in binary. i.e. given a TM  $M$  we write  $[M] \in \{0, 1\}^*$  for its binary encoding. We assume that the encoding contains its length such that we know when subsequent input on the tape starts.

Now we define the following language:

$$L_{\text{halt}} = \{[M]w \mid [M \text{ holds on input } w]\}$$

It is easy to define a TM which accepts this language: we just simulate  $M$  and accept if  $M$  stops.

However, Turing showed that there is no TM which decides this language. To see this let us assume that there is a TM  $H$  which decides  $L$ . Now using  $H$  we construct a new TM  $F$  which is a bit obnoxious:  $F$  on input  $x$  runs  $H$  on  $xx$ . If  $H$  says yes then  $F$  goes into a loop otherwise ( $H$  says no)  $F$  stops.

The question is what happens if we run  $F$  on  $[F]$ ? Let us assume it terminates, then  $H$  applied to  $[F][F]$  returns yes and hence we must conclude that  $F$  on  $[F]$  loops???. On the other hand if  $F$  with input  $[F]$  loops then  $H$  applied to  $[F][F]$  will stop and reject and hence we have to conclude that  $F$  on  $[F]$  will stop?????

This is a contradiction and hence we must conclude that our assumption that there is a TM  $H$  which decides  $L_{\text{halt}}$  is false. We say  $L_{\text{halt}}$  is undecidable.

#### 4.0 CONCLUSION

In this unit you have been introduced to Turing machines and the context-sensitive grammars. In the next unit you will be learning about the last class of grammars, unrestricted grammars and the machines that can recognise them.

#### 5.0 SUMMARY

In this unit, you learnt that:

- a grammar is context-sensitive (or type 1) if the left hand side of a production is at least as long as the right hand side. That is for each  $\alpha \rightarrow \beta \in P$  we have  $|\alpha| \leq |\beta|$
- The Halting Problem says that no computer program or Turing machine can determine if ALL computer programs or Turing machines will halt or not halt on ALL inputs
- Decision problems are stated as questions where the answer is binary, 0 or 1, False or True, No or yes, Reject or Accept and so forth
- Basically Godel proved that when trying to add axioms to a formal system in order to prove all true theorems within the formal system, eventually the system will become inconsistent before it becomes complete
- A complete formal system is a formal system where all true theorems can be proved
- An inconsistent formal system is a formal system where at least one false statement can be proved within the formal system

## 6.0 TUTOR-MARKED ASSIGNMENT

- 1) State Godel incompleteness theorem
- 2) What do you understand by halting problem?
- 3) What mathematical concepts will you use in proving that the halting problem is unsolvable?
- 4) Define context-sensitive grammars
- 5) What do you understand by decision problems?
- 6) What does it mean to say a formally stated problem is
  - a) Unsolvable
  - b) undecidable

## 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.

**Module 4: Turing Machines****Unit 3: Unrestricted Grammars***CONTENTS*

1.0	<i>Introduction</i>
2.0	<i>Objectives</i>
3.0	<i>Main Content</i>
3.1	Unrestricted Grammars
3.2	From Grammars to Turing Machines
3.3	From Turing Machines to Grammars
4.0	<i>Conclusion</i>
5.0	<i>Summary</i>
6.0	<i>Tutor-Marked Assignment</i>
7.0	References/Further Reading

**1.0 INTRODUCTION**

In this the previous unit you learnt about Turing machines and context-sensitive grammars. In this unit, which is the concluding unit of this course, you will learn about Turing machines and the last class of grammars referred to as unrestricted grammars. It will be shown that Turing machines and unrestricted grammars are equivalent.

Now let us go through your study objectives for this unit.

**2.0 OBJECTIVES**

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

- Define unrestricted grammars
- Demonstrate the relationship between Turing machines and unrestricted grammars

**3.0 MAIN CONTENT****3.1 Unrestricted Grammars**

The productions of an unrestricted grammar have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*$$

The other grammar types we have considered (left linear, right linear, linear, context free) restrict the form of productions in one way or another. An *unrestricted grammar* does not.

In what follows, we will attempt to show that unrestricted grammars are equivalent to Turing machines. Bear in mind that:

- A language is *recursively enumerable* if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language.
- "Does not accept" is *not* the same as "reject" – the Turing machine could go into an infinite loop instead, and never get around to either accepting *or* rejecting the string.

Our plan of attack is to show that the languages generated by unrestricted grammars are precisely the recursively enumerable languages.

### 3.2 From Grammars to Turing Machines

**Theorem 1:** Any language generated by an unrestricted grammar is recursively enumerable.

This can be proven as follows:

1. If a procedure exists for enumerating the strings of a language, then the language is recursively enumerable. (We proved this earlier in unit 2 of this module.)
2. There exists a procedure for enumerating all the strings in any language generated by an unrestricted grammar. (We will demonstrate the procedure shortly in this unit.)
3. Therefore, any language generated by an unrestricted grammar is recursively enumerable.

Here is a review of the argument for (1) above. We prove the language is recursively enumerable by constructing a Turing machine to accept any string  $w$  of the language.

- Build one Turing machine that generates the strings of the language in some systematic order.
- Build a second Turing machine that compares its input to  $w$  and accepts its input if the two strings are identical.
- Build a composite Turing machine that incorporates the two machines above, using the output of the first as input to the second.

Now we have to systematically generate all the strings of the language. For other types of grammars it worked to generate shortest strings first; we do not know how to do that with an unrestricted grammar, because some productions could make the sentential form shorter. It might take a million steps to derive  $\lambda$ .

Instead, we order the strings *shortest derivation first*. First we consider all the strings that can be generated from  $S$  in one derivation step, and see if any of them are composed entirely of terminals. (We can do this because there are only a finite number of productions.) Then we consider all the strings that can be derived in two steps, and so on. Q.E.D.

### 3.3 From Turing Machines to Grammars

We have shown that a Turing machine can do anything that an unrestricted grammar can do. Now we have to show that an unrestricted grammar can do anything a Turing machine can do. This can be done by using an unrestricted grammar to emulate a Turing machine. We will give only the barest outline of the proof.

Recall that a *configuration* of a Turing machine can be written as a string

$$x_i \dots x_j q_m x_k \dots x_l$$

where the  $x$ 's are the symbols on the tape,  $q_m$  is the current state, and the tape head is on the square containing  $x_k$  (the symbol immediately following  $q_m$ ). It makes sense that a grammar, which is a system for rewriting strings, can be used to manipulate configurations, which can easily be written as strings.

A Turing machine accepts a string  $w$  if :

$$q_0 w \xrightarrow{*} x q_f y$$

for some strings  $x$  and  $y$  and some final state  $q_f$ , whereas a grammar produces a string if:

$$S \xrightarrow{*} w.$$

Because the Turing machine starts with  $w$  while the grammatical derivation ends with  $w$ , the grammar we build will run "in reverse" as compared to the Turing machine.

Recall that a Turing machine accepts a string  $w$  if

$$q_0 w \xrightarrow{*} x q_f y$$

and that our grammar will run "backwards" compared to the Turing machine.

The productions of the grammar we will construct can be logically grouped into three sets:

1. **Initialization:** These productions construct the string  $\dots B \$ x q_f y B \dots$  where  $B$  indicates a blank and  $\$$  is a special variable used for termination.
2. **Execution:** For each transition rule of  $\mathcal{D}$  we need a corresponding production.

3. **Cleanup:** Our derivation will leave some excess symbols  $q_0$ ,  $B$ , and  $\$$  in the string (along with the desired  $w$ ), so we need a few more productions to clean these up.

For the terminals  $T$  of the grammar we will use the input alphabet  $\Sigma$  of the Turing machine.

For the variables  $V$  of the grammar we will use

- $\Gamma - \Sigma$ , the tape alphabet minus the symbols we took for  $T$ .
- A symbol  $q_i$  for each state of the Turing machine.
- $B$  (blank) and  $\$$  (used for termination).
- $S$  (for a start symbol) and  $A$  (used for initialization).

**Initialization:** We need to be able to generate any string of the form

$$B \dots B \$ x q_f y B \dots B$$

Since we need an arbitrary number of "blanks" on either side, we use the productions

$$S \rightarrow BS \mid SB \mid \$A$$

(The  $\$$  is a marker we will use later.) Next we use the  $A$  to generate the strings  $x, y \in \Gamma$ , with a state  $q_f$  somewhere in the middle:

$$A \rightarrow sA \mid As \mid q_f, \text{ for all } s \in \Gamma.$$

**Execution:** For each transition rule of  $\delta$  we need a corresponding production. For each rule of the form:

$$\delta(q_i, a) = (q_j, b, R)$$

we use a production:

$$b q_j \rightarrow q_i a$$

and for each rule of the form:

$$\delta(q_i, a) = (q_j, b, L)$$

we use a production:

$$q_j c b \rightarrow c q_i a$$

for every  $c \in \Gamma$  (the asymmetry is because the symbol *to the right* of  $q$  is the one under the Turing machine's tape head.)

**Cleanup:** We end up with a string that looks like  $B...B\$q_0wB...B$ , so we need productions to get rid of everything but the  $w$ :

$$B \rightarrow \lambda$$

$$\$q_0 \rightarrow \lambda$$

#### 4.0 CONCLUSION

In this last unit of this course, you have been introduced to unrestricted grammars and the machines that can recognise them. It has been shown that a Turing machine can do anything that an unrestricted grammar can do and vice versa. Any other thing you need to know about computation will be discussed in another course on computation. If you are interested it is advisable you take the course on Theory of Computation.

#### 5.0 SUMMARY

In this unit, you learnt that:

- The productions of an unrestricted grammar have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*$$

- An *unrestricted grammar* does not restrict the form of productions
- a Turing machine can do anything that an unrestricted grammar can do and vice versa i.e. unrestricted grammars are equivalent to Turing machines

#### 6.0 TUTOR-MARKED ASSIGNMENT

- 1) Define unrestricted grammars
- 2) When is a grammar recursively enumerable?
- 3) Prove that any language generated by an unrestricted grammar is recursively enumerable.

#### 7.0 REFERENCES/FURTHER READING

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Pearson Education.

- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.
- James P. Schmeiser, David T. Barnard (1995). *Producing a top-down parse order with bottom-up parsing*. Elsevier North-Holland.