THEORETICAL COMPUTER SCIENCE II

SCHOOL OF COMPUTING

Only study guide for

COS2601

UNIVERSITY OF SOUTH AFRICA; PRETORIA

Printed and published by the
University of South Africa
Muckleneuk, Pretoria

COS2601/1/2018–2019

70450609

Layout done by the department

PHOTOGRAPHS
Ilze Botha and Shutterstock.com

GRAPHIC DESIGNER
Ella Viljoen

CONTENTS

1 Introduction

Dear Student

As noted in Tutorial Letter 101, this is a blended module, and therefore much of your module material is available on myUnisa. However, in order to support you in your learning process, we have also provided this study material in printed format.

Below you will find all the material that is available on the COS2601 site on myUnisa.

2 Home page

COS2601 – Theoretical Computer Science II

Welcome to the myUnisa module site for COS2601. Your lecturer for this module is Colin Pilkington.

In this module we will focus on the basic ideas of automata theory and also show you how to apply recursive definitions and inductive proofs. We trust that you will find the module interesting.

There are two versions of the prescribed book, and you can use either one of them – see the Learning Units, Unit 0 for more details on the prescribed book. In this module we will cover part 1 of the book and in the third-year module (COS3701) we will cover parts 2 and 3.

Statistics from previous years show that most students who mastered the required study material and who attempted all the assignments passed the module. So, this should inspire you to do the same.

Assessment

You will be assessed in various ways in this module.
- There are three hand-in assignments, and it is important to complete all these assignments.
- There are three self-tests that you should also do. We will provide the answers to these self-tests.
- There is a two-hour written exam.

A work schedule is provided in Additional Resources. This schedule gives you an indication of the pace required so that you will be able to hand in assignments on time, and are left with enough time for exam preparation.

General information

This module site is dedicated to support your learning for this module. Please make a regular habit of checking this site.
- **Home page**: This is the page you are currently on, and you will always start on this landing page of the COS2601 myUnisa site.
- **Learning Units**: This is where you will find all the work for this semester, including what has to be studied and extra notes. The MO001 document that will be sent to you, and can be downloaded from Additional Resources, is a document version of these pages.
- Use the **Discussion Forums** to interact with other students about assignments, study groups, self-tests, and the exam. Please post your discussions on the appropriate forum and topic discussion line; do not use this forum for idle chatter. We will be monitoring the discussions periodically and will contribute occasionally when necessary. If your questions are not addressed here, contact the lecturer via email or telephone.
- **Assignments**: Use this page to submit all three your hand-in assignments for the semester. You can also see your assignment marks here. The assignment questions can be found in Tutorial letter 101.
- **Announcements**: These are made regularly when there is an issue that we want to tell you about or to provide useful information – so please keep checking these. The contents of the announcement will be e-mailed to your Unisa myLife account – make sure that you check this regularly too.

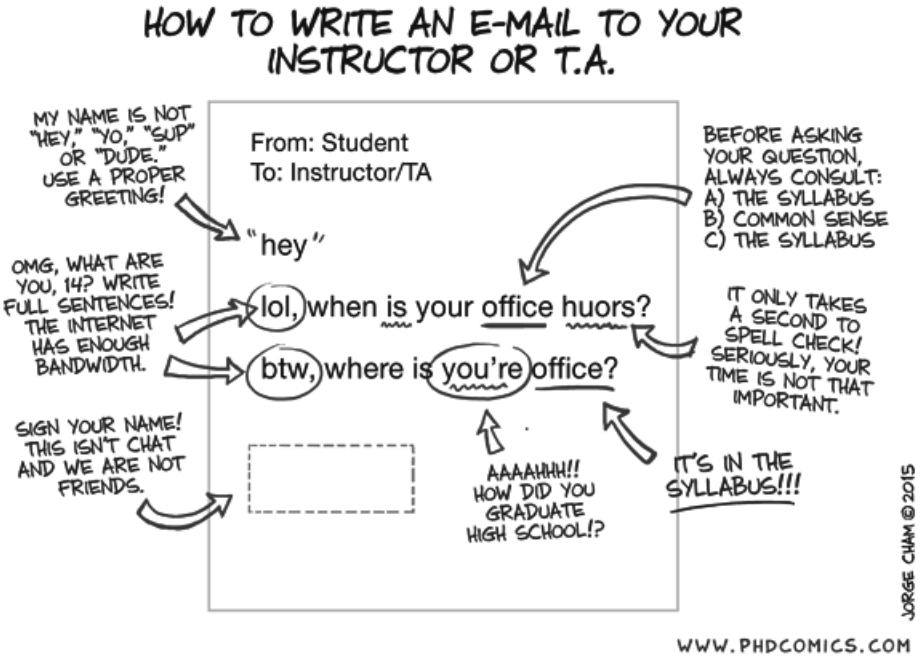
- Go to **Official Study Material** to find PDF versions of some tutorial letters and past exams.
- The **Additional Resources** page is where you will find other relevant documents such as assignment solutions and discussions, the work schedule, among other documents and videos.
- The **Schedule** will remind you of important current events and information like assignment and exam dates.

Contacting us

You can contact us, or other module lecturers, by using the following resources.

- Tutorial Letter 301 provides important information about the School of Computing (SOC) including all the lecturers' contact information.
- You are welcome to visit the SOC website (http://osprey.unisa.ac.za) where you can view the lecturers' contact information and availability (http://osprey.unisa.ac.za/reg.htm). (Right click the link and select <<Open in New Window>> to view the site properly.) Please phone the School's general number (011 670 9200) to leave a message if a lecturer is not available.

Take a look at the comic strip below that provides some guidelines when contacting us. Our equivalent of a T.A. is an e-tutor.



"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

3 Learning Unit 0 – Orientation

Introduction

Welcome
Welcome to the learning units that will make up your study material for the semester. Here you will find out about the prescribed book, the work that you are expected to study from it, and the assignments that need to be done. It is important that you check the assessment and study plan page, as here you will find a guide to getting through all the material in the allotted time (and submitting the assignments on time).

In this module we will discuss a simple abstract model of a computer, namely a finite automaton (FA). We will familiarise ourselves with the simple languages that finite automata can accept as input, namely regular languages. You will also be introduced to two very useful mathematical tools: induction and recursion. Non-regular languages together with the pumping lemmas will also be dealt with.

Your familiarity with basic set theory, relations, functions, proof techniques, and logic, acquired via your COS1501 module, will be useful, thus keep the COS1501 study material close at hand. COS2601 prepares you for COS3701 in which more realistic models of computers are investigated and some limitations of computers are discovered.

This module does not involve practical work on a computer. However, there is a strong emphasis on practising the skills acquired by doing the exercises provided in the study guide as well as the assignments. By completing all the assignments, you are actually preparing for the examinations. It is very important to work consistently throughout the semester in order to master the contents of this very interesting module.

This first unit (Unit 0) covers more general topics that relate to the module. The rest of the learning units will guide you through the work that has to be studied as part of this module. You can use the table of contents to move to the unit that you want to view.

Once you start moving around these pages, you can always return to the Table of Contents by clicking the link at the top or bottom centre of each page. There are also links here to the previous and next page in the sequence of pages in the learning units.

Purpose
Qualifying students can apply fundamental knowledge and skills from Applied Mathematics (like set theory) to computing fields such as programming to assist in the development of correct algorithms that can be implemented as computer programs. Students will use set theory and other relevant applied mathematical tools, to define regular expressions that generate regular languages. Students have to construct regular language acceptor machines (finite automata, Moore and Mealy machines, transition graphs and nondeterministic finite automata), which are considered to be simple abstract models of computers. Students will also use mathematical tools (induction and recursion techniques) to provide mathematical evidence (proof) of a statement.

This module forms part of the theory of a computer science major, supporting further studies and applications in the sector of computer programming, bioinformatics and linguistics. These concepts and skills contribute to the development of the computing field in southern Africa, Africa, and globally.

Outcomes
Specific outcome 1: Define regular languages (as formal mathematical presentations) using a variety of defined mathematical tools for evidence (including definitions, theorems and operators).

Specific outcome 2: Construct mathematical proofs in a clear and concise way using abstract mathematical reasoning techniques like direct proofs (induction) and proof by contradiction (pumping lemma).

Specific outcome 3: Construct regular language machine acceptors by drawing these machine acceptors through applying the relevant definitions and theorems.

Specific outcome 4: Critically analyse and synthesise regular expressions, regular languages, and machines that accept regular languages.

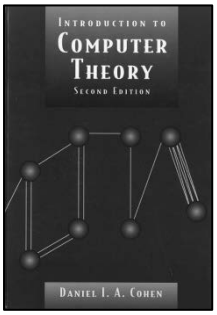
Specific outcome 5: Apply algorithms on machines by performing algorithms on transition graphs to obtain the regular expressions that define the language accepted by the transition graphs and by performing an algorithm on finite automata (to obtain a product, intersection, sum, Kleene closure) of a maximum of two finite automata (in order to construct new language acceptors from old language acceptors). If applied in real life, problems which can be solved include pattern matching problems within the context of security, bioinformatics as well as linguistics.

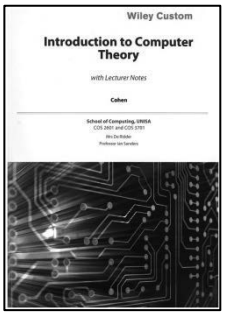
e-Tutors

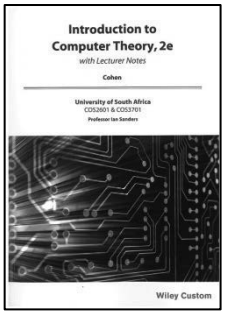
Unisa offers online tutorials (e-tutoring) to students registered for modules at NQF level 5, 6, and 7; that is, qualifying first-year, second-year, and third-year modules. Once you have been registered for a qualifying module, you will be allocated to a group of students with whom you will be interacting during the tuition period as well as an e-tutor who will be your tutorial facilitator. Thereafter you will receive an SMS informing you about your group, the name of your e-tutor and instructions on how to log onto myUnisa in order to receive further information on the e-tutoring process. Of course, you can still contact either of the module lecturers if you need to.

We strongly encourage you to use your e-tutors – do the exercises that they post online, email them when you have problems, and discuss the module content on the e-tutor discussion forums. The point of the e-tutor is to help you, and it would be a pity if you were not to use this valuable resource.

Books







Prescribed book
You are fortunate enough to have a really excellent prescribed book.
The prescribed book for COS2601 is:
Cohen, Daniel I.A.
1997
Introduction to Computer Theory, 2nd edition.
John Wiley & Sons.

You may purchase the 2014 or 2016 Unisa custom editions.

The prescribed book covers the introduction to automata theory in great detail, with many explanatory examples. In the learning units we will, in the main, restrict ourselves to brief comments that indicate connections with other modules, to summaries of some of the longer algorithms, and to the odd additional example.

The exception occurs in learning unit 3 that deals somewhat superficially with the idea of recursion. We have included an in-depth discussion of recursion and induction in the notes in this learning unit, which you should treat as important examinable material. Note that we cover only part I of the textbook. (Parts II and III are covered in COS3701.)

The following parts in Cohen are excluded from the COS2601 syllabus:

- From the section "The Myhill-Nerode Theorem" on page 196 up to just before the problems on page 203.
- All problems based on the omitted sections above.

Note that the solutions to the recommended problems provided at the end of each unit can be found in a file under Additional Resources.

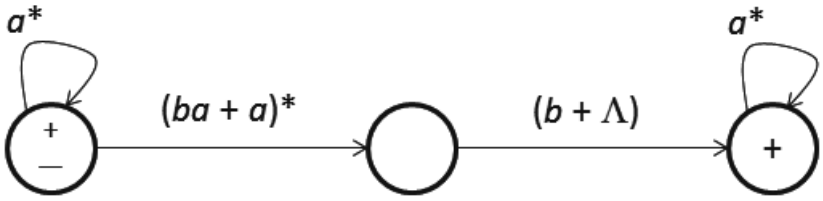
Errata

The following corrections should be made in the Cohen book.

Page 56: The transition function should be $\delta(q_i, x_j) = q_k$ instead of $\delta(q_i, x_j) = x_k$.

Page 71: Line 8 reads, "The input *desiccate* will go through the sequence of states: 1-1-1-1-1-2-3-4-4…" It should be 1-1-1-1-1-2-**2**-3-4-4.

Page 86: One of the edges of the TG on the lower part of page 86 has an incorrect label. The label of the edge from the start state to the middle state should be $(ba + a)^*$.

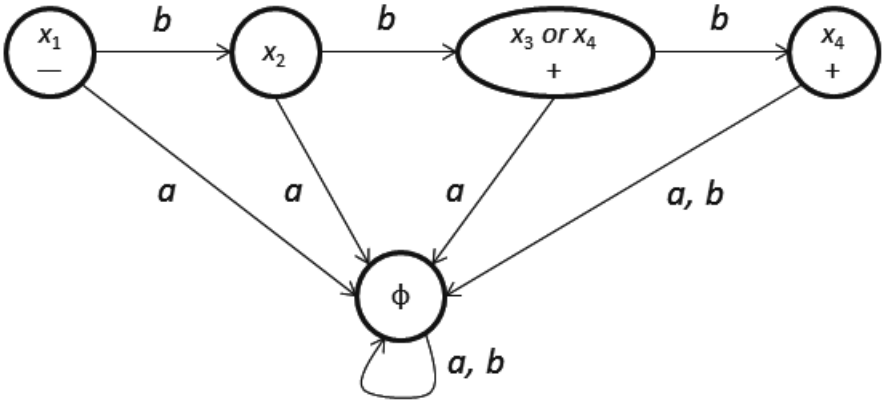


Page 98: On line 5, the word *unless* should be replaced with *useless*.

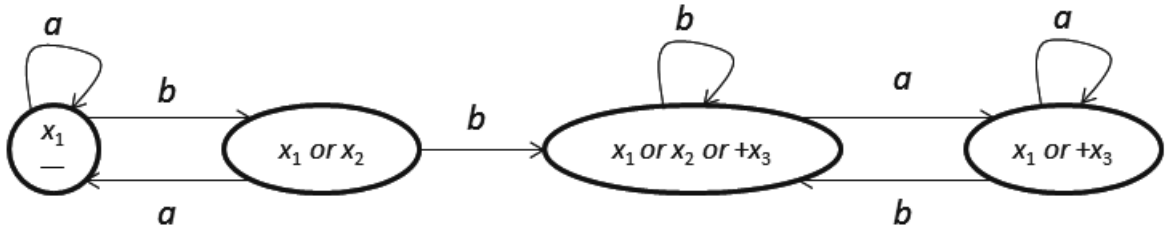
Page 131: Line 11 reads, "If we are in z_2...". It should read, "If we are in z_4...".

Pages 139 & 140: There are some errors in each of the three examples. If we apply the algorithm correctly, all three resulting FAs are different from those in the book to a greater or lesser extent.

- The first NFA on page 139 does indeed change into the given FA but the edge from state x_4 to the dead-end state should have *a*, *b* as label (not *a* alone).

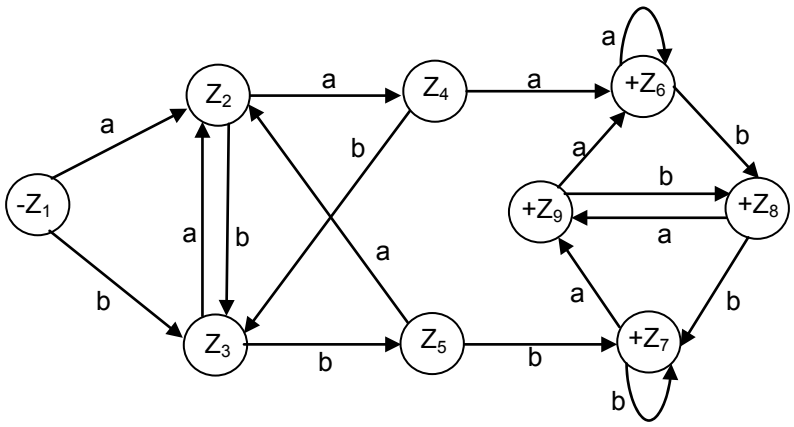


- The second NFA on page 139 changes to an FA with **four** states:
  The state on the right-hand side of the FA in the book should be marked as
  x_1 *or* x_2 *or* $+x_3$
  and the label of the loop should be *b* (and not *a, b*). There should be an edge with label *a* from this state to a new state marked
  x_1 *or* $+x_3$.
  At this new state there should be a loop with an *a* label and an edge returning to the (x_1 *or* x_2 *or* $+x_3$) state with label *b*.



- The NFA on page 140 changes to an FA with two states more than the NFA itself. We obtain the following:

New state	Old states	Read an a	Read a b
-z1	1	z2 = 1 or 2	z3 = 1 or 5
z2	1 or 2	z4 = 1 or 2 or 3	z3 = 1 or 5
z3	1 or 5	z2 = 1 or 2	z5 = 1 or 5 or 6
z4	1 or 2 or 3	+z6 = 1 or 2 or 3 or 4	z3 = 1 or 5
z5	1 or 5 or 6	z2 = 1 or 2	+z7 = 1 or 5 or 6 or 4
+z6	1 or 2 or 3 or 4	+z6 = 1 or 2 or 3 or 4	+z8 = 1 or 5 or 4
+z7	1 or 5 or 6 or 4	+z9 = 1 or 2 or 4	+z7 = 1 or 5 or 6 or 4
+z8	1 or 5 or 4	+z9 = 1 or 2 or 4	+z7 = 1 or 5 or 6 or 4
+z9	1 or 2 or 4	+z6 = 1 or 2 or 3 or 4	+z8 = 1 or 5 or 4



Page 181: For the FA at the bottom of the page, the edge from state 11 to state 8 is missing an a label.

Recommended book

Should you wish to know more about a particular topic, you may consult the following book. (This book may not necessarily be included in the study collection of the Unisa library.)

Martin, J. 2013. *Introduction to Languages and the Theory of Computation.* 3rd edition. McGraw-Hill.

Books online

You can also find books on the theory of computation in the online resources from the library.

- Go to oasis.unisa.ac.za
- Click on Library Links → Search for Information Resources → A-Z list of electronic resources → s → Safari Business and Tech Books Online
- In the search box at the top right of the page, search the entire site for “theory of computation”.
- You can then click on a book’s title, which should take you to the table of contents for that book. From here you can click on the chapter you want to read.

Electronic resources

Computer-aided instruction tutorials
The computer-aided instruction (CAI) tutorials named "Automata" and "Pumping lemmas" are provided on a CD that you should have received in your study package. It is not essential for you to work through these supplementary tutorials. They can, however, help you to understand the relevant study material better.

The minimum system requirements to run these tutorials is a Pentium II computer, preferably with 32MB RAM, a CD-ROM drive, a mouse and a 16-bit (High Colour) colour display, with at least Microsoft Windows 95.

How to run the tutorial:
- Insert the CD-ROM into your CD-ROM drive.
- Navigate to the CD drive in Windows Explorer.
- Double-click on the executable file Relation.exe.

Note: If you have problems in running this tutorial (e.g. in Windows 7 an irritating pop-up is repeatedly displayed), you can copy the whole cos2601 directory to the hard disk of your computer, and then double-click on the executable file Automata.exe or Pumplema.exe.

To download the tutorial:
You can also download it from http://osprey.unisa.ac.za/TechnicalReports/cos2601/cos2601.zip if you want to – it is 8.7 MB.
- Go to the given web link.
- Save cos2601.zip to your computer.
- Double click on the saved cos2601.zip file.
- Choose *Extract all files* from top row of buttons on the opened page.
- Choose a destination for the extracted files.
- Click on the Extract Button.
- Double click on the cos2601 folder, double click on the Automata or Pumplema folder, and then double click on the Automata.exe or Pumplema.exe icon (it looks like a round ball with a red ribbon around it).
- You can now navigate through the tutorial.

We have tried these steps without experiencing any problems. Depending on your browser and operating system, there might be a slight variation in these steps. Ask someone experienced with computers to help you if you struggle.

Problems while running the tutorial:
- If the first screen is unattractive and spotty, your screen setting may be of a too low quality. Change your computer's display settings to 16-bit or higher.
- The bottom part of the screen and left or right of the screen appear to be "missing". Your computer is probably set at the incorrect resolution. This should be changed to at least 800x600.
- The picture on your screen appears in a window in the centre of the screen with open space around it. Your resolution setting is higher than what is required. It is not essential that you reset it. Reset it if you want a bigger picture on your screen. (See procedure above.)

Osprey web server
The School of Computing also uses the Osprey web server (http://osprey.unisa.ac.za/). The purpose of this server is to provide information about the School. It does not offer student administration services. There is also a discussion forum here that can be used.

Assessment

Assessment plan
You will be assessed in several ways in this module.

- There are three assignments.
    - The first assignment is a multiple-choice assignment, and counts 20% towards the semester mark.
    - The second assignment is a written assignment, and counts 60% towards the semester mark.
    - The third assignment is a multiple-choice assignment, and counts 20% towards the semester mark.
    - These should be done individually and not in groups.
    - These must be submitted to Unisa for marking.
    - Together, these three assignments count 20% towards the final module mark.
- There are three self-test assignments.
    - These should not be submitted.
- There is one written exam.
    - This exam counts 80% towards the final module mark.
    - You must submit at least the first assignment by the due date to obtain admission to the exam.

Why do assignments? In the first place, we need to provide proof to the Department of Higher Education and Training that you are an active student. Therefore it is compulsory to submit Assignment 01 by its due date. Furthermore, experience has shown that a student who does not work systematically during the semester is likely to give up and does not even attempt to write the examination.

Your final mark will be calculated as follows:
Semester mark (out of 100) × 20% + Examination mark (out of 100) × 80%

In order to pass this module, a final mark of at least 50% is required.

Example: The following example shows how the assessment system works, assuming that assignments 01 and 02 were submitted.

Assignment	Mark	× Weight	Contribution to semester mark
01	90%	× 0.20	18%
02	90%	× 0.60	54%
03	90%	× 0.20	18%
Semester mark			90%

The resulting semester mark is 90%.
Suppose you obtain 80% in the examination. The final module mark will be calculated as follows:
(90 × 0.20)% + (80 × 0.80)% = (18.0 + 64.0)% = 82%.

Note: The semester mark will not contribute towards the results of students writing a supplementary examination.

Due dates
You will find a list of due dates for both the hand-in and self-test assignments in tutorial letter 101.

Submission of assignments
You may submit written assignments by **post**, or hand them in at a **regional centre**, or submit them **electronically** via myUnisa. For detailed information and requirements as far as assignments are concerned, refer to *myStudies @ Unisa* which you received with your study package.

You need to follow the steps below when you submit an assignment via myUnisa:

- Go to myUnisa at https://my.unisa.ac.za
- Log on with your student number and password.
- Choose the relevant module (COS2601) in the orange block.
- Click on assignments in the menu on the left-hand side of the screen.
- Click on the assignment number of the assignment that you want to submit.
- Follow the instructions.

If you submit your assignment via the postal system, remember that you have to take into account the time the post will take to reach Unisa.

Note that administrative assignment enquiries (e.g. whether or not the university has received your assignment or the date on which an assignment was returned to you) should be addressed as stipulated in the *myStudies @ Unisa* brochure and not to the academic department.

Plagiarism of assignments

When your assignment has been marked, a percentage will be awarded. This is an indication of how correct your answers were, as well as the quality of your assignment. Copying other students' solutions, or the official solutions from a previous or current semester is plagiarism, which is a punishable offence that may lead to expulsion from the university. At the very least you will receive 0% for a plagiarised assignment.

Study plan

There is a study plan in tutorial letter 101 (also available as a PDF document in the Additional Resources), which you can use to guide your studies through the semester. By following the plan, you will be able to get all the assignments done by their due dates.

Examination

Consult your *my Studies @ Unisa* brochure for general examination guidelines and examination preparation guidelines.

Owing to regulatory requirements, in order to be considered for examination admission a student must submit at least Assignment 1.

The examination consists of one, 2-hour examination paper, and will be set in English only. You will be examined on all the content covered in the assignments (including the self-test assignments), as well as all the content in the prescribed book, unless specifically stated otherwise.

The maximum mark for each question will be indicated in brackets next to the question. However, note that the mark allocated to each question does **not** necessarily reflect the length of your solution or the time you should spend on it.

Read all the questions carefully and then answer them in any order. Remember to number the questions correctly.

Most of the examination questions will closely resemble the exercises in the prescribed book (solutions to some of these questions appear in the latter part of the study guide) and assignment questions.

Keep in mind that you had the prescribed book and learning units open next to you when you attempted the assignments. They will not be there when you write the examination paper! Therefore, when you have reached the point in your preparation for the examination where you consider yourself to have mastered the material, we suggest that you try answering past examination papers **without** consulting the books or study material.

Note: We have included an example examination paper in the learning units. There are other downloadable examination papers available on myUnisa as part of the official study material. The model solutions for these papers will not be made available. However, you are welcome to ask specific questions on specific topics that you do not understand.

We further recommend that you do not attempt to memorise anything that you do not understand during your preparation for the examination. Instead you should take the time to thoroughly understand the study material in order to be able to apply it.

During your preparation for the examination, we suggest that you pay special attention to the following topics:

- The understanding and analysis of different languages over the alphabet $\Sigma = \{a, b\}$ and the application of the Kleene closure.
- Induction (discussed in depth in the learning units).
- Recursion (recursive definitions of regular languages are discussed in detail in the learning units).
- Regular expressions: you should be able to provide a regular expression for a required language and to analyse a regular expression in order to determine the language generated.
- Machines: FAs (Finite Automata), TGs (Transition Graphs), NFAs (Non-deterministic Finite Automata), Mealy and Moore machines. You should be able to draw a required machine and to analyse a provided machine in order to determine which language is being accepted by the machine in question, or to determine the output in the case of a Mealy or a Moore machine. You should also be able to convert a Mealy machine to a Moore machine and *vice versa*. You should also be able to draw a Mealy or Moore machine when given the output of a machine.
- All the algorithms inside Kleene's theorem are of the utmost importance. Chapter 7 should be studied in detail.
- You should be able to determine the intersection of two regular languages by applying the algorithm as described in chapter 9 in Cohen.
- The application of the pumping lemma with length as described in the study guide is important.
- The theory and applications in chapter 11 should be studied.

Those of you who are considering registering for COS3701 next semester can get an idea of the contents of the module by looking at Parts II and III in *Cohen*. These two parts are the prescribed material for COS3701 (you will be informed of certain sections that should be left out.)

We wish you many fruitful hours of examination preparation and wish you success in the examination!

Activity

One of the things that myUnisa does not have is the ability for you to create an online profile for yourself. This means that other students (and even your lecturers) know very little about you.

Please take the time to go the Discussion forum, and to the Introductions forum. In a new topic, please introduce yourself to the group.

- Tell us who you are: your name, and age (if you want to).
- Maybe tell us a little about your background: where are you from, in which city you live, do you have a full-time job, and so on.
- Also, read some of the introductions and respond to the introductions of the other students.

Progressing through the learning units

Each of the learning units that follows this introductory section is divided into three parts:

- Study material: Here you will find which chapter of the prescribed book you are expected to study in the unit to meet the assessment outcomes. There will also be an indication of how long you have to work with the material in this unit in order to keep up to date with your studies.
- Notes: Any particular points of interest will be picked up in this section, and so this part may provide extra reading and other material that you should have a look at.
- Recommended problems. For each learning unit, we will suggest some exercises that you should do yourself. You will find the solutions to these problems in a separate learning unit. We urge you never to look at a solution before you have spent at least 20 minutes grappling with the problem. Remember that there is no effortless way to acquire worthwhile skills.

We suggest that you tackle each learning unit as follows: Read fairly rapidly through the chapter in the prescribed book. Do not spend too much time on something you do not understand completely. Then go carefully through the corresponding notes, doing the activities as you come across them. Finally return to the prescribed book and work systematically through the entire chapter, making sure you understand everything and then do the recommended problems provided at the end of each learning unit. Do not tackle a chapter before you have mastered the previous one. You will not understand the theory properly if you fail to do the problems.

The learning units that deal with the assignments and self-tests will be structured differently, and here you will find the following:

- Scope: This section will detail what work must be completed in order to attempt the assignment, as well as information on how to submit your answers.
- Assignment questions: This section will list all the questions that you will have to answer for the assignment. The questions will also be made available as a PDF file in the Additional Resources.
- Assignment solutions: After the closing date, a discussion of the assignment will be posted as a tutorial letter (we will let you know when it has been posted via an announcement), and the sample solutions will be posted to the Additional Resources page.

4 Learning Unit 1 - Background

Study Material

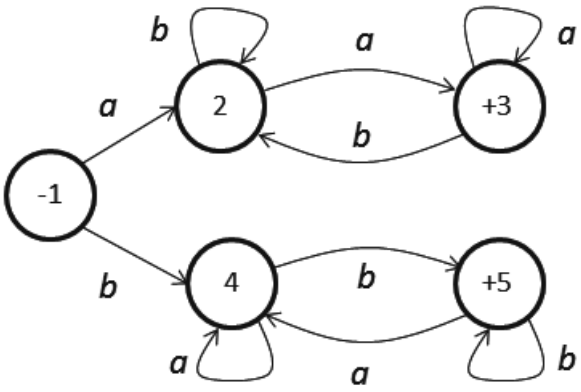
Cohen
You need to study chapter 1 in the prescribed book.

Time allocated
You will need one week to study this unit and unit 2.

Notes

Background
Some background to automata theory is given in this chapter, which we trust you will find absorbing.

In this module we define and study the class of theoretical machines known as finite automata. Although they are limited in power, they are capable of recognising numerous symbol patterns, which we identify as the class of regular languages. In the next learning unit you will learn what a “language” is. Below is a sketch of a finite automaton which recognises some specific language. You will meet these automata in learning unit 5.



Finite automata and regular language are at the lowest level of a hierarchy of machines and languages. In COS3701 a broader range of machines and languages is studied. See page 574 in Cohen to get an idea of what is covered in COS2601 and COS3701.

Recommended problems

There are no recommended problems for this learning unit.

5 Learning Unit 2 – Languages

Study Material

Cohen
You need to study chapter 2 in the prescribed book.

Time allocated
You will need one week to study unit 1 and this unit.

Notes 2.1

The idea of a formal language is introduced in this learning unit. After studying these notes, you should be able to explain

- what an alphabet, a word, a language and the empty string are;
- what is meant by the length of a string;
- how to concatenate strings;
- what the closure of a set is, and how to form it; and
- what Cohen means when he speaks of "proof by constructive algorithm".

Alphabet
For the duration of this module we will be interested only in finite alphabets, although the languages themselves may be infinite, like L_1, L_2, L_3, L_4 and PALINDROME (*Cohen*, pages 10 - 13). It is impossible to write down all the words in an infinite language, although every word is finite. Most of the languages discussed in the prescribed book are, in fact, based, like PALINDROME, on the very simple alphabet {*a b*}. Note: when in doubt, assume the alphabet is {*a b*}.

Specification of languages
The languages you will encounter in this module will generally be described in one of two ways: either an indication will be given of how any string of characters can be tested for membership of the language (and this will normally amount to describing a "machine" that can recognise strings of the language), or else a rule will be provided that indicates how the strings of the language can be constructed.

Concatenation and closure
The notion of *concatenation* is probably already familiar to you from your first-year modules. This notion is used to define the concept of *closure*. Given any non-empty set S (which may be either an alphabet or a set of words), the closure of S is the set S* consisting of all strings that can be obtained by choosing zero, one, or more strings from S and concatenating them in the order in which they are chosen. Note that S* *always contains the empty string* Λ, because that's what you get if you choose no strings from S and concatenate them. Also note that unless S is empty or the only member of S is the empty string, S* will contain an infinite number of strings. In learning unit 3 we discuss strings and concatenation in greater detail.

Activity

Activity 2.1
1. Let S = {*ab bba aa*}. Write down all words in S* consisting of four or fewer letters.
2. Provide an example of two sets S and T which are such that $S^* \cup T^* \neq (S \cup T)^*$.

Discussion of activity 2.1
1. Λ *ab aa bba abab abaa aaaa aaab*
   Note that the empty string is included.

2. Of course there are many correct answers. Below we provide two possibilities:
   S = {*a*} and T = {*b*}. In this case S* = {*a*}* contains only words consisting solely of *a*'s or Λ, and T* = {*b*}* contains words consisting solely of *b*'s or Λ, whereas (S $\cup$ T)* = {*a b*}* contains words with both *a*'s and *b*'s, together or separately, and Λ is also a word in the language.

   S = {*ab ba*} and T = {*abb ba*}. The word *ababb* is in (S $\cup$ T)* but not in either S* or T*, thus, also not in S* $\cup$ T*.

   Note that the following are NOT correct answers.
   S = {*a b ab*} and T = {*a b ba*} where S* = T* = {*a b*}*
   S = {*ab*} and T = {*a b*}
   S = {a} and T = {*aa*}.
   Are you able to see why not?

Notes 2.2

Proof by constructive algorithm
The idea of a "constructive algorithm" is quite simple. If we claim that something exists, then of course we have to justify our claim by providing proof of its existence. Many types of proofs exist. If one proves that something exists by actually showing how to build it, rather than by using "*reductio ad absurdum*" (which is used in learning unit 10) for instance, then one is said to have given a proof by constructive algorithm. The proof of theorem 5 (on page 44 in the prescribed book) is one of the easiest examples of proof by constructive algorithm. We say more about this proof in learning unit 4.

The symbol $\subseteq$
Notice that on page 18 Cohen uses the symbol $\subset$ whereas we use $\subseteq$. In the latest set theory books, $\subseteq$ stands for "is a subset of and is maybe equal to". You should change the incorrect symbol to the correct one whenever you come across it in the prescribed book. We use $\subset$ to say "is a subset of, but definitely not equal to".

Recommended problems 2.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
1, 2, and 3 on page 19.

Recommended problems 2.1 – solutions

Problem 1

S = {*a b*}. S* consists of Λ and all finite strings of *a*'s and *b*'s in any order. The strings of length 2 are

ab, *ba*, *aa* and *bb* ... $4 = 2^2$ of them.

The strings of length 3 are

aba, *abb*, *aaa*, *aab*, *baa*, *bab*, *bba*, *bbb* ... $8 = 2^3$ of them.

The number of strings of length n equals the number of combinations of length n over {*a b*}. There are 2^n such combinations, thus 2^n strings.

Problem 2

We look at the sets of strings of length $k \in \mathbb{Z}^{\geq}$ over the alphabet S = {*aa b*}. We indicate a language by T_i and the number of words in the language by $|T_i|$.

k = 0: Language $T_0 = \{\Lambda\}$, therefore $|T_0| = 1$.

k = 1: $T_1 = \{b\}$, therefore $|T_1| = 1$.

k = 2: Concatenate *aa* to the right of each string in T_0 and *b* to the right of each string in T_1. We get $T_2 = \{aa\ bb\}$, so $|T_2| = 2$.

k = 3: Concatenate *aa* to the right of each string in the language T_1 (i.e. the language with words of length 3-2) and concatenate *b* to the right of each string in T_2 (i.e. the language with words of length 3-1): $T_3 = \{baa\ aab\ bbb\}$, so $|T_3| = 3$.

k = 4: Concatenate *aa* to the right of each string in T_{4-2} and *b* to the right of each string in T_{4-1}:
$T_4 = \{aaaa\ bbaa\ baab\ aabb\ bbbb\}$, $|T_4| = 5$.

k = 5: $T_5 = \{baaaa\ aabaa\ bbbaa\ aaaab\ bbaab\ baabb\ aabbb\ bbbbb\}$, $|T_5| = 8$.

k = 6: $T_6 = \{aaaaaa\ bbaaaa\ baabaa\ aabbaa\ bbbbaa\ baaaab\ aabaab\ bbbaab\ aaaabb\ bbaabb\ baabbb\ aabbbb\ bbbbbb\}$, $|T_6| = 13$.

In general: $|T_n| = |T_{n-1}| + |T_{n-2}|$, $n \geq 2$.

Problem 3

$\{\Lambda\ ab\ ba\ abab\ abba\ baab\ baba\ ababab\ ababba\ abbaab\ baabab\ abbaba\ baabba\ \ babaab\ bababa\}$

We cannot have *aaa* because each *a* must either be preceded by a *b* or followed by a *b*. Therefore it is impossible to have three consecutive *a*'s. Similarly, *bbb* is impossible because each *b* must either be preceded by an *a* or followed by an *a*. Thus note that any string in the language can contain *at most* two consecutive *a*'s (i.e. substring *aa*). The same goes for all *b*'s (i.e. substring *bb*).

The two smallest words not in the language are *a* and *b*.

Recommended problems 2.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
7(ii) and 7(iii) on page 19.

Recommended problems 2.2 – solutions

Problem 7

(ii) $y^3 = yyy$. We are given that $y^3 \in$ PALINDROME. That means reverse(yyy) = yyy.

Suppose y has length n and we indicate its letters by $y_1y_2y_3 \ldots y_n$.

If n = 0, it follows directly that $y \in$ PALINDROME. So let n > 0
Then
reverse($y_1y_2 \ldots y_n\ y_1y_2 \ldots y_n\ y_1y_2 \ldots y_n$) = $y_ny_{n-1} \ldots y_1\ y_n \ldots y_1\ y_n \ldots y_1$.

Now we must have
$y_1 = y_n$
$y_2 = y_{n-1}$
et cetera.

So, $y \in$ PALINDROME.

(iii) Suppose z has length k and we indicate its letters by

$z_1z_2 \ldots z_k$
If k = 0, it follows directly that $z \in$ PALINDROME. So, let k > 0. Then
$z_1z_2 \ldots z_k\, z_1 \ldots z_k\, z_1 \ldots z_k \ldots z_1 \ldots z_k = z_k \ldots z_1\, z_k \ldots z_1 \ldots z_k \ldots z_1$ (n times)

Thus $z_1 = z_k$
$z_2 = z_{k-1}$
et cetera.

So, $z \in$ PALINDROME.

Recommended problems 2.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
9(ii), 11, 12 and 13 on pages 19 and 20.

Recommended problems 2.3 – solutions

Problem 9(ii)

It is clear that *bbb* $\in T \subseteq T^*$, **but** *bbb* $\notin S^*$. Therefore, $S^* \neq T^*$.

Since S* consists of all concatenations of *ab* and *bb*, and *ab*, *bb* $\in$ T implies that T* contains all concatenations of *ab* and *bb*, $S^* \subseteq T^*$.

Problem 11

(i) We know from Theorem 1 that $(S^*)^* = S^*$. Now S^+ is S* without Λ, so $(S^+)^*$ has Λ in it and all concatenations of members of S^+ in it. But these concatenations are just those of S* [ignoring Λ] because concatenations with Λ do not change them. It follows that $(S^+)^* = (S^*)^* = S^*$.

(ii) Since S^+ consists of all concatenations of words of S, without Λ, $(S^+)^+$ also has precisely all concatenations of words of S, without Λ. It follows that $S^+ = (S^+)^+$.

In both parts (i) and (ii) we can be more formal, proving mutual inclusion of the given sets in each case. Try it.

(iii) S* contains Λ and all [finite] concatenations of members of S. Since $\Lambda \in S^*$, $(S^*)^+$ also contains Λ plus precisely the concatenations in S* [without Λ]. The $^+$ adds no more to S*, i.e. $(S^*)^+ = S^*$.

S^+ has all concatenations over S without Λ. The * adds Λ to these but does not produce any other concatenations that cannot be found in S^+. Thus $(S^*)^+ = (S^+)^*$.

Problem 12

From S = {*a, bb, bab, abaab*} it is clear that any concatenations of elements of S will either contain no *b*'s or an even number of *b*'s since each element of S contains either zero *b*'s or an even number of *b*'s. In particular, the two strings given in the problem statement are *not* in S* since they contain 5 and 7 *b*'s respectively.

Problem 13

We assume that L contains at least two different nonempty words x and y. Under the assumption that $w_1w_2 \in L$ whenever $w_1 \neq w_2$, we have $xy \in L$ and $(y)(xy) \in L$, and $(x)(yxy) \in L$.
But $(x)(yxy) = (xy)(xy) \in L$, so if we take $w_1 = xy$ we have $w_1w_1 \in L$. We have found a counterexample.

Recommended problems 2.4

Do the following problems to consolidate your knowledge of the work in this learning unit:
17(i), 17(ii), and 19 on page 20.

Recommended problems 2.4 – solutions

Problem 17(i)

S* is the language of all words of even (including zero) length consisting of *a*'s and *b*'s.

Problem 17(ii)

S = {*aaa, bbb, aab, aba, baa, abb, bab, bba*}.

Problem 19

Here is a simple counterexample that shows that the algorithm does not work correctly. Choose (aba)(aba) $\in$ S*.

Step 1. Cross off *abaab*. We are left with *a*.
Step 2. We can go no further.

The algorithm then says that *abaaba* $\notin$ S*!

6 Learning Unit 3 – Recursive Definitions

Study Material

Cohen
You need to study chapter 3 in the prescribed book.

Time allocated
You will need one week to study this unit and unit 4.

Notes

The concepts of "recursive definition" and "inductive proof" are introduced in these notes. After studying this learning unit you ought to be able to

- apply the definition of recursion; and
- formulate proofs by induction.

We assume you have already read chapter 3 in Cohen. Therefore you should have an intuitive idea of what a recursive definition is. By studying this learning unit you will learn more about recursive definitions and you will also learn how to apply mathematical induction.

3.1 What makes a definition recursive?

For starters, read the first paragraph on page 21 in Cohen again. As a point of departure we now consider the recursive definition of the set EVEN given on page 21 in the prescribed book.

EVEN	The set EVEN is defined by these three rules: • Rule 1 2 is in EVEN. • Rule 2 If x is in EVEN then so is x + 2. • Rule 3 The *only* elements in the set EVEN are those that can be produced from the two rules above. It is hard to get excited about a recursive definition of EVEN, because the set EVEN has a simple alternative description, namely EVEN = { x $\mid$ x = 2n for some positive integer n}. (The set EVEN, as used in these notes, is the set of all *positive* even integer numbers, while {…-4, 2, 0, 2, 4, 6, …} is the set of *all* even integer numbers.) Recursive definitions only stir the passions when there are no simpler alternatives. However, to gain an idea of how recursive definitions work, it is best to start with easy examples, particularly since we can check whether the recursive description does indeed give us the correct set.
The word "definition"	The word "definition" is not a word to fear. A definition of something is just a very clear description of that thing. It is always possible to give more than one description of something. For example, we have seen two descriptions of EVEN so far and more exist, such as the description of EVEN as the set that remains after we have discarded the odd integers from the set of positive integers: EVEN = $\mathbb{Z}^+ \setminus$ { x $\mid$ x = 2n – 1 for some n $\in \mathbb{Z}^+$ }. (We denote the set of integers by $\mathbb{Z}$, the set of positive integers by $\mathbb{Z}^+$ and the set of non-negative integers by $\mathbb{Z}^{\geq}$.)

Recursive definitions	What makes a definition *recursive*? Do we call Cohen's definition of EVEN recursive because it has the form of three rules? Would the following be a recursive definition of EVEN? The set EVEN is defined by these three rules: • Rule 1 8 is in EVEN • Rule 2 x is in EVEN if x is an even positive integer. • Rule 3 The only elements in the set EVEN are those which must belong to EVEN by virtue of the above two rules. The above certainly does describe the set of even positive integers. Equally, certainly, it does *not* qualify as a recursive definition. (In the first example of this learning unit a recursive definition of EVEN is provided.) It is not enough (in fact, it is not even essential) to state a definition in terms of rules. To be recursive, a definition has to describe **how, by working in a certain way, one could build the set one is interested in.**
The equipment and how to build	In general, a recursive definition of a set says, "We're talking about the set that you could build by using the following *equipment* in the following way". Then the definition tells us what equipment we've got, and how we're assumed to know more or less what's going on – all the definition provides are some shorthand hints that the informed reader will be able to make sense of. What equipment should a recursive definition provide? Briefly, the idea is that one can describe a set recursively if the set has a sort of starting point or first element from which all the other elements can be obtained by applying some process over and over. It is common practice to call the starting point a *generator*, the process a *function*, and the environment in which the process is applied a *universal set*.
Part 1 of the equipment: universal set	Thus we need a *universal set*, containing all the things we're interested in, possibly together with a whole lot of things in which we're not interested. For example, in the case of EVEN we might take the set of all real numbers, $\mathbb{R}$, to be our universal set. $\mathbb{R}$ contains all the even integers, so it is a satisfactory choice, although it also contains many numbers such as ½ and √3 that are irrelevant to our purpose. It is quite common for people to say nothing about the universal set if they think that a satisfactory universal set is easy to choose, and so you will search in vain for any mention of a universal set in *Cohen's* recursive definition of EVEN. Nevertheless, it's an essential item of equipment.
Part 2 of the equipment: function	Secondly, we need a *function* that eats elements in the universal set and spits out the same sort of thing. For example, we might choose to use the function f: $\mathbb{R} \to \mathbb{R}$ defined by f(x) = x + 2. Cohen*'s* recursive definition hints, in Rule 2, that this is the function to use. He gives the formula x + 2, at any rate, although he doesn't give the domain and the codomain of the function. The informed reader will use her or his choice of universal set to provide the domain and codomain.
Part 3 of the equipment: generator	Thirdly, we need a *generator*. In other words, we have to choose an element of the universal set to be the "first" member of the set we plan to build. In the case of EVEN, we choose the number 2. Cohen's recursive definition tells us, in Rule 1, that this is what we should do. To summarise, we have the following equipment: • a universal set, namely (in this case) $\mathbb{R}$; • a function defined on the universal set, in this case the function f which eats any real number x and spits out the real number x + 2; • a generator, in this case the number 2.

How should the equipment be used?	Now let's see *how we are supposed to use* the equipment. The idea is that we want to characterise EVEN as the set which contains *nothing but 2* and the numbers one can get from 2 by means of f. (See Cohen's Rule 3.) There is a specific way to build such a set. Briefly, you form the *smallest* subset of your universal set that contains 2 and all the things obtainable from 2 by means of f. This is achieved as follows:
Two possible properties	Consider the following two properties that may be properties of a subset A of the universal set $\mathbb{R}$: • $2 \in A$; • if $x \in A$, then $f(x) = x + 2 \in A$ (i.e. A is closed under f). Certainly the set EVEN must have these properties. But there are many subsets of $\mathbb{R}$ that possess these two properties. Take the set $\mathbb{Z}$ consisting of all integers, for example. $2 \in \mathbb{Z}$, so $\mathbb{Z}$ has the first property. And if x is an integer, then certainly x + 2 is also an integer, i.e. for every $x \in \mathbb{Z}$, $f(x) = x + 2$ also belongs to $\mathbb{Z}$. Therefore $\mathbb{Z}$ is closed under f. Some of the other subsets of $\mathbb{R}$ that have the two properties are • the set of all rational numbers x such that $x \geq -5/3$; • the set of all positive integers. But all three of the subsets we've mentioned contain elements such as the number 113 that we don't want in EVEN. These subsets are too big. We need to go from subsets like $\mathbb{Z}$ to something smaller by throwing away unnecessary elements. One possibility is to take a subset like $\mathbb{Z}$ and to throw away the unnecessary elements one by one. The difficulty is that we may not be able to reach the set we're interested in by this step-by-step process, because we might start from a set, like $\mathbb{Z}$, with an infinite number of unnecessary elements. Moreover, we would like to find a *general method*, not one that works only for getting EVEN from $\mathbb{Z}$. Fortunately there is a method that can always be applied.
General method	Take the intersection of all the subsets having the two properties. Let's think about this for a moment. You know that the intersection of two subsets A and B is the subset $A \cap B = \{ x \mid x \in A \text{ and } x \in B\}$ which consists of the elements common to both A and B. We can apply this idea to find the intersection of more than two, or even an infinite number of, subsets. The set we need should consist of the elements common to all the subsets that have the two properties, therefore it has the following mathematical description: $\{ x \mid$ for every $A \subseteq \mathbb{R}$, if A has the two properties that $2 \in A$ and that A is closed under f, then $x \in A\}$.
The way to use the equipment	Here, then, is the way to *use the equipment*. The generator and the function help us to describe the subsets of the universal set that are relevant, and then we build from them the set we want by taking their intersection. Call this intersection `I` for short.

Is it the case that EVEN = I?	How do we know that I is exactly the set we were trying to define? We can use a straightforward definition of EVEN to check. Recall that the straightforward non-recursive definition given earlier was EVEN = { x | x = 2n for some n $\in \mathbb{Z}^+$}. **Proof that EVEN = I**: Firstly, we must show that EVEN $\subseteq$ I. To show this, we demonstrate that each element of EVEN is common to all subsets with the two properties. If x $\in$ EVEN, then x has the form 2n, for some positive integer n. There are two possible cases. Either n = 1 or n > 1. Suppose n = 1. Then x = 2 and so x $\in$ A for every subset A with the two relevant properties. Hence x is common to all those subsets, so x $\in$ I. Now consider the case where n > 1. Then x = 2n can be obtained by adding 2 to the number 2 a total of n - 1 times, i.e. x = f(f(…f(2)…)) where f is applied n - 1 times. So x belongs to every subset closed under f, and thus x $\in$ I. Conversely, we should show that I $\subseteq$ EVEN. To begin with, note that 2 $\in$ EVEN, since 2 = 2n for n = 1. Next, note that if x $\in$ EVEN, then x + 2 $\in$ EVEN. After all, if x $\in$ EVEN then it means that x = 2n for some integer n. But x + 2 = 2(n + 1), so x + 2 qualifies for membership of EVEN. Hence EVEN is one of the subsets with the two properties. Thus every element of I (the intersection of all such subsets) can be found in EVEN. Thus we have proved that EVEN = I.
Summary	The equipment and the way to use the equipment can be summarized as follows: EVEN = { x | x $\in$ A for every A $\subseteq \mathbb{R}$ such that 2 $\in$ A and A is closed under the function f: $\mathbb{R} \to \mathbb{R}$ defined by f(x) = x + 2}. This definition or description is recursive because of what it says, even though it doesn't look like Cohen's definition with rules. The idea of breaking up this description into rules is to make the definition more readable. So, for example, we may write: EVEN is the intersection of all subsets A of $\mathbb{R}$ which satisfy the following two conditions: • 2 $\in$ A • If x $\in$ A, then f(x) $\in$ A, where f: $\mathbb{R} \to \mathbb{R}$ is defined by f(x) = x + 2. Do you see the resemblance between the above and Cohen's description on page 21? Immediately below the definition Cohen declares that his last rule is redundant because it is always presumed. **However, we require that you always include such a rule if you give a recursive definition in Cohen's way.** (When you provide a definition in the form explained above, the "smallest subset" part takes care of Cohen's last rule.) To conclude this section, let us reiterate what a recursive definition is.

Any description of a set is a **recursive definition** if it characterises that set as the subset of some universal set that is the smallest of all those subsets which contain a certain starting point and are closed under some given function.

3.2 Using recursive definitions with confidence

All you need is practice. Let's look at another example. Suppose we want to give a nice mathematical definition of the set of positive integers. Just as was the case with EVEN, one can give a straightforward definition like

$\mathbb{Z}^+ = \{ x \mid x > 0 \text{ and } x \in \mathbb{Z}\}$.
What would a recursive description of $\mathbb{Z}^+$ look like? Is such a description possible at all?

Yes. The set $\mathbb{Z}^+$ does have an obvious starting point, namely the number 1. Furthermore, each of the other elements can be obtained by adding 1 to the previous element. For instance, 113 can be obtained by adding 1 to 112. So it appears that a recursive definition would be quite natural. Here goes an attempt.

$\mathbb{Z}^+$ is the smallest subset of the following universal set that contains the generator given below and is closed under the given function:
Universal set: $\mathbb{R}$
Generator: 1
Function: g: $\mathbb{R} \to \mathbb{R}$ defined by g(x) = x + 1.

Of course, we could write the definition in the style of Cohen:
$\mathbb{Z}^+$ is defined by the following rules:

- Rule 1: 1 is in $\mathbb{Z}^+$.
- Rule 2: If $x \in \mathbb{Z}^+$ then $x + 1 \in \mathbb{Z}^+$.
- Rule 3: Nothing else is in $\mathbb{Z}^+$.

(Note the third rule.) Just as we checked that the recursive definition of EVEN gave us the same set as the straightforward non-recursive definition, you can check that the recursive definition above gives the set $\mathbb{Z}^+$.

Additional exercise 0
Do so.

Additional exercise 1
Provide a recursive description of the set $\mathbb{Z}^{\geq}$, which, in case you have forgotten, consists of the non-negative integers 0, 1, 2, 3, …

Additional exercise 2
Provide a recursive description of the set ODD which consists of the odd positive integers 1, 3, 5, …

Up to now we have discussed a way to describe sets by saying more or less, "The set that you get if you do such and such with the following equipment." The equipment has been very primitive: a universal set such as $\mathbb{R}$, a function like f: $\mathbb{R} \to \mathbb{R}$ where f(x) = x + 2, and a generator like 1. In order to describe more exciting sets recursively, we need to use more complicated equipment. We may need a more complicated universal set, or we may need to use more than one function, or the functions may themselves be things like binary operations, or we may want more than one generator. Let's look at some examples.

Sequences	Let's look at examples that have to do with sequences. For our purposes, a sequence may be regarded as a function having either $\mathbb{Z}^{\geq}$ or $\mathbb{Z}^{+}$ as its domain. So, for instance, the function f: $\mathbb{Z}^{+} \rightarrow \mathbb{Z}$ defined by f(x) = 3x - 4 is a sequence. It is common to describe a sequence such as f by giving the first few images as well as the formula (or "general term", as it is often called): -1, 2, 5, …, 3n - 4, … From this description we can recover not only the formula defining the function, namely 3n - 4, but also the domain: The first term is -1, which can be obtained by substituting 1 for n in the formula, so the domain of the function is $\mathbb{Z}^{+}$. It is also common practice to use a variable symbol such as n or m or k instead of x or y or w when talking about sequences. But you can use x or y if you like. Now, how would we set about a recursive description of a sequence like f?
Universal set is a set of ordered pairs	Well, the first thing to bear in mind is that the set we're interested in is a function, that is, a set of ordered pairs. So our universal set should be a set of ordered pairs. One possible choice would be to take our universal set to be $\mathbb{Z} \times \mathbb{Z}$. (Yes, you may use $\mathbb{R} \times \mathbb{R}$ if you wish.) Next we want to specify a generator. The obvious thing to do is to take the pair of f that has the number 1 as first co-ordinate. The first term of the sequence is the number -1. So our generator is the pair (1, f(1)) = (1, -1). Instead of writing down the statement $(1, -1) \in f$ we could write $f(1) = -1$. What's left to specify? We need some function that eats ordered pairs in $\mathbb{Z} \times \mathbb{Z}$ and spits out new ordered pairs in $\mathbb{Z} \times \mathbb{Z}$. We also want to keep things as simple as possible, so we'll be a bit informal in talking about this function. Which function is appropriate? Well, the sequence we're interested in is an arithmetic one, by which we imply that every term is obtained from the previous term by adding some constant number. By looking at the sequence, you will see that this "common difference" is exactly 3. Therefore our function on ordered pairs should eat a pair (n, f(n)) and spit out the pair (n + 1, f(n) + 3). More concisely, $f(n + 1) = f(n) + 3$ Now we can give our recursive definition of the sequence f: The sequence f is the smallest subset of $\mathbb{Z} \times \mathbb{Z}$ such that • $f(1) = -1$ • $f(n + 1) = f(n) + 3$ Or, we could write in the style of Cohen: The sequence f is given by the rules • Rule 1: $f(1) = -1$ • Rule 2: $f(n + 1) = f(n) + 3$ • Rule 3: No other ordered pairs live in f.

Additional exercise 3
Provide a recursive description of the arithmetic sequence
4, 11, 18, …, 7n - 3, …

Geometric sequence	What about a geometric sequence like 2, -2, 2, -2, …, $(-1)^{n+1}(2)$, …? It's just as easy to handle as an arithmetic sequence. The sequence is the smallest subset of $\mathbb{Z} \times \mathbb{Z}$ such that • $x_1 = 2$ • $x_{n+1} = (-1)x_n$ The first rule tells us that 2 is the first term (or starting point) and the second rule tells us that -1 is the common ratio. Does it puzzle you to see references to x_1 and x_n rather than to f(1) and f(n)? The fact is that when people write about sequences, they often prefer to use letters such as x or a with subscripts, rather than the standard function notation.

Additional exercise 4
Provide a recursive description of the arithmetic sequence
2, 6, 18, …, $2(3)^{n-1}$, …

More than one function	We've seen some examples in which the universal set was a Cartesian product. Let's look at an example in which we have more than one function on the universal set. Consider the set $\mathbb{Z}$ of integers. How would we describe $\mathbb{Z}$ recursively? Our experience with $\mathbb{Z}^+$ suggests that $\mathbb{R}$ is an appropriate universal set. To find a suitable generator, it is convenient to think of $\mathbb{Z}$ as something that spreads out from the centre: …….|____|____|____|____|____|____|____|….. -3 -2 -1 0 1 2 3 Starting at 0, we can get positive integers by adding 1 and we can get negative integers by adding -1. So we take the number 0 as our generator. The functions we need are obvious: Take • f: $\mathbb{R} \to \mathbb{R}$ defined by f(x) = x + 1 • g: $\mathbb{R} \to \mathbb{R}$ defined by g(x) = x - 1 So $\mathbb{Z}$ is the smallest subset of $\mathbb{R}$ such that • $0 \in \mathbb{Z}$ • if $x \in \mathbb{Z}$ then $f(x) = x + 1 \in \mathbb{Z}$ • if $x \in \mathbb{Z}$ then $g(x) = x - 1 \in \mathbb{Z}$ In the style of Cohen, we may write: $\mathbb{Z}$ is defined by the rules • Rule 1: $0 \in \mathbb{Z}$ • Rule 2: If $x \in \mathbb{Z}$ then $x + 1 \in \mathbb{Z}$ • Rule 3: If $x \in \mathbb{Z}$ then $x - 1 \in \mathbb{Z}$ • Rule 4: Nothing else is in $\mathbb{Z}$.

Additional exercise 5
Provide a recursive definition of the function that, given $n \in \mathbb{Z}^+$, the sum 1 + 2 + ... + n is generated.

Additional exercise 6
Determine f(6) where f is the function defined recursively by

- f(1)= 2 and f(2) = 1
- f(n + 1) = 3f(n - 1) - 2f(n)

Solutions to additional exercises 0 – 6

Exercise 0

$\mathbb{Z}^+ = \{ x \mid x > 0 \text{ and } x \in \mathbb{Z}\}$

S = { x | for every $A \subseteq \mathbb{R}$, if A has the property that $1 \in A$ and A is closed under $f : \mathbb{R} \to \mathbb{R}$
defined by f(x) = x + 1, then $x \in A$}

We know that $\mathbb{Z} \subseteq \mathbb{R}$

Is $\mathbb{Z}^+ \subseteq S$?
Assume $x \in \mathbb{Z}^+$, then we may say x > 0. If x = 1, then $x \in S$, and if x > 1, then we apply the function f, x - 1
times:
f(f(f(...(1)...))) = x, hence $x \in S$
$\mathbb{Z}^+ \subseteq S$ follows.

Is $S \subseteq \mathbb{Z}^+$?

It is clear that $1 \in \mathbb{Z}^+$. Furthermore, if $x \in \mathbb{Z}^+$, then also $x + 1 \in \mathbb{Z}^+$ (because x + 1 > x > 0). Thus, $\mathbb{Z}^+$ is one of the subsets of $\mathbb{R}$ with the required property and therefore $S \subseteq \mathbb{Z}^+$.

Since $\mathbb{Z}^+ \subseteq S$ and $S \subseteq \mathbb{Z}^+$, it follows that $\mathbb{Z}^+ = S$ and that is what we wanted to prove.

Exercise 1

$\mathbb{Z}^{\geq}$ is the smallest subset of $\mathbb{R}$ that contains the generator 0 and is closed under the function f: $\mathbb{R} \to \mathbb{R}$ defined by f(x) = x + 1.

or

(a) $0 \in \mathbb{Z}^{\geq}$
(b) If $x \in \mathbb{Z}^{\geq}$, then $x + 1 \in \mathbb{Z}^{\geq}$
(c) The only elements in $\mathbb{Z}^{\geq}$ are those that can be produced from (a) and (b).

Exercise 2

ODD is the smallest subset of $\mathbb{R}$ that contains the generator 1 and is closed under the function f: $\mathbb{R} \to \mathbb{R}$ defined by f(x) = x + 2.

or

(a) $1 \in$ ODD
(b) If $x \in$ ODD, then also $x + 2 \in$ ODD
(c) Nothing else is in ODD.

Exercise 3

The sequence f is the smallest subset of $\mathbb{Z}$ x $\mathbb{Z}$ such that
f(1) = 4 and
f(n + 1) = f(n) + 7

or

(a) f(1) = 4
(b) f(n + 1) = f(n) + 7
(c) No other ordered pairs are in f.

Exercise 4

The sequence f is the smallest subset of $\mathbb{Z}$ x $\mathbb{Z}$ such that
$f(1) = x_1 = 2$ and
$f(n + 1) = x_{n+1} = 3x_n$

Exercise 5

The function f is the smallest subset of $\mathbb{Z}$ x $\mathbb{Z}$ such that
f(1) = 1 and
f(n + 1) = f(n) + (n + 1)

Exercise 6

f(3) = 3(2) - 2(1) = 6 - 2 = 4
f(4) = 3(1) - 2(4) = 3 - 8 = -5
f(5) = 3(4) - 2(-5) = 12 + 10 = 22
f(6) = 3(-5) - 2(22) = -15 - 44 = -59

3.3 Strings

Definition of "string"	What exactly is a string? Although we have already encountered strings in chapter 2, Cohen never actually defines strings. Indeed, one can go a long way with just an intuitive idea, but if someone were to ask you, "Yes, but what exactly do you mean when you use the word 'string'", then you would presumably like to be in a position to formulate a satisfactory answer. **Definition:** Given any set X, a *string* over X is a function f: { 1, 2, … ,n} $\rightarrow$ X where n can be any positive integer. The number n is the *length* of the string. Suppose, for example, X = {a, b, c, d}. Then the function f: {1, 2, 3} $\rightarrow$ X defined by f(1) = c, f(2) = a and f(3) = d is the string we would normally represent by writing cad and the length of the string is 3. Suppose, as second example, X = {0, 1, 2}. Then the function f: {1, 2} $\rightarrow$ X defined by f(1) = 2 and f(2) = 2 is the string we would normally represent by writing just 22. The length of the string is 2.
Λ	An important special case is the *empty string* over X, which is the function Λ: Ø $\rightarrow$ X, where the symbol Ø of course denotes the empty set. The function Λ is extremely simple. Since its domain is the empty set, it has no ordered pairs at all. In other words, the function is the empty set. In general, the length of a string is equal to the number of elements in its domain, so it makes sense to regard Λ as having length 0.

Closure	By the *set of all strings over X* we understand the set of all functions of the form f: {1, 2, …, n} $\rightarrow$ X, where n is allowed to be any positive integer, together with the function Λ. Call this set X* for short. In learning unit 2 we defined X* as the *closure* of the set X. The set of all *nonempty* strings over X, which we denote by X^+, is just the set of all f: { 1, 2, …, n} $\rightarrow$ X, with Λ excluded because its domain is not of the right kind.
Concatenation	What does one do with strings? Basically, all we want to do with these things is *concatenate* them. One can think of concatenation as a binary operation on the set of all strings. If we call this operation CONCAT for short, then it works as follows:
CONCAT	Suppose we are given a set X. CONCAT: X* $\times$ X* $\rightarrow$ X* must be able to eat any pair (f, g) where f and g are strings in X*, and CONCAT must spit out a string (let's call it CONCAT(f, g) for short) that somehow looks like f followed by g (very loosely speaking!). To make this more precise, suppose that we take f: {1, 2, …, n} $\rightarrow$ X and g: {1, 2, …, k} $\rightarrow$ X. Then CONCAT(f, g): {1, 2, …, n + k} $\rightarrow$ X is the string such that if i ≤ n then CONCAT(f, g)(i) = f(i) and if i > n then CONCAT(f, g)(i) = g(i - n).
Example	Let's see whether an example will help us to clarify this mouthful. Let X = {0, 1, 2}, and let f be the function f: {1, 2} $\rightarrow$ X defined by f(1) = 2 and f(2)=2 (in other words, f is the string 22) and let g be the function g: {1} $\rightarrow$ X defined by g(1) = 0 (so g is the string 0). Then CONCAT(f, g) is a function with domain {1, 2, 3} such that • the first element in the string CONCAT{f, g) is CONCAT(f, g)(1) = f(1) = 2 • the second element in CONCAT{f, g) is CONCAT(f, g)(2) = f(2) = 2 • the third element in CONCAT{f, g) is CONCAT(f, g)(3) = g(3 - 2) = g(1) = 0 Hence CONCAT(f, g) is the string we would normally write as 220. This is exactly what the intuitive notion of concatenation introduced by Cohen on pages 10 – 11 in the prescribed book would lead us to expect. Of course, the description of CONCAT given above is not yet complete. We ought to specify what CONCAT(f, g) is when either (or both) of f and g is the empty string Λ. This is easy. If f = Λ, then CONCAT(f, g) = g, else CONCAT(f, g) = f.
Languages are set of strings	What does all this have to do with recursive definitions? Well, on page 25 Cohen gives recursive definitions of the languages L_1, L_2, L_3 and L_4 (where L_3 = INTEGERS). In order to understand these definitions fully, we need to think about the *generator*, the *function* and the *universal set* appropriate for each.

L_1	In the case of L_1, • the universal set is X* where X = {x}; • the function is CONCAT; • the generator is x. So a recursive definition in our more pedantic style would be: L_1 is the smallest subset of {x}* such that • $x \in L_1$; • if $Q \in L_1$, then so is CONCAT(x, Q).
L_4	In the case of L_4, • the universal set is again {x}*; • the function is again CONCAT; • the generator is Λ. So a suitable recursive definition would be: L_4 is the smallest subset of {x}* such that • $\Lambda \in L_4$; • if $Q \in L_4$, then so is CONCAT(x, Q).
L_2	In the case of L_2, • the universal set is again {x}*; • the function is again CONCAT; • the generator is x. So a suitable recursive definition would be: L_2 is the smallest subset of {x}* such that • $x \in L_2$; • if $Q \in L_2$, then so is CONCAT(CONCAT((x, x), Q)).
L_3	In the case of L_3, • the universal set is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}*; • the function is CONCAT; • the generators are 1, 2, 3, 4, 5, 6, 7, 8, 9. For the first version of L_3 (described on page 11 in Cohen): L_3 is the smallest subset of {1, 2, 3, 4, 5, 6, 7, 8, 9}* such that • $1, 2, 3, 4, 5, 6, 7, 8, 9 \in L_3$; • if $Q \in L_3$, then CONCAT(Q, 0), CONCAT(Q, 1), CONCAT(Q, 2), CONCAT(Q, 3), CONCAT(Q, 4), CONCAT(Q, 5), CONCAT(Q, 6), CONCAT(Q, 7), CONCAT(Q, 8), CONCAT(Q, 9) $\in L_3$.

Activity 3.1

Provide a recursive definition for the language AB consisting of all words containing the substring *ab*, over the alphabet {*a, b*}.

Discussion of activity 3.1
Let X= {*a, b*}.
Universal set: X*
Generator: *ab*
Function: CONCAT as defined previously.

Definition:
We first give the recursive definition in our style and then in *Cohen's* style:

(i) Let AB be the smallest subset of X* such that *ab* $\in$ AB, and
if Q $\in$ AB, then CONCAT(Q, *a*), CONCAT(Q, *b*), CONCAT(*a*, Q)
and CONCAT(*b*, Q) are in AB.

(ii) Rule 1: *ab* $\in$ AB.
Rule 2: If Q $\in$ AB then CONCAT(Q, *a*) $\in$ AB.
Rule 3: If Q $\in$ AB then CONCAT(Q, *b*) $\in$ AB.
Rule 4: If Q $\in$ AB then CONCAT(*a*, Q) $\in$ AB.
Rule 5: If Q $\in$ AB then CONCAT(*b*, Q) $\in$ AB.
Rule 6: Only words generated by rules 1 to 5 are in AB.

Additional Exercise 7
Let X = {*a, b*}, and let f: {1, 2, 3} $\rightarrow$ X be the function defined by f(1) = f(3) = a and f(2) = b, and let g: {1, 2} $\rightarrow$ X be the function defined by g(1) = b, g(2) = a. Find the following strings:
CONCAT(f, g), CONCAT(f, f), CONCAT(g, f), CONCAT(g, g), CONCAT(Λ, g), CONCAT(Λ, Λ).

Additional Exercise 8
Tackle exercise 19 on page 30 in the prescribed book.

Solutions to additional exercises 7 and 8

Exercise 7

CONCAT(f, g) = *ababa*
CONCAT(f, f) = *abaaba*
CONCAT(g, f) = *baaba*
CONCAT(g, g) = *baba*
CONCAT(Λ, g) = *ba*
CONCAT(Λ, Λ) = Λ

Exercise 8(i)

The null string Λ has a length of zero (an even number) and must also be included. Our definition is:

EVENSTRING is the smallest subset of {a, b}* such that
Λ, *aa*, *ab*, *ba* and *bb* $\in$ EVENSTRING and
if w_1 and w_2 $\in$ EVENSTRING then also
CONCAT(w_1, w_2) $\in$ EVENSTRING.

or

EVENSTRING is the smallest subset of {a, b}* such that
Λ $\in$ EVENSTRING and
if w_1 $\in$ EVENSTRING, then also CONCAT(w_1, *aa*), CONCAT(w_1, *bb*),
CONCAT(w_1, *ab*), CONCAT(w_1, *ba*) $\in$ EVENSTRING.

Exercise 8(ii)

ODDSTRING is the smallest subset of {a, b}* such that
a, *b* ∈ ODDSTRING and
if w ∈ ODDSTRING then also CONCAT(*aa*, w), CONCAT(*ab*, w),
CONCAT(*ba*, w), CONCAT(*bb*, w) ∈ ODDSTRING.

Exercise 8(iii)

AA is the smallest subset of {a, b}* such that
aa ∈ AA and
if w ∈ AA then also CONCAT(w, *a*), CONCAT(*a*, w),
CONCAT(w, *b*), CONCAT(*b*, w) ∈ AA.

Exercise 8(iv)

NOTAA is the smallest subset of {a, b}* such that
Λ, *a*, *b* ∈ NOTAA and
if w ∈ NOTAA then also CONCAT(w, *b*) and
CONCAT(w, *ba*) ∈ NOTAA.

3.4 Inductive proofs

We have seen recursive definitions of sets of numbers like EVEN and of languages such as L_3. It's time to discuss the principal benefit that a recursive definition offers. Although it won't be completely clear to you at this stage, we can briefly state what this benefit is.

Principal benefit of recursive definitions	*For every recursively defined set there are properties of associated induction principles that can be used to prove interesting things about sets.* As an illustration, consider the set of positive integers. You will recall that $\mathbb{Z}^+$ is the smallest subset of $\mathbb{R}$ such that • $1 \in \mathbb{Z}^+$ and • if $k \in \mathbb{Z}^+$ then $k + 1 \in \mathbb{Z}^+$. Focus on that word "smallest". Now suppose something is true of some positive integers. For example, it is easy to see that $n^3 + 2n$ is divisible by 3 when n = 1 and also when n = 2. How could we hope to show that *all* positive integers have this property? We could certainly not show it by substituting successive values like n = 3, n = 4, and so on, because there are infinitely many positive integers. Investigate the subset of $\mathbb{Z}^+$ containing the words that do have the property (i.e. the property that if you substitute them for n in $n^3 + 2n$, then the result is a multiple of 3). Check whether this subset • contains 1 • contains k + 1 whenever the positive integer k belongs to it. If the subset has these two characteristics, then it must be *at least as big* as $\mathbb{Z}^+$. But it is a subset of $\mathbb{Z}^+$, thus it can't be any bigger than $\mathbb{Z}^+$. Therefore they must be the same size, in fact, equal. Which means that *every* positive integer must have the property we started with (namely that if you substitute it for n in the expression $n^3 + 2n$ then the result is divisible by 3).

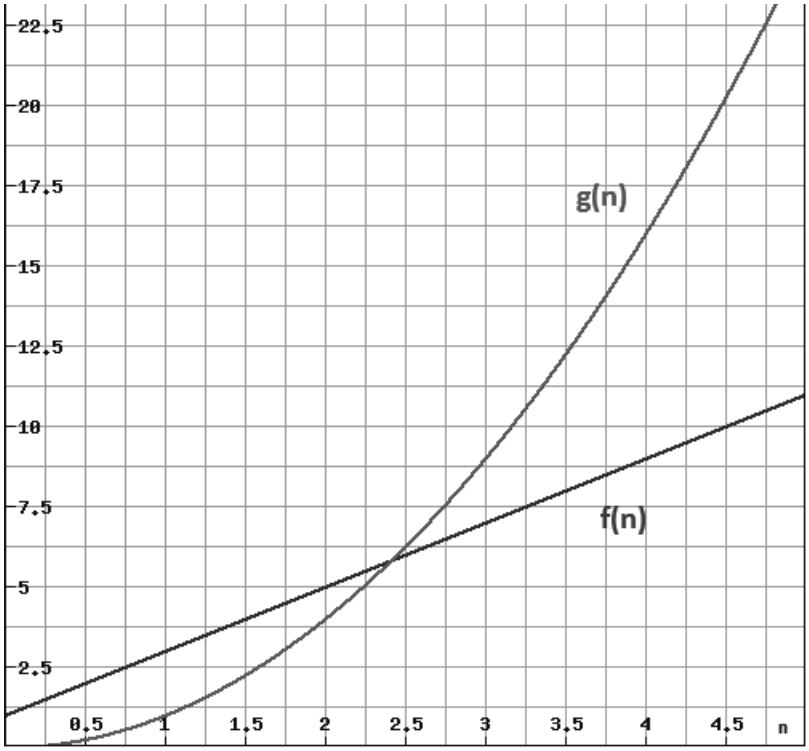
	Let's go through the whole argument in detail. **Problem**: We've discovered that *some* positive integers have an interesting property, namely that if you substitute them for n in $n^3 + 2n$, then the result is a multiple of 3. For example, if n = 2, then $n^3 + 2n$ becomes $2^3 + 2(2) = 12$, which is certainly divisible by 3. What is more, we have a feeling that maybe all positive integers have this property, and we're wondering whether there is a way to justify the feeling. Catching sight of the word "smallest" in the recursive definition of $\mathbb{Z}^+$, we proceed to argue as follows. **Our reasoning**: We will work step by step.
Step 1	We want a convenient name for the subset of numbers that have the property. Let A be the subset of $\mathbb{Z}^+$ containing all values of n such that $n^3 + 2n$ is divisible by 3, that is $A = \{ n \in \mathbb{Z}^+ \mid n^3 + 2n = 3m \text{ for some } m \in \mathbb{Z}\}$. (This is just a nice straightforward, non-recursive definition of A.)
Step 2	We check whether $1 \in A$. Well, $1^3 + 2 = 3 = 3(1)$, so 1 does indeed qualify for membership of A.
Step 3	We assume that $k \in A$ and check whether $k + 1 \in A$. Well, if $k \in A$ then we know that $k^3 + 2k$ is divisible by 3. How about $(k+1)^3 + 2(k+1)$? Hmmm... $(k+1)^3 + 2(k+1)$ $= k^3 + 3k^2 + 3k + 1 + 2k + 2$ $= (k^3 + 2k) + 3k^2 + 3k + 3$. But $(k^3 + 2k)$ and $3k^2$ and 3k and 3 are all multiples of 3, so their sum is also a multiple of 3 (because we could take out a common factor, namely 3). Hence $(k + 1)^3 + 2(k + 1)$ is a multiple of 3, which means that $k + 1 \in A$.
Step 4	We notice that $A \subseteq \mathbb{Z}^+$ since the members of A are all positive integers. On the other hand, A is a subset of $\mathbb{R}$ such that $1 \in A$ and such that $k + 1 \in A$ whenever $k \in A$. Since $\mathbb{Z}^+$ is the intersection of all such subsets, it follows that $\mathbb{Z}^+ \subseteq A$. Hence, $A = \mathbb{Z}^+$. This implies that every positive integer n has the property that $n^3 + 2n$ is divisible by 3.

The argument above is based on the following principle:
Principle of mathematical induction for $\mathbb{Z}^+$:
If a subset of $\mathbb{Z}^+$ contains the element 1 and is such that it contains k + 1 whenever it contains k, then that subset in fact equals $\mathbb{Z}^+$.

More examples	Let's look at more examples in which we use proof by induction.

An example: summation function	In Additional Exercise 5 you were asked to give a recursive definition of the summation function. $$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n.$$ Is it possible to give a non-recursive definition of this function? Let us set ourselves the task of trying to find such a closed form definition which will require thinking, but not induction, and then of showing that the formula we find gives the correct function, which will require induction, but not much thinking. Our first step must be to try to identify a pattern in the sums. 1 = 1 1 + 2 = 3 1 + 2 + 3 = 6 1 + 2 + 3 + 4 = 10 We do not see much of a pattern here. Maybe we should draw pictures? Let's visualize each sum as a pile of bricks. Then 1 looks like ▪ 1 + 2 looks like ▪ ▪ ▪ 1 + 2 + 3 looks like ▪ ▪ ▪ ▪ ▪ ▪ 1 + 2 + 3 + 4 looks like ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ Does this help us to find a pattern? Maybe. Each pile looks like a triangle. Each of the triangles looks like half a rectangle. To put it differently, from each triangle we can form a rectangle by putting another copy of the triangle on top. For instance, from the triangle representing 1 + 2 + 3 we get ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ ▪ In general, if we double a triangle representing the sum 1 + 2 + … + n we get a rectangle of n bricks wide and n + 1 bricks high. So the rectangle contains n(n + 1) bricks. Therefore the original triangle contains n(n + 1)/2 bricks. This is the pattern we were looking for.

Proof?	Can we now claim that for *every* $n \in \mathbb{Z}^+$, $1 + 2 + \ldots + n = n(n + 1)/2$? Well, we certainly have good reason to suspect that this is the case, but we don't yet have an argument that would convince a sceptic. While we ourselves may feel convinced that doubling the triangle that represents $1 + 2 + ... + n$ gives a rectangle n bricks wide and $n + 1$ bricks high, the fact remains we have *not proved* it. The situation is that, with the aid of some creative thinking helped along by pictures, we've detected a pattern, and now we can use induction to *verify* that all positive integers conform to the pattern. The argument runs as follows: We want to show that the summation function defined recursively in Additional Exercise 5 is exactly the same as the function $g: \mathbb{Z}^+ \to \mathbb{Z}^+$ given by the formula $g(n) = n(n + 1)/2$. Therefore we should show that for each element n of $\mathbb{Z}^+$, $1 + 2 + \ldots + n = g(n)$.
Step 1	Let $A = \{ n \mid n \in \mathbb{Z}^+$ and $1 + 2 + ... + n = n(n + 1)/2\}$.
Step 2	Is $1 \in A$? Well, what does the sum $1 + 2 + \ldots + n$ look like when $n = 1$? Very simple. The sum has only one term in it, namely 1. On the right-hand side of the equality, the formula $n(n + 1)/2$ reduces to $1(1 + 1)/2 = 2/2 = 1$, so the left and right sides are equal, i.e. $1 \in A$.
Step 3	Assume $k \in A$. Is $k + 1 \in A$? Well, we know that $1 + 2 + ... + k = k(k + l)/2$ because $k \in A$. Now $1 + 2 + ... + k + (k + l) = k(k + 1)/2 + (k + 1)$ $= [k(k + 1) + 2(k + l)]/2$ $= [(k + 1)(k + 2)]/2$ $= (k + 1)[(k + 1) + 1]/2$ which is precisely what one gets if one substitutes $k + 1$ for n in the formula $n(n + 1)/2$. Hence, $k + 1 \in A$.
Step 4	Thus $A = \mathbb{Z}^+$, which in turn means that for every $n \in \mathbb{Z}^+$, $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$ $= n(n + 1)/2$

Another example: algorithm analysis	Our next example is related to *algorithm analysis*. As you know from your programming modules, a task like sorting a list can be performed by a variety of algorithms. Some of these algorithms are more efficient than others. The efficiency of an algorithm can be described in terms of the number of elementary actions performed (the number of additions or the number of comparisons, for example), given an input list of length n. In other words, for each algorithm one works out its *time complexity function*, which might be something like f(n) = 2n + 1 or g(n) = n^2. And then one compares these functions to see whether the one algorithm is "better" than the other. In the case of two algorithms with time complexity functions, respectively defined by f(n) = 2n + 1 and g(n) = n^2, a glance at the graphs of the corresponding real-valued functions suggests that after a certain point, g increases more rapidly than f.  If we can *prove* that f(n) < g(n) for all n greater than a certain amount, then we would have justified the claim that the algorithm corresponding to f is more efficient than the one corresponding to g for inputs longer than the "certain amount". The question is, how can we prove that f(n) < g(n)? The answer is, you guessed it, by induction. Investigating the first few values suggests that when n gets bigger than 2 it so happens that g passes f. So let us try to prove by induction that $n^2 > 2n + 1$ for all $n \geq 3$. **Proof**:
Step 1	Let $A = \{ n \in \mathbb{Z}^+ \mid \text{if } n \geq 3 \text{ then } n^2 > 2n + 1\}$.

Step 2	Is $1 \in A$? Yes, the reason for this is interesting. The property that a number n must possess in order to qualify for membership of A has the form of an "if … then …" statement. From COS1501 you may recall that a statement of the form $p \rightarrow q$ can be rewritten in the logically equivalent form $(\neg p) \vee q$. Therefore the statement "if $n \geq 3$ then $n^2 > 2n + 1$" can be rewritten as "$n < 3$ or $n^2 > 2n + 1$". Does the value n = 1 satisfy the latter statement? Of course. 1 is less than 3, so the statement "$1 < 3$ or $1^2 > 2(1) + 1$" is true. And so the number 1 qualifies for membership of A.
Step 3	Next, assume that $k \in A$. Is it the case that $k + 1 \in A$? Well, what information do we have? We know that either $k < 3$ or $k^2 > 2k + 1$. Now let's look at k + 1. If $k + 1 < 3$, then $k + 1 \in A$. So let's concentrate on the case where $k + 1 \geq 3$. This case falls into two subcases: $k + 1 = 3$: Since $3^2 = 9$ whereas $2(3) + 1 = 7$, we see that $3^2 > 2(3) + 1$ and so $k + 1 \in A$. $k + 1 > 3$: Clearly k has to be at least 3 in this case, and since $k \in A$ this means that $k^2 > 2k + 1$. Now $(k + 1)^2 = k^2 + 2k + 1$ $> (2k + 1) + 2k + 1$ $= 2k + 2 + 2k$ $= 2(k + 1) + 2k$ $> 2(k + l) + 1$ (since $k \geq 3$).
Step 4	In all cases, the assumption that $k \in A$ allows us to infer that $k + 1 \in A$ and therefore we can conclude that $A = \mathbb{Z}^+$.

Activity 3.2

Provide the recursive definition of $\mathbb{Z}^+$ and the associated induction principle. Then apply the principle to prove that for all $n \in \mathbb{Z}^+$,

$$1^3 + 2^3 + 3^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2.$$

Discussion of activity 3.2

Recursive definition

$\mathbb{Z}^+$ is the smallest subset of $\mathbb{R}$ such that

- $1 \in \mathbb{Z}^+$ and
- if $k \in \mathbb{Z}^+$ then $k + 1 \in \mathbb{Z}^+$.

Another correct *recursive definition* is:

- Rule 1: $1 \in \mathbb{Z}^+$
- Rule 2: If $k \in \mathbb{Z}^+$, then also $k+1 \in \mathbb{Z}^+$.
- Rule 3: Only elements generated by the above rules are in $\mathbb{Z}^+$.

Induction principle

If a subset A of $\mathbb{Z}^+$ is such that $1 \in A$ and if $k \in A$, then also $k + 1 \in A$, then $A = \mathbb{Z}^+$.

Proof:
Step 1:
Define $A \subseteq \mathbb{Z}^+$ as follows:
$A = \{ n \in \mathbb{Z}^+ \mid 1^3 + 2^3 + 3^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2\}$.
We want to prove that A is actually equal to $\mathbb{Z}^+$. By definition we know that $A \subseteq \mathbb{Z}^+$. We now have to prove that $\mathbb{Z}^+ \subseteq A$. (We will use the fact, previously shown, that $1 + 2 + \ldots + k = k(k + 1)/2$.)

Step 2:
$1 \in \mathbb{Z}^+$. Is $1 \in A$? Yes, since $1^3 = 1$ and $1^2 = 1$.

Step 3:
Assume $k \in A$, thus $1^3 + 2^3 + 3^3 + \ldots + k^3 = (1 + 2 + \ldots + k)^2$
Is $k + 1 \in A$? Consider
$(1 + 2 + ... + k + (k + 1))^2 = (1 + 2 + ... + k)^2 + 2\cdot(1 + 2 + ... + k)\cdot(k + 1) + (k + 1)^2$ ①
$= (1 + 2 + ... + k)^2 + 2\cdot(k(k + 1)/2)\cdot(k + 1) + (k + 1)^2$
$= (1 + 2 + ... + k)^2 + (k^3 + 2k^2 + k) + (k^2 + 2k + 1)$
$= (1 + 2 + ... + k)^2 + (k^3 + 3k^2 + 3k + 1)$
$= (1^3 + 2^3 + ... + k^3) + (k + 1)^3$. (from the assumption)
We have proven that $k + 1 \in A$.
① Remember, $(a + b)^2 = (a^2 + 2ab + b^2)$ with $a = (1 + 2 + ... + k)$ and $b = (k + 1)$.

Step 4:
We can now conclude that $\mathbb{Z}^+ \subseteq A$, and therefore that $\mathbb{Z}^+ = A$.

Additional exercise 9
The above proof shows that $n^2 > 2n + 1$ (for $n \geq 3$) can be simplified a great deal. Firstly note that one is only interested in integers greater than 2. The set of integers greater than 2 has a recursive definition, and therefore an induction principle of its own. Provide the *recursive* definition and *induction principle* for this set, and then apply it to *prove* that $n^2 > 2n+1$ for all n in this set.
(See activity 3.3 for a similar problem.)

Activity 3.3
Provide a *recursive definition* of the set of positive integers greater than 2. Formulate the *appropriate induction principle*, and apply the latter to *prove* that
$2n + 1 < 2^n$, for all $n \geq 3$.

Discussion of activity 3.3
Recursive definition
S is the smallest subset of $\mathbb{Z}$ such that
- $3 \in S$
- and if $x \in S$, then also $x + 1 \in S$.

Induction principle
If a subset A of S contains the element 3 and is such that if it contains the element k + 1 whenever it contains the element k, then this subset A equals S.

Proof
Step 1:
Let $A = \{ n \mid n \in S, 2n + 1 < 2^n\}$

Step 2:
Is $3 \in A$? Yes, $2(3) + 1 = 7$ and $2^3 = 8$, so we have $7 < 8$.

Step 3:
Suppose $k \in A$, thus $\mathbf{2k + 1 < 2^k}$ (and $k \geq 3$).

Is $k + 1 \in A$? Let us see:
$2(k + 1) + 1 = (2k + 2) + \mathbf{1}$
$= (\mathbf{2k + 1}) + 2$
$< \mathbf{2^k} + 2$ (from our assumption)
$< 2^k + 2^k$ (since $2 < 2^k$ for $k \geq 3$)
$= 2 \cdot 2^k$
$= 2^{k+1}$
Therefore, $k + 1 \in A$.

Step 4:
We can conclude that A = S, thus that the claim is true for every element in S.

Additional exercise 10
Provide a *recursive definition* of the set of positive integers n such that $n \geq 5$. Formulate the *appropriate induction principle*, and apply the latter to *prove* that
$2^n > n^2$, for all $n \geq 5$.

Additional exercise 11
Use induction on $\mathbb{Z}^+$ to prove that for all $n \in \mathbb{Z}^+$,
a. $1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n + 1)(2n + 1)/6$
b. $1 + 5 + 9 + \ldots + (4n - 3) = n(2n - 1)$
c. $6^{n+2} + 7^{2n+1}$ is divisible by 43 without a remainder
d. $x - y$ is a factor of the polynomial $x^n - y^n$.

Solutions to additional exercises 9 – 11

Exercise 9

Recursive definition:

S is the smallest subset of $\mathbb{Z}$ such that
$3 \in S$
and if $x \in S$, then also $x + 1 \in S$.

Induction principle:

If a subset A of S contains the element 3 and is such that if it contains the element k, then it also contains the element k + 1, then this subset equals S.

PROOF:

Let $A = \{ n \mid n \in S, n^2 > 2n + 1\}$.

Is $3 \in A$? Yes, since $3^2 = 9$.
$2(3) + 1 = 7$
and $9 > 7$.

Suppose $k \in A$, thus $k^2 > 2k + 1$ and $k \geq 3$.

What about k + 1?
$(k + 1)^2 = k^2 + 2k + 1$
$> 2k + 1 + 2k + 1$
$= 2(k + 1) + 2k$

$> 2(k + 1) + 1$ since $k \geq 3$.

Therefore, $k + 1 \in A$.

We conclude that A = S.

Exercise 10

Recursive definition:

S is the smallest subset of $\mathbb{Z}$ such that
$5 \in S$, and
if $x \in S$ then also $x + 1 \in S$.

Induction principle:

If a subset A of S contains the element 5 and is such that if $k \in A$, then also $k + 1 \in A$, then this subset equals S.

PROOF:

Let $A = \{ n \mid n \in S, 2^n > n^2\}$

Is $5 \in A$? We have $2^5 = 32$ and $5^2 = 25$. Since $32 > 25$, it follows that $5 \in A$.

Assume $k \in A$, thus $2^k > k^2$ and $k \geq 5$.

What about k + 1?
$2^{k+1} = 2 \cdot 2^k > 2 \cdot k^2 = k^2 + k^2$.

In study unit 3 part 5 we proved that $k^2 > 2k + 1$ for all $k \geq 3$. This inequality certainly holds for $k \geq 5$.

Subsequently we have
$2^{k+1} > k^2 + k^2$
$> k^2 + 2k + 1$
$= (k + 1)^2$
Hence, $k + 1 \in A$.

We conclude that A = S.

Exercise 11(a)

Let $S = \{ n \mid n \in \mathbb{Z}^+, 1^2 + 2^2 + \ldots + n^2 = n(n + 1)(2n + 1)/6\}$.

Is $1 \in S$? We have $1^2 = 1$ and $1(1 + 1)(2 + 1)/6 = 6/6 = 1$, thus $1 \in S$.

Assume $k \in S$. What about k + 1? We have
$1^2 + 2^2 + \ldots + k^2 + (k + 1)^2$
$= k(k + 1)(2k + 1)/6 + 6(k + 1)^2/6$
$= (k + 1)(2k^2 + k + 6k + 6)/6$
$= (k + 1)(2k^2 + 7k + 6)/6$
$= (k + 1)(k + 2)(2k + 3)/6$
$= (k + 1)(k + 1 + 1)(2(k + 1) + 1)/6$

Hence, $k + 1 \in S$. Thus $S = \mathbb{Z}^+$.

Exercise 11(b)

Let $S = \{ n \mid n \in \mathbb{Z}^+, 1 + 5 + ... + (4n - 3) = n(2n - 1)\}$.

Is $1 \in S$? Yes, since $4 - 3 = 1$ and also $1(2 - 1) = 1$.

Assume $k \in S$, hence $1 + 5 + ... + (4k - 3) = k(2k - 1)$.

Is $k + 1 \in S$?

$$\begin{aligned} 1 + 5 + ...+ (4k - 3) + (4(k + 1) - 3) \\ &= k(2k - 1) + 4k + 4 - 3 \\ &= 2k^2 - k + 4k + 1 \\ &= 2k^2 + 3k + 1 \\ &= (k + 1)(2k + 1) \\ &= (k + 1)(2(k + 1) - 1) \end{aligned}$$

Hence, $k + 1 \in S$.

Thus $S = \mathbb{Z}^+$.

Exercise 11(c)

Let $S = \{ n \mid n \in \mathbb{Z}^+, 6^{n+2} + 7^{2n+1} = 43k \text{ where } k \in \mathbb{Z}\}$.

Is $1 \in S$? We have,

$6^{1+2} + 7^{2+1} = 6^3 + 7^3 = 216 + 343 = 559 = 43(13)$.

Hence, $1 \in S$.

Assume $m \in S$, thus $6^{m+2} + 7^{2m+1} = 43 \cdot k$ for some $k \in \mathbb{Z}$.

Is $m + 1 \in S$? Let us investigate:

$$\begin{aligned} 6^{m+1+2} + 7^{2m+2+1} &= 6 \cdot 6^{m+2} + 7^2 . 7^{2m+1} \\ &= 6^{m+2} + 7^{2m+1} + 5 \cdot 6^{m+2} + 48 \cdot 7^{2m+1} \end{aligned}$$

From our assumption we see that $6^{m+2} = 43 \cdot k - 7^{2m+1}$ and if we substitute it in the above, we have

$$\begin{aligned} 6^{m+1+2} + 7^{2m+2+1} &= 6^{m+2} + 7^{2m+1} + 5 \cdot (43 \cdot k - 7^{2m+1}) + 48 \cdot 7^{2m+1} \\ &= 43 \cdot k + 5 \cdot 43 \cdot k + (43 \cdot 7^{2m+1}). \end{aligned}$$

Every term is divisible by 43 and therefore $m + 1 \in S$.

Hence, $S = \mathbb{Z}^+$.

Exercise 11(d)

Let $S = \{ n \mid n \in \mathbb{Z}^+, x\text{-}y \text{ is a factor of } x^n - y^n\}$.

Is $1 \in S$? Yes, since $x - y = x^1 - y^1$.

Assume $k \in S$, hence $x - y$ is a factor of $x^k - y^k$.

Is $k+1 \in S$? We want to know whether $x - y$ is a factor of $x^{k+1} – y^{k+1}$. Let us have a look:

$x^{k+1} - y^{k+1}$
$= x \cdot x^k - y \cdot y^k$
$= x \cdot x^k - y \cdot y^k + y \cdot x^k - y \cdot x^k$ (our trick)
$= (x - y) \cdot x^k + y(x^k - y^k)$

It is clear that x - y is a factor of the first term and from our assumption we know that x - y is a factor of the second term.

Hence, $k + 1 \in S$.

Thus $S = \mathbb{Z}^+$.

3.5 Cohen's silent use of induction

Cohen tends to use induction without mentioning it. Let us examine the proofs of theorems 2, 3, and 4. These theorems all deal with the language of arithmetic expressions. The language AE is given a recursive definition on pages 25-26 in the prescribed book. (Note that Cohen again left out the important rule 4: "AE contains only those things which can be created by the three rules above.")

$\sum^*$	The universal set of which AE is a subset is $\sum^*$, where $\sum$ consists of the digits 0, 1, 2, …, 9 and the characters +, -, *, /, (,). From the recursive definition it is clear that AE has many generators, namely all strings of digits that represent integers. It is also clear that several functions are involved in the definition, for example the function that eats a pair of strings (x, y) and spits out the string CONCAT(CONCAT ((x, +), y)) which Cohen casually denotes by x + y. Note that the letters x and y are not part of the language AE; we are not dealing with polynomials.
Appropriate induction principle	What is the appropriate induction principle for AE? Simply this. If any subset of AE contains all the generators (i.e. the strings of digits representing integers) and is closed under the relevant functions (e.g. if x and y are strings belonging to the subset, then the concatenation x + y must belong to the subset), then the subset is equal to the whole of AE.
Proof of Theorem 2	Consider the proof of theorem 2. We want to show that no strings in AE contain the character $. Because we know that our universal set, of which AE is a subset, is the set of all functions of the form $f: \{1, 2, 3, \ldots, n\} \rightarrow \{0, 1, \ldots, 9, +, -, *, /, (,)\}$ we can immediately assert that no string containing the character $ belongs to AE (because no such string is to be found in the universal set). Cohen cannot really use our simple proof, because he hasn't said anything about the universal set or the nature of strings. In any case, his proof does say something quite interesting, namely that *even if* $\sum$ *contained the character $*, so that there were strings containing $ in $\sum^*$, AE still would not possess any strings with $ in them. The proof works as follows. Let A be the subset of AE consisting of strings without $. All the generators of AE belong to A. Moreover, A is closed under the functions mentioned in Rules 2 and 3 of the recursive definition of AE. Hence A = AE.
Proof of Theorem 3	The proof of theorem 3 is similar. Cohen implicitly considers the subset of AE, consisting of strings which neither begin nor end with /. He shows that the generators of AE all belong to the subset, and then that the subset is closed under the functions

	mentioned in Rule 2 and 3. He concludes that the subset equals AE.
Proof of Theorem 4	Theorem 4 is an attempt to avoid using induction, an attempt which fails. The key step in the proof occurs when Cohen claims that if there are strings in AE containing //, then there must be a *shortest* string in AE containing //. This sounds obvious, but if a sceptic asks you to justify the claim, you will very soon find out that you have to use induction.

3.6 Some words of caution

When we gave recursive definitions of functions, we were interested in *sequences*, i.e. functions having as domain either $\mathbb{Z}^{\geq}$ or $\mathbb{Z}^{+}$. We never checked carefully that the sets of ordered pairs we constructed were indeed functions (recall what functions are from COS1501: each input value is related to exactly one output value). In fact, all the examples we looked at were functions. But it is possible for things to go wrong, so that the recursive definition does indeed describe a subset built in a certain way, but this subset is not a function. We will not go into the matter more deeply here. However, should you do a course in set theory, recursive function theory or advanced discrete mathematics at some stage, keep your eyes open for something called the *Recursion Theorem*, which tells under which conditions you may be sure that the relation you recursively define will be a function.

Monochromatic horses?	A careless use of induction can also have its pitfalls. Consider the following "proof" that all horses are the same colour. For convenience of expression, let us introduce the term "monochromatic" (one-colour). Now we will use induction on the number of horses. Let A be the subset of $\mathbb{Z}^{+}$ consisting of number n such that every set of n horses is *monochromatic*. Then $1 \in A$, since every set of one horse is clearly monochromatic. Assume $n \in A$. Is $n + 1 \in A$? Well, take any set of n + 1 horses, and call the horses $a_1, a_2, \ldots, a_{n+1}$. Remove a_{n+1} for a moment. What remains is a set of n horses, which by assumption is monochromatic. So $a_1, a_2, \ldots, a_n$ are all the same colour, and in particular, a_1 is the same colour as a_2. Now put back a_{n+1} and this time remove a_1. What remains is a set of n horses which must of course be monochromatic. So $a_2, \ldots, a_n, a_{n+1}$ are all the same colour. Hence $a_1, a_2, \ldots, a_n, a_{n+1}$ are all the same colour (because they all have the same colour as a_2). Thus $n + 1 \in A$. So $A = \mathbb{Z}^{+}$. The result is ridiculous. We know that horses are not all the same colour. What has gone wrong?

The error	Well, recall that we used the assumption that $n \in A$ to infer that all the horses in the set $\{a_1, a_2, \ldots, a_n\}$ were the same colour. Then we used the same assumption to show that all the horses in $\{a_2, \ldots, a_{n+1}\}$ were the same colour. Then we bridged the gap between the two sets by using a_2 (in fact, any common element would have done) to show that the colour shared by the horses in $\{a_1, \ldots, a_n\}$ is the same as the colour shared by those in $\{a_2, \ldots, a_{n+1}\}$. **But what if n + 1 = 2**? Then the first set is just $\{a_1\}$ and the second set is just $\{a_2\}$, and there is no common element to tie together the colour shared by the horses of the first set with the colour shared by the horses of the second set. So the argument fails.

Additional exercise 12
Prove the following by induction and then criticise your proof:
Every set of balls containing a green ball consists entirely of green balls.

Solution to additional exercise 12

Exercise 12

PROOF:

Let A= { n | n $\in \mathbb{Z}^+$ and every set of n balls that contains a green ball, contains only green balls}.

1 $\in$ A, since a set of one green ball contains only green balls.

Assume k $\in$ A, thus assume that every set of k balls that contains a green ball, contains only green balls.

Let us now investigate a set of k + 1 balls, say

$b_1, b_2, \ldots, b_k, b_{k+1}$

that contains a green ball. Suppose the green ball is b_1. Now remove b_{k+1}. We are left with a set of k balls and one of these balls (b_1) is green. According to our assumption, all the balls must be green. In particular ball b_k is green.

Add ball b_{k+1} to our set of k balls and remove ball b_k. Again we are left with a set of k balls of which at least one ball (b_1) is green and all the other balls must also be green. In particular ball b_{k+1} is green.

Add ball b_k to this set of k balls. We now have a set of k+1 balls and we have shown that all of them are green. Hence, k + 1 $\in$ A.

Thus A = $\mathbb{Z}^+$.

ERROR:

What happens if k = 1, that is if k + 1 = 2? If we remove ball $b_{k+1} = b_2$, we are left with only ball b_1 and if we remove $b_k = b_1$, then we are left with only b_2. There exists no common ball in both of the sets to link the colour. Our so-called argument therefore fails.

Recommended problems 3.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
5, 7, and 8 on page 29.

Recommended problems 3.1 - solutions

Note: {...-4, -2, 0, 2, 4,...} is the set of even integer numbers. The set EVEN in chapter 3 is, however, the set of all positive even integer numbers, thus {2, 4, ...}.

Problem 5

We start with “EVEN is the smallest subset of $\mathbb{Z}$ such that:”
and then we have only one generator:

2 $\in$ EVEN
If x $\in$ EVEN then f(x) = x + 2 $\in$ EVEN

OR we have 2 generators:

2, 4 $\in$ EVEN
If x $\in$ EVEN then f(x) = x + 4 $\in$ EVEN

OR we have 3 generators:

2, 4, 6 $\in$ EVEN

If x $\in$ EVEN then f(x) = x + 6 $\in$ EVEN. Et cetera.

OR we have n generators:

2, 4, 6, ... 2n $\in$ EVEN.

If x $\in$ EVEN then f(x) = x + 2n $\in$ EVEN.

Problem 7

1. Any number is in POLYNOMIAL(x, y).
2. The variable x is in POLYNOMIAL(x, y).
3. The variable y is in POLYNOMIAL(x, y).
4. If p and q are in POLYNOMIAL(x, y), then so are p + q, p - q, (p) and pq.
5. The only members of POLYNOMIAL(x, y) are those that can be produced using rules 1, 2, 3 and 4.

Problem 8

(a)

x $\in$ ALEX	Rule 1
2 $\in$ ALEX	Rule 1
x + 2 $\in$ ALEX	Rule 1 or Rule 2(iii)
(x + 2) $\in$ ALEX	Rule 1 or Rule 2(i)
3 $\in$ ALEX	Rule 1
3x $\in$ ALEX	Rule 1 or Rule 2(v)
$(x + 2)^{3x} \in$ ALEX	Rule 2(vii)

(b) Elementary calculus enables us to differentiate any polynomial, and we know that

$(f(x) \pm g(x))' = f' \pm g'$

$(f(x)g(x))' = f'g + g'f$

$(f(x)/g(x))' = (f'g - g'f)/g^2$

$(f(x)^{g(x)})' = e^{glnf}\,(g'lnf + g/f(f)')$

$(f(g(x))' = df/dg \cdot dg/dx$

Since we can differentiate the basic elements (polynomials) as well as all other expressions produced by Rule 2, we can therefore differentiate all algebraic expressions.

(c) No. If we substitute x with an algebraic expression in an algebraic expression, and apply the other rules then we again have an algebraic expression.

Recommended problems 3.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
9, 10, and 13 on page 29.

Recommended problems 3.2 - solutions

Problem 9

3
(3)
(3)x (multiplication)
((3)x)
((3)x) + 7
(((3)x) + 7)
(((3)x) + 7)x (multiplication)
((((3)x) + 7)x)
((((3)x) + 7)x) – 9
(((((3)x) + 7)x) – 9)

$x^8 + x^4 = x^4(x^4 + 1) = xxxx(xxxx + 1)$

Steps:
x
xx
xxxx
xxxxxxxx
xxxxxxxx + xxxx

Hence 5 steps.

$7x^7 + 5x^5 + 3x^3 + x = (((7x^2 + 5)x^2 + 3)x^2 + 1)x$
Steps:
x
xx
$7x^2$
$7x^2 + 5$
$(7x^2 + 5)$
$(7x^2 + 5)xx$
$(7x^2 + 5)xx + 3$
$((7x^2 + 5)x^2 + 3)$
$((7x^2 + 5)x^2 + 3)x^2$
$((7x^2 + 5)x^2 + 3)x^2 + 1$
$(((7x^2 + 5)x^2 + 3)x^2 + 1)$
$(((7x^2 + 5)x^2 + 3)x^2 + 1)x$

12 steps.

Problem 10

We have to consider x^n where $n \in \{0, 1, 2, ... 30\}$.
a) ***n is even***.

Let's look at the case where n = 30:

Step 1:	$x \in$ POLYNOMIAL	(Rule 2)
Step 2:	$xx = x^2 \in$ POLYNOMIAL	(Rule 3)
Step 3:	$x^2x^2 = x^4 \in$ POLYNOMIAL	(Rule 3)
Step 4:	$x^4x^2 = x^6 \in$ POLYNOMIAL	(Rule 3)
Step 5:	$x^6x^6 = x^{12} \in$ POLYNOMIAL	(Rule 3)
Step 6:	$x^{12}x^{12} = x^{24} \in$ POLYNOMIAL	(Rule 3)
Step 7:	$x^{24}x^6 = x^{30} \in$ POLYNOMIAL	(Rule 3)

For all the other cases where n is even we follow a similar procedure. It will often be necessary to use x^8. (x^8 can be obtained in 4 steps, namely steps 1 to 3 and then $x^4x^4 = x^8$.)

Let's look at the case where n = 22:
(i) Get x^{16} in 5 steps.
(ii) $x^{16}\ x^4 = x^{20}$ (6 steps)
(iii) $x^{20}\ x^2 = x^{22}$ (7 steps)

(b) ***n is odd.***

Let's look at the case where n = 27:

Step 1:	$x \in$ POLYNOMIAL	(Rule 2)
Step 2:	$xx = x^2 \in$ POLYNOMIAL	(Rule 3)
Step 3:	$x^2x = x^3 \in$ POLYNOMIAL	(Rule 3)
Step 4:	$x^3x^3 = x^6 \in$ POLYNOMIAL	(Rule 3)
Step 5:	$x^6x^6 = x^{12} \in$ POLYNOMIAL	(Rule 3)
Step 6:	$x^{12}x^{12} = x^{24} \in$ POLYNOMIAL	(Rule 3)
Step 7:	$x^{24}x^3 = x^{27} \in$ POLYNOMIAL	(Rule 3)

For all other cases where n is odd we follow a similar procedure. Sometimes we need x^5 (which can be obtained by steps 1 to 3 and then $x^3x^2 = x^5$ (4 steps)).

Consider the case n = 23:

x
x^2
x^3
x^5
x^{10}
x^{20}
x^{23}.

Problem 13

The "problem" lies in Rule 2 at the bottom of page 25 in *Cohen* because, for example, if (x) $\in$ AE so is ((x))
by Rule 2(i) and if -(x) $\in$ AE so is -(-(x)) by Rule 2(i) and (ii). We modify Rule 2(i) as follows:
If $x \in$ AE then so is (x) provided that $x \neq (u)$ where $u \in$ AE.
Keep Rule 2(ii) as is.

Recommended problems 3.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
15 and 16 on pages 29 and 30.

Recommended problems 3.3 - solutions

Problem 15

(i) Modify Rule 1 to:
Λ, a, b $\in$ PALINDROME.

Keep Rule 2 and add a third rule:
Rule 3: The only members of PALINDROME are those produced by the above two rules.

(ii) EVENPALINDROME is the smallest subset of {a, b}* such that

$\Lambda \in$ EVENPALINDROME
If x $\in$ EVENPALINDROME, then also
axa, bxb $\in$ EVENPALINDROME.

Problem 16

(i) ODD is the smallest subset of $\mathbb{Z}$ such that
1 $\in$ ODD
If x $\in$ ODD then also f(x) = x + 2 $\in$ ODD.

(ii) S is the smallest subset of
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* such that
1, 2, 3, 4, 5, 6, 7, 8, 9 $\in$ S
If x $\in$ S, then also CONCAT(x, 0), CONCAT(x, 1), CONCAT(x, 2), CONCAT(x, 3), CONCAT(x, 4), CONCAT(x, 5), CONCAT(x, 6), CONCAT(x, 7), CONCAT(x, 8), CONCAT(x, 9) $\in$ S.

7 Learning Unit 4 – Regular Expressions

Study Material

Cohen
You need to study chapter 4 in the prescribed book.

Time allocated
You will need one week for learning unit 3 and this unit.

Notes

The concept of a "regular expression" is introduced in these notes. After studying this learning unit, you should be able to

- recognise whether an expression is regular;
- describe the language associated with a given regular expression;
- write down the regular expression that defines a given language; and
- prove that any finite language is defined by a regular expression.

Regular expression = rule for building a language	The basic idea in this learning unit is that one can sometimes describe a language by giving a very simple kind of rule that specifies how the strings belonging to the language can be built up. If you like, you may think of the rule as shorthand for a recursive definition. The kind of rule we're interested in is called a *regular expression*.
Recursive definition of regular expressions	The set of all possible regular expressions (over an alphabet $\sum$) is given by a recursive definition on pages 35-36 in Cohen. The relevant universal set is T*, where T consists of the letters in $\sum$ together with the characters +, (,), and *. The generators are the letters of $\sum$ and the empty word Λ, and there are several functions involved, namely the one that eats a string x and spits out (x), CONCAT, the function that inserts the character + between two strings, and the function that tacks on * behind a string. Note that the definition of ST given on page 41 in Cohen could be expressed very clearly with the aid of CONCAT: $ST = \{w \mid w = CONCAT(w_1, w_2) \text{ where } w_1 \in S \text{ and } w_2 \in T\}$.
Proof of theorem 5	As promised in learning unit 2, let us take a quick look at the proof of theorem 5 on page 44 of the prescribed book. This is a proof by constructive algorithm since Cohen shows how a regular expression for L *can be built*. Please note that the proof consists of a single sentence in this case. The rest of the text on pages 44 and 45, up to the ▪, is just an illustration of the given algorithm.
EVEN-EVEN	A language that you will often come across in this module is the language EVEN-EVEN. On pages 48-49 in Cohen, EVEN-EVEN is described as the language of all words containing an even number of *a*'s and an even number of *b*'s. The empty word Λ is an element of EVEN-EVEN (the length of Λ is 0, and **0 is an even number**).

The following comment is very important:
A regular expression for a language L must be able to generate **all** the words in L and **no other words**. Keep this in mind when attempting to solve the problems at the end of chapter 4 in Cohen. This also applies if you are given a regular expression **r** and you have to determine whether the language L is defined by **r**. If L is not defined by **r**, then you can prove it by providing a ***counterexample***.

Counterexample	What is a counterexample? A counterexample, in this case, is either a word that is in L but cannot be generated by **r**, or it is a word that is not in L but can be generated by **r**.

Activity

Activity 4.1

1. Does the regular expression
   a*(**ba**)***a***(**bba**)*(**b** + **bb** + Λ)**a***
   define the language of all words where the substring *bbb* does not occur anywhere?
   If not, give a counterexample.

2. Give a regular expression that generates the language consisting of all words containing exactly one occurrence of the substring *aa*.

Discussion of activity 4.1

1. When you look carefully at the given regular expression, you will find that no word containing three consecutive *b*'s can be generated. Even so, the answer to the question is NO. Why? Because the expression cannot generate *all* words without *bbb*. Many permissible words are not generated. Examples are: *bbaabb*; *bbababa*; *bbababba*.

   Are you able to see that these (permissible) words are NOT generated by the given regular expression?

2. Note that a substring such as *aaa* will not be permissible; it contains two (overlapping) double *a*'s. Somewhere in the expression we must have a substring *aa*, thus
   (something1) **aa** (something2).
   The (something1)-substring may not contain any double *a*, therefore only single *a*'s, but it may have any number of *b*'s and must end on *b* (otherwise we will have three *a*'s next to one another). The (something1)-substring may also be empty. Similarly, the (something2) may not contain any double *a*'s, but may have any number of *b*'s and must start with *b*. It may also be empty. Then our answer is (**b** + **ab**)***aa**(**b** + **ba**)*.

Recommended problems 4.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
2, 5(i), 6, 8, 9, and 10 on page 49.

Recommended problems 4.1 - solutions

Problem 2

(**b***(**aaa**)*)* or ((**aaa**)***b***)*

Problem 5(i)

(**a** + **b**)* (**aa** + **bb**)

Problem 6

(Λ + **b**)(**ab**)***aa**(**ba**)*(Λ + **b**) + (**a** + Λ)(**ba**)***bb**(**ab**)*(**a** + Λ)

The first term generates all words with a double *a*, and the second term generates all words with a double *b*.

Problem 8

a*((**b** + **bb**)**aa***)*(Λ + **b** + **bb**) **aaa a***((**b** + **bb**)**aa***)*(Λ + **b + bb**)
+
b*((**a** + **aa**)**bb***)*(Λ + **a** + **aa**) **bbb b***((**a** + **aa**)**bb***)*(Λ + **a** + **aa**)

The first term makes provision for words with at least 3 successive *a*'s at least once and never more than 2 successive *b*'s. The second term generates words with at least 3 successive *b*'s at least once and never more than 2 successive *a*'s.

Problem 9(i)

b***a***

Problem 9(ii)

Here we make provision firstly for all words not containing the substring *bba* and, secondly, for all words not containing substring *abb*:

a*(**baa***)***b*** + **b***(**a*****ab**)***a***

Problem 10

(**b*****ab*****ab*****ab***)*

Recommended problems 4.2

Do the following problem to consolidate your knowledge of the work in this learning unit:
12 on page 49.

Recommended problems 4.2 - solutions

Problem 12(i)

The given expression defines a language where every word contains at least one *a* which is followed at some stage by a *b*. This means that every word contains the substring *ab* with anything in front of it and anything following it. This is exactly what is generated by the regular expression in (i) and we may conclude that the two expressions define the same language.

Problem 12(ii)

The expression in (i) generates all possible strings except those where all the *b*'s occur before all the *a*'s. If we, therefore, add the term **b*****a*** to the expression, we include all possible strings in the language. The right-hand side defines the language that consists of all possible strings and we may conclude that the expressions on the left-hand and right-hand sides are equivalent.

Problem 12(iii)

The first term on the LHS (left-hand side) generates all strings with at least 2 *ab*-substrings
(anything) *ab* (anything) *ab* (anything)
and all strings with at least 1 *ab*-substring
(anything) *ab* (*b*'s followed by *a*'s).
The second term on the LHS generates all strings without any *ab*-substring. Again no string that can be built up from the alphabet is excluded and therefore the language is exactly the same as the language defined by the RHS (right-hand side) expression.

Problem 12(iv)

There are many beasts left. One example is

(**a** + **b**)***ab**{(**a** + **b**)***ab**[(**a** + **b**)***ab**(**a** + **b**)* + **b***a*] + **b***a*} + **b***a*.

Recommended problems 4.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
16, 17(iv), and 17(v) on page 50.

Recommended problems 4.3 - solutions

Problem 16(i)

Λ^* generates one word only, namely Λ, and therefore Λ^* and Λ are equivalent.

Problem 16(ii)

Firstly, we show that any word that can be generated by the left-hand side (LHS) can also be generated by the right-hand side (RHS) and, secondly, we show that any word that can be generated by the RHS can also be generated by the LHS.

The LHS defines a language where we have zero or more *a*'s followed by a *b* and this pattern can be repeated as many times as we like and then the word ends with zero or more *a*'s. Each of these words can also be generated by the RHS, namely *a*'s and a *b* and as many *a*'s as required from the second **a***, then a *b* again, etc. We end with *a*'s.

The RHS defines a language where each word consists of *a*'s and then a *b* and *a*'s, where the *b* and *a*'s may be repeated as many times as we like. All these words can also be generated by the LHS: Take as many *a*'s as required and then a *b*, again as many *a*'s as necessary before the following *b*, etc. Ensure that we end with *a*'s.

Note that the empty string can be generated by both sides.

Problem 16(iii)

This is exactly the same as problem 16(ii) except that we take 3 *b*'s (instead of one) each time.

Problem 17(iv)

Any word generated by the LHS looks as follows:

a(*ba* or *a*)(*ba* or *a*) ... (*ba* or *a*)*b*

where the (*ba* or *a*) term may occur any number of times. This word can also be generated by the RHS: Choose *a*. Then choose *a*'s if required or else the *b* followed by the first *a* from **aa***b. If *a*'s are now required, choose as many as necessary from **a***, else the *b* and again the first *a*. Repeat as many times as necessary and end with the *b*.

Any word generated by the RHS looks as follows:

(*a*(*a*...*a*)*b*)(*a*(*a*...*a*)*b*)...(*a*(*a*...*a*)*b*)

where we may have any number of *a*'s in the inner brackets but the (*a*(*a*...*a*)*b*) term must occur at least once. This word can also be produced by the LHS: Choose *a* and then as many *a*'s as necessary from the inner factor (**ba** + **a**)*, then *ba* and again as many *a*'s as necessary, etc. End with *b*.

Problem 17(v)

The empty string can be produced by both the LHS and RHS. Now we consider the other words. Suppose the word is generated by the LHS. It can be done in two ways.

(a) The letter *a*, followed by any number of *a*'s and *b'*s arranged in any way. This word can also be generated by the RHS: Choose, for example, the *a* from **ab*** and zero *b*'s. If the word never has two or more *a*'s next to each other, we can choose all the other *a*'s and *b*'s also from **ab***. If, however, more than one *a* occurs next to each other in one or more places, we choose the *a*'s and *b*'s as required from (**b***a**)* until we come to the very last *aa*. The first of these two *a*'s is also chosen from (**b***a**)* and the second *a* from **ab***, also as many *b*'s as required and again an *a* followed by *b*'s, etc.

(b) Alternatively the word may look as follows:
(anything)*aa*(anything).
Such words can also be produced by the RHS: The first "anything" is chosen from (**b***a**)*. We go on like this until we arrive at the very last occurrence of *aa*. The first *a* of *aa* is again chosen from the (**b***a**)* and the second a from **ab***. The other "anything" is chosen from the (**ab***)* part if (**b***a**)* generates Λ.

Now we look at words generated by the RHS: ((*b*...*b*)*a*(*b*...*b*)*a*...(*b*...*b*)*aa*(*b*...*b*)
and the whole pattern can be repeated any number of times. If we carefully study the expression we see that the word must have at least one *aa* if we start with a *b*.

These words can also be produced by the LHS: Any word starting with an *a* can be generated by the **a**(**a** + **b**)* term. If the word starts with a *b*, it must have a double *a* and thus it can be generated by the last term on the LHS.

Recommended problems 4.4

Do the following problems to consolidate your knowledge of the work in this learning unit:
18(iii), 18(v), and 19 on page 50.

Recommended problems 4.4 - solutions

Problem 18(iii)

Each word in this language has a clump of *a*'s (an odd number of them) followed by a clump of *b*'s (an odd number of them) followed by an (odd) clump of *a*'s followed by an (odd) clump of *b*'s, etc. We always start with an *a*-clump and end with a *b*-clump (so we have the same number of *a*-clumps as *b*-clumps) and the empty string is also a word in this language.

Problem 18(v)

This language has all the words of the language in (iii) but we also may have any number of *b*'s in front and/or any number of *a*'s at the end.

Problem 19(i)

What does R = SR + T mean? It means R is the union of SR and T. So

(a) $T \subseteq R$.
(b) If any element of S is concatenated with an element of R, we again get an element of R.

First we want to prove that $S^*T \subseteq R$.
Let $x \in S^*T$. Then we can write $x = wv$, where $w \in S^*$ and $v \in T \subseteq R$. If $w = \Lambda$, we have $x \in T \subseteq R$ and that is what we want to have. Say, however, $w \neq \Lambda$. Then $w = w_1w_2 \ldots w_k$, $k \geq 1$, and each $w_i \in S$.

Because $v \in R$, we have from (b) above

that $w_k v \in R$
and $w_{k-1}(w_k v) \in R$
and $w_{k-2}(w_{k-1} w_k v) \in R$
…
and $w_1(w_2 \ldots w_k v) \in R$, so $x \in R$.

We may then conclude that $S^*T \subseteq R$.

Secondly, we want to show that $R \subseteq S^*T$.
Let $x \in R$. From the premise it follows that $x \in SR$ or $x \in T$. If $x \in T$ then certainly $x \in S^*T$ since $\Lambda \in S^*$.
If $x \in SR$, then $x = v_1 w_1$ where $v_1 \in S$ and $w_1 \in R$.

If $w_1 \in R$, it follows that $w_1 \in SR$ or $w_1 \in T$. If $w_1 \in T$ then certainly $x = v_1 w_1 \in S^*T$ since $v_1 \in S$. If $w_1 \in R$, then $w_1 = v_2 w_2$ where $v_2 \in S$ and $w_2 \in R$.

If $w_2 \in R$ it follows that $w_2 \in SR$ or $w_2 \in T$. If $w_2 \in T$ then certainly $x = v_1 v_2 w_2 \in S^*T$. If $w_2 \in SR$ then $w_2 = v_3 w_3$ where $v_3 \in S$ and $w_3 \in R$.
Et cetera.

This process must stop after a finite number of steps because x is of finite length and at each step we remove a non-empty string v_i. At some stage we have

$v_m w_m \in SR$, where $v_m \in S$ and $w_m \in R$

and where w_m is the smallest possible string in R and thus cannot be written as $v_{m+1} w_{m+1}$ according to our premise $w_m \in T$. So we have that $x \in S^*T$ and may draw the conclusion that $R \subseteq S^*T$.

From the above it follows that $R = S^*T$ if $R = SR + T$ is given.

Problem 19(ii)

Let $x \in R$. Then x may be written as $x = wv$ where $w \in S^*$ and $v \in T$. If $w = \Lambda$, we have that $x \in T$, so certainly $x \in SR + T$. If $w \neq \Lambda$, we have $x = w_1 w_2 \ldots w_k v$ where $k \geq 1$ and each $w_i \in S$. Now according to our premise

$w_k v \in R$
$w_{k-1}(w_k v) \in R$
…
$w_2(w_3 \ldots w_k v) \in R$
$x = w_1(w_2 \ldots w_k v) \in SR$.

So, $x \in SR + T$ and we may conclude that $R \subseteq SR + T$.

Let $x \in SR + T$. This means either $x \in SR$ or $x \in T$. If $x \in T$, then certainly $x \in S^*T$ (because $\Lambda \in S^*$) and then we know $x \in R$. If $x \notin T$, then $x \in SR$. We may write $x = wv$ where $w \in S$ and $v \in R$. But according to our premise, we then have $v \in S^*T$, so v can be written as $v = yz$ where $y \in S^*$ and $z \in T$. So

$x = wv = wyz \in SS^*T = S^*T = R$.

Thus, $x \in R$, and we may conclude that $SR + T \subseteq R$.

From the above it follows that $R = SR + T$ if $R = S^*T$ is given.

8 Self-test A

Self-test A scope

What is covered?
This self-test covers chapters 1 – 4 in Cohen and learning units 1 – 4.

Self-test submission
Please note that you should not submit this self-test to Unisa for marking. These assignments are intended for you to test yourself. Please contact your e-tutor or one of the lecturers if you need help.

Time allocated
You will need one week to complete this self-test and assignment 1.

Due date
Check Tutorial Letter 101 for the due date for this self-test.

Self-test A Questions

Question A.1
Consider the two sets S = {*a b aa*} and T = {*a b ab*} as well as the following four statements:
A. T* is a subset of S*
B. S* is a subset of T*
C. S* = T*
D. S ∩ T = {*a*, *b*}

Select the correct answer from the list below:
1. All statements (A - D) are true.
2. All statements (A - D) are false.
3. Only statements A and B are true.
4. Only A is true.
5. Only statements A and D are true.

Question A.2
Let MULTIPLEofTHREEAA be the language defined over {*a b*} that constitutes all words with the length of a multiple of three and containing the *aa*-substring. We want to compile a recursive definition for MULTIPLEofTHREEAA. Which words are all generators of the language?
(Remember, the generators of MULTIPLEofTHREEAA should enable us to generate all the words in the language, provided that our recursive definition is correct.)
1. *baa*, *aab*, *aaa*, *bbaabb*, *abaabb*, *bbaaba* and *abaaba*
2. *aaa*, *aabaa*
3. *baa*, *aab* and *aaa*
4. *abb*, *bba* and *bbb*
5. *aaa*, *babab*, *abaab* and *baaba*

Question A.3
Consider the statements below:
A. Λ is called the empty string or null string.
B. The length of Λ is 1.
C. Λ is usually part of the alphabet.
D. The language that has no words is represented by $\emptyset$.
E. The closure of $\emptyset$ does not contain any elements.

Select the correct answer from the list below:
1. Only statements A, B and C are true.
2. Only statements A, B and C are false.

3. Only statement C is true.
4. Only statements A and C are true.
5. Only statements A and D are true.

Question A.4

Which of the following regular expressions generates only a subset of the language PALINDROME? The selected regular expression should generate only words that belong to PALINDROME but does not have to generate all the words in PALINDROME.

1. (**a** + **b**)*
2. (**aa** + **bb**)*
3. (**ababb** + **ab**)*
4. (**a** + **aa**)*
5. (**bba** + **ba**)*

Question A.5

Which of the following statements is false?

1. Hilbert was not merely satisfied that every mathematical provable result should be true; he also presumed that every true result was provable.
2. Gödel proved that there was no algorithm to provide proofs for all true statements in mathematics.
3. The development of the Turing Machine and the proof that Turing provided that there were mathematically definable fundamental questions about the Turing Machine itself that the machine cannot answer destroyed all hope of ever achieving Hilbert's program of mechanizing mathematics.
4. One of the mathematical problems Church, Kleene, Post, Markov, Von Neumann and Turing worked independently on was to determine which mathematical statements do have proofs and how to generate these proofs.
5. Alan Mathison Turing was not involved with codebreaking of Nazi code during World War 2.

Question A.6

Consider the sets S = {*a aa aaa*} and T = {*a aaa*} as well as the following five statements:

A. T* is a subset of S*
B. S* is a subset of T*
C. S* = T*
D. S* ∩ T* **=** {*a aaa*}
E. S ∈ S*

Select the correct answer from the list below:

1. All statements (A - E) are true.
2. All statements (A - E) are false.
3. Only statements A, B and C are true.
4. Only statements B and E are true.
5. Only statements A and D are true.

Question A.7

Consider the statements and regular expressions provided in A, B and C.

A. The regular expression, (**a** + **b**)***bb**(**a** + **b**)*, generates all the words where the substring *bb* occurs exactly once in the alphabet Σ = {*a b*}*.
B. The regular expression, (**ba** + **bb***a**)*, generates all the words that do not contain the *aa*-substring over the alphabet Σ = {*a b*}*.
C. The regular expression, (**ab** + **ba** + **aa** + **bb**) (**ab** + **ba** + **aa** + **bb**)*, generates all non-empty words of even length over the alphabet Σ = {*a b*}*.

Select the correct option from the list below:

1. All the statements (A-C) are true.
2. All the statements (A-C) are false.
3. Only statement C is true.
4. Only statements A and C are true.

5. Only statements A and B are true.

Question A.8

Consider the statements below:

A. The most fundamental component of computer theory is the theory of mathematical logic.
B. Georg Cantor invented set theory and discovered simultaneously some paradoxes regarding what seemed to be rigorously proven mathematical theorems.
C. Noam Chomsky created the subject of mathematical models for the description of languages.
D. The mathematical model two neurophysiologists (W. McCulloch and W. Pitts) constructed for the way in which sensory receptor organs in animals behave, was of the same nature as the one Turing invented.
E. The birth of the computer was wholly independent of other events in the previous century.

Select the correct answer from the list below:

1. All the statements (A-E) are true.
2. All the statements (A-E) are false.
3. Only statements A, B, C and D are true.
4. Only statements C and D are true.
5. Only statements A, B and C are true.

Question A.9

Which one of the following languages, L, contains the most four-letter words?

1. If Σ = {*a b*}, then L = Σ*.
2. If Σ = {*aa bb ab ba*}, then L = Σ*.
3. If Σ= {*a b c*}, then L = Σ*.
4. If Σ = {*aaa bbb ccc abc bbc cbc*}, then L = Σ*.
5. If Σ = {*aaaa bbbb*}, then L = Σ*.

Question A.10

Consider the following statements regarding any sets S and T.

A. $(S^+)^* = (S^*)^*$.
B. $(S^+)^+ = (S^+)$.
C. If $S \subseteq T$ and $T \subseteq S$, then S = T.

Select the correct answer from the list below:

1. All statements (A – C) are false.
2. Only statements A and B are true.
3. Only statements B and C are true.
4. Only statement A is true.
5. All statements (A – C) are true.

Question A.11

A recursive definition for the language ODDAA over the alphabet Σ = {*a b*} must be compiled where ODDAA consists of all words of odd length that **contain** the substring *aa*.

Give (i) an appropriate universal set,
(ii) the generator(s) of ODDAA, and
(iii) an appropriate function on the universal set, and then
(iv) use these concepts to write down a recursive definition of the language ODDAA.

Question A.12

A recursive definition for the language EVENnotBAB over the alphabet Σ = {*a b*} must be compiled where EVENnotBAB consists of all words of even length **and** does not contain the substring *bab*.

Give (i) an appropriate universal set,
(ii) the generator(s) of EVENnotBAB, and
(iii) an appropriate function on the universal set, and then
(iv) use these concepts to write down a recursive definition of the language EVENnotBAB.

Solutions to Self-test A

The solutions to the questions are presented below, one answer to a page.

Question A.1

Consider the two sets S = {*a b aa*} and T = {*a b ab*} as well as the following four statements:

A. T* is a subset of S*
B. S* is a subset of T*
C. S* = T*
D. S ∩ T = {*a b*}

Select the correct answer from the list below:

1. All statements (A - D) are true.
2. All statements (A - D) are false.
3. Only statements A and B are true.
4. Only A is true.
5. Only statements A and D are true.

Answer: Option 1

Discussion:

We are given two sets S = {*a b aa*} and T = {*a b ab*}. We begin by determining S* and T*.

S* = {*a b aa*}* = {Λ *a b aa bb ab ba aaa bbb abb bba aab baa* …}, comprising all concatenations of *a* and *b*, together and separately.
T* = {*a b ab*}* = {Λ *a b aa bb ab ba aaa bbb abb bba aab baa* …}, comprising all concatenations of *a* and *b*, together and separately.

Thus, S* = T*. We now need to determine which statements are true.

A. T* is a subset of S* is true because all elements of T* are elements of S*.
B. S* is a subset of T* is true because all elements of S* are elements of T*.
C. S* = T* is true from the above.
D. S ∩ T = {*a b*} is true because *a* and *b* are the only elements that belong to both sets S and T.

Thus, we can conclude that Option 1 should be selected.

Question A.2

Let MULTIPLEofTHREEAA be the language defined over Σ = {*a b*} that constitutes all words with a length of multiple of three and containing the *aa*-substring. We want to provide a recursive definition for MULTIPLEofTHREEAA. Which words are all generators of the language?
(Remember, the generators of MULTIPLEofTHREEAA should enable us to generate all the words in the language, provided that our recursive definition is correct.)

1. *baa*, *aab*, *aaa*, *bbaabb*, *abaabb*, *bbaaba* and *abaaba*
2. *aaa*, *aabaa*
3. *baa*, *aab* and *aaa*
4. *abb*, *bba* and *bbb*
5. *aaa*, *babab*, *abaab* and *baaba*

Answer: Option 3

Discussion:

Let us consider the above possible generators:

1. The words *baa*, *aab*, *aaa*, *bbaabb*, *abaabb*, *bbaaba* and *abaaba* are not all generators of MULTIPLEofTHREEAA. All the generators *baa*, *aab* and *aab* of the language are provided in this option, thus strictly speaking all the words in the language can be generated. However, the words

bbaabb, *abaabb*, *bbaaba* and *abaaba* are not the smallest words that are generators of the language. This option is **false**.

2. Although *aaa* is a generator of MULTIPLEofTHREEAA, all the generators are not provided – *baa* and *aab* are omitted. Furthermore, the word *aabaa* does not belong to the language at all. This option is **false**.

3. The words *baa*, *aab* and *aaa* are all the generators of MULTIPLEofTHREEAA because they are the smallest words with multiples of three length that contain the *aa*-substring.**True**.

4. The words *abb*, *bba*, and *bbb* are not generators of MULTIPLEofTHREEAA because they do not contain the *aa*-substring. This option is **false**.

5. Not all the words *aaa*, *babab*, *abaab* and *baaba* are generators of MULTIPLEofTHREEAA. Only *aaa* is a generator of the language. The words *babab*, *abaab* and *baaba* do not belong to MULTIPLEofTHREEAA at all – their length is 5. Thus this option is **false**.

From the above we can conclude that Option 3 is correct.

Question A.3

Consider the statements below:

A. Λ is called the empty string or null string.
B. The length of Λ is 1.
C. Λ is usually part of the alphabet.
D. The language that has no words is represented by $\emptyset$.
E. The closure of $\emptyset$ does not contain any elements.

Select the correct answer from the list below:

1. Only statements A, B and C are true.
2. Only statements A, B and C are false.
3. Only statement C is true.
4. Only statements A and C are true.
5. Only Statements A and D are true.

Answer: Option 5

Discussion:

Let us consider the five statements:

A. This statement is **true**. Refer to Cohen, p 8, paragraph 3.

B. This statement is **not true**. The length of the null string is per definition 0. Refer to Cohen, p 12, the fifth last line.

C. This statement is **not true**. The symbol Λ is not usually allowed to be part of the alphabet of any language. Refer to Cohen, p 8, paragraph 3.

D. This statement is **true**. Refer to Cohen, p 8, paragraph 4.

E. This statement is **not true**. The closure of an empty set contains Λ, the null string. Refer to Cohen, p 16, paragraph 7.

Only statements A and D are true. Thus Option 5 is the correct option.

Question A.4

Which one of the following regular expressions generates a subset of the language PALINDROME? The selected regular expression should generate only words that belong to PALINDROME but does not need to generate all the words in PALINDROME.

1. (**a** + **b**)*.
2. (**aa** + **bb**)*.
3. (**ababb** + **ab**)*.
4. (**a** + **aa**)*.
5. (**bba** + **ba**)*.

Answer: Option 4
Discussion:
The language PALINDROME over the alphabet Σ = {*a b*} is defined on page 13 in Cohen:
PALINDROME = {Λ and all strings *x* such that reverse(*x*) = *x*}
= {Λ *a b aa bb aaa bbb aba bab aaaa abba* …}

For options 1 to 5 we will consider the given regular expressions:

1. Let L be the language defined by (**a** + **b**)*. L is the set of all possible concatenated strings of letters from the alphabet {*a b*} and contains words that are not members of PALINDROME. We provide a counterexample: *ab*∈L, but *ab*∉PALINDROME. Thus L is **not a subset** of PALINDROME.

2. Let M be the language defined by (**aa** + **bb**)*. M is the set of words consisting of Λ and concatenations of *aa* or *bb* or both. Some words are elements of M but not of PALINDROME. We provide a counterexample: *aabb* ∈ M, but *aabb* ∉ PALINDROME. Thus M is **not a subset** of PALINDROME.

3. Let T be the language defined by (**ababb** + **ab**)*. T is the set of words consisting of Λ and concatenations of *ababb* or *ab* or both. Some words are elements of T but not of PALINDROME. We provide a counterexample: *ab* ∈ T, but *ab* ∉ PALINDROME. Thus T is **not a subset** of PALINDROME.

4. Let F be the language defined by (**a** + **aa**)*. F = {Λ *aa aaa aaaa* ...}. All these words are in PALINDROME. Due to the fact that this regular expression only generates Λ and words that are concatenations of *a*, it generates only words that belong to PALINDROME. Thus F is **a subset** of PALINDROME.

5. Let P be the language defined by (**bba** + **ba**)*. P is the set of words consisting of Λ and concatenations of *bba* or *ba* or both. Some words are elements of P but not of PALINDROME. We provide a counterexample: *ba* ∈ P, but *ba* ∉ PALINDROME. Thus P is **not a subset** of PALINDROME.

Thus, Option 4 is the correct answer.

Question A.5
Which of the following statements is **false**?
1. Hilbert was not merely satisfied that every mathematical provable result should be true; he also presumed that every true result was provable.
2. Gödel proved that there was no algorithm to provide proofs for all true statements in mathematics.
3. The development of the Turing Machine and the proof that Turing provided that there were mathematically definable fundamental questions about the Turing Machine itself that the machine cannot answer, destroyed all hope of ever achieving Hilbert's program of mechanizing mathematics.
4. One of the mathematical problems Church, Kleene, Post, Markov, Von Neumann and Turing worked independently on was to determine which mathematical statements have proofs and how to generate these proofs.
5. Alan Mathison Turing was not involved with code breaking of Nazi code during World War 2.

Answer: Option 5
Discussion:
We are given five statements:
1. This statement is **true**. Refer to Cohen page 4, paragraph 3.

2. This statement is **true**. Refer to Cohen page 4, paragraph 7.
3. This statement is **true**. Refer to Cohen page 5, paragraphs 1 and 2.
4. This statement is **true**. Refer to Cohen page 4, the last paragraph and page 5, paragraph 1.
5. This statement is **false**. Refer to Cohen page 5, paragraph 3. Turing was involved in the construction of the machine which was used in breaking the German secret code.

Thus, Option 5 should be selected.

Question A.6

Consider the sets S = {*a aa aaa*} and T = {*a aaa*} together with the following five statements:

A. T* is a subset of S*
B. S* is a subset of T*
C. S* = T*
D. S* ∩ T* **=** {*a aaa*}
E. S ∈ S*

Select the correct answer from the list below:

1. All statements (A - E) are true.
2. All statements (A - E) are false.
3. Only statements A, B and C are true.
4. Only statements B and E are true.
5. Only statements A and D are true.

Answer: Option 3

Discussion:

We are given two sets S = {*a aa aaa*} and T = {*a aaa*}. We begin by determining S* and T*.

S* = {*a aa aaa*}* = {Λ *a aa aaa aaaa aaaaa* …}, comprising all lengths of concatenations of *a* to itself together with Λ.
T* = {*a aaa*}* = {Λ *a aa aaa aaaa aaaaa* …}.

Thus, S* = T*. We now need to determine which statements are true.

A. The statement T* is a subset of S* is **true** because every element in T* is an element of S*.

B. The statement S* is a subset of T* is **true** because every element in S* is an element of T*.

C. The statement S* = T* is **true** because all elements of S* are in T*, and all elements of T* are in S*.

D. The statement S* ∩ T* = {Λ *a aa aaa aaaa aaaaa* …} (i.e. all lengths of concatenations of *a* to itself together with Λ). Thus, S* ∩ T* **=** {*a aaa*} is **not true**.

E. The statement S ∈ S* is **not true**. S is a set – in fact a subset of S*. The elements of S* are words, not sets.

Only statements A, B and C are **true**. Thus, Option 3 is correct.

Question A.7

Consider the statements provided in A, B and C.

A. The regular expression (**a** + **b**)***bb**(**a** + **b**)* generates the language over the alphabet Σ = {*a b*} consisting of all the words where the substring *bb* occurs exactly once.
B. The regular expression (**ba** + **bb***a**)* generates the language over the alphabet Σ = {*a b*} consisting of all the words that do not contain the *aa*-substring.
C. The regular expression (**ab** + **ba** + **aa** + **bb**) (**ab** + **ba** + **aa** + **bb**)* generates the language over the alphabet Σ = {*a b*} consisting of all non-empty words of even length.

Select the correct answer from the list below:
1. All the statements (A - C) are true.
2. All the statements (A - C) are false.
3. Only statement C is true.
4. Only statements A and C are true.
5. Only statements A and B are true.

Answer: Option 3
Discussion:
We should consider the three languages described in statements A, B and C. We need to determine whether any of the statements A, B, and C are **true**.

A. The regular expression (**a** + **b**)***bb**(**a** + **b**)* generates the language consisting of words where we have: (any concatenations of *a*'s and *b*'s)*bb*(any concatenations of *a*'s and *b*'s). Words generated include: *bbb*, *abbba*, *bbabb*. These words have more than one occurrence of the *bb*-substring. Thus, it generates the language over the alphabet Σ = {*a b*} consisting of all words where the substring *bb* occurs **at least** once but not **exactly** once. Therefore the given statement is **not true**.

B. The regular expression (**ba** + **bb***a)* generates the language consisting of words including Λ, *ba*, *bba*, *baba*, *babba*, *bbaba*, *bbabba*,... The words *ab* and *b* do not contain the *aa*-substring but the given regular expression cannot generate these words (there are also other words that do not contain the *aa*-substring but cannot be generated). Thus, the language over the alphabet Σ = {*a b*} does **not** contain **all** the words that do not contain the *aa*-substring. Therefore the given statement is **not true**.

C. The regular expression (**ab** + **ba** + **aa** + **bb**)(**ab** + **ba** + **aa** + **bb**)* generates the language consisting of words including *ab*, *ba*, *aa*, *bb*, *abab*, *abba*, *abaa*, *abbb*, *ababab*, ... Thus this regular expression generates the language over the alphabet Σ = {*a b*} consisting of all words that are non-empty and have even length. This statement is **true**.

Only statement C is **true**. Thus, Option 3 is the correct answer.

Question A.8
Consider the statements below:
A. The most fundamental component of computer theory is the theory of mathematical logic.
B. Georg Cantor invented set theory and discovered simultaneously some paradoxes regarding what seemed to be rigorously proven mathematical theorems.
C. Naom Chomsky created the subject of mathematical models for the description of languages.
D. The mathematical model two neurophysiologists (W. McCulloh and W. Pitts) constructed for the way in which sensory receptor organs in animals behave, was of the same nature as the one Turing invented.
E. The birth of the computer was wholly independent of other events in the previous century.

Select the correct answer from the list below:
1. All the statements (A - E) are true.
2. All the statements (A - E) are false.
3. Only statements A, B, C and D are true.
4. Only statements C and D are true.
5. Only statements A, B and C are true.

Answer: Option 3
Discussion:
We decide whether the five given statements are true or false:

A. This statement is **true**. Refer to Cohen, p 3, paragraph 5.
B. This statement is **true**. Refer to Cohen, p 3, paragraph 5.

C. This statement is **true**. Refer to Cohen, p 6, paragraph 2.
D. This statement is **true**. Refer to Cohen, p 5, second last paragraph.
E. This statement is **false**. The first computer was constructed during World War II. Refer to page 5, paragraph 3.

Only statements A, B, C and D are **true**. Thus, Option 3 is the correct answer.

Question A.9

Which one of the following languages, L, contains the most four-letter words?

1. If Σ = {*a b*}, then L = Σ^*.
2. If Σ = {*aa bb ab ba*}, then L = Σ^*.
3. If Σ = {*a b c*}, then L = Σ^*.
4. If Σ = {*aaa bbb ccc abc bbc cbc*}, then L = Σ^*.
5. If Σ = {*aaaa bbbb*}, then L = Σ^*.

Answer: Option 3

Discussion:

Let us consider Σ and L in each case.

1. If Σ = {*a b*}, then L = {Λ *a b aa bb ab ba* …} comprising all concatenations of *a* and *b*, together and separately, and Λ. Thus, there are 2^4 = 16 different four-letter words.

2. If Σ = {*aa bb ab ba*}, then L = {Λ *aa bb ab ba aaaa bbbb abab baba* …}. There are 4^2 = 16 different four-letter words.

3. If Σ = {*a b c*}, then L = {Λ *a b c aa bb cc ab ac ba bc ca cb* …} comprising all concatenations of *a*, *b*, and *c*, together and separately. Thus, there are 3^4 = 81 different four-letter words.

4. If Σ = {*aaa bbb ccc abc bbc cbc*}, then L = {Λ *aaa bbb ccc abc bbc cbc aaaaaa aaabbb* …}. All the defined words are of a length that is a multiple of three. Thus, there are no four-letter words.

. If Σ = {*aaaa bbbb*}, then L = {Λ *aaaa bbbb aaaaaaaa bbbbbbbb* …}. Thus, only 2 four-letter words belong to Σ.

If Σ = {*a b c*}, then 81 different four-letter words are generated. Therefore Option 3 is the correct answer.

Question A.10

Consider the following statements regarding any languages S and T over the alphabet
Σ = {*a b*}.

D. $(S^+)^* = (S^*)^*$.
E. $(S^+)^+ = S^+$.
F. If $S \subseteq T$ and $T \subseteq S$, then S = T.

Select the correct answer from the list below:

1. All statements (A - C) are false.
2. Only statements A and B are true.
3. Only statements B and C are true.
4. Only statement A is true.
5. All statements (A - C) are true.

Answer: Option 5

Discussion:

We are given any languages S and T. Let us consider all the statements.

A. Let us first look at an example. Let S = {*a*}.
$(S^+)^* = \{a^+\}^* = \{a\ aa\ aaa\ \ldots\}^* = \{\Lambda\ a\ aa\ aaa\ \ldots\}$

$(S^*)^* = \{a^*\}^* = \{\Lambda\ a\ aa\ aaa\ \ldots\}^* = \{\Lambda\ \Lambda\Lambda\ \ldots\ a\ aa\ aaa\ \ldots\} = \{\Lambda\ a\ aa\ aaa\ \ldots\}$
Thus, $(S^+)^* = (S^*)^*$ in this example.

Let S be any language. Consider the statement, $(S^+)^* = (S^*)^*$. (We do not give a formal proof.) $(S^*)^* = S^*$ according to the theorem on page 18 of Cohen. If S is a language without the word Λ, then S^+ is the language S^* without the word Λ. If we determine $(S^+)^*$, the word Λ is included in the language together with all the words of S^+, thus $(S^+)^* = S^*$. If S is the language that contains Λ, then $S^+ = S^*$ and thus $(S^+)^* = (S^*)^*$. This statement is **true** for any set S. Refer to Cohen, pages 17 and 18.

B. Let us first look at an example. Let $S = \{a\}$.
$(S^+)^+ = \{a^+\}^+ = \{a\ aa\ aaa\ \ldots\}^+ = \{a\ aa\ aaa\ \ldots\}$
$S^+ = \{a^+\} = \{a\ aa\ aaa\ \ldots\}$
Thus, $(S^+)^+ = (S^+)$ in this example.

Let S be any language. Consider the statement, $(S^+)^+ = S^+$. (We do not provide a formal proof.) If S is a language without the word Λ, then S^+ is the language S^* without the word Λ. If we determine $(S^+)^+$ the word Λ is excluded from this language. Only the words of S^+ appear in the final set of words $(S^+)^+$, thus $(S^+)^+ = S^+$. If S is the language that contains Λ, then $S^+ = S^*$ and thus $(S^+)^+ = (S^*)^+ = S^+$ (Λ does not appear in the final set of words). This statement is **true** for any set S. Refer to Cohen, pp 17 and 18.

C. Consider the statement, *If $S \subseteq T$ and $T \subseteq S$, then $S = T$*. Let T and S be any languages.
If $S \subseteq T$ then every element of S is an element of T, and if $T \subseteq S$ then every element of T is an element of S. Thus S = T according to the principles of set theory. This statement is **true** for any sets S and T.

All three statements A, B and C are true. Therefore Option 5 is the correct answer.

Question A.11
A recursive definition for the language ODDAA over the alphabet $\Sigma = \{a\ b\}$ must be compiled where ODDAA consists of all words of odd length that **contain** the substring *aa*.
Give (i) an appropriate universal set,
(ii) the generator(s) of ODDAA, and
(iii) an appropriate function on the universal set, and then
(iv) use these concepts to write down a recursive definition of the language ODDAA.

Answer
(i) The set $\{a\ b\}^*$ will be suitable because it contains, along with other words, all the words that are in the language ODDAA.
(ii) The generators of ODDAA should be of odd length and should contain the substring *aa*. Thus *aaa*, *aab* and *baa* are suitable generators. Λ is not odd and does not contain the *aa*-substring, therefore Λ is not a word in ODDAA. Note that the generator(s) is/are always the smallest word(s) in a language.
(iii) The function CONCAT as defined in learning unit 3 will be suitable.
(iv) We give two possible recursive definitions. Keep in mind that the length of all words in ODDAA should be odd. The generators all have an odd number of letters. Therefore, to keep the length of generated words odd, and to enable us to generate all possible words belonging to ODDAA, we concatenate two letters at a time.

ODDAA is the smallest subset of $\{a\ b\}^*$ such that
aaa, *aab*, *baa* $\in$ ODDAA, and
if *w* $\in$ ODDAA, then also
CONCAT(*w*, *aa*), CONCAT(*aa*, *w*),
CONCAT(*w*, *bb*), CONCAT(*bb*, *w*),
CONCAT(*w*, *ab*), CONCAT(*ab*, *w*),
CONCAT(*w*, *ba*), CONCAT(*ba*, *w*) $\in$ ODDAA.
or

Rule 1: *aaa*, *aab*, *baa* $\in$ ODDAA.
Rule 2: If *w* $\in$ ODDAA, then also
CONCAT(*w*, *aa*), CONCAT(*aa*, *w*),
CONCAT(*w*, *bb*), CONCAT(*bb*, *w*),
CONCAT(*w*, *ab*), CONCAT(*ab*, *w*),
CONCAT(*w*, *ba*), CONCAT(*ba*, *w*) $\in$ ODDAA.
Rule 3: Only words generated by rules 1 and 2 are in ODDAA.

Question A.12

A recursive definition for the language EVENNOTBAB over the alphabet Σ = {*a b*} must be compiled where EVENNOTBAB consists of all words of even length **and** not containing the substring *bab*.

Give (i) an appropriate universal set,
(ii) the generator(s) of EVENNOTBAB, and
(iii) an appropriate function on the universal set, and then
(iv) use these concepts to write down a recursive definition of the language EVENNOTBAB.

Answer

(i) The set {*a b*}* will be suitable because it contains, along with other words, all the words that are in the language EVENNOTBAB.

(ii) The generators should be of even length and should **not** contain the substring *bab*. One possible choice is Λ. Another possible choice is *aa*, *ab*, *ba* and *bb*. Note that zero is an even number.

(iii) The function CONCAT as defined in learning unit 3 will be suitable.

(iv) We give two possible recursive definitions. Note that all words in EVENNOTBAB should have an even number of letters. The generators all have an even number of letters, therefore, to keep the length of new words even, we concatenate two letters at a time.

EVENNOTBAB is the smallest subset of {*a b*}* such that
Λ $\in$ EVENNOTBAB, and
if *w* $\in$ EVENNOTBAB, then also
CONCAT(*w*, *aa*) $\in$ EVENNOTBAB,
if *w* $\in$ EVENNOTBAB and ***w* does not end on a *ba***, then
CONCAT(*w*, *bb*), CONCAT(*w*, *ba*) $\in$ EVENNOTBAB, and
if *w* $\in$ EVENNOTBAB and ***w* does not end on a *b***, then
CONCAT(*w*, *ab*) $\in$ EVENNOTBAB.

or

Rule 1: Λ, *aa*, *ab*, *ba*, *bb* $\in$ EVENNOTBAB.
Rule 2: If *w* $\in$ EVENNOTBAB, then also
CONCAT(*aa*, *w*), CONCAT(*w*, *aa*) $\in$ EVENNOTBAB,
if *w* $\in$ EVENNOTBAB and ***w* does not start with an *ab***, then
CONCAT(*bb*, *w*), CONCAT(*ab*, *w*) $\in$ EVENNOTBAB,
if *w* $\in$ EVENNOTBAB and ***w* does not start with a *b***, then
CONCAT(*ba*,*w*) $\in$ EVENNOTBAB,
Rule 3: Only words produced by rules 1 and 2 are in EVENNOTBAB.

9 Assignment 1

Assignment 1 scope

What is covered?
This assignment is a multiple-choice question (MCQ) assignment, and covers chapters 1 – 4 in Cohen.

Assignment submission
This assignment should be submitted either electronically via myUnisa (the preferred route), or by filling in a mark-reading sheet and submitting it via one of the regional centres or the post office.

Time allocated
You will need one week to complete self-test A and this assignment.

Due date
Check Tutorial Letter 101 for the due date and unique assignment number for this assignment.

Assignment 1 questions

You can find the assignment question in tutorial letter 101.

Assignment 1 solutions

After the closing date, a discussion of the assignment will be posted to the Additional Resources page. You will be informed of this via an announcement on myUnisa.

10 Learning Unit 5 – Finite Automata

Study Material

Cohen
You need to study chapter 5 in the prescribed book.

Time allocated
You will need one week to study this unit.

Vodcasts
You can find a video demonstrating a tool (JFLAP) which you can use to build and test finite automata in the Videos folder in Additional Resources.

Notes

These notes will introduce the concept of a "finite automaton" (or FA). After studying this learning unit, you should be able to

- check whether something is a finite automaton;
- test whether a given word is accepted by a given finite automaton;
- design a finite automaton to recognise (i.e. accept, or define) a given language; and
- describe the language accepted by a given finite automaton.

Components of an FA	Cohen defines a finite automaton on page 53 and again on pages 55 - 56 to be a structure built up from several components. There is a set S of *states*, containing in particular the *start state* s_0 and having a subset F of *final states*. Then there is also a set $\sum$ of allowable *input letters* and, most important of all, there is a set T of *transitions*. You may think of each transition as a rule that says, "If you are in this state and you read this input letter you must tootle off to that state". A very simple way to write such a rule is as an ordered pair ((s, c), t) where s, t $\in$ S and c $\in \sum$. Since we must have a transition for every state and every input letter, and since the transitions give us no choice about where to go, the set T turns out to be a *function* from S $\times \sum$ to S. In COS1501 you learned that a function is a special type of relation. The relation from S $\times \sum$ to S is a set of ordered pairs. The first co-ordinate is also an ordered pair with its first entry a state from S and its second entry a letter from $\sum$. The second co-ordinate of each element of the relation is a state from S. A *function* from S $\times \sum$ to S is thus a relation in which *each* element of the domain (S $\times \sum$) is mapped onto *one* element of the co-domain. (At the top of page 56 in the prescribed book this function is indicated by δ.)
5-tuple	So the definition of a finite automaton could be expressed very concisely as a 5-tuple: FA = (S, $\sum$, T, s_0, F) where S is a non-empty finite set (called the set of states), $\sum$ is a finite set (called the alphabet of input letters), T: S $\times \sum \rightarrow$ S (called the set of transitions), $s_0 \in$ S (called the start state) and F $\subseteq$ S (called the set of final states).

Examples of the function T	In the case of the example on page 54 in Cohen, T is the function such that T(x, a) = y T(x, b) = z T(y, a) = x T(y, b) = z T(z, a) = z T(z, b) = z.
Pictures	Let us indicate each state in S with a circle. The function can be represented by drawing edges between the states (or a loop on a state). Label each edge with a letter from ∑. Since we are representing a function, *there must be **precisely one** edge for each letter in ∑ leaving **every** state. Furthermore, each edge leaving a particular state must have a different label*. This pictorial representation is called the transition diagram of the finite automaton.
Transition diagrams ≠ transition graphs	Please take note that these transition diagrams are NOT the same as the transition graphs (TGs) defined in the next learning unit. These transition diagrams are indeed examples of transition graphs, but not every transition graph is a graph of a finite automaton. In other words, every FA is a special case of a transition graph but not every transition graph is an FA. The difference between the two will be explained in the next learning unit.

The following comment is important:
The empty word, Λ, is *not* allowed to be a label of a transition (i.e. an edge) of an FA.

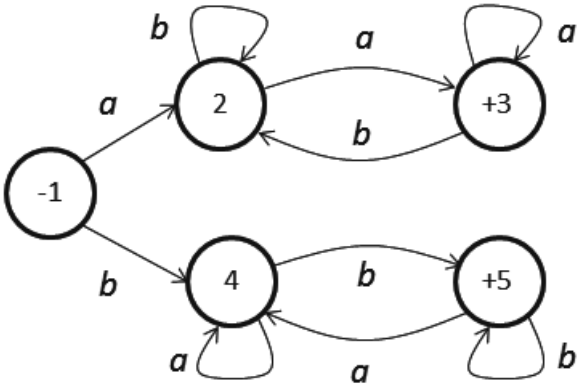
Specification of a language	In learning unit 2 we mentioned two ways in which a language can be specified. We considered the second way (that is, regular expressions) in learning unit 4. We start the discussion of the other way in this learning unit – i.e. establishing a machine that tests whether a given string of letters is a word in a language.
An example of an FA	Several FAs are built in the prescribed book. Let us build a different one here:

Activity

Activity 5.1
Build an FA that accepts the language consisting of all words which start and end on the same letter. All words in the language must have at least two letters.

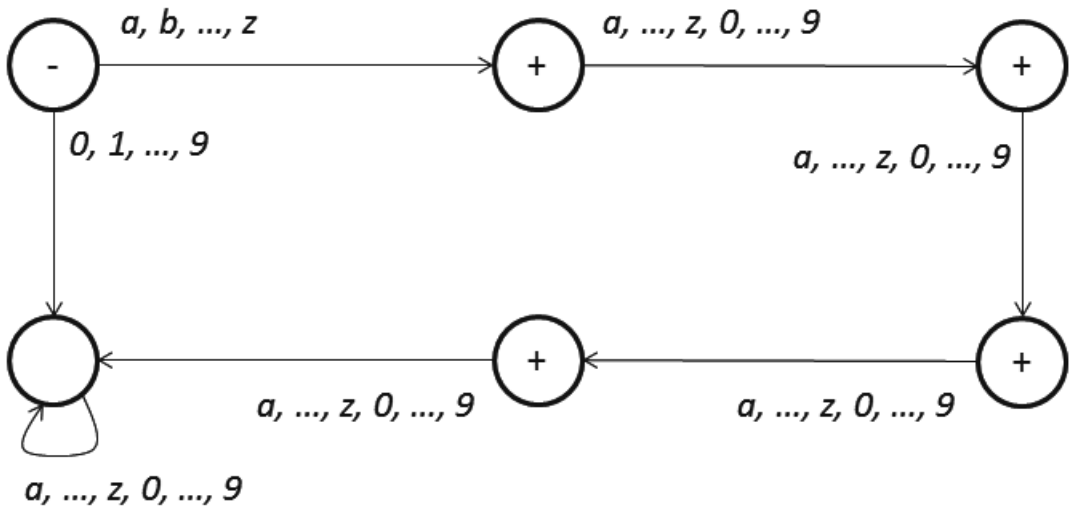
Discussion of Activity 5.1
Which words are members of this language? Well, when the first letter of a word is *a*, then the last letter must also be *a* and if the first letter is *b*, then the last letter must also be *b*. We will therefore have two separate edges leaving the start state, say state 1, one with label *a* and the other with label *b*. Suppose an *a* takes us to state 2. On reading an *a* in state 2 we move to a final state, say state 3, because this *a* may be the last letter of the word and if so, the word must be accepted (recall that the first letter was an *a*). However, if we read a *b* in state 2 we should not move to a final state. It is unnecessary to go to a different state – we can simply stay in state 2. What should happen if we read an *a* in state 3? We stay there since the word should be accepted if this *a* is the last letter of the word. Suppose we read a *b* in state 3. Then we cannot stay there because if this is the last letter, the word should not be accepted. We don't have to create a new state; we move back to state 2.

Let us return to the start state and suppose we read a *b*. We have to move to a new state, say state 4. We stay in state 4 on reading an *a* but move to a final state, say state 5, on reading a *b*. We stay in state 5 if we read a *b* there, but move back to state 4 if we read an *a*. We can now draw the machine. *Note that each state has exactly two outgoing edges, one with label* a *and one with label* b.



Another example

A second example of an FA	This notion of describing a language by giving a machine that accepts precisely that language is important in compiler design. Suppose, for instance, that we have a programming language whose variable names can consist of one, two, three or four characters. Suppose, moreover, that the first character must be a letter and that the remainder may be either letters or digits. We can easily construct a machine (given below) that will test whether a given string is an acceptable variable name in the language. Our alphabet is {*a, b, c, …, z, 0, 1, 2, …, 9*}.

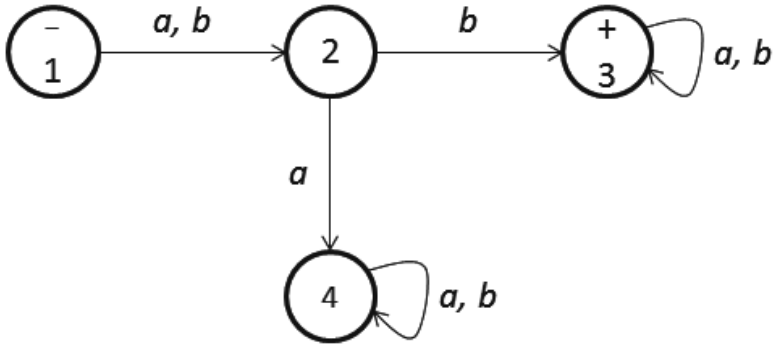


Recommended problems 5.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
2, 5(i), 5(ii), and 6 on page 71.

Recommended problems 5.1 – solutions

Problem 2

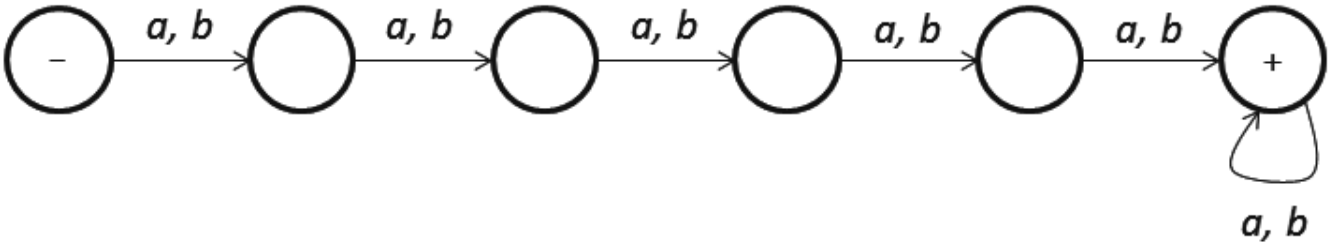


E

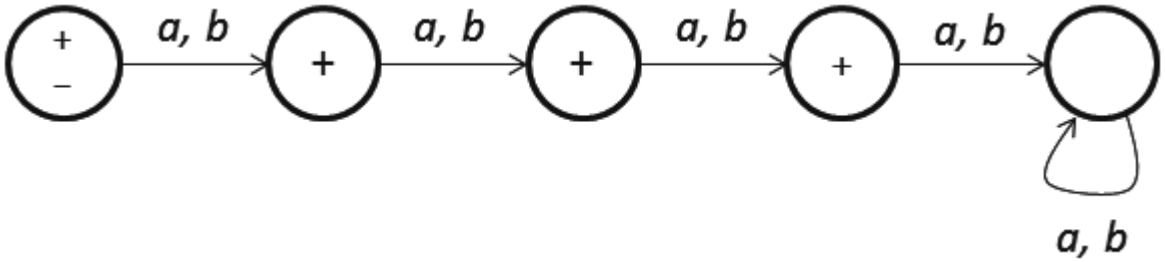
	a	b
1 (initial)	2	2
2	4	3
3 (final)	3	3
4	4	4

(**a** + **b**)**b**(**a** + **b**)*

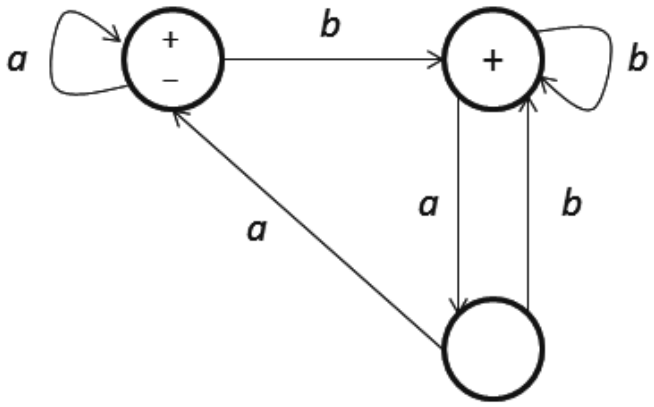
Problem 5(i)



Problem 5(ii)



Problem 6



Recommended problems 5.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
17(ii) and 17(iii) on page 73.

Recommended problems 5.2 – solutions

Problem 17(ii)

All words end on *a* and are of positive (>0) even length.

Problem 17(iii)

All words are of positive (>0) even length and every second letter is an *a*.

11 Learning Unit 6 – Transition graphs

Study Material

Cohen
You need to study chapter 6 in the prescribed book.

Time allocated
You will need one week to study this unit.

Notes

These notes will introduce the concept of a "transition graph" (or TG). After studying this learning unit you should be able to

- check whether something is a transition graph;
- test whether a given string is accepted by a given transition graph;
- design a transition graph to accept a given language;
- describe the language accepted by a given transition graph;
- explain what non-determinism is.

5-tuple	As in the case of a finite automata, we may view a transition graph as a 5-tuple: $TG = (S, \sum, T', S_0, F)$ Where S is a non-empty finite set (the set of states), $\sum$ is a finite set (the input alphabet), T' is a relation from $S \times \sum^*$ to S (the set of transitions), S_0 is a non-empty subset of S (the set of start states), and $F \subseteq S$ (the set of final states).
What is the difference between TGs and FAs?	The basic differences between an FA and a TG are that • we may have more than one start state in the case of a TG; • the transition-relation T' of a TG **need not be a function**; and • the labels on edges may be **strings from** $\sum^*$ rather than just being letters from $\sum$. The fact that T' need not be a function has two consequences. Firstly, the domain of T' need not be the whole of $S \times \sum^*$. Therefore, for some states, we may have no edges leaving. Secondly, a given element of $S \times \sum^*$ may have associated with it more than one image (second co-ordinate). Thus for some states we may have more than one edge leaving the given state with the same label.

Please note that:
The empty word, Λ, may be a label in a TG.

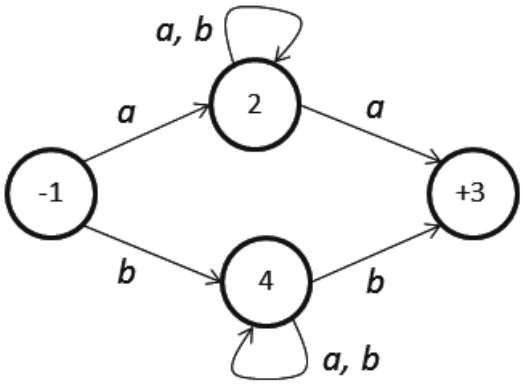
Every FA is TG	Cohen states on page 81 that, "It is extremely important for us to understand that every FA is also a TG". How can we achieve this insight? Well, note that the FA's function T: $S \times \sum \rightarrow S$ is a subset of $(S \times \sum^*) \times S$, so the function T is a relation from S x $\sum^*$ to S. Furthermore, it changes nothing if we say that $\{s_0\}$ is the set of start states rather than saying that s_0 is the start state. So the transition diagram of the FA is a special case of a TG.

Non-determinism	TGs are the first examples of non-deterministic machines. (Human) choice becomes a factor in selecting the path through the machine; the machine does not make all its own determinations and the path does not depend on the input alone.

Activity

Activity 6.1
Build a TG that accepts the language consisting of all words which start and end on the same letter. All words in the language should have at least two letters.

Discussion of Activity 6.1
We need one start state only, say state 1. If we read an *a* in state 1 we move to state 2 (say). Suppose we read an *a* in state 2. If this *a* is the last letter of the word, then we have to move to a final sate, say state 3. If, however, this *a* is not the last letter of the word, then we may simply stay in state 2. Thus there will be two edges leaving state 2 with label *a*. If we read a *b* in state 2, then we also simply stay there. It is not necessary to have any edges leaving state 3. Suppose we read a *b* in state 1. Then we go to a separate state, say state 4. If we read an *a* in state 4, then we stay there. Suppose we read a *b* in state 4 and suppose this *b* is the last letter of the word, then we have to move to a final state. We need not create another final state – state 3 will do perfectly well because it has no outgoing edges. The machine is drawn below. *Note that*, in contrast with the FA in activity 5.1, *the TG has one state with no outgoing edges and other states with more than one outgoing edge with the same label*.



Recommended problems 6.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
2, 5, and 7 on pages 94 and 95.

Recommended problems 6.1 – solutions

Problem 2

(i) Λ is accepted by TG_2 and TG_5.
(ii) *a* is accepted by TG_4.
(iii) *b* is accepted by TG_1 , TG_2 and TG_4.
(iv) *aa* is accepted by TG_1 , TG_3, TG_5 and TG_6.
(v) *ab* is accepted by TG_1 , TG_2, TG_4 and TG_6.
(vi) *aba* is accepted by TG_1 , TG_5 and TG_6.
(vii) *abba* is accepted by TG_1 , TG_5 and TG_6.
(viii) *bab* is accepted by TG_5.
(ix) *baab* is accepted by TG_5.
(x) *abbb* is accepted by TG_3, TG_4 and TG_6.

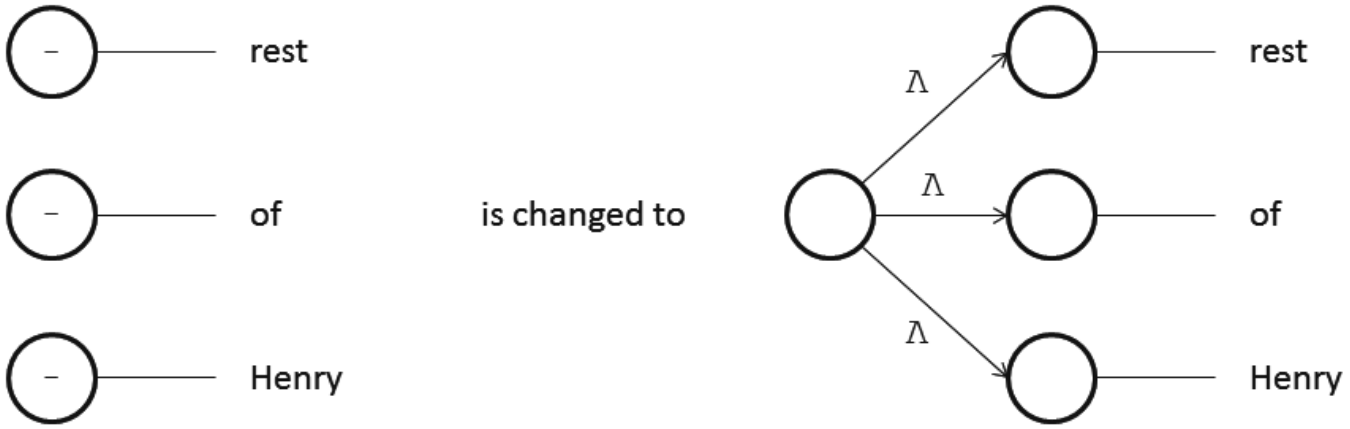
We do not give rigorous proofs in questions 5 and 7, but indicate how it should be done.

Problem 5

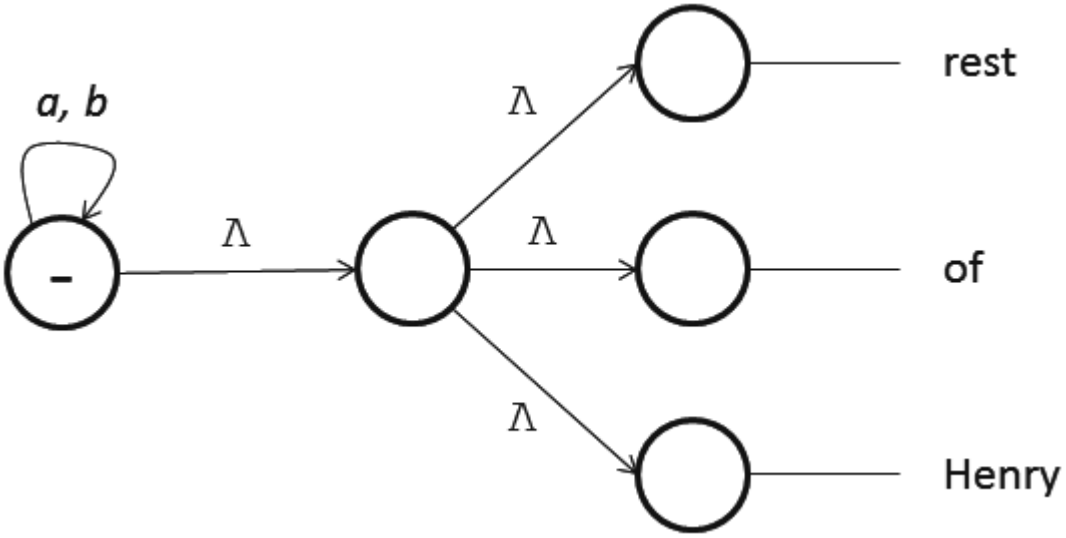
If we have more than one final state, we remove all the + signs from the final states and add an additional state to the TG - the new (single) final state. We then make transitions from the old final states to the new final state and label them with the empty string Λ.

Problem 7

If Henry has more than one start state we change it to a TG with one start state by Λ transitions:



and then we proceed as follows:



Recommended problems 6.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
11, 16, and 19 on pages 90 and 91.

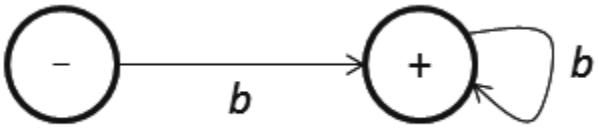
Recommended problems 6.2 – solutions

Problem 11

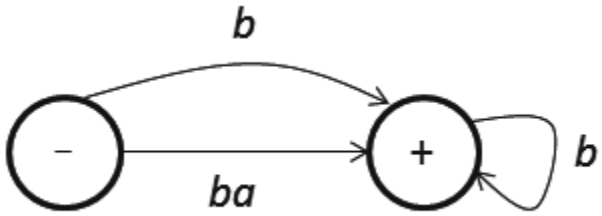
The answer is L^+ (thus L* but without Λ except if L already contains the empty string).

Problem 16(i)

Suppose the language is **bb***. The TG is:



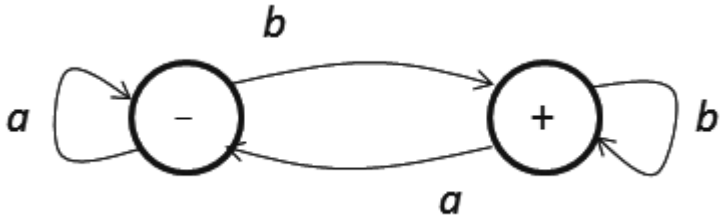
If we change the TG to



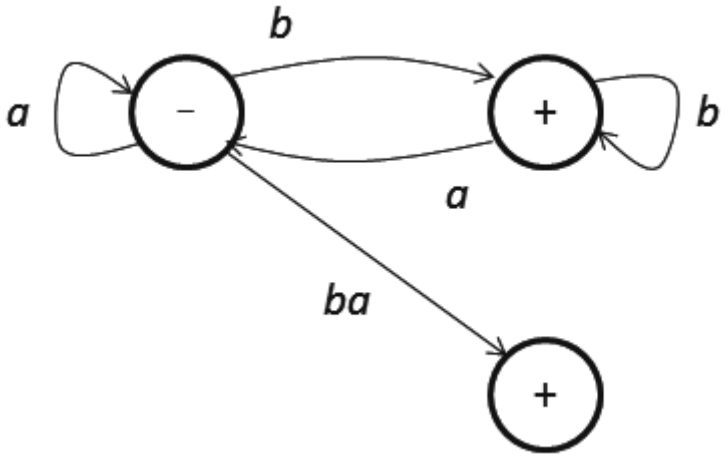
the language now is **bb*** and **ba** and **bab*** and this is not what we want.

Problem 16(ii)

Suppose the TG is:



and we change it to:



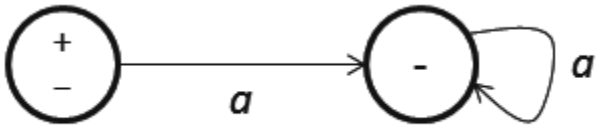
The new TG also accepts words like *aaba* which is of course incorrect.

Problem 16(iii)

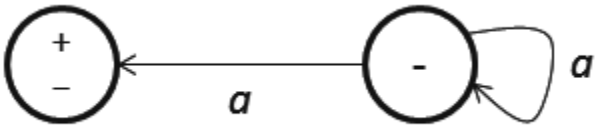
If the start state has loops or incoming edges, add an additional start state **and** an additional final state. Connect the two new states with an edge labelled **ba**. The new TG consists of two separate parts.

If, however, the start state does not have any loops or incoming edges, just add an additional final state. Connect the start state with the new final state with an edge labelled **ba**.

Problem 19



changes to



12 Assignment 2

Assignment 2 scope

What is covered?
This assignment is a written assignment and covers chapters 3 – 6 in Cohen.

Assignment submission
This assignment should be submitted either electronically via myUnisa (the preferred route), or by placing it in an assignment cover and submitting it via one of the regional centres or the post office.

Time allocated
You will need one week to complete this assignment.

Due date
Check Tutorial Letter 101 for the due date and unique assignment number for this assignment.

Assignment 2 questions

You can find the assignment question in tutorial letter 101.

Assignment 2 solutions

After the closing date, a discussion of the assignment will be posted to the Additional Resources page. You will be informed of this via an announcement on myUnisa.

13 Learning Unit 7 – Kleene's Theorem

Study Material

Cohen
You need to study chapter 7 in the prescribed book.

Time allocated
You will need 2 weeks to study this unit. Note that this is a very long chapter (over 50 pages), and you will need to make sure that you do not fall behind at this point.

Vodcasts
You can find videos showing examples of using the algorithms for proving Kleene's Theroem, part 3, in the Videos folder in Additional Resources. It is recommended that you work through the sections in the prescribed book, and this document, before watching the videos.

Notes

These notes aim to achieve the following.

- To state and prove the important theorem of Kleene (pronounced "cleany"). This theorem states that any language that can be defined by either a regular expression, or by a finite automaton, or by a transition graph, can in fact be defined by all three methods.
- To introduce the concept of a non-deterministic finite automaton and to examine the question whether such a machine can do anything an ordinary (i.e. deterministic) FA cannot do.

After studying this learning unit, you should be able to

- sketch the proof of Kleene's theorem;
- apply the algorithms used in the proof to build FAs or regular expressions;
- define a non-deterministic automaton (NFA);
- describe the language accepted by a given NFA;
- design an FA that is equivalent to a given NFA;
- prove that FA = NFA (which means you'll first have to understand what the equation is really saying).

We want to prove that A = B = C	What is the basic strategy used in the proof of Kleene's theorem? Well, there are three sets under consideration (call them A, B and C for the moment) and we want to show that $A = B = C$ If we were first-years doing COS1501, we would approach the problem as follows: first we would show that A = B (which amounts to showing that $A \subseteq B$ and $B \subseteq A$), and then we would show that B = C (which amounts to showing that $B \subseteq C$ and $C \subseteq B$). However, Cohen shows us a more efficient approach. Instead of giving four pieces of proof establishing the inclusions $A \subseteq B$, $B \subseteq A$, $B \subseteq C$ and $C \subseteq B$ respectively, he gives only three pieces of proof, namely $A \subseteq B$, $B \subseteq C$ and $C \subseteq A$. The saving is possible because the inclusions $B \subseteq A$ and $C \subseteq B$ are guaranteed by the transitivity of the relation $\subseteq$: • if $B \subseteq C$ and $C \subseteq A$ then $B \subseteq A$ • if $C \subseteq A$ and $A \subseteq B$ then $C \subseteq B$.

What are A, B and C?	What are the sets we have called A, B and C? Well, each is a set of languages over some fixed common alphabet. The set A contains all the languages (over the given alphabet) that can be defined by designing finite automata to accept them; the set B contains the languages that can be described in terms of transition graphs; and the set C consists of the languages that can be defined by regular expressions. Clearly the point of proving A = B = C is that it shows three standard ways of describing languages to be equivalent; any language we can describe in terms of a regular expression, say, could also have been described in terms of an FA or a transition graph. Now let's take a closer look at the proof of Kleene's theorem.

7.1 Summary of the proof of Kleene's theorem

(i) $A \subseteq B$	We noted in the previous study unit that the transition diagram of an FA is a transition graph, so it's pretty obvious that any language defined by an FA is also defined by a transition graph. It is thus obvious that $A \subseteq B$ and this concludes the first part of the proof of Kleene's theorem.
(ii) $B \subseteq C$	In order to prove the second part, we use a proof by constructive algorithm. We start with a transition graph (i.e. with a language from B) and then we construct a regular expression that defines the same language. By doing this we have proved that the language from B is in fact also a language from C. Since this conversion can be done for any language from B (thus, for any transition graph), $B \subseteq C$ holds.
(iii) $C \subseteq A$	For the third part of the proof, we start with a regular expression (i.e. with a language from C) and we build a finite automaton that accepts the same language. We have thus shown that the language from C is in fact also a language from A. This can be done for any language from C, therefore $C \subseteq A$ holds.

By the above three steps we have proved: A = B = C (Kleene's theorem).

We will now discuss the proofs of the second and third parts in more detail.

7.2 Second part of the proof

This part of the proof is the **algorithm for building a regular expression from a transition graph**.

This algorithm	An algorithm is given which allows us to take any transition graph and to use it to build, step by step, a regular expression defining the same language. We can summarise the algorithm as follows:
Step 1	Given any transition graph, change it to an equivalent transition graph with just one start state.
Step 2	Change the TG generated by the previous step to an equivalent TG with just one final state (which differs from the start state).
Step3	If, in the TG generated by the previous step, there is a state with more than one loop and the loops bear the labels $\mathbf{r}_1$, $\mathbf{r}_2$,…,$\mathbf{r}_n$ (where the $\mathbf{r}_i$ are regular expressions or simple strings), then replace all the loops on that state by a single loop bearing the label: $\mathbf{r}_1 + \ldots + \mathbf{r}_n$.

Step 4	If, in the TG generated by the previous step, there are states connected by more than one edge (in the same direction) having the labels $\mathbf{r}_1$, $\mathbf{r}_2$,…,$\mathbf{r}_n$, then replace the n parallel edges by a single edge bearing the label: $\mathbf{r}_1 + \mathbf{r}_2 + \ldots + \mathbf{r}_n$.
Step 5	If the TG generated by the previous step consists only of a start state and a final state connected by a single edge, we are done and the appropriate regular expression is given by the label on the single remaining edge. However, if the TG does not consist of two states with a single edge, then we must eliminate states one by one as follows. (The procedure also eliminates edges, of course.) Choose any state other than the start state or the final state. Eliminate that state by applying the *bypass operation*, i.e. by taking all edges entering that state directly to all the states that can be reached by following an edge from the chosen state. If an edge into the chosen state is labelled by $\mathbf{r}_1$ and an edge out of the chosen state is labelled by $\mathbf{r}_2$, then the single edge replacing these two edges must be labelled by $\mathbf{r}_1\mathbf{r}_2$. If one of the edges is a loop, say the edge labelled by $\mathbf{r}_1$, then the new label must be $\mathbf{r}_1{}^*\mathbf{r}_2$.) **Of course, great care must be taken not to overlook any paths in the original TG**.
Repeat steps 3, 4 and 5	Steps 3, 4 and 5 are repeated until we are left with just the initial state and the final state. The resulting regular expression can now be written down. Since each step eliminates either a state or an edge (or several, if one is lucky), it follows that the algorithm will terminate, i.e. will reach the stage at which we are left with two states and a single edge. Note that on page 106, Cohen also gives a summary of the algorithm. It is, however, differently numbered.

7.3 Third part of the proof

We want to show that **for every regular expression, there exists an FA that defines the same language.** Given some fixed alphabet, this can be proved by induction on the set of regular expressions over the fixed alphabet, since the set of regular expressions can be defined recursively.

Proof by induction	Cohen, of course, does this without admitting it publicly. To illustrate his use of induction, observe the following: For the sake of convenience, let us use R to denote the set of regular expressions over $\sum$ and S to denote the subset of R containing all the regular expressions whose languages can be recognised by FAs. (We want to show that R = S.) *Rule 1* (pages 108 – 109) shows that the generators of R belong to S (take a peep at *Rule* 1 on page 36 if you need to refresh your memory as to the generators of R). *Rule 2* (pages 109 – 117) shows that if $\mathbf{r}_1 \in$ S and $\mathbf{r}_2 \in$ S, then $\mathbf{r}_1 + \mathbf{r}_2 \in$ S. *Rule 3* (pages 117 – 125) shows that if $\mathbf{r}_1 \in$ S and $\mathbf{r}_2 \in$ S, then $\mathbf{r}_1\mathbf{r}_2 \in$ S. *Rule 4* (pages 125 – 134) shows that if $\mathbf{r}_1 \in$ S, then $\mathbf{r}_1{}^* \in$ S. Therefore S = R, i.e. every regular expression defines a language that can be accepted by an FA.

But this is not all Cohen*'s* argument achieves. Mixed up with the inductive argument summarised above, he gives an algorithm that allows us to take **any regular expression and to *construct* the appropriate FA**.

The algorithm reflects the recursive definition of R. Cohen therefore begins on pages 108-109 by considering the generators of R. If the generator is a regular expression **x** obtained from the letter *x* in $\sum$, then it is quite easy to design an FA to accept the language defined by the expression since the language contains only the one-letter word *x*. Similarly, he shows how to design an FA to accept the language defined by the regular expression Λ, which doesn't require much ingenuity because the language again contains only a single word, namely Λ.

More ingenuity is displayed by the next three pieces of the algorithm. Cohen indicates how to use FAs that can accept the languages defined by $\mathbf{r}_1$ and $\mathbf{r}_2$ to build an accepting FA for each of the languages defined by the regular expressions $\mathbf{r}_1 + \mathbf{r}_2$, $\mathbf{r}_1\mathbf{r}_2$ and $\mathbf{r}_1$*.

We will first outline the algorithm and then we will discuss steps 2 to 4 in more detail.

The algorithm for building a finite automaton from a regular expression.

Consider a regular expression **r** that may contain $\mathbf{\Lambda}$ and any number of other letters.

1. For every letter (and $\mathbf{\Lambda}$, if relevant) from **r**, an FA is built that accepts it.

2. Suppose the factor $\mathbf{r}_1 + \mathbf{r}_2$ appears in **r**, and we have already built two FAs accepting $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively. Build the FA for $\mathbf{r}_1 + \mathbf{r}_2$.

3. Suppose the factor $\mathbf{r}_1\mathbf{r}_2$ appears in **r**, and we have already built two FAs accepting $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively. Build the FA for $\mathbf{r}_1\mathbf{r}_2$.

4. Suppose the factor $\mathbf{r}_1$***** appears in **r**, and we have already built an FA accepting $\mathbf{r}_1$. Build the FA for $\mathbf{r}_1$*.

Steps 2, 3 and 4 are repeated until the required FA has been completed.

Setup	Let's tackle the execution of steps 2, 3 and 4. Assume that you have an FA with states x_1, x_2, … to recognize the language defined by $\mathbf{r}_1$, and an FA with states y_1, y_2, … to recognize the language defined by $\mathbf{r}_2$, with initial states x_1, and y_1 respectively. Both FAs have a finite number of states. The FA to recognize the language defined by $\mathbf{r}_1 + \mathbf{r}_2$ (or by $\mathbf{r}_1\mathbf{r}_2$ or by $\mathbf{r}_1$*) will have states z_1, z_2, … with z_1 as start state.
"Or"-notation	The key to arriving at the states z_1, z_2, … lies in understanding the "or"-notation Cohen uses from page 110 onwards. An equation like $z_i = x_j$ or y_k says that an input letter will take us to state z_i in the new FA if the letter would have taken us to x_j in the FA for $\mathbf{r}_1$ or to y_k in the FA for $\mathbf{r}_2$. So, the way to design the transitions of the new FA is to keep in mind the transitions that are possible on the old FAs. For instance, consider the assignment $z_4 = x_3$ or y_1 or y_2 at the top of page 120 in Cohen. This tells us that state z_4 on the new machine has been reached either by ending up in state x_3 on machine 1, or in state y_1 on machine 2, or in state y_2 on machine 2. On page 120 an indication is also given of how the three possible states can be reached: We can end up in state x_3 by staying in machine 1, we can end up in state y_1 if we have just left machine 1 and are starting in machine 2, and we can end up in

	state y_2 if we are running in machine 2. As promised, we will now discuss steps 2, 3 and 4 in the third part of the proof of Kleene's theorem.

Step 2 of third part: Building an FA for $r_1 + r_2$

Every state z_i of the new FA has to have the form

$x_{something}$ or $y_{another}$.

The start state is $z_1 = x_1$ or y_1.

If either $x_{something}$ or $y_{another}$ is a final state and z_i is the state "$x_{something}$ or $y_{another}$" then z_i is a final state of the new FA.

If we are in a state

$z_i = x_{something}$ or $y_{another}$

on the new FA, and we then read an input letter, say p, then we must go to the state

$z_{new} = x_{new1}$ or y_{new2}

where x_{new1} is the state in the FA of $\mathbf{r}_1$ that we should go to if we read the letter p while in state $x_{something}$, and, similarly, y_{new2} is the state in the FA of $\mathbf{r}_2$ that we would go to if we read the letter p while in state $y_{another}$.

Step 3 of the third part: Building an FA for r_1r_2

For every *non-final* state x_i we need a corresponding state z_i for the new machine $\mathbf{r}_1\mathbf{r}_2$. When we strike a final state in FA_1, then it may be that we want to start moving inside FA_2, or it may be that we want to stay around in FA_1. So we use the "or"-notation. So, in general the remaining z-states are described in the following way:

$z_i = x_{something}$ or y_1 or $y_{another}$

with the idea that $x_{something}$ is the state we would get to if we want to stick to FA_1, y_1 is the start of FA_2, and $y_{another}$ is the state we would get to if we had previously crossed over to FA_2. One of the last two "or" parts of the state z_i must sometimes be omitted, and in other cases there will be MORE THAN ONE "or" part of the last type. This will become clear when you attempt the recommended problems.

Since the start state x_1 of FA_1 may also be a final state, the start state z_1 of the new FA is given either by the equation

$z_1 = x_1$

if x_1 is not a final state of FA_1, or by

$z_1 = x_1$ or y_1

if x_1 is a final state of FA_1.

Step 4 of third part: Building an FA for r_1*

Λ is always a member of the closure of a set.

The language defined by $\mathbf{r}_1$ may or may not have Λ in it, so the start state x_1 of the corresponding FA_1 may or may not be a final state as well. Whichever is the case, the language defined by $\mathbf{r}_1$* will contain Λ and so the start state z_1 of the new FA must be a final state too. So we specify that

$z_1 = x_1$ and is a final state (as well as being the start state, of course).

If x_1 was a final state then we can now move on to think of the next state we'd go to in FA_1. On the other hand, if x_1 was not a final state then we must now have a second state on the new FA that corresponds to x_1 on FA_1 but that is not a final state:

$z_2 = x_1$ and is not a final state.

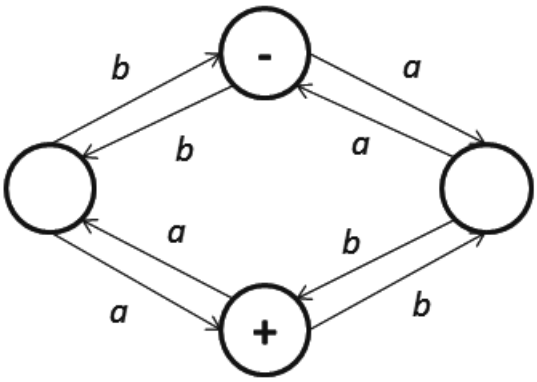
The idea is that z_1 is used to recognise Λ. If reading the input while moving inside FA_1 ever brings us back to the start state x_1 again (remember, x_1 is not final), then in the new FA we represent this by coming to z_2.

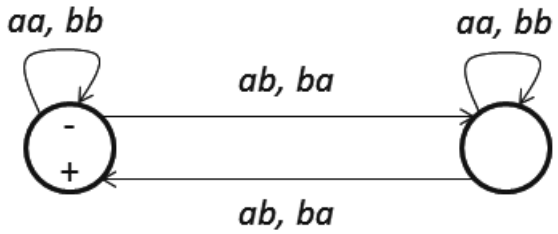
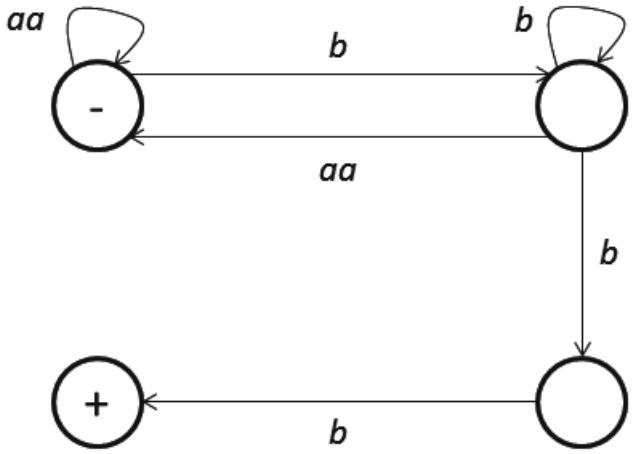
The remaining z-states are now constructed as combinations of x-states having the form

"$x_{something}$"
or the form
"$x_{something}$ and we continue moving in FA_1 because we're busy with a piece of $\mathbf{r}_1$* of the form $\mathbf{r}_1$,
or
x_1 since we've just finished a piece of $\mathbf{r}_1$* of the form $\mathbf{r}_1$ and for the next piece we must go back to the beginning of FA_1 and start anew,
or
…".

Fortunately there will be only a finite number of possibilities, but it can still get pretty tedious.

7.4 Some remarks

Practise	To conclude, let us remark *that it is not sufficient merely to read through these algorithms. We expect you to be able to apply them. So practise by doing the recommended problems*. Please attempt them yourself before looking at our solutions. Then we strongly recommend that you compare your solutions to ours!
An example of a regular language	In study unit 5 we gave the diagram of an FA that accepts all the variable names of an imaginary programming language. It is possible to build an FA that accepts all the acceptable C^{++} variable names. From this we know that the set of acceptable C^{++} variable names is a regular language, and (according to Kleene's theorem) we can find a regular expression for the FA that accepts this language.
Why THREE ways?	You may have wondered why we don't stick to one simple way of defining languages. The reason for using more than one way is that it is in some cases easier to use a regular expression, and in other cases easier to build a machine. Let's consider a few examples. The regular expression **a*** that defines the language of all possible strings containing *a*'s is so simple to understand that we do not require a machine. The same goes for the regular expression (**a** + **b**)***aa**(**a** + **b**)* that defines the language of all words containing the substring *aa* at least once. Now consider the following simple FA that accepts ODD-ODD: 

	See whether you can transform the FA to a regular expression defining ODD-ODD (problem 11(iii) on page 49 in Cohen). It is much, much easier to describe the language in terms of an FA than in terms of a regular expression. The simplest way to define the language EVEN-EVEN is by means of a transition graph:  The language consisting of all words ending in at least 3 *b*'s and where all *a*'s appear in even clumps is also much easier to define in terms of a transition graph: 

7.5 Nondeterministic FAs

5-tuple	To begin with, we can make Cohen's definition a bit more precise: a *non-deterministic* FA can be described as a 5-tuple: NFA = (S, ∑, T′, s_0, F) where S is a non-empty finite set (the set of states), ∑ is a finite set (the input alphabet), T′ is a relation from S x ∑ to S (the transitions), $s_0 \in$ S (the start state) and F $\subseteq$ S (the set of final states).
Basic difference between FA and NFA	The basic difference between an FA and an NFA is that T′ need not be a function from S x ∑ to S. In other words, there does not have to be an edge for every letter in ∑ leaving every state, and there may be more than one edge with the same label leaving a particular state. Clearly every FA is an NFA, since the transition function of an FA is a relation from S x ∑ to S.

Every NFA is a TG	Note that an NFA is a special case of a TG. In fact, the transition diagram of an NFA is just a TG in which the labels on the edges are *single letters from* $\sum$ (and *not* Λ). An NFA also differs from a TG in that an NFA only has a single initial state.

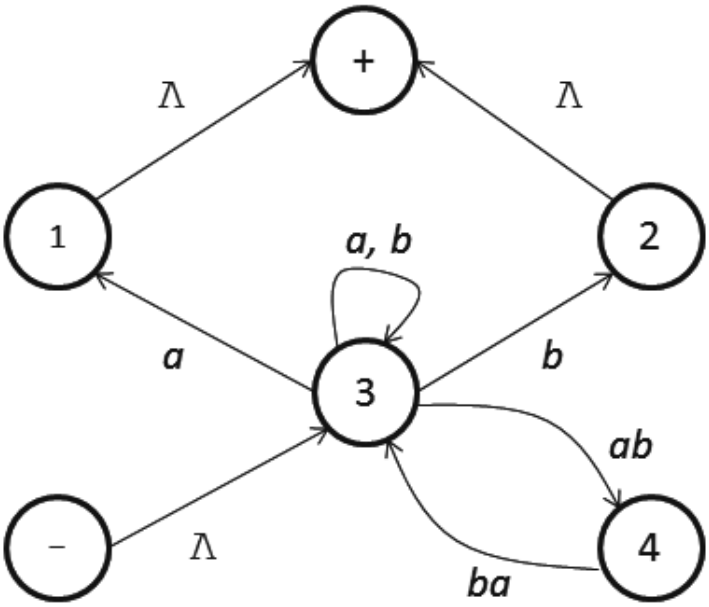
Recommended problems 7.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
1(iii) and 1(v) on page 142.

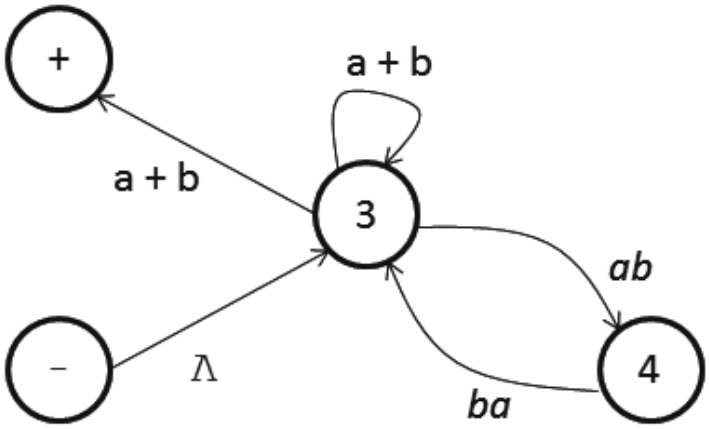
Recommended problems 7.1 – solutions

Problem 1(iii)

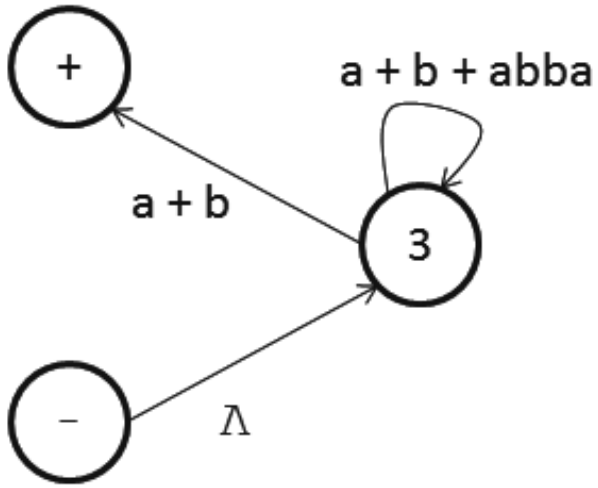
Although it is not essential, we recommend that you always create unique start and final states:



Eliminate states 1 and 2 and deal with the loop at state 3:



Eliminate state 4:



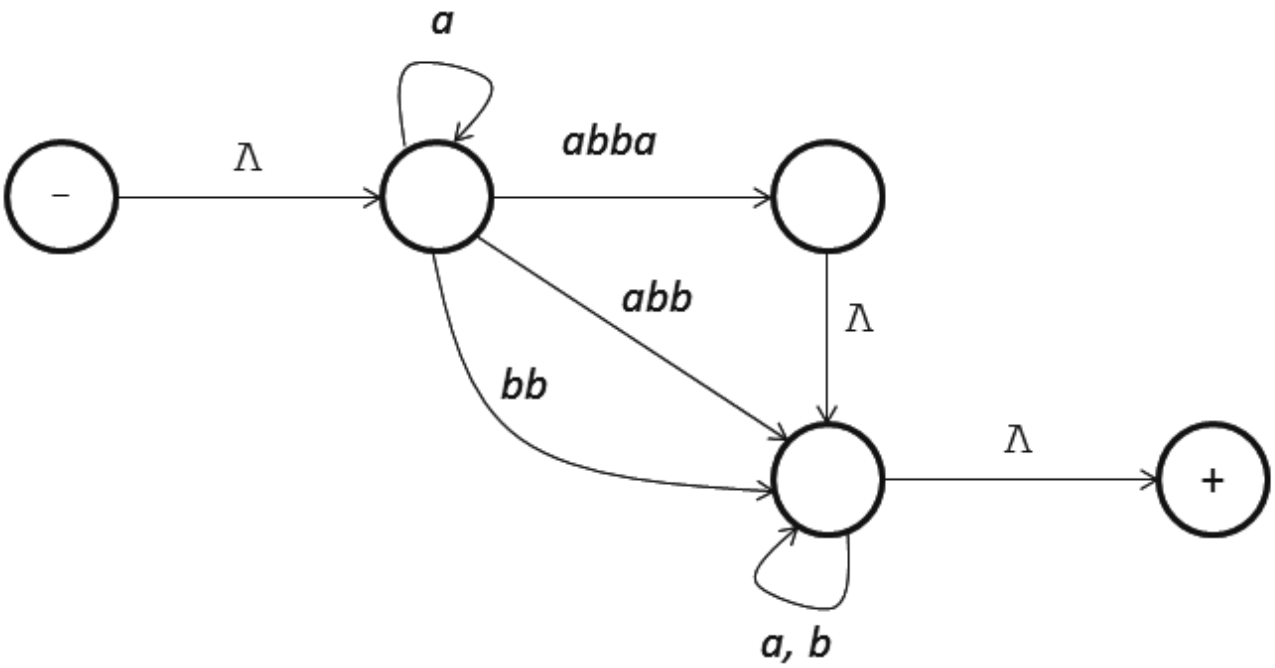
After state 3 has been eliminated, the regular expression is

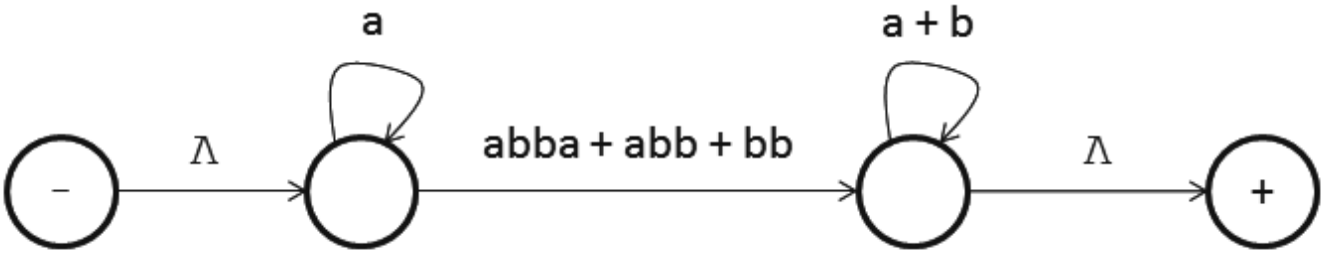
(**a** + **b** + **abba**)*(**a** + **b**)

which is equivalent to

(**a** + **b**)*(**a** + **b**).

Problem 1(v)





The regular expression is

a*(**abba** + **abb** + **bb**)(**a** + **b**)*

and it can be simplified to

a***bb**(**a** + **b**)*.

Recommended problems 7.2

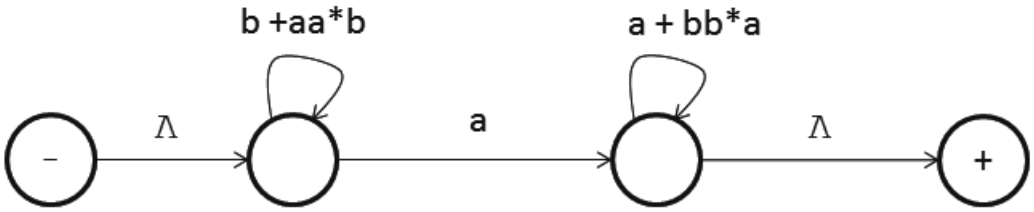
Use the techniques of the first proof of Kleene's theorem for the following:
Find regular expressions for each of FA_1, FA_2, and FA_3 given on page 143 in Cohen.

Recommended problems 7.2 – solutions

Conversion of three FAs into regular expressions

Keep in mind that every FA is a TG. We are going to use the same approach as in questions 1(iii) and 1(v).

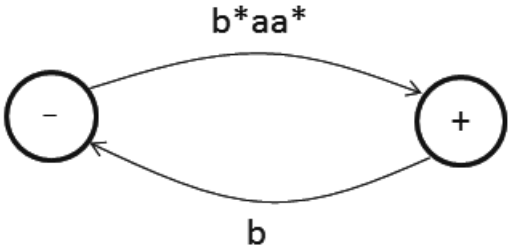
FA_1



The regular expression is

(**b** + **aa***b**)***a**(**a** + **bb***a**)*.

Note that a simpler regular expression can be derived as follows:

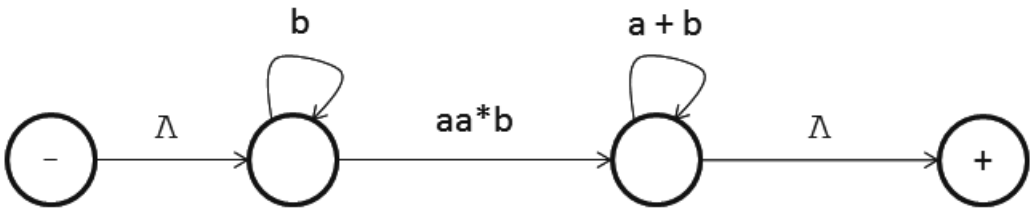


b*aa*(**bb***aa*)*.

In this case the expression is correct, but this method will not always produce a correct expression. The first method will, however, always produce a correct expression.

This language consists of all words that end with an *a*.

FA_2

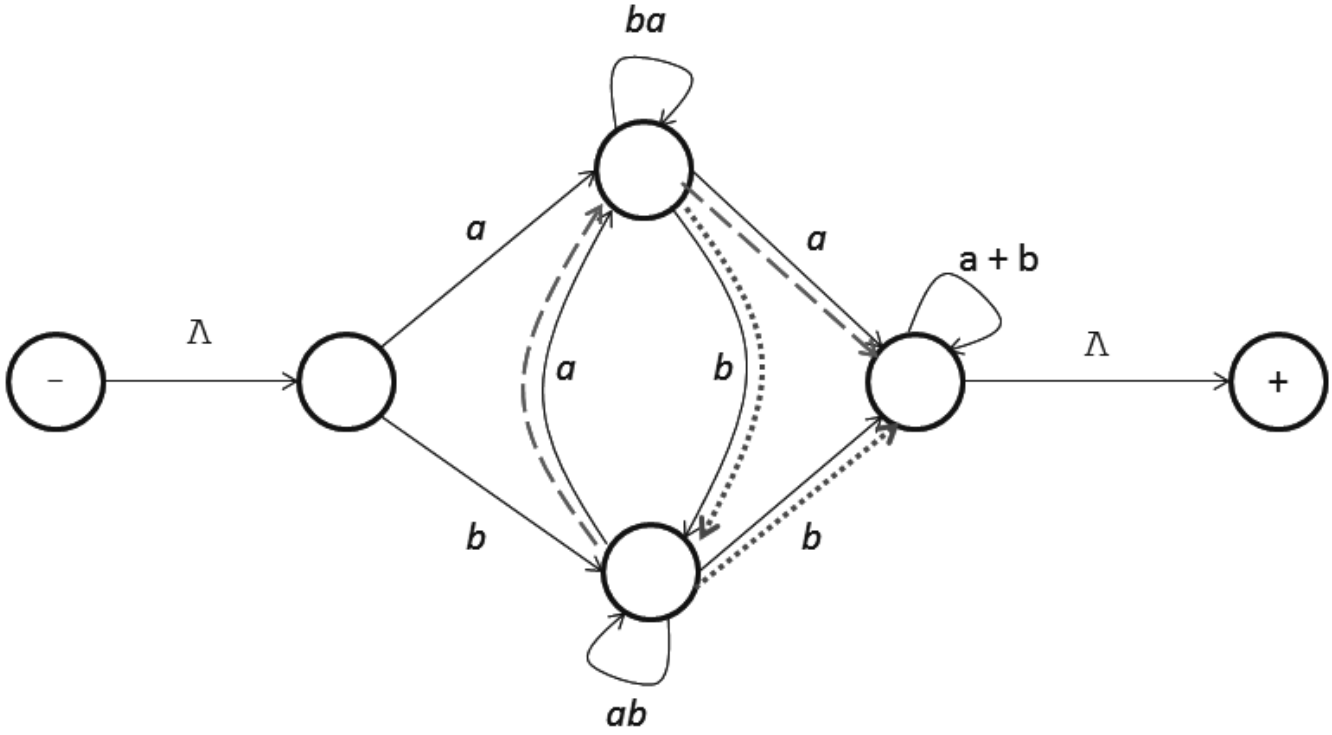


The regular expression is:

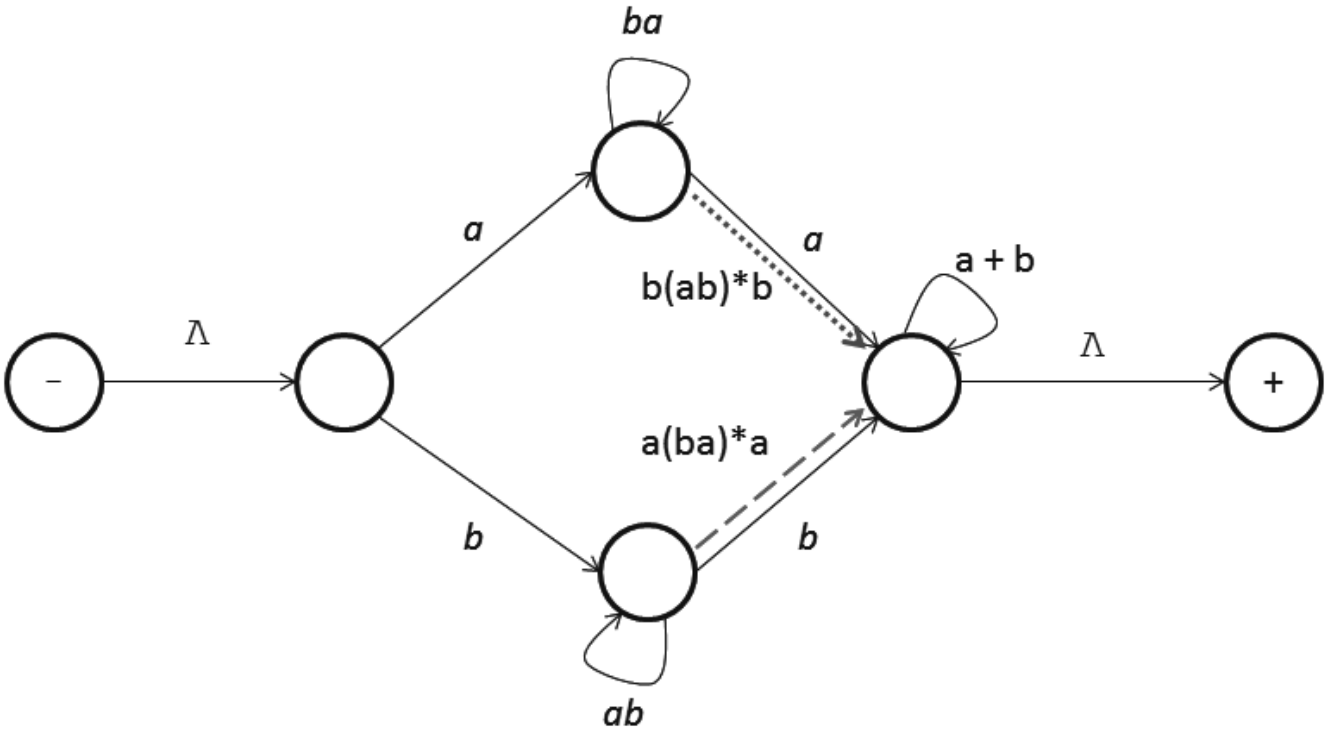
b*(**aa***b**)(**a** + **b**)*

This language consists of all words that contain the substring *ab*.

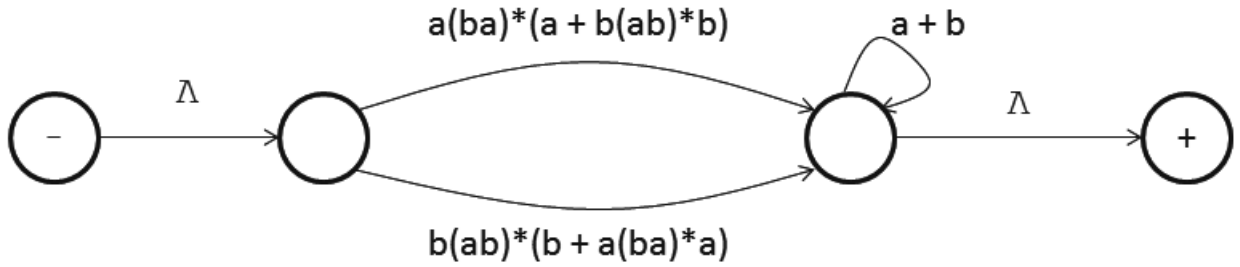
FA_3 We must ensure that all possible paths between the initial and final states are preserved. Before we eliminate the two intermediate states, we must deal with the vertical edges:



We may not eliminate the vertical edges without making provision for the paths indicated by the dotted lines in the figure. We deal with those edges as follows:



We can now eliminate the intermediate states:



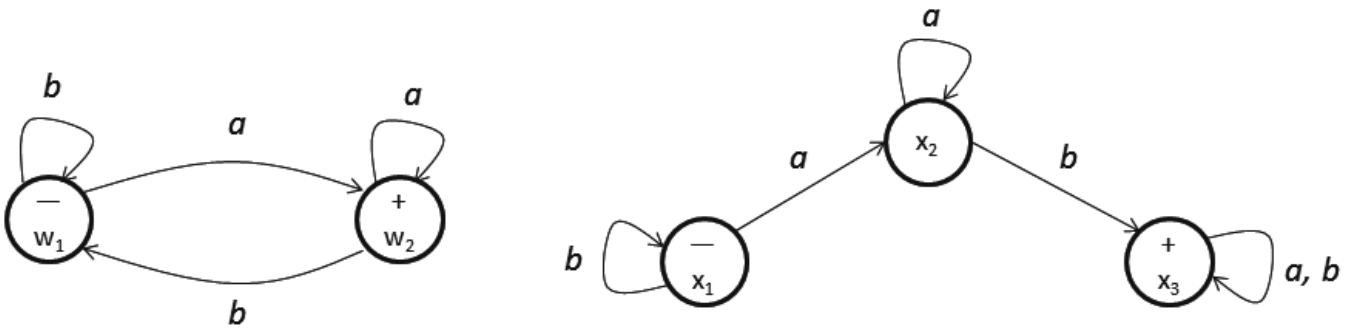
The expression is: [**a**(**ba**)*(**a** + **b**(**ab**)*(**b**) + **b**(**ab**)*(**b** + **a**(**ba**)*(**a**)](**a** + **b**)*
This language consists of all words that contain a double *a* or a double *b*.

Recommended problems 7.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
3(i) and 5(ii) on pages 143 and 144.

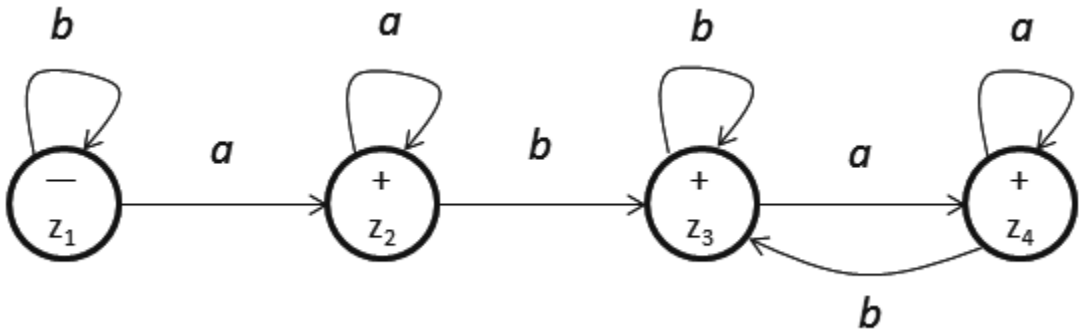
Recommended problems 7.3 – solutions

Problem 3(i)



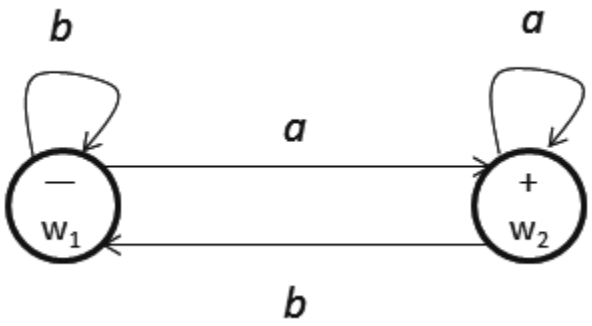
The transition table:

New state	Read an *a*	Read a *b*
$-z_1 = w_1$ or x_1	$+z_2$	z_1
$+z_2 = w_2$ or x_2	$+z_2$	$+z_3$
$+z_3 = w_1$ or x_3	$+z_4$	$+z_3$
$+z_4 = w_2$ or x_3	$+z_4$	$+z_3$

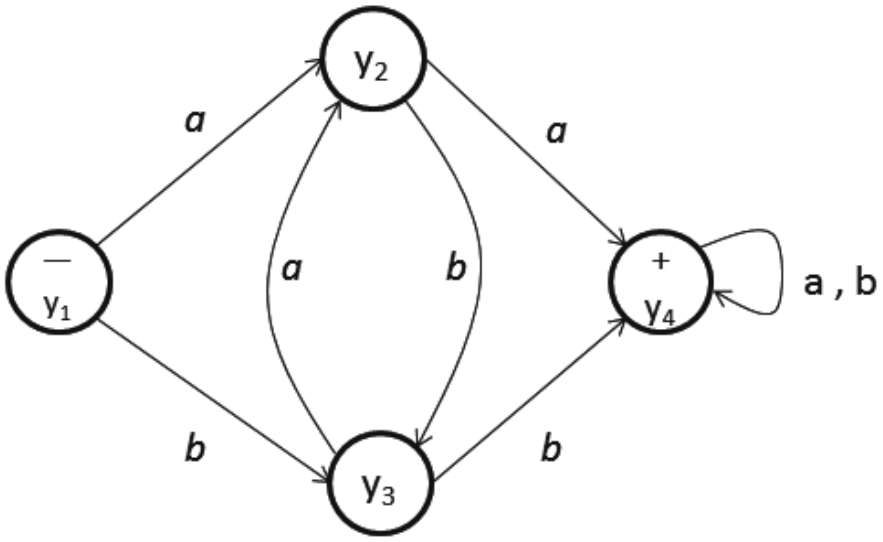


Problem 5(ii)

We want to determine a new FA for $\mathbf{r}_1\mathbf{r}_3$ where the FA for $\mathbf{r}_1$ is:



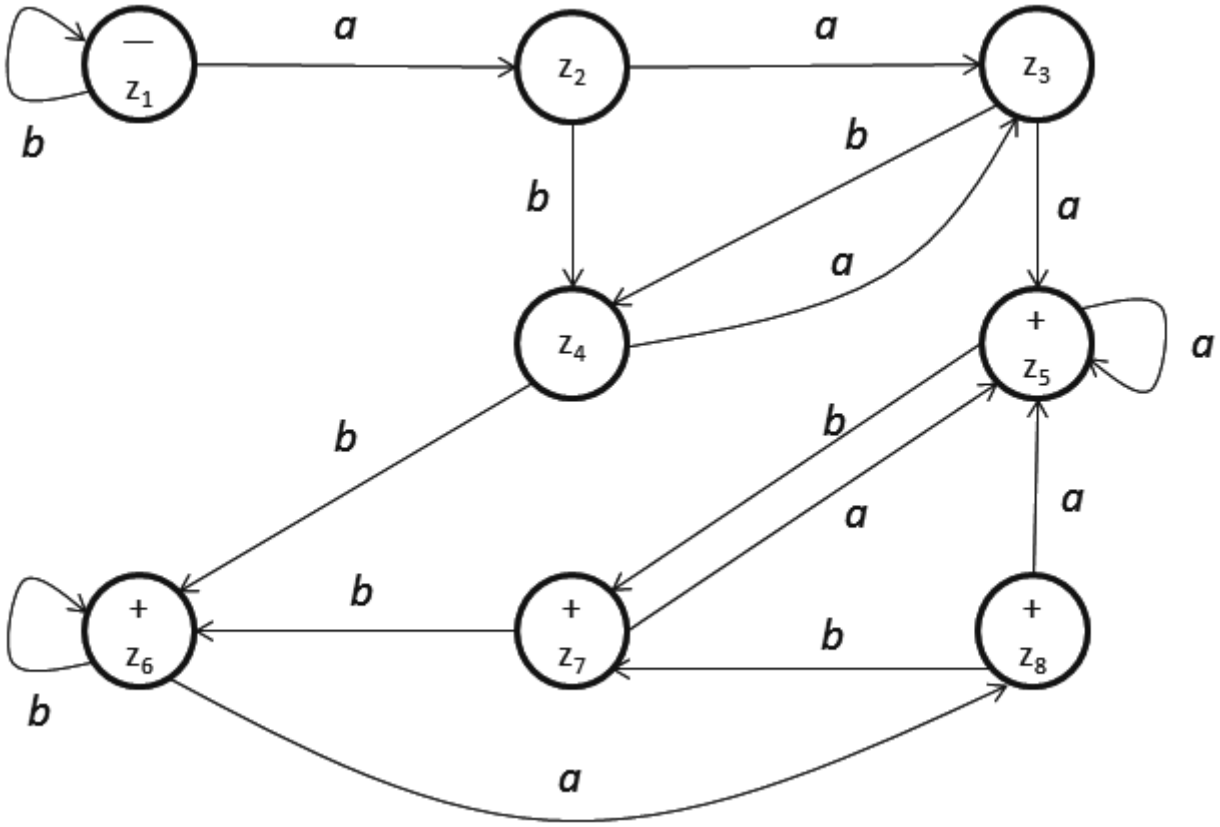
and the FA for $\mathbf{r}_3$ is:



The transition table for the new FA:

New state	Read an *a*	Read a *b*
$-z_1 = w_1$	z_2	$-z_1$
$z_2 = w_2$ or y_1	z_3	z_4
$z_3 = w_2$ or y_1 or y_2	$+z_5$	z_4
$z_4 = w_1$ or y_3	z_3	$+z_6$
$+z_5 = w_2$ or y_1 or y_2 or y_4	$+z_5$	$+z_7$
$+z_6 = w_1$ or y_4	$+z_8$	$+z_6$
$+z_7 = w_1$ or y_3 or y_4	$+z_5$	$+z_6$
$+z_8 = w_2$ or y_1 or y_4	$+z_5$	$+z_7$

We can now construct the transition diagram for the new FA.



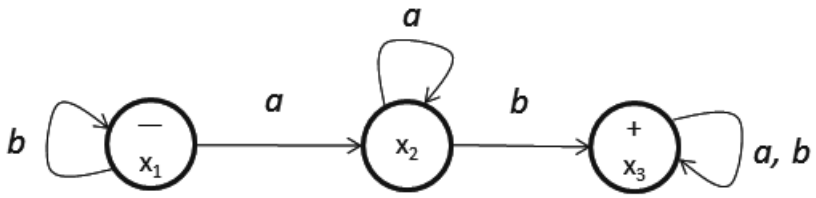
Recommended problems 7.4

Use the technique of the first proof of Kleene's theorem for the following:
Construct an FA for the product language FA_2FA_3 with FA_2 and FA_3 as given on page 143 in Cohen.

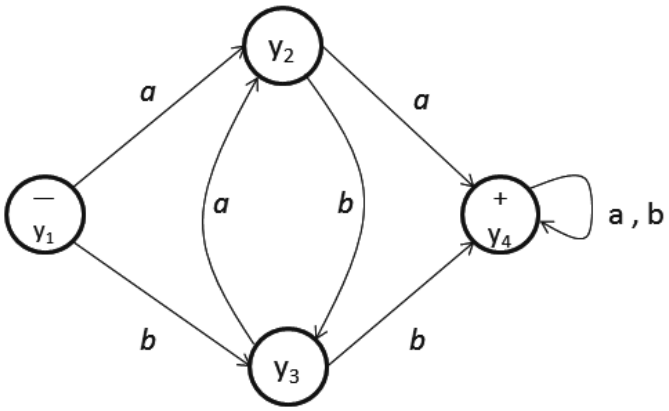
Recommended problems 7.4 – solutions

Construct an FA for the product language FA_2FA_3

FA_2:



FA_3:



The new FA for FA_2FA_3:

$z_1 = x_1$:

Read an *a* → $z_2 = x_2$

Read a *b* → $z_1 = x_1$

$z_2 = x_2$:

Read an *a* → $z_2 = x_2$

Read a *b* → $z_3 = x_3$ or y_1

$z_3 = x_3$ or y_1:

Read an *a* → $z_4 = x_3$, y_2 or y_1

Read a *b* → $z_5 = x_3$, y_3 or y_1

$z_4 = x_3$, y_2 or y_1:

Read an *a* → $+z_6 = x_3$, y_4, y_1 or y_2

Read a *b* → $z_5 = x_3$, y_3 or y_1

$z_5 = x_3$, y_3 or y_1:

Read an *a* → $z_4 = x_3$, y_2 or y_1

Read a *b* → $+z_7 = x_3$, y_4, y_3 or y_1

$+z_6 = x_3$, y_4, y_1 or y_2:

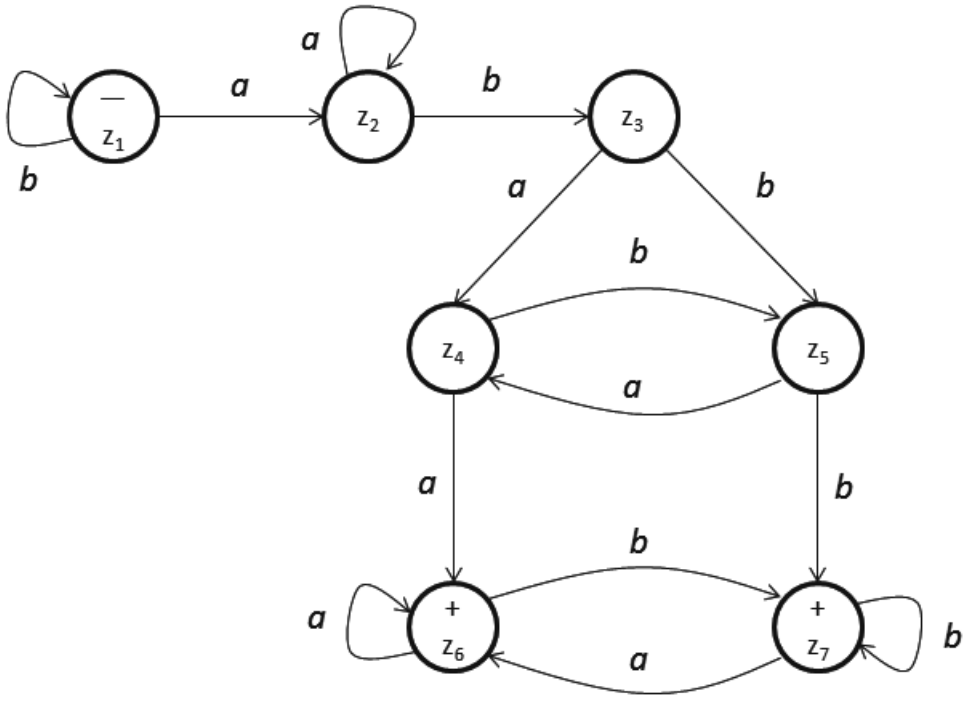
Read an *a* → $+z_6 = x_3$, y_4, y_2 or y_1

Read a *b* → $+z_7 = x_3$, y_4, y_1 or y_3

$+z_7 = x_3$, y_1, y_3 or y_4:

Read an *a* → $+z_6 = x_3$, y_1, y_2 or y_4

Read a *b* → $+z_7 = x_3$, y_1, y_3 or y_4

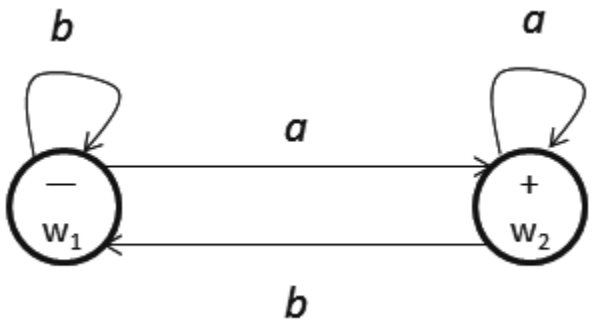


Recommended problems 7.5

Do the following problems to consolidate your knowledge of the work in this learning unit:
6(i), 13(ii)(a), 14(ii), 14(v), 14(ix), and 19 on pages 144 - 147.

Recommended problems 7.5 – solutions

Problem 6(i)



We create a **separate** initial state which is also a final state because Λ must also be accepted:

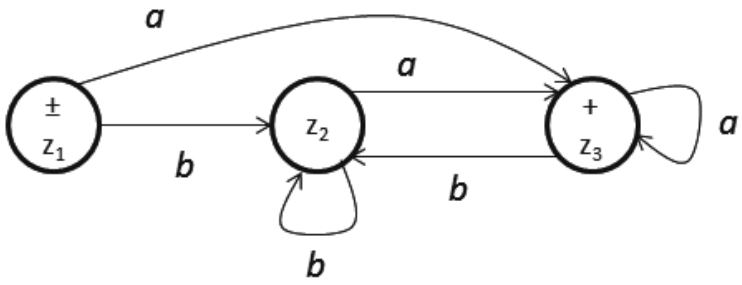
$\pm z_1 = w_1$ and $z_2 = w_1$.

$\pm z_1$:
a is read: w_1 goes to $+w_2$ or back to w_1 (because w_2 is a final state). Thus it goes to $+z_3 = w_1$ or $+w_2$.
b is read: w_1 goes to $w_1 = z_2$.

$z_2 = w_1$:
a is read: w_1 goes to $+w_2$ or (because w_2 is a final state) to w_1, thus $+z_3$.
b is read: w_1 goes to $w_1 = z_2$

$+z_3 = w_1$ or w_2:
a is read: w_1 goes to $+w_2$ or (as above) to w_1
and
w_2 goes to $+w_2$ or w_1, again $+z_3$.
b is read: w_1 goes to w_1
and
w_2 goes to w_1, thus z_2.

New state	Read an *a*	Read an *b*
$\pm z_1 = w_1$	$+z_3$	z_2
$z_2 = w_1$	$+z_3$	z_2
$+z_3 = w_1$ or w_2	$+z_3$	z_2



Problem 13(ii)a

We have the following identity:
(It is a form of the binomial theorem, but you do not have to remember it.)

$$(1 + 1)^k = C(k, 0) + C(k, 1) + C(k, 2) + C(k, 3) + ... + C(k, k)$$
$$= 1 + C(k, 1) + C(k, 2) + C(k, 3) + ... + C(k, k)$$

$FA_1 + FA_2$:

Refer to page 113 in *Cohen.* It can be seen that the maximum number of states in the automaton $FA_1 + FA_2$ is $n \cdot m$.

FA_1FA_2:

Refer to page 121 in *Cohen.* The maximum number of final states in FA_1 is n. Note that we may get several combinations of states of FA_2 at "are in a set of $y_{something}$ continuing on FA_2". (You only have to look at some of the previous examples.) The maximum number of states in the automaton FA_1FA_2 is:

$$n \cdot 1 \cdot (C(m, 1) + C(m, 2) + C(m, 3) + ... + C(m, m))$$
$$= n \cdot 1 \cdot ((1 + 1)^m - 1)$$
$$= n(2^m - 1).$$

$(FA_1)^*$:

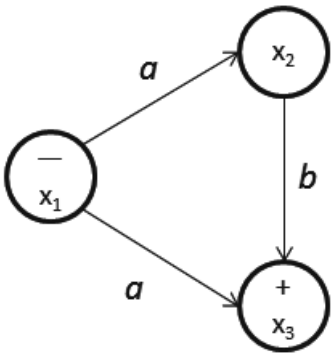
Refer to the relevant pages in the study guide. Every state of FA_1^* consists of a combination of states of FA_1 itself. (See problem 6(1) above for an example.) FA_1 has a maximum of n final states and FA_1^* therefore has the following maximum number of states:

$$(C(n, 1) + C(n, 2) + C(n, 3) + ... + C(n, n)) + 1$$
$$= ((1 + 1)^n - 1) + 1$$
$$= 2^n$$

You can see from the above that the new machine can have a finite number of states only. It may, however, be a considerable number!

Problem 14(ii)

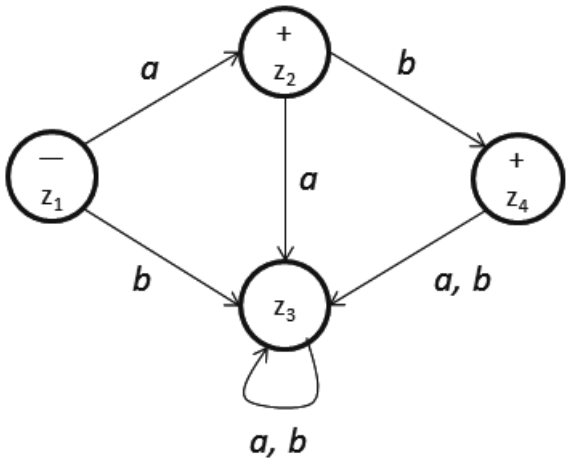
Let us number the states of the given NFA.



Now we set up the following table:

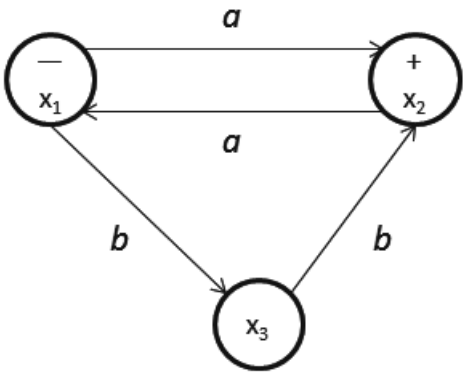
New state	Read an *a*	Read an *b*
$-z_1 = x_1$	$+z_2$	z_3
$+z_2 = x_2$ or x_3	z_3	$+z_4$
z_3 (dead-end)	z_3	z_3
$+z_4 = x_3$	z_3	z_3

So we get the following FA:



Problem 14(v)

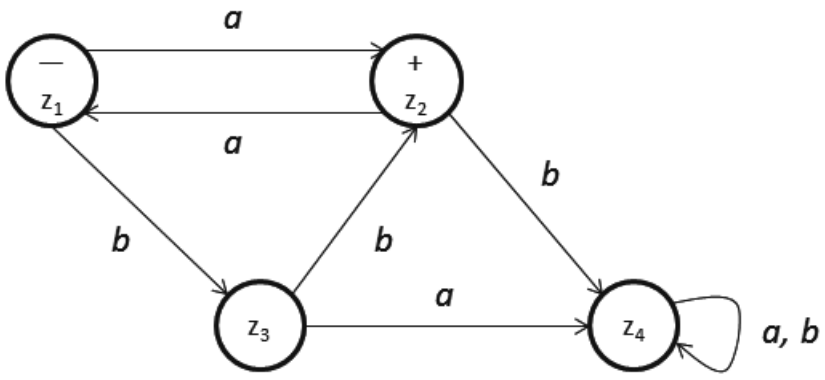
The given NFA:



The following table is compiled:

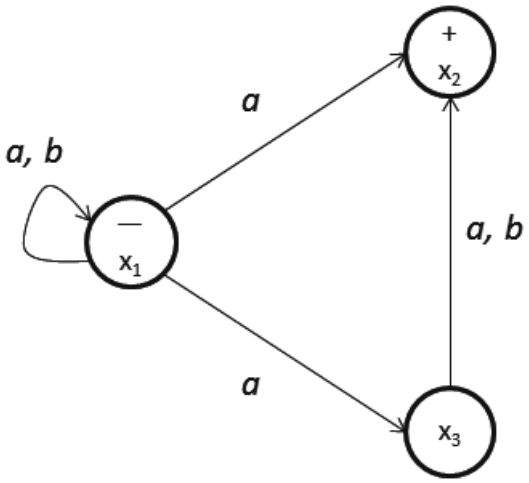
New state	Read an *a*	Read an *b*
$-z_1 = x_1$	$+z_2$	z_3
$+z_2 = x_2$	$-z_1$	z_4
$z_3 = x_3$	z_4	$+z_2$
z_4 (dead-end)	z_4	z_4

So we get the following FA:



Problem 14(ix)

The given NFA:

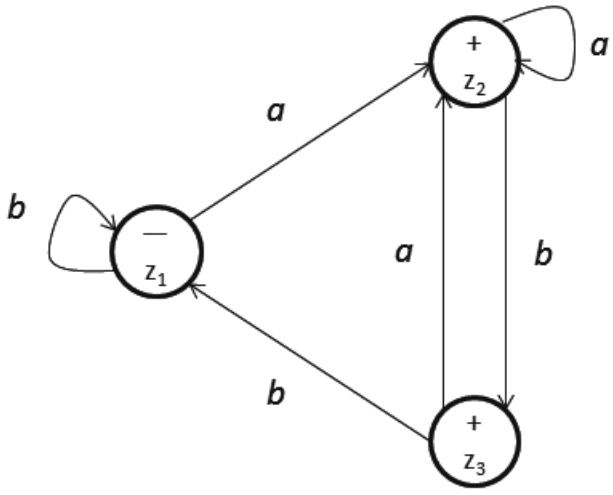


We get the following table:

New state	Read an *a*	Read an *b*
$-z_1 = x_1$	$+z_2$	$-z_1$
$+z_2 = x_1$ or x_2 or x_3	$+z_2$	$+z_3$
$+z_3 = x_1$ or x_2	$+z_2$	$-z_1$

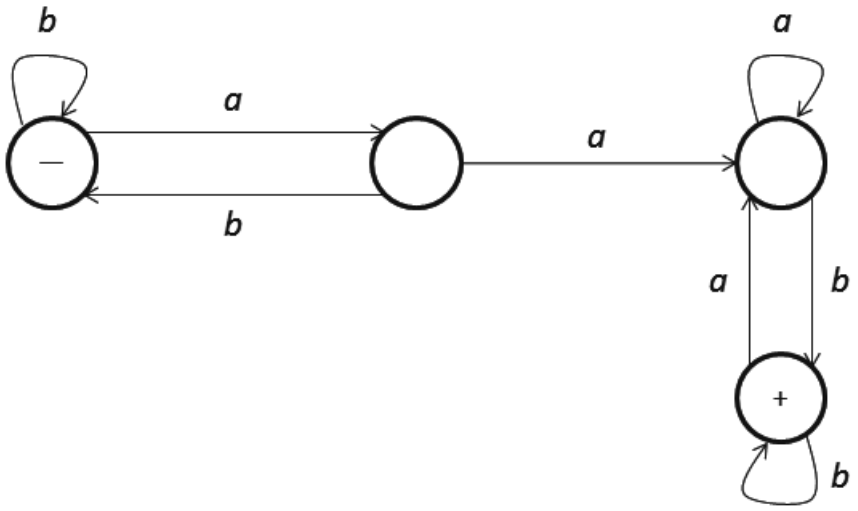
We see that no dead-end state is necessary here.

The FA looks as follows:

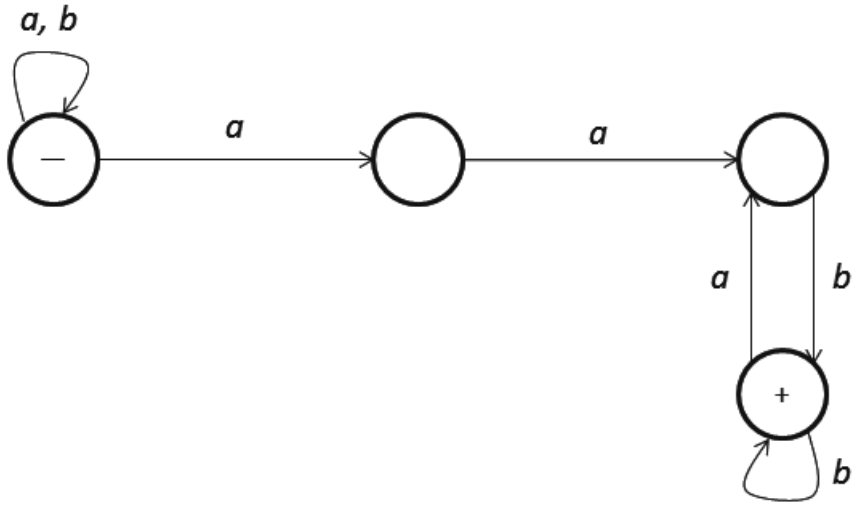


Problem 19

The state with the *a*-loop can be eliminated without changing the language that is accepted by the FA. We get the following FA:



What language is accepted? Well, if you look carefully at the machine you will see that every word that is accepted ends on *b* and somewhere in the word the substring *aab* occurs. The following NFA (with seven edges) accepts the same language:



14 Learning Unit 8 – Finite Automata with Output

Study Material

Cohen
You need to study chapter 8 in the prescribed book.

Time allocated
You will need one week to study this unit and do self-test B.

Notes

These notes will introduce the idea of an FA with output. After studying this learning unit, you should be able to

- define a Moore machine;
- define a Mealy machine;
- prove that Me = Mo;
- design an Me that is equivalent to a given Mo;
- design an Mo that is equivalent to a given Me;
- design an Me or an Mo for a given sequential circuit.

Moore machine as a 6-tuple	Let's start by tightening up the definition of a Moore machine given on page 150 in Cohen. We have to define the set of states S, the initial state q_0, the set of input letters $\sum$ and the set of output letters Γ. Finally, we have the output table that indicates which output letter must be used for every state that is entered. By a Moore machine we thus understand a 6-tuple: $Mo = (S, \sum, T, q_0, \Gamma, f)$ Where S is a non-empty finite set (the set of states), $\sum$ is a non-empty set (the input alphabet), $T: S \times \sum \rightarrow S$ (the transition table), $q_0 \in S$ (the start state) Γ is a non-empty set (the output alphabet) and $f: S \rightarrow \Gamma$ (the output table). Note that both T and f are functions.
Mealy machine as a 6-tuple	Next we can tighten up the definition of a Mealy machine given on page 152 in Cohen. By a Mealy machine we understand a 6-tuple: $Me = (S, \sum, T, q_0, \Gamma, f)$ Where S, $\sum$, T, q_0 and Γ are exactly as in the definition of a Moore machine, but $f: S \times \sum \rightarrow \Gamma$.
No final states	Note that these machines do NOT need final states since we have a function f which tells us what output to give.
The difference between Mo and Me	The basic difference between these two machines is that the output of a Moore machine depends only on the current state, while the output of a Mealy machine depends on both the current state *and* on the input letter being read.

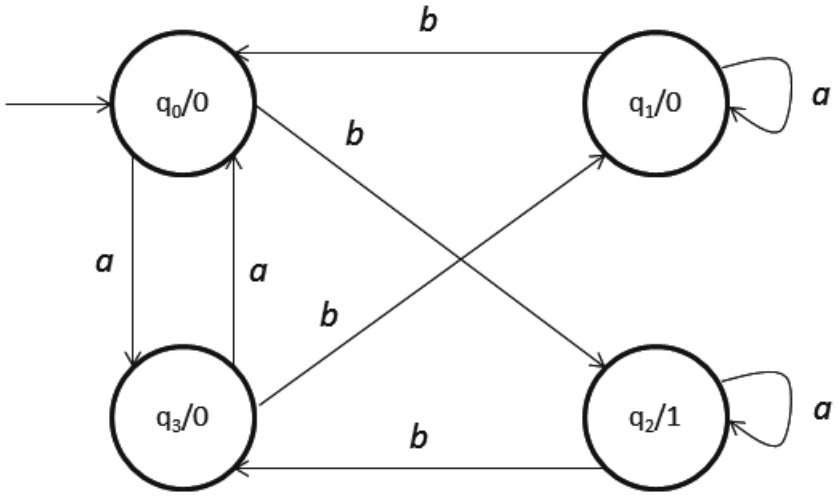
Binary arithmetic	In your first year you were introduced to binary numbers and the 1-complement of an input string and you learned how to subtract one binary number from another. In chapter 8 in Cohen you are shown how to accomplish this with a Mealy machine. You also learned about flip-flops, sequential circuits and AND, NAND and OR gates in your first year. This chapter in Cohen shows how Mealy and Moore machines can be applied in this context.

Recommended problems 8.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
1(iv), 3(iii), 4 (for 1(iv) and 3(iii)) and 5 on pages 164 and 165.

Recommended problems 8.1 – solutions

Problem 1(iv)



PROBLEM 3(iii)

	a	*b*	output
q_0	q_1	q_2	0
q_1	q_1	q_1	1
q_2	q_1	q_0	0

Problem 4

1(iv): 0 0 0 1 1 0
3(iii): 0 1 1 1 1 1

Problem 5

We already know that any Mo can be converted into an equivalent Me. We convert the LESS-machine in the same way into an Me. The only difference is that we need not start by saying that the first character is not printed. It is also possible to construct an algorithm to convert an Me into a LESS-machine.

Recommended problems 8.2

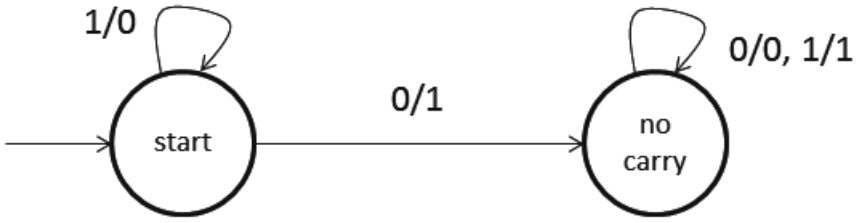
Do the following problems to consolidate your knowledge of the work in this learning unit:
6(ii) and 7 on page 166.

Recommended problems 8.2 – solutions

Problem 6(ii)

	q_0	q_1	q_2
q_0		*b*/1	*a*/0
q_1	*a*/0		*b*/0
q_2	*a*/1		*b*/1

Problem 7

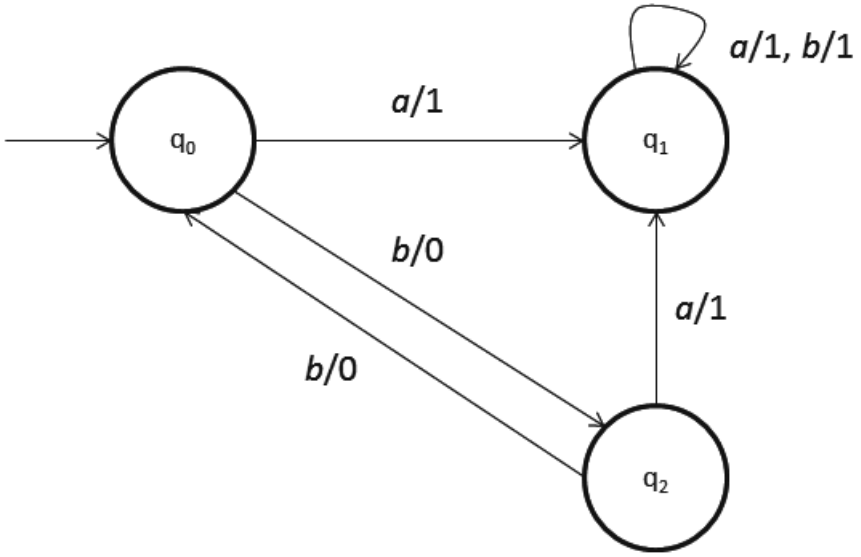


Recommended problems 8.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
8 (for 3(iii)), 9 (for 6(ii)), and 10 on page 167.

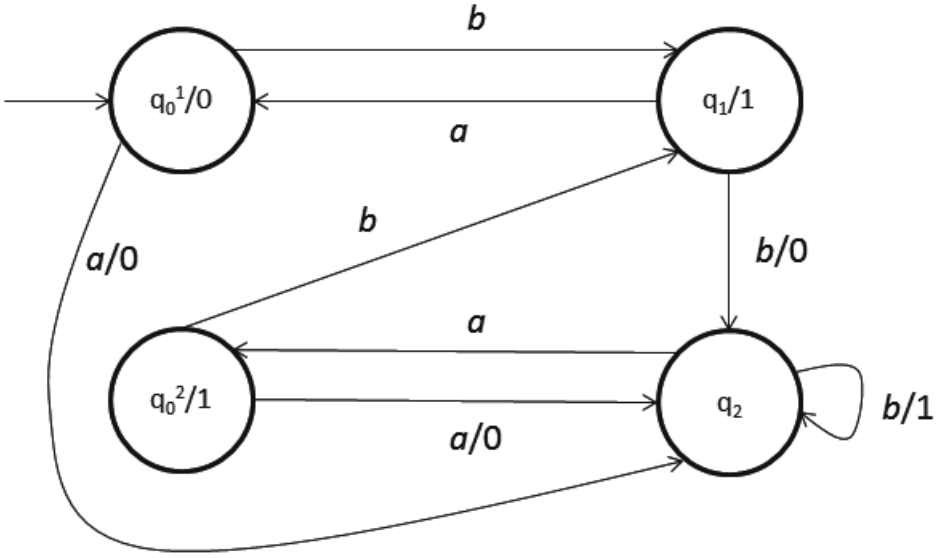
Recommended problems 8.3 – solutions

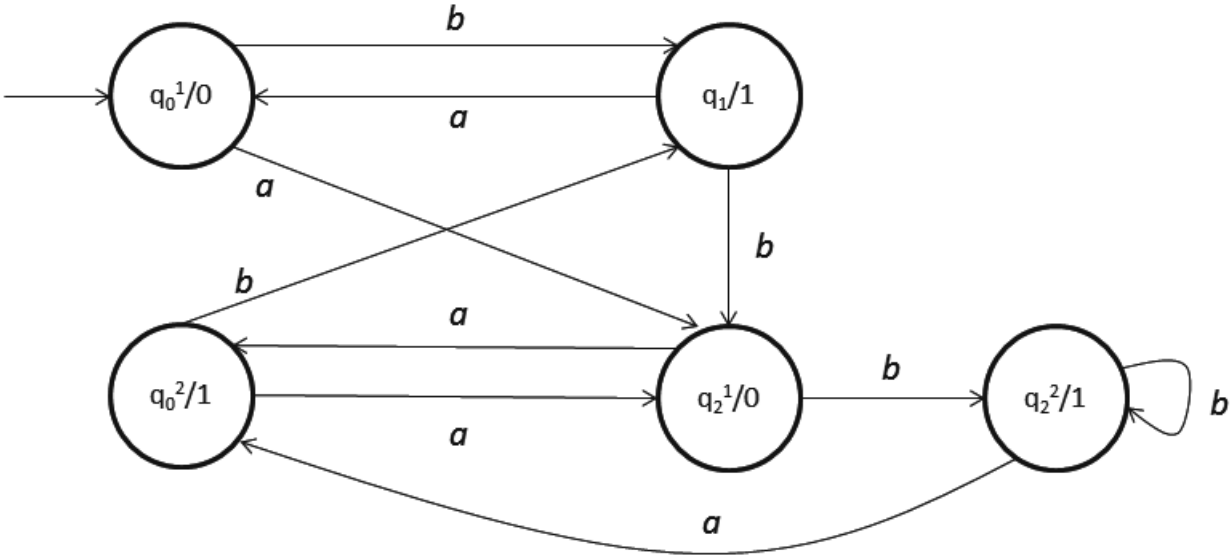
Problem 8 for 3(iii)



Problem 9 for 6(ii)

We do it in two steps.





Problem 10

q_0: A = 0 B = 0
q_1: A = 0 B = 1
q_2: A = 1 B = 0
q_3: A = 1 B = 1

new B = old A
new A = input OR (old A OR old B)
output = input AND old B

q_0: input 0: new B = old A = 0 q_0
new A = 0 OR (0 OR 0) = 0
output = 0 AND 0 = 0

q_0: input 1: new B = 0 q_2
new A = 1 OR (0 OR 0) = 1
output = 1 AND 0 = 0

q_1: input 0: new B = 0 q_2
new A= 0 OR (0 OR 1) = 1
output = 0 AND 1 = 0

q_1: input 1: new B = 0 q_2
new A = 1 OR (0 OR 1) = 1
output = 1 AND 1 = 1

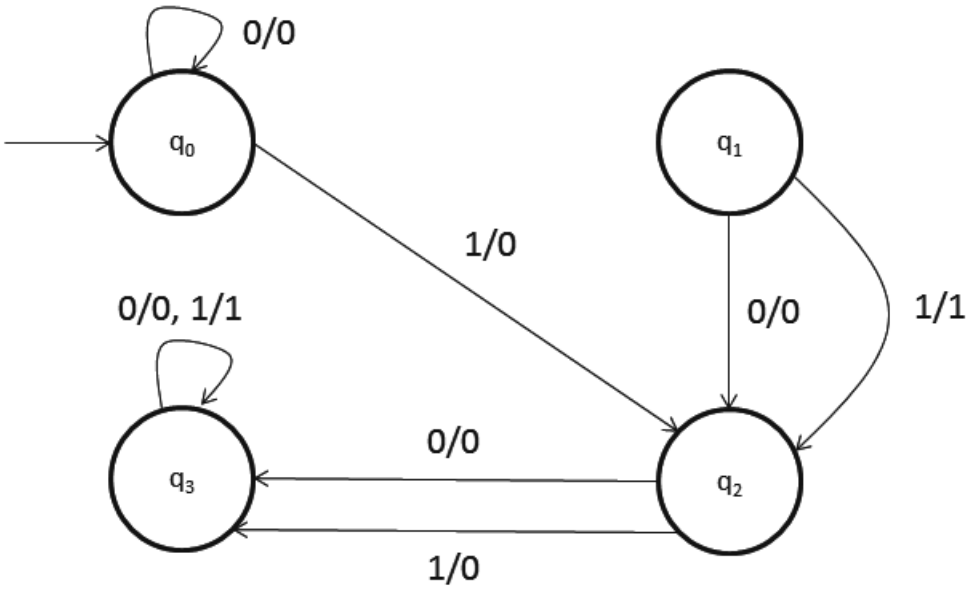
q_2: input 0: new B = 1 q_3
new A = 0 OR (1 OR 0) = 1
output = 0 AND 0 = 0

q_2: input 1: new B = 1 q_3
new A = 1 OR (1 OR 0) = 1
output = 1 AND 0 = 0

q_3: input 0: new B = 1 q_3
new A = 0 OR (1 OR 1) = 1
output = 0 AND 1 = 0

q_3: input 1: new B = 1 q_3
new A = 1 OR (1 OR 1)= 1
output = 1 AND 1 = 1

	After input 0		After input 1	
Old state	**New state**	**Output**	**New state**	**Output**
q_0	q_0	0	q_2	0
q_1	q_2	0	q_2	1
q_2	q_3	0	q_3	0
q_3	q_3	0	q_3	1



15 Self-test B

Self-test B scope

What is covered?
This self-test covers chapters 7 – 8 in Cohen and learning units 7 – 8.

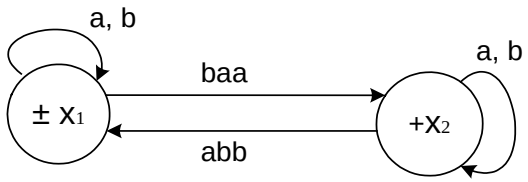
Self-test submission
Please note that you should not submit this self-test to Unisa for marking. These assignments are intended for you to test yourself. Please contact your e-tutor or one of the lecturers if you need help.

Time allocated
You will need one week to complete chapter 8 and this self-test.

Due date
Check Tutorial Letter 101 for the due date for this self-test.

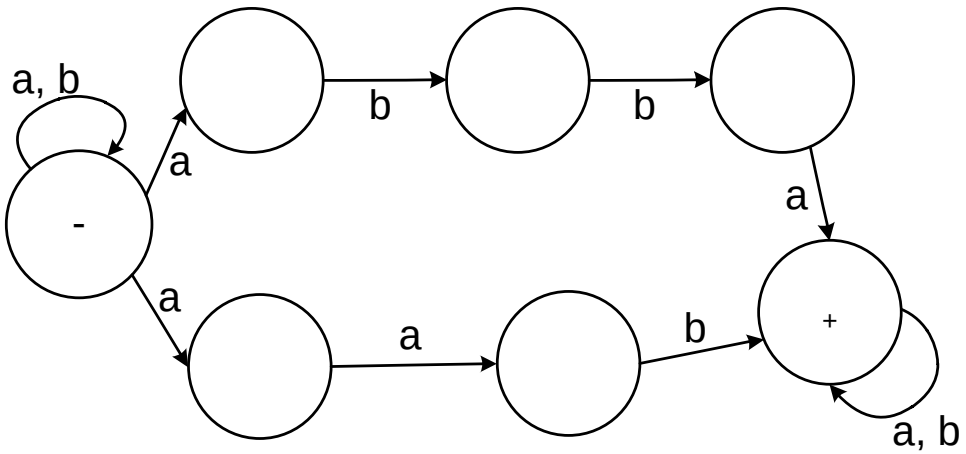
Self-test B Questions

Question B.1
Use Kleene's theorem to find a regular expression that generates the language accepted by the following TG (Transition Graph):



Question B.2
Problem 5(i) on page 143 in the 1997 edition.

Question B.3
Build a (deterministic) FA which accepts the same language as the following NFA:



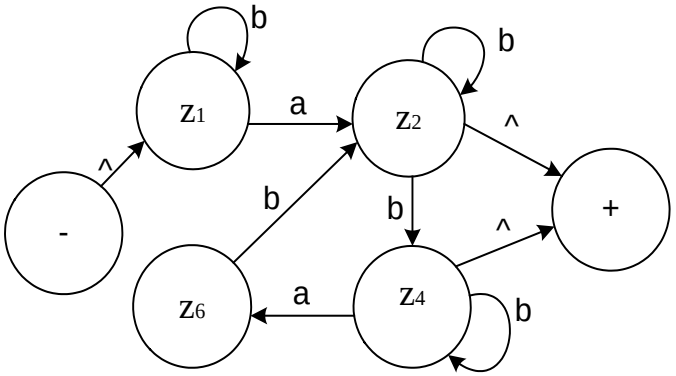
Question B.4
Problem 14(vii) on page 146 in the 1997 edition. Do not use the constructive algorithm presented in Proof 2 of Theorem 7. Instead analyse the NFA, decide which language is accepted by the NFA, and then provide an FA that accepts the same language as the NFA.

Question B.5
Refer to problem 3(v) on page 165 in the 1997 edition: Convert the Moore machine to a Mealy machine.

Solution to Self-test B

Hints for finding a regular expression that generates a language accepted by a given TG (transition graph). (Refer to the algorithm in Kleene's theorem provided on p 106 in Cohen.)
Hints:

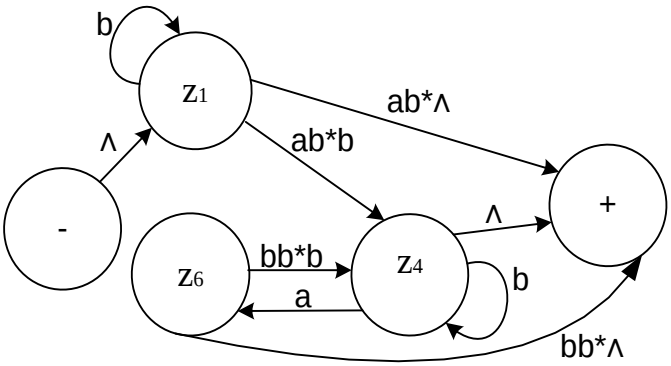
- Create a unique start state and final state if necessary.
- Possible dead end states can be omitted from the TG from which we will then deduce the regular expression.
- In the process of obtaining a regular expression for the language, we should eliminate states one by one. The start and final state should be present in the final TG.
- **Each incoming** edge to a state must be connected to **each outgoing** edge from this state to create new paths. For example, if there are 2 incoming edges to a state z_2 (say) and 2 outgoing edges from this state, 4 new paths should be created. A new path begins at the state where the incoming edge came from (began) and the new path ends where the outgoing edge ended. If a loop is present at such a state z_2 then the path should go through the loop. We will look at an example by referring to the following TG:



State z_2 has 2 incoming edges and 2 outgoing edges. We create 4 new paths:
We get

- *an edge with label ab*Λ from state z_1 to the final state,*
- *an edge with label bb*Λ from state z_6 to the final state,*
- *an edge with label ab*b from state z_1 to state z_4, and*
- *an edge with label bb*b from state z_6 to state z_4.*

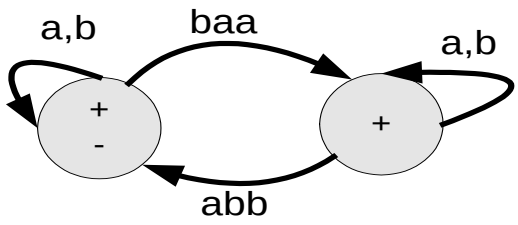
Note that all the new paths go through the loop on state z_2 so that b is included in all the new labels. Take note of where the new paths begin and end. A TG with the new paths:*



Now you can complete the task by eliminating the states one by one.

Question B.1

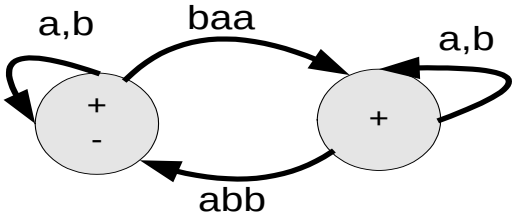
Use Kleene's theorem to find a regular expression that generates the language accepted by the following TG (Transition Graph):



Answer

Note: When a state must be eliminated, each incoming edge must be connected to each outgoing edge. We are required to follow the algorithm given in Cohen, page 106.

We have to find a regular expression which is equivalent to the language accepted by the following transition graph.

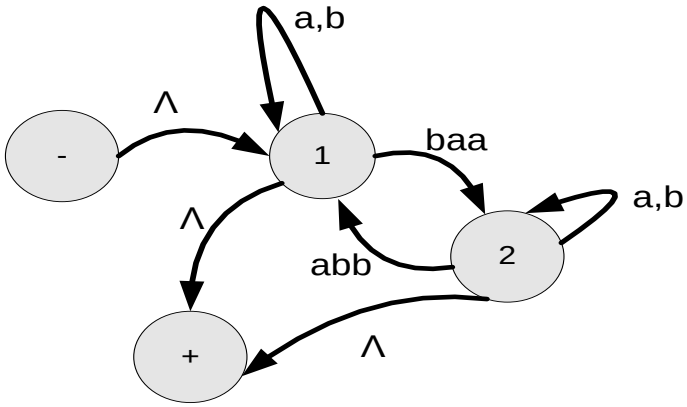


It is obvious that all possible words consisting of 0 or more *a*'s and/or *b*'s are accepted, thus

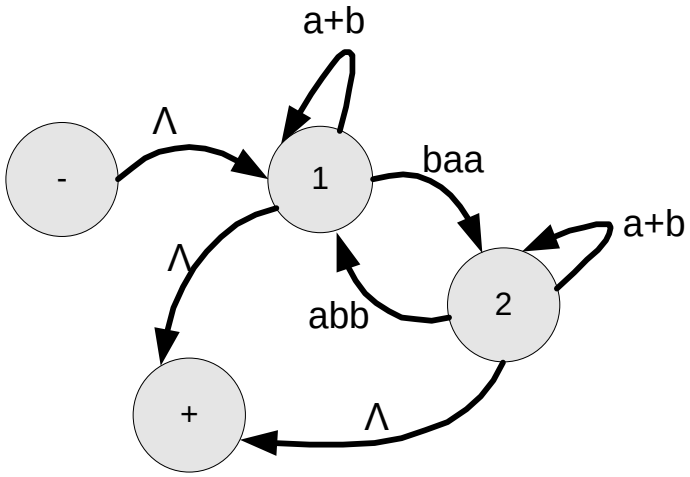
(**a** + **b**)*

will be a correct regular expression. However, we are required to use the algorithm within Kleene's theorem and we proceed as follows:

Step 1: Create a unique start state and a unique final state. (To simplify the explanation the states are numbered.)



Step 2: The labels of state 1 and state 2 can be changed as follows:



Step 3: We eliminate state 2:

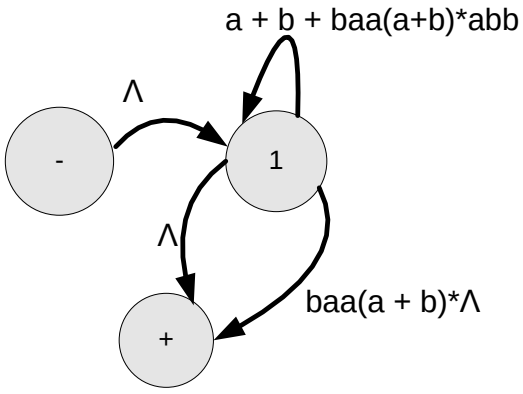
Each incoming edge to state 2 must be connected to each outgoing edge from state 2. The label on the resultant edge is the concatenation of the label on the incoming edge with the Kleene star of the label on the loop edge, which is then concatenated with the label on the outgoing edge. (Refer to Cohen, the algorithm (Step 2), page 106.)

There is one incoming edge with label *baa* to state 2, a loop edge on state 2 with label (**a** + **b**) (we must introduce the Kleene star here - refer to Cohen, page 96), and there are two outgoing edges from state 2 with labels *abb* and Λ respectively. Thus two edges will bypass state 2.

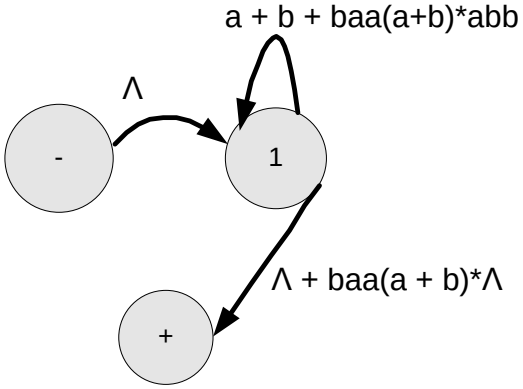
We determine the labels of these two resultant edges:

(i) Concatenate the label on the incoming edge (*baa*) with the Kleene star of the label on the loop edge ((**a** + **b**)*) and the label on the outgoing edge (*abb*). The result is a loop edge on state 1 with label **baa**(**a** + **b**)***abb**. (Note that the incoming *baa*-edge *begins* at *state 1* and the outgoing *abb*-edge *ends* at *state 1*.) The label on the loop on state 1 now becomes **a** + **b** + **baa**(**a** + **b**)***abb**. (Refer to Cohen, page 95).

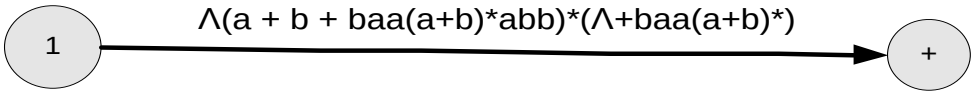
(ii) Concatenate the label on the incoming edge (*baa*) with the Kleene star of the label on the loop (**a** + **b**)* and the label on the outgoing edge (Λ). The result is an edge with label **baa**(**a** + **b**)*Λ. This edge connects state 1 and the final state. (Note that the incoming *baa*-edge *begins* at *state 1* and the outgoing Λ-edge *ends* at the *final state*.) The label on this edge is **baa**(**a** + **b**)*Λ. (Refer to Cohen, page 95).



Step 4: We can combine the two outgoing edges from state 1: (Refer to Cohen, page 95.)



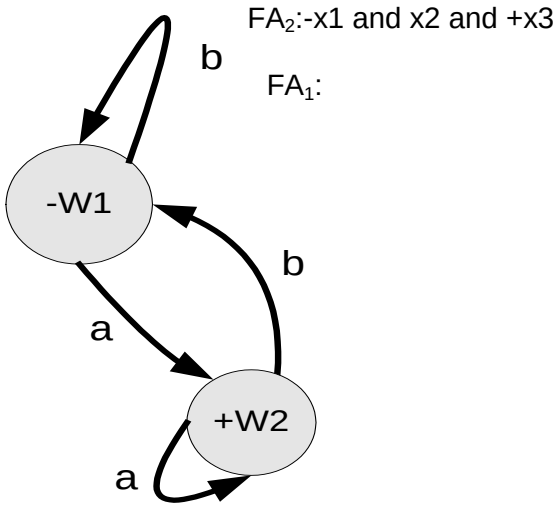
Step 5: Finally we eliminate state 1:
There is one incoming edge to state 1 and one outgoing edge from state 1. Note the Kleene star at the label of the loop: (**a** + **b** + **baa**(**a** + **b**)***abb**)*.



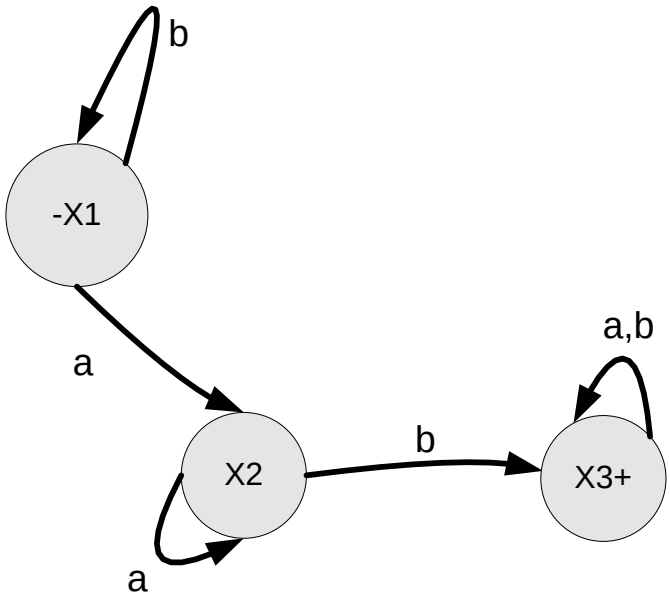
Thus a correct regular expression is:
(**a** + **b** + **baa**(**a**+**b**)***abb**)*(Λ +**baa**(**a**+**b**)*)
We may simplify this to: (**a** + **b**)*.

Question B.2
Problem 5(i) on page 143 in the 1997 edition.

Answer
The states of the respective machines are numbered as follows:
FA_1:-w1 and +w2



FA_2:



The start state of the new machine is the start state of FA_1, thus -z1 = -w1.
If we now read an *a*, we move to state w2. *Because w2 is a final state of FA_1, we must immediately add* **or x1**. Thus, if an *a* is read in state z1, we move to state z2 = w2 **or x1**.
If we read a b in state z1 = w1, we move to state w1, thus back to z1.

Let us now examine the states:

State z2 = w2 or x1
If we read an *a* in state z2 and we have been in w2, we go to w2 *and again we have to add "or x1"*. If we have been in x1, we move to state x2. Therefore we have a new z state, namely z3 = w2 or x1 or x2.
Suppose we read a *b* in state z2 then if we have been in w2, we go to state w1 and if we have been in state x1, we stay in state x1. So, if a *b* is read, we go to z4 = w1 or x1.

State z3 = w2 or x1 or x2
If an *a* is read in w2, we remain in w2 and again add "or x1". If we have been in x1, we go to x2. If we have been in x2, we stay in x2. This means that, if an *a* is read in z3, we stay in z3 where z3 = w2 or x1 or x2.
Suppose a *b* is read in z3. If we have been in w2, we move to w1. If we have been in x1, we stay in x1 and if we have been in x2, we move to state x3. *Because x3 is a final state of the second FA, this z state is also a final state.* Therefore, if we read a *b* in state z3, we go to a state +z5 where +z5 = w1 or x1 or x3.

State z4 = w1 or x1
Suppose we read an *a* in this state. If we have been in w1, we go to "w2 or x1" and if we have been in x1, we go to x2. Thus if an *a* is read in z4, we go to w2 or x1 or x2 which is equal to state z3.
Suppose we read a *b* in z4. If we have been in w1, we stay there and if we have been in x1, we stay there. Therefore if a *b* is read in z4, we stay in state z4.

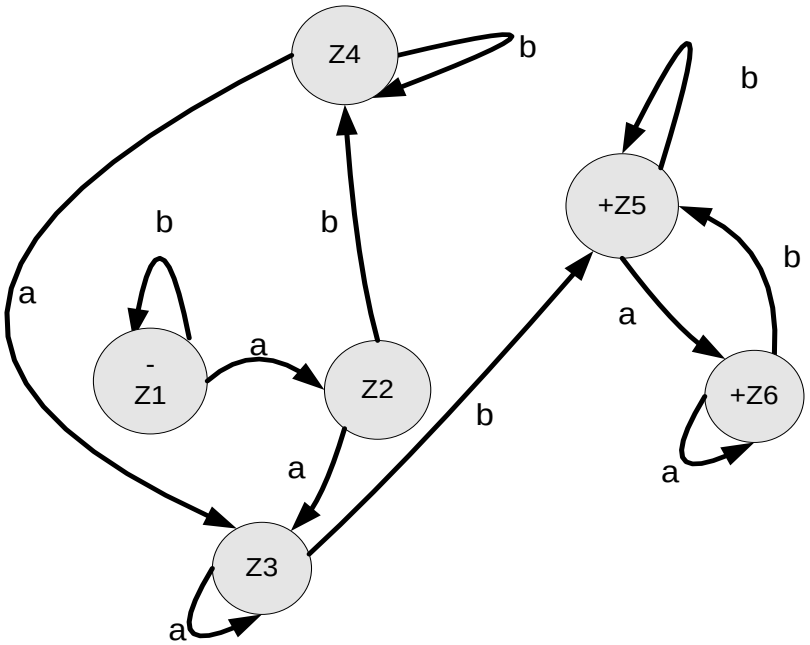
State +z5 = w1 or x1 or x3
If an *a* is read in w1 we move to w2 or x1. If we have been in x1 we go to x2. If we have been in x3 we stay in x3. Therefore we introduce a new z state, + z6 = w2 or x1 or x2 or x3. State z6 is a final state because one of the states it consists of, namely x3, is a final state of FA2.
If a *b* is read in state z5, we stay in w1 if we have been there. If we have been in x1, we also stay there and similarly for x3. Therefore if a *b* is read in state z5, we remain there.

State +z6 = w2 or x1 or x2 or x3
Suppose we read an *a* in this state. If we have been in w2, we stay there. Thus we start with “w2 or x1”. If we have been in x1, we go to x2. If we have been in x2, we stay there and if we have been in x3, we stay there. Therefore we go to w2 or x1 or x2 or x3 which is precisely state z6.
Suppose we read a *b* in state z6. If we have been in w2, we move to w1 and if we have been in x1, we stay there. If we have been in state x2, we move to x3. If we have been in x3, we stay there. This means that if reading a *b* in state z6, we go to w1 or x1 or x3, thus to state z5.

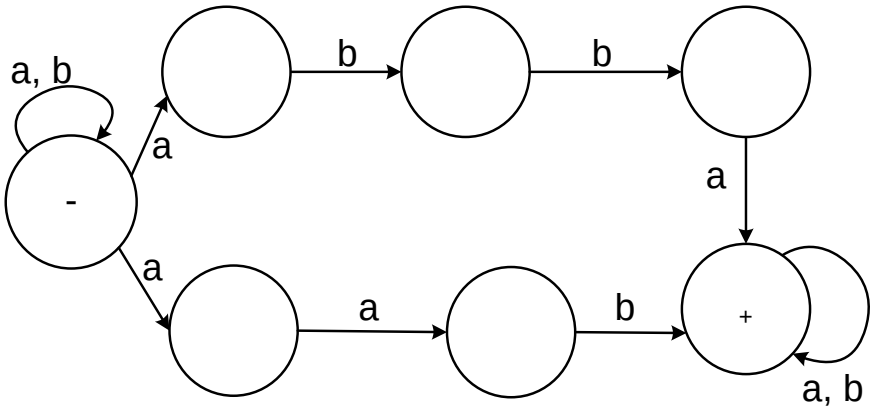
A tabular representation of FA_1FA_2 follows:

New state	**Old states**	**Read an *a***	**Read a *b***
-z1	w1	z2 = w2 or x1	-z1 = w1
z2	w2 or x1	z3 = w2 or x1 or x2	z4 = w1 or x1
z3	w2 or x1 or x2	z3 = w2 or x1 or x2	+z5 = w1 or x1 or x3
z4	w1 or x1	z3 = w2 or x1 or x2	z4 = w1 or x1
+z5	w1 or x1 or x3	+z6 = w2 or x1 or x2 or x3	+z5 = w1 or x1 or x3
+z6	w2 or x1 or x2 or x3	+z6 = w2 or x1 or x2 or x3	+z5 = w1 of x1 or x3

The resultant FA looks as follows:



Question B.3
Build a (deterministic) FA which accepts the same language as the following NFA:



Answer
First of all we have to determine which language is accepted by the given NFA. Well, it is clear that all words that are accepted should somewhere contain the substring

abba

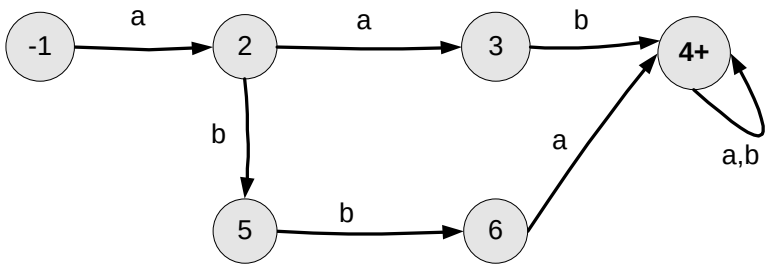
or the substring

aab.

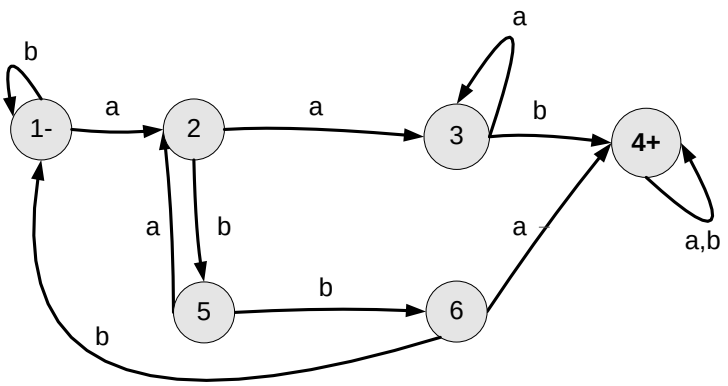
Let us label our start state 1. If we read an *a*, we move to state 2. The letter *a* may be the first letter of either of the two characterising substrings. If we read another *a* in state 2, we go to state 3. It is possible that we have now read the first two letters of the substring *aab*. If we, therefore, read a *b* in state 3, this word must be accepted and we go to a final state, say state 4. We never leave the final state - it does not matter how many (if any) *a*'s and *b*'s follow this substring.

At this stage the FA looks as follows:

Suppose we read a *b* in state 2. This *b* may be part of the substring *abba*. Thus we go to state 5 and if we read another *b* in state 5 then we go to state 6. If we read an *a* in state 6, we know the word should be accepted. Thus we go to the final state. Up to now we have constructed the following:



Of course this is not the complete FA. What happens when we read a *b* in state 1 or an *a* in state 5, et cetera? Let us see:
Both the characteristic substrings start with the letter *a*. Therefore if we read a *b* in state 1, we have to stay there until we eventually encounter an *a*. Suppose we read an *a* in state 3. Well, it is possible that we may still have a substring *aab* and the best move will be to stay in state 3. However, if we read an *a* in state 5, we go back to state 2, because this may be the first letter of one of the characterising substrings. Suppose we read a *b* in state 6. This letter cannot be the start of either of the substrings and we have to go back to the beginning (thus to state 1) and again wait for an *a*. The complete FA now looks as follows:



Question B.4
Problem 14(vii) on page 146 in the 1997 edition, but do **not** use the constructive algorithm presented in Proof 2 of Theorem 7. Instead analyse the NFA, decide which language is accepted by the NFA, and then provide an FA that accepts the same language as the NFA.

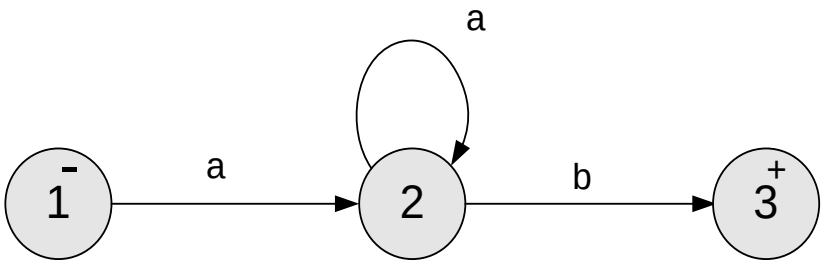
Answer
First of all we have to determine which language is accepted by the given NFA. Well, it is clear that all words that are accepted should start with *a* and end in *b.* Thus the shortest word accepted is:

ab.

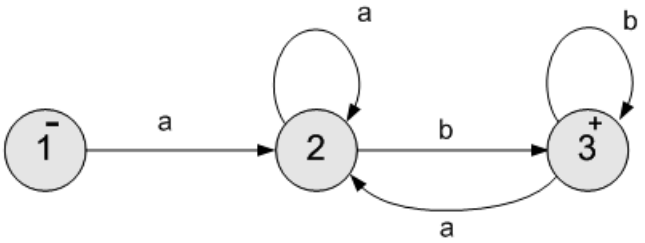
A regular expression that generates the language accepted by the given NFA is:

a(**a** + **b**)*__b__.

Let us label our start state 1. If we read an *a*, we move to state 2. If we read another *a* in state 2, we loop back into state 2. If we are in state 2 and we read a *b* we move to state 3. State 3 is a final state. Thus far the FA looks as follows:



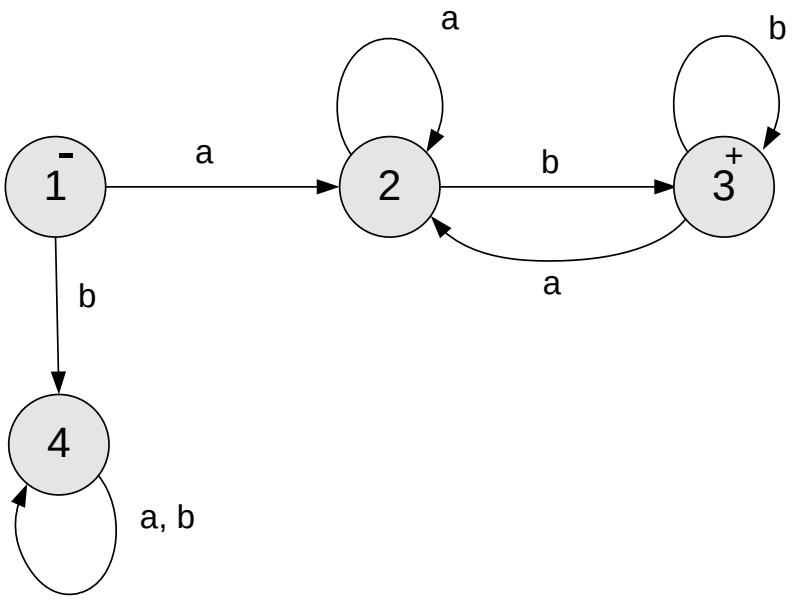
Suppose we read a *b* in state 3. Then we loop right back into state 3 – words like *abb* and *abbbb* should be accepted. However, if we read an *a* in state 3 then we move to state 2 because words ending with an *a* should not be accepted.
The FA looks now as follows:



Of course this is not the complete FA. What happens when we read a *b* in state 1?

If we read a *b* in state 1, we have to move to a new state, state 4. Words starting with a *b* cannot be accepted. Thus if we are in state 4 and we read either an *a* or a *b*, then we loop back to state 4.

Our complete FA:

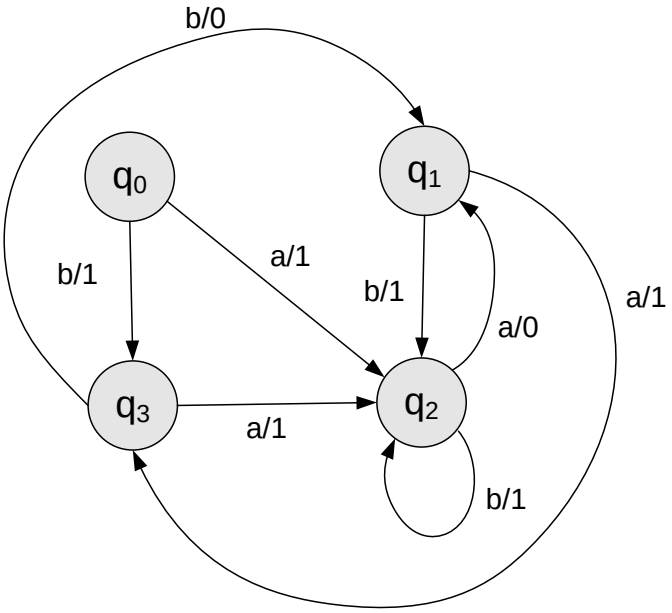


Question B.5

Convert the Moore machine in problem 3(v) on page 165 in the 1997 edition to a Mealy machine.

Answer

The equivalent Mealy machine of the given Moore machine:



16 Learning Unit 9 – Regular Languages

Study Material

Cohen
You need to study chapter 9 in the prescribed book.

Time allocated
You will need one week to study this unit.

Vodcasts
You can find a video showing an example of using the algorithms for finding the intersection language of two languages. It is recommended that you work through the sections in the prescribed book, and this document, before watching the videos.

Notes

In these notes we will examine some properties of regular languages. After studying this learning unit, you should be able to

- show that the set of regular languages is closed under the operations of union, concatenation, closure, complementation and intersection;
- find, for given regular languages L_1, and L_2, a regular expression and an FA defining $L_1 \cap L_2$.

Union, intersection and laws	The introduction to set theory provided in COS1501 will be needed to understand this chapter in Cohen. In particular, De Morgan's laws $(A \cap B)' = A' \cup B'$ and $(A \cup B)' = A' \cap B'$ and the involution law $(A')' = A$ will be used. Note that we use the algorithm applied by Cohen in "Good proof of theorem 12".
Complement	Of course, $A' = U - A$ Where U is some universal set. So before we can talk sense about the complement A' of A, we need to be clear about the universal set in the relevant context. The sets in this chapter are all sets of strings over some alphabet $\sum$, so the universal set U is always the set of *all* strings over $\sum$.

Recommended problems 9.1

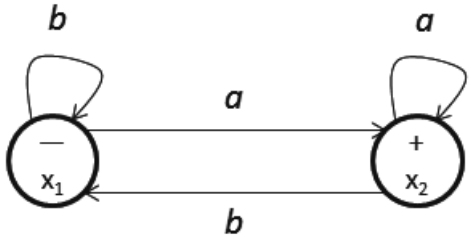
Do the following problems to consolidate your knowledge of the work in this learning unit:
2 (long method) on page 185,
4, 17, 19(ii) (do these three using the short method) on page 185 and 186, and
20 on page 186.

Recommended problems 9.1 – solutions

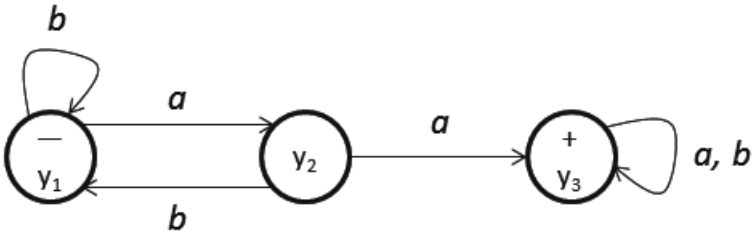
Problem 2

We want to determine $L_1 \cap L_2 = (L_1' + L_2')'$.

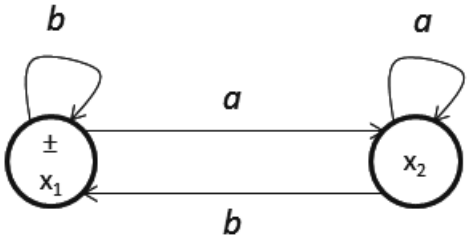
L_1: (**a** + **b**)***a**



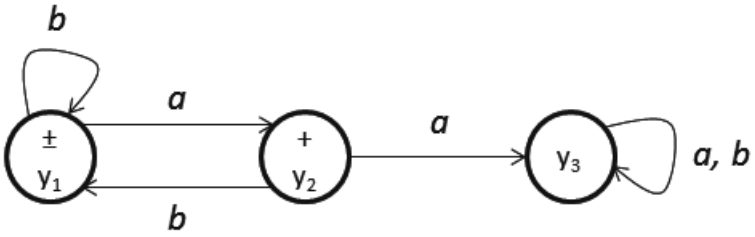
L_2: (**a** + **b**)***aa**(**a** + **b**)*



From the above we have:
L_1':

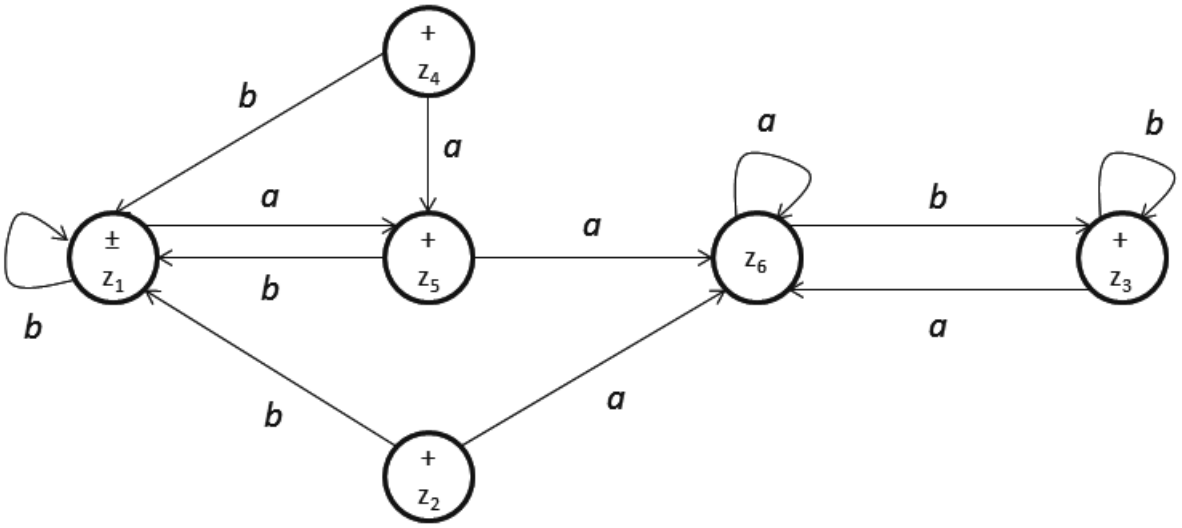


L_2':

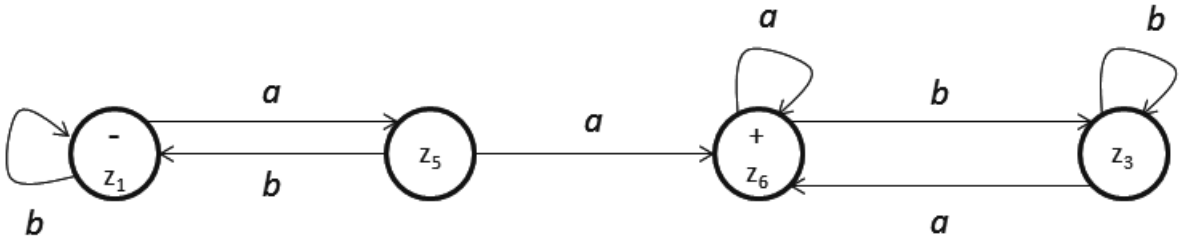


Now we have $L_1' + L_2'$:

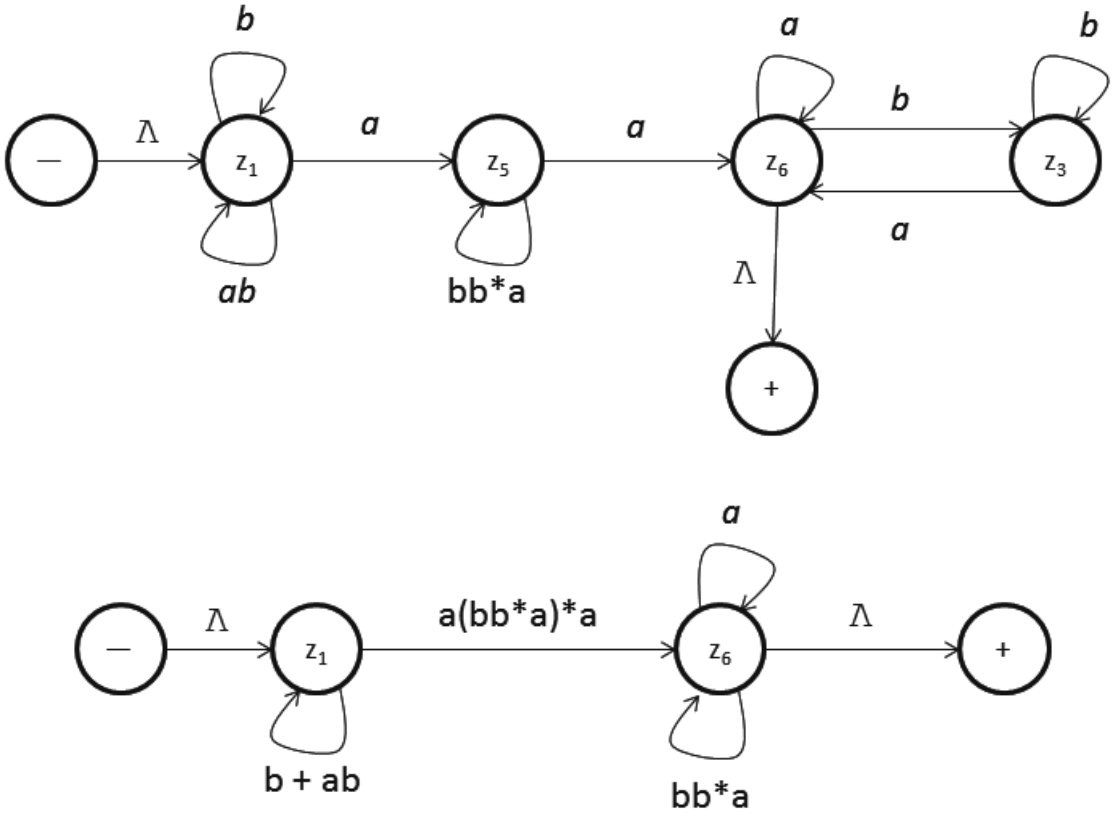
New state	Read an *a*	Read an *b*
$\pm z_1$ = x_1 or y_1	z_5	z_1
$+z_2$ = x_1 or y_2	z_6	z_1
$+z_3$ = x_1 or y_3	z_6	z_3
$+z_4$ = x_2 or y_1	z_5	z_1
$+z_5$ = x_2 or y_2	z_6	z_1
z_6 = x_2 or y_3	z_6	z_3

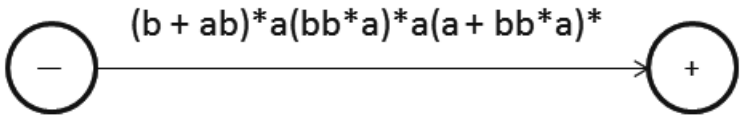


$(L_1' + L_2')'$: We may ignore z_2 and z_4 as there are no edges going to them.



Now we must find a regular expression.



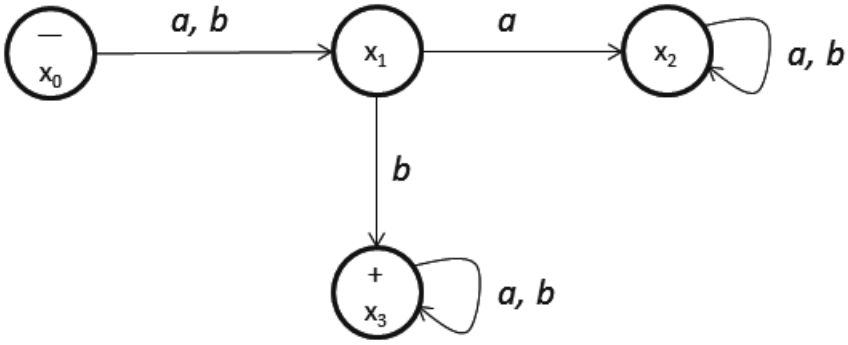


Thus (**b** + **ab**)**a*(**bb******a**)**a*(**a** + **bb******a**)* is a correct regular expression.

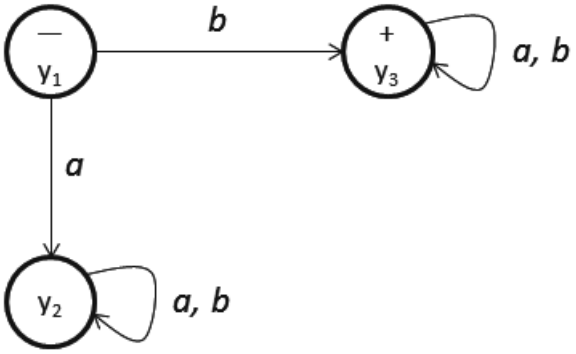
Problem 4

We are going to determine $L_1 \cap L_2$ with the short method. Every final state of $L_1 \cap L_2$ will consist of *a final state of L_1 and a final state of L_2.*

L_1: (**a** + **b**)**b**(**a** + **b**)*

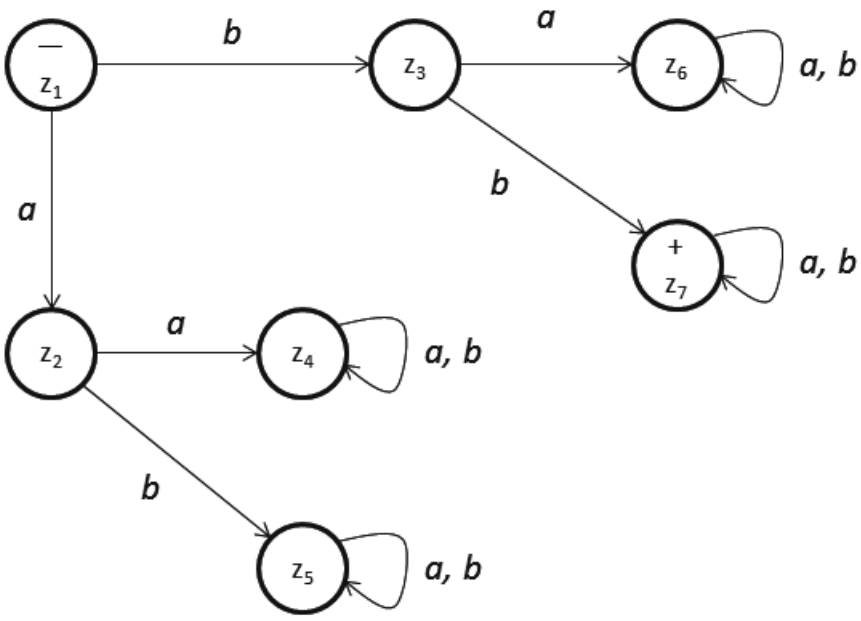


L_2: **b**(**a** + **b**)*



$L_1 \cap L_2$:

New state	Read an *a*	Read an *b*
$-z_1 = x_0$ or y_1	z_2	z_3
$z_2 = x_1$ or y_2	z_4	z_5
$z_3 = x_1$ or y_3	z_6	$+z_7$
$z_4 = x_2$ or y_2	z_4	z_4
$z_5 = x_3$ or y_2	z_5	z_5
$z_6 = x_2$ or y_3	z_6	z_6
$+z_7 = x_3$ or y_3	$+z_7$	$+z_7$



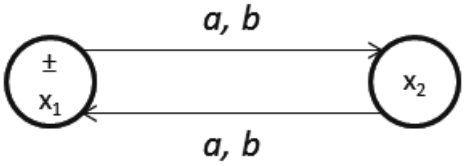
In order to write a regular expression, we immediately ignore the states z_2, z_4, z_5 and z_6 because we cannot reach the final state $+z_7$ from any of them. Our regular expression is:

bb(**a** + **b**)*.

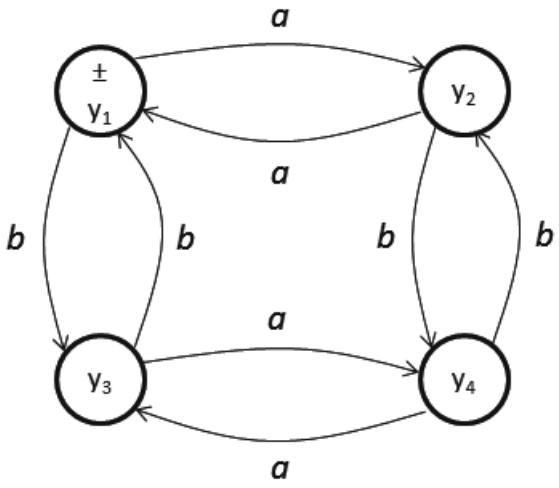
Problem 17

We know $L_2 \subseteq L_1$. Therefore $L_2 \cap L_1 = L_2$. We do it, however, in any case.

L_1:

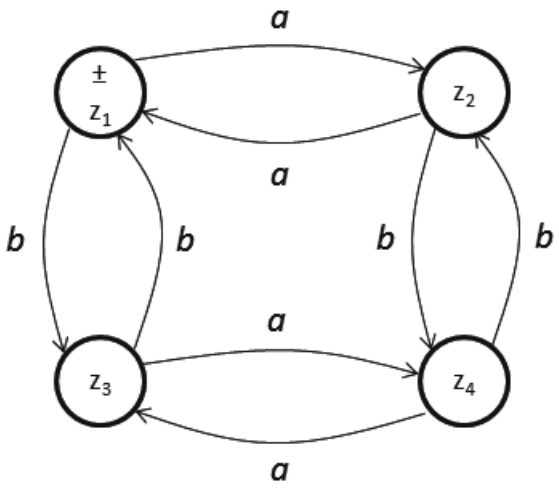


L_2:



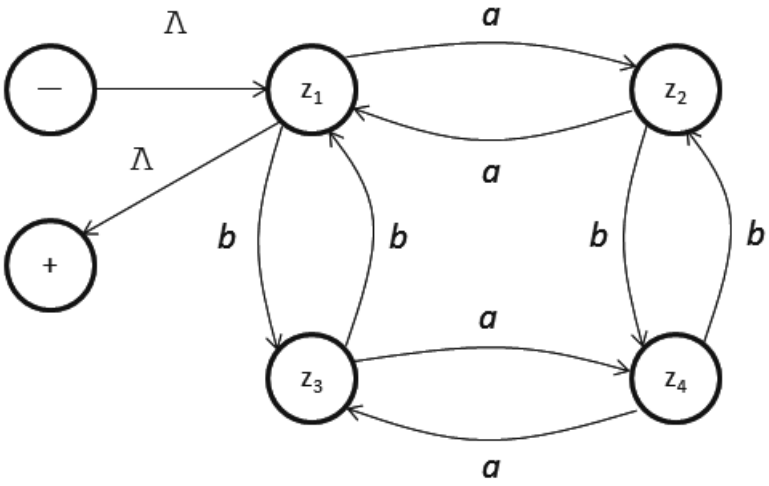
$L_1 \cap L_2$:

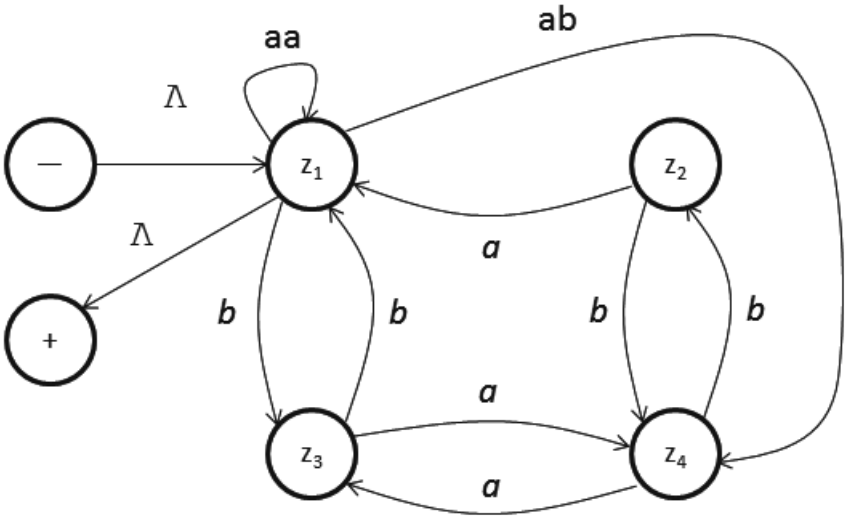
New state	Read an *a*	Read a *b*
$\pm z_1 = \pm x_1$ or $\pm y_1$	z_2	z_3
$z_2 = x_2$ or y_2	$+z_1$	z_4
$z_3 = x_2$ or y_3	z_4	$+z_1$
$z_4 = x_1$ or y_4	z_3	z_2

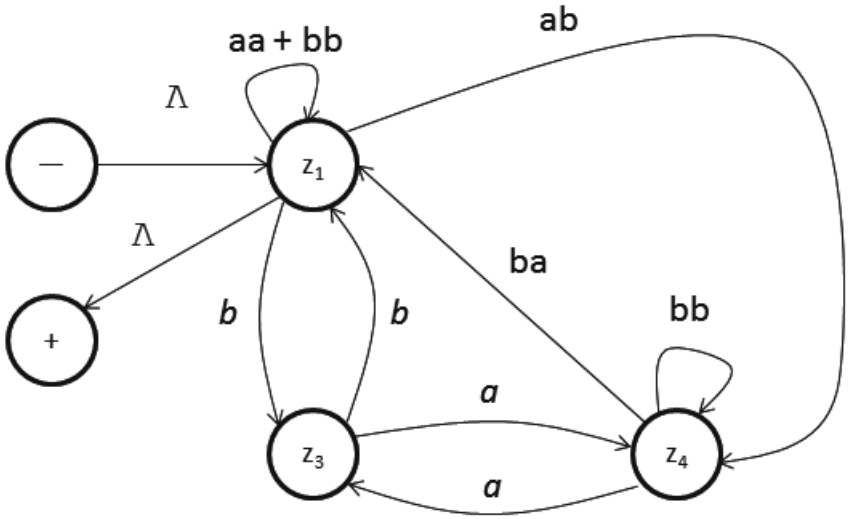


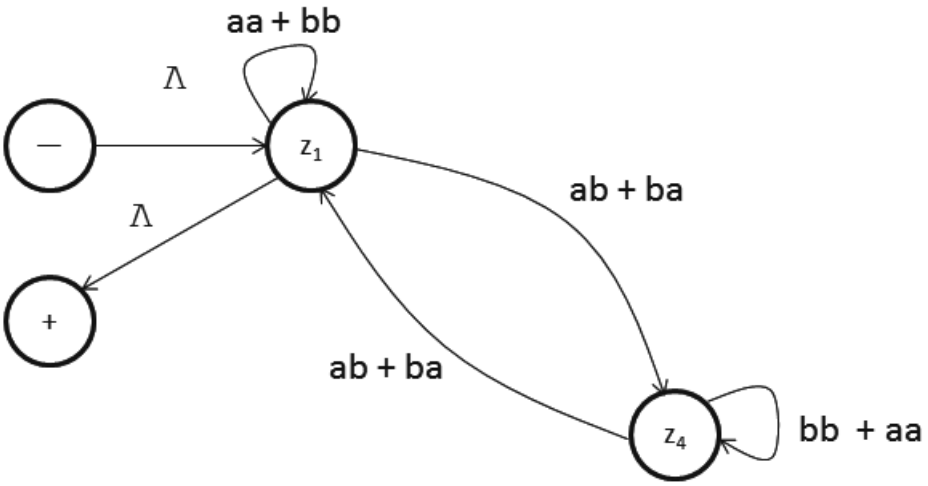
As expected, this is the same as the FA for L_2.

Let us build a regular expression from this:

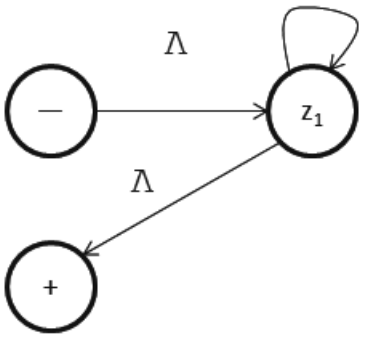



aa
ab
Λ
z1
z2
Λ
a
b
b
b
b
+
a
z3
z4
a


aa + bb
ab
Λ
z1
Λ
ba
b
b
bb
+
a
z3
z4
a


aa + bb
Λ
z1
ab + ba
Λ
+
ab + ba
z4
bb + aa

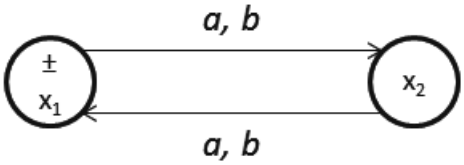
aa + bb + ((ab + ba) (aa + bb)* (ab + ba))



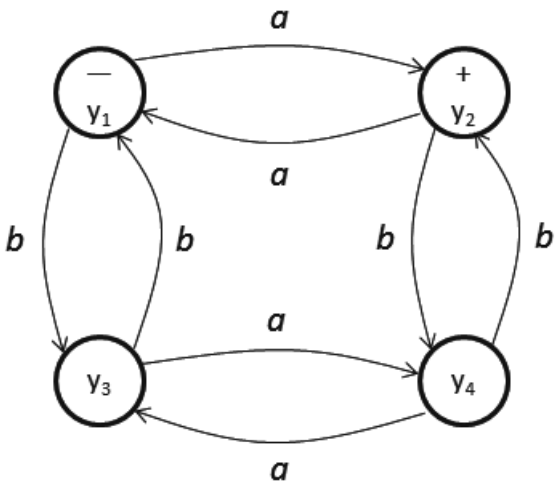
So the regular expression is: (**aa** + **bb** + (**ab** + **ba**)(**aa** + **bb**)*(**ab** + **ba**))*

Problem 19(ii)

L_1:



L_2:



We know that $L_1 \cap L_2 = \emptyset$. Let us see if our method yields the same result:

New state	Read an *a*	Read an *b*
$-z_1 = x_1$ or y_1	z_2	z_3
$z_2 = x_2$ or y_2	z_1	z_4
$z_3 = x_2$ or y_3	z_4	z_1
$z_4 = x_1$ or y_4	z_3	z_2

As expected there is no final state.

Problem 20

Consider the set V of all languages consisting of *a*'s and *b*'s (thus each member of V is a specific language). Now the following subsets of V are examples of collections of languages that are closed under intersection and union but not closed under complement:

(a) the subset of V which consists of all the finite languages (the union and the intersection of any two such languages will be a finite language but the complement of a finite language will include all the infinite languages),

(b) the subset of V which consists of all the languages where the words are of even length,

(c) the subset of V which consists of all the languages where the words start with an *a*.

17 Learning Unit 10 – Non-regular Languages

Study Material

Cohen
You need to study chapter 10 in the prescribed book.

Time allocated
You will need one week to study this unit and unit 11.

Notes

In these notes we will introduce the idea of a *non-regular* language and investigate a technique (called the pumping lemma) for showing that a language is non-regular. After studying this learning unit you should be able to

- give examples of non-regular languages;
- apply the pumping lemma with length and the pumping lemma without length to show that a given language is non-regular.

The pumping lemma

What is the basic idea behind the pumping lemma? Suppose you have a set of strings over some alphabet and you can't find a regular expression or finite automaton or transition graph defining the set. Then you may begin to suspect that the set is not regular. The problem is, how can you *prove* that it is non-regular?

10.1 Method of proof

Reductio ad absurdum	To establish the non-regularity of a language, you can try to use *reductio ad absurdum*. You will recall from COS1501 that *reductio ad absurdum* is one method of proof. It involves finding a situation in which there are only two possibilities: a good one and a bad one. For example, there are only two possibilities for a set of strings: either it is a regular language or it is non-regular. The *good* possibility is the one that corresponds to what you're trying to prove which, in this case, is that the set is non-regular. Good = non-regular Bad = regular Now one fastens onto the *bad* possibility and tries to eliminate it. How? Well, both versions of the pumping lemma tell us that if our infinite set is regular, then we can build a certain kind of string inside it. Therefore our strategy is to try and build a string of this type and to show that it violates the description of the language with which we began. This gives us a contradiction: we've assumed the set is regular and so the string *has* to belong to it, but the string can't belong to it because its original description excludes such strings. Once we have shown a contradiction we discard our assumption. That leaves only one possibility, namely that the set is non-regular. Which is what we wanted to prove.

Example of *reductio ad absurdum* and pumping lemma in Cohen	On page 193 Cohen uses this technique to prove that the infinite language $L = \{ a^n b^n \mid n \geq 0\}$ is non-regular. He begins by assuming that L is regular (i.e. the bad possibility). Then his argument shows that L has to contain a word having two substrings of the form *ab* or L has to contain a word having more *a*'s than *b*'s or L has to contain a word having more *b*'s than *a*'s, but this clearly violates the definition of L. Thus we get a contradiction and are led to discard the assumption that L is regular.
Example of *reduction ad absurdum* in Cohen	Similarly, we can use *reduction ad absurdum* (although it is not necessary to use the pumping lemma) to show that EQUAL is non-regular. We assume EQUAL is regular (the bad possibility), but this would imply that $L = \{ a^n b^n \mid n \geq 0\}$ is regular, which contradicts a previously established fact. Thus we are led to discard the assumption that EQUAL is regular.
More examples in Cohen	This technique was also used to prove that the languages PALINDROME and PRIME are non-regular.

10.2 The pumping lemma: Theorems 13 and 14

The pumping lemma is used to prove languages are non-regular. In the first place, the language had better be an infinite one, otherwise it would be silly to suspect it of non-regularity. Remember, way back in learning unit 4 we showed that all finite languages are regular. Let's consider the pumping lemma without length (given by Theorem 13) first.

Theorem 13 (The pumping lemma without length)

What does theorem 13 say?	Theorem 13 says: Let L be any regular language that has *infinitely* many words (each of finite length). Then there exist some strings *x*, *y* and *z*, with length(*y*) > 0 (i.e. $y \neq \Lambda$), such that *all* the strings $xy^n z$ with $n \in \{1, 2, 3,\ldots\}$ belong to L. In other words, the theorem says: IF L is regular, THEN there exist some strings x, y and z with the above properties. Note that the theorem DOES NOT SAY: "If there exist strings with the above properties, then L is regular". In other words, if L conforms to the pumping lemma (L has the pumping property), then the theorem *tells us nothing about whether L is regular or not*. If L does not conform to the pumping lemma, then we can conclude that L is non-regular. Note that the theorem says that there exist SOME strings *x*, *y* and *z*, if L is regular. It does NOT say that we can choose ANY word in L, and divide that word into *x*, *y* and *z* substrings in ANY way we choose.
The use of the pumping lemma without length	How must we use this theorem to prove that a language is non-regular? We do the following:
(i)	**Assume the language is regular.** Let us consider the language $L = \{ a^n b^{n+1}\} = \{abb, aabbb, aaabbbb, \ldots\}$ as an example. Assume L is regular.

(ii)	**Choose an arbitrary word from the given language to work with**. This means that we choose a typical word, not a specific one. We can choose w = $a^k b^{k+1}$ as a typical word from L (*not* a string like $a^5 b^6$).
(iii)	**Consider each possible type of choice of y, and show in each case that the string $xy^n z$ is not in the language for at least one n > 1**. (It is also sufficient to consider only the instance where n = 2.) Why do we only check every possible choice of *y*? When the substring *y* has been chosen, *x* is simply the part of the word before *y,* and *z* is the part after *y*. Let us consider our example again. We have to consider each possible type of choice of y for our chosen word w. The three cases are: (a) *y* consists only of *a*'s (b) *y* consists only of *b*'s (c) *y* contains the substring *ab* Remember that length(*y*) > 0. We must now show that all three cases lead to a contradiction when *y* is pumped. We can write our arbitrary word $w = a^k b^{k+1}$, with k > 0, as: $w = a^k b^{k+1} = xyz$ with length(*y*) > 0. According to the lemma, the string xy^2z must also be in L. (a) The string xy^2z contains more than k *a*'s but the number of *b*'s is still k + 1 – this is a contradiction since $xy^2z \notin$ L. (b) The string xy^2z contains more than k + 1 *b*'s, but only k *a*'s, thus $xy^2z \notin$ L. We have found a contradiction. (c) The string xy^2z contains the substring *ab* twice, thus $xy^2z \notin$ L. We have found a contradiction.
(iv)	**If every possible type of choice of y leads to a contradiction, we can conclude that our assumption in (i) is false, i.e. the language is non-regular.** In our example, every possible type of choice of *y* leads to a contradiction when *y* is pumped. We can conclude that L is non-regular.
Rather use Theorem 14	It is actually much easier to use Theorem 14 to prove that a language is non-regular. Let's see how it is done.

Theorem 14 (The pumping lemma with length)

What does Theorem 14 say?	Theorem 14 says: Let L be an infinite language accepted by an FA with N states. (Thus L is a regular language!) Then, for *all words w $\in$ L that have more than N letters* there are strings *x*, *y* and *z* with length(*y*) > 0 (i.e. $y \neq \Lambda$) and length(*x*) + length(*y*) ≤ N such that $w = xyz$ and *all* strings $xy^n z$ with n $\in$ {1, 2, 3, …} belong to L. Note again that the theorem does NOT say: "If there exist some strings with the above properties, then the language is regular".

Use of pumping lemma with length	The technique to prove that a language is non-regular by using the pumping lemma with length is a follows:
(i)	**Assume that the language is regular. Then it is accepted by an FA with (say) N states.** Let us again consider the language $L = \{ a^n b^{n+1} \mid n > 0\}$ as an example. Assume L is regular and L is accepted by an FA with N states. (It is not acceptable to assume that this FA has (say) 10 states.)
(ii)	**Choose an arbitrary word with more than N letters and write it as *xyz*. We know that length(*x*) + length(*y*) ≤ N and length(*y*) > 0**. Therefore we can choose our word in such a way that we only have to deal with one possible type of choice of *y*. This is sufficient since the lemma holds for *all* words with *more than* N letters. In our example we can choose $w = a^N b^{N+1}$ as a typical word with more than N letters, and write it as $w = a^N b^{N+1} = xyz$ with length(*x*) + length(*y*) ≤ N and length(*y*) > 0. (It is NOT acceptable to choose (say) $w = a^{10} b^{11}$.)
(iii)	**We now look for a contradiction by trying to show that, for every such choice of *y*, there is at least one n > 1 for which $xy^n z$ is not in the language**. (It is sufficient to consider only the case where n = 2.) If we have chosen our word *w* correctly, we only have to consider one type of choice of *y*. Let us discuss our example again. We have $w = a^N b^{N+1} = xyz$ with length(*x*) + length(*y*) ≤ N. This implies that both the strings *x* and *y can only consist of a's.* Now *xyyz* must also be in the language. When *y* is pumped, we obtain a string with more than N *a*'s, but only N + 1 *b*'s. This string is not in L.
(iv)	**If every possible type of choice of *y* leads to a contradiction, we can conclude that our assumption in (i) is false, i.e. the language is non-regular**. We have found that the only possible type of choice of *y* in our example leads to a contradiction. Our assumption in (i) is clearly false and we can conclude that L is non-regular.

Note that Cohen's comment on page 195 just before the last marker, ■, concerns an illustration. It is unacceptable with regard to a proof. A claim CANNOT be proved to be true by giving an example. (We may, however, prove a claim to be **not true** by giving a **counterexample**.)

10.3 Examples

Let's tackle a few of the problems at the end of chapter 10 in Cohen.

Problem 6(i) (Cohen, page 204)	We solve this problem by using the pumping lemma with length. Assume that SQUARE is regular. Then there exists some FA which accepts SQUARE. The FA has some number, say k, states. The string $w = a^{k \cdot k}$ belongs to SQUARE. By theorem 14 there is some way to write *w* as *xyz* (in other words, as $a^i a^m a^j$, since all the strings are strings of *a*'s) with $k \cdot k = i + m + j$ and $m > 0$ and $i + m \leq k$, such that $xy^n z \in$ SQUARE for all $n \geq 1$. So in particular $xyyz = a^i a^m a^m a^j \in$ SQUARE, which means there must exist some $l > k$ such that $i + m + m + j = l^2$. Now $i + m + j = k^2$, so $l^2 = k^2 + m$. But m cannot be greater than k, so $k^2 + m \leq k^2 + k$. Now $l > k$, so l cannot be smaller than k + 1, i.e. we have that $l^2 \geq (k + 1)^2$ and that $k^2 + m = l^2 \leq k^2 + k$. This means that $(k + 1)^2 = k^2 + 2k + 1 \leq k^2 + k$, which is clearly false. Hence we conclude that SQUARE cannot be regular.
Problem 1(v) (Cohen, page 203)	Let $L = \{ a^n b^n a^m \mid n \geq 0$ and $m \geq 0\}$, and assume that L is regular. We will first try to solve this problem by using Theorem 13.
Solution 1 (Pumping lemma without length)	Assume L is regular. The string $w = a^k b^k a^m = xyz$ is clearly in L. There are five possible cases: (a) *y* consists only of *a*'s from the first group of *a*'s. In this case xy^2z is a string with more than k *a*'s in the first group, but only k *b*'s. The string xy^2z is not in L and we have thus found a contradiction. (b) *y* consists only of *b*'s. In this case xy^2z is not in L since it has more than k *b*'s, but only k *a*'s. We have found a contradiction. (c) *y* contains the substring *ab*. The string xy^2z contains more than one *ab* substring and is not a word in L – a contradiction. (d) *y* contains the substring *ba*. The string xy^2z contains more than one *ba* substring, and is not a word in L – a contradiction. (e) *y* consists only of *a*'s from the second group of *a*'s. The number of *a*'s in the last group is increased when y is pumped, but there are still k *a*'s in the first group of *a*'s and k *b*'s. This results in a string that *is a word* in L, thus this case does *not* lead to a contradiction. This means that *we can conclude nothing about L* by applying the pumping lemma without length. Let us rather try to use the pumping lemma with length.

Solution 2 (Pumping lemma with length)	Assume L is regular. Thus L is accepted by some FA with k states. The string $w = a^k b^k a^m$ is clearly in L. Therefore by Theorem 14 there must be a way to write w as xyz, with length(x) + length(y) ≤ k and length(y) > 0, such that $xy^p z \in$ L for every p ≥ 1. Since length(x) + length(y) ≤ k, we can infer that x and y consist solely of a's. Thus if we consider xy^2z, then the number of initial a's has been increased. Whereas in $w = xyz$ the number of initial a's was the same as the number of b's, since $w = a^k b^k a^m$, it is clear that in xy^2z the number of initial a's is now greater than the number of b's. So $xyyz$ cannot belong to L. This gives the contradiction we seek. Hence L is non-regular. We have thus reached a conclusion by using the pumping lemma with length.

10.4 The language PRIME and problem 16

Primes	Cohen shows that PRIME is non-regular on pages 195 - 196. He describes PRIME as $\{ a^p \mid$ p is prime$\}$. This doesn't have any meaning if you don't know what a prime number is. Basically, a prime is an integer p bigger than 1 that has no factors other than 1, -1, p and –p. The set of prime numbers is defined as $P = \{ p \in \mathbb{Z}^+ \mid p > 1 \wedge p = ka \Rightarrow k = \pm 1, a = \pm p\}$. In other words, if you try to write a prime p as a product mn, then m and n will have to be chosen from the set {1, -1, p, -p}. Clearly an integer like 6 is not prime because we can write 6 as the product 2.3. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19. You are invited to find the next few primes, say up to 43. (If you think of a prime p as an integer possessing precisely *four* factors, namely 1, -1, p and –p, then it is clear that the integer 1 is not prime, for 1 has only two factors, namely 1 and -1.)
Problem 16 (page 205)	Note that in every word a^n, n is either a prime number or n is not a prime number. This means that PRIME′ is the complement of PRIME. We know that PRIME is non-regular (Cohen, pages 195-196). We don't need the pumping lemma for this one, although *reduction ad absurdum* is still the way to go.
16(i)	Assume PRIME′ is regular. Then, by Theorem 11 on page 173, PRIME is regular. But we have already shown PRIME to be non-regular. So we conclude that PRIME′ is non-regular.
16(ii)	Notice that the pumping lemmas are of limited usefulness – a number of non-regular languages satisfy the conditions of the theorems. For instance, consider the non-regular language PRIME′. Any string a^k in PRIME′ can be written as $a^k = xyz$ with $x = a^0$, $y = a^k$, and $z = a^0$. Now all strings of the form $xy^nz = a^0a^{nk}a^0 = a^{nk}$ will belong to PRIME′, since nk is not prime. We have thus shown that PRIME′ does satisfy the pumping lemma without length and *PRIME′ is non-regular*.

16(iii)	The moral of the story is that *all* regular languages behave in the way stipulated by the pumping lemmas, but so do *some* non-regular languages. To prove that these non-regular languages are in fact non-regular, some means other than the pumping lemmas must be employed.

Note

The following sections in Cohen, starting on page 196, and the problems based on them are not prescribed study material:
"The Myhill-Nerode theorem" and "Quotient Languages".

Recommended problems 10.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
1(i), 1(ii), 4(i), 5(i) for ODDPALINDROME, 7(i), and 10(i) on pages 203 to 205.

Recommended problems 10.1 – solutions

Problem 1(i)

Assume that S = $\{a^n b^{n+1}\}$ is a regular language. Say the FA that accepts this language (we know there must be such an FA from Kleene's theorem) has k states. We have

$w = a^k b^{k+1} \in S$

which we may write as

$w = xyz$

with length(y) > 0 and length(x) + length(y) $\leq$ k.

This implies that x and y consist of a's only. Now we must have

$xyyz \in S$,

but this is a contradiction because this word has more than k a's and still only k+1 b's. Thus S is not regular.

Problem 1(ii)

Assume that S = $\{a^n b^n a^n\}$ is regular and that the FA that accepts the language has k states.

So

$w = a^k b^k a^k \in S$

which we may write as

$w = xyz$

with length(x) + length(y) $\leq$ k and length(y) > 0.

Obviously both x and y consist of a's only. Now we must also have that $xyyz \in S$
but this cannot be true because this word has more than k a's at the beginning, then k b's and then k a's. Such words are *not* elements of S.

We conclude that our assumption was not correct and that S, therefore, cannot be regular.

Problem 4(i)

Assume that TRAILING-COUNT is regular. Then there exists an FA which accepts the language. The FA has a finite number of states and let us say the number of states is k.

Let $w = b^k$. Then

$wa^k \in$ TRAILING-COUNT.

We may write this word as $wa^k = xyz = b^k a^k$.

But length(x) + length(y) $\leq$ k, and length(y) > 0, so x and y consist of b's only. Say

$x = b^i$
$y = b^j \quad j \neq 0$
$z = b^m a^k, \quad$ so k = i + j + m.

Now we must have that also
$xyyz = b^i b^{2j} b^m a^k \in$ TRAILING-COUNT.

But k = i + j + m $\neq$ i + 2j + m.
So $xyyz \notin$ TRAILING-COUNT. Our initial assumption leads to a contradiction.
So TRAILING-COUNT is not regular.

Problem 5(i) for ODDPALINDROME

Assume that ODDPALINDROME is regular and that it is accepted by an FA with k states. We have that
$w = a^k b a^k \in$ ODDPALINDROME.

We may write w as
$w = xyz$
with length(x) + length(y) $\leq$ k and length(y) > 0.

This means that x and y consist of a's only. Now we must also have that
$xyyz \in$ ODDPALINDROME.
But this word has more than k a's, followed by one b followed by k a's and we know such a word *cannot* be in the language. Our assumption was incorrect. The language is *not* regular.

Problem 7(i)

Assume DOUBLESQUARE is regular and is accepted by an FA with k states.

We have that
$w = a^{k \cdot k} b^{k \cdot k} \in$ DOUBLESQUARE
and we may write w as
$w = xyz$
with length(x) + length(y) $\leq$ k and length(y) > 0.

This implies that both x and y consist of a's only. Now we must have that the word
$xyyz$
is an element of the language, but this is impossible since this word has more a's than b's. Our assumption that DOUBLESQUARE was regular was incorrect. The language is non-regular.

Problem 10(i)

Assume that S = $\{a^n b^n c^n\}$ is regular, thus there exists an FA with k states that accepts the language. Now we have that
$w = a^k b^k c^k \in$ S
and the word w may be written as
$w = xyz$
with length(x) + length(y) $\leq$ k and length(y) > 0.

This implies that y consists of a's only. Now we must also have that

$xyyz \in$ S.

But this word has more *a*'s than *b*'s and *c*'s, so it cannot be a word in S and we draw the conclusion that the language is non-regular.

Recommended problems 10.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
15, 17(i), 17(ii), and 17(iii) on pages 205 and 206.

Recommended problems 10.2 – solutions

Problem 15

(i) Let R be the language generated by the regular expression **a*****b*** (thus a regular language).
Let N = $\{a^n b^n\}$ (thus a non-regular language).
Then R + N = R, thus regular.

(ii) Remember that a finite language is always regular. Let R = {*aba aabaa bbbaaabaaabbb*} and
N = ODDPALINDROME.

Then R + N = N, thus non-regular.

Problem 17(i)

The regular language is generated by a regular expression, say **r**. We simply add all the other words (a finite number, say n) to the regular expression, thus

$\mathbf{r} + w_1 + w_2 + \ldots + w_n$.

Again we have a regular expression and the language generated by such an expression is, of course, regular.

Problem 17(ii)

Say our regular language is L_1 and the (finite, regular) language consisting of the words we want to subtract is L_2. According to Theorem 11, L_2' is regular too. The language that we have when we subtract L_2 from L_1 is

$L_1 \cap L_2'$

and this is regular according to Theorem 12.

Problem 17(iii)

We have an infinite set, say N, for which we cannot build an FA. If we add a finite number of words, say a set S, to this set, we get a new set

T = N + S.

Let us assume T is regular. If we subtract S from this we must get a regular language according to 17(ii). But we are left with N when we subtract S from T and N is non-regular! Our assumption that T is regular is incorrect.

18 Learning Unit 11 – Decidability

Study Material

Cohen
You need to study chapter 11 in the prescribed book.

Time allocated
You will need one week for learning unit 10 and this unit.

Notes

These notes will introduce the concept of decidability. After studying this learning unit, you should be able to

- define the notions of "decision procedure" and "decidable problem";
- determine whether two FAs accept the same language or not;
- determine whether a given FA accepts any string at all;
- determine whether a given FA accepts a finite or an infinite language.

De Morgan's law and the product rule	Note that on page 212 Cohen uses *De Morgan's laws* with the symbol + being used to denote union (i.e. $\cup$). The *product rule* in combinatorics is essential to understand, and why on page 216 there are m^N strings of length N, and m^{N+1} strings of length N + 1 and so on. The machine has N states and the alphabet has m letters. Think of building an input string with N letters, then there are m possible ways of choosing the first letter, m choices for the second letter, …, and m choices for the Nth letter, which means that there are m·m·….m (N times) i.e. m^N ways in which the string can be built.
Equivalence of FAs	When are two FAs equivalent? In other words, when are two languages, L_1 and L_2, accepted by two given FAs, equivalent? The answer: *If the FA for* $(L_1' + L_2)' + (L_1 + L_2')'$ *accepts any words, then the two FAs for L_1 and L_2 are not equivalent*.
Procedure for showing equivalence	To decide whether the two FAs are equivalent, we follow these steps:
(i)	Build the first part of the above machine, $(L_1' + L_2)'$.
(ii)	IF the part you built in (i) accepts any words, then the FAs for L_1 and L_2 are not equivalent. Stop. ELSE build the second part of the machine, $(L_1 + L_2')'$.

(iii)	IF	the second part accepts any words, then the FAs for L_1 and L_2 are not equivalent. Stop.
	ELSE	the FAs for L_1 and L_2 are equivalent. Stop.
		Notice that you may build the second part $(L_1 + L_2')'$ in step (i) and then build the first part $(L_1' + L_2)'$ in the ELSE part of step (ii). It is clearly never necessary to build the machine for the union of the two parts.

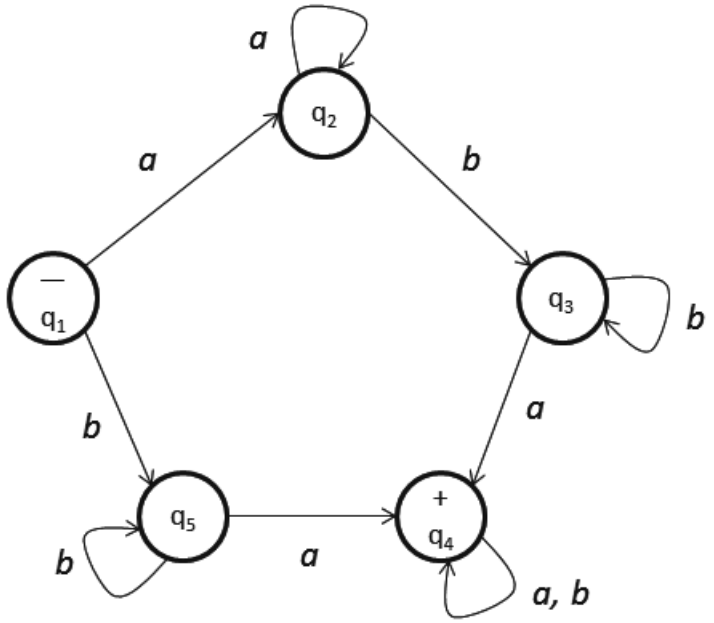
Recommended problems 11.1

Do the following problems to consolidate your knowledge of the work in this learning unit:
1 and 3 on page 217.

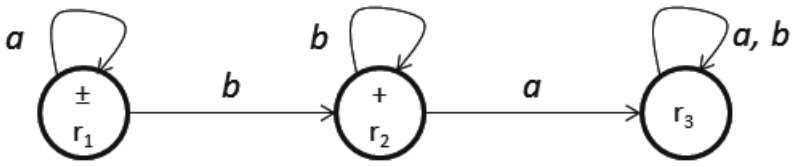
Recommended problems 11.1 – solutions

Problem 1

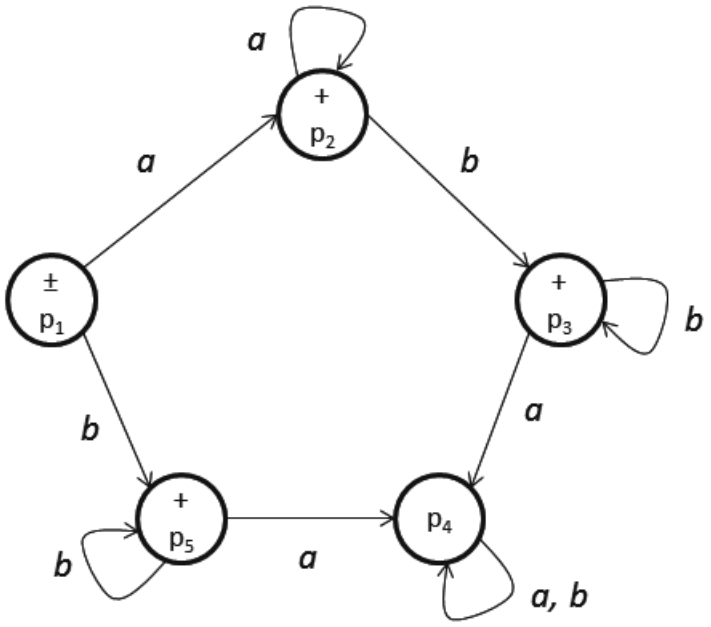
FA_1':



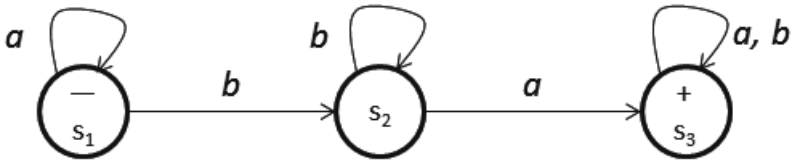
FA_2:



FA_1:



FA_2':



$(FA_1' + FA_2)$:

New state	Read an *a*	Read an *b*
$\pm z_1 = -q_1$ or $+r_1$	$+z_2$	$+z_3$
$+z_2 = q_2$ or $+r_1$	$+z_2$	$+z_4$
$+z_3 = q_5$ or $+r_2$	$+z_5$	$+z_3$
$+z_4 = q_3$ or $+r_2$	$+z_5$	$+z_4$
$+z_5 = +q_4$ or r_3	$+z_5$	$+z_5$

This means that $(FA_1' + FA_2)'$ will have no final states.

$(FA_1 + FA_2')'$:

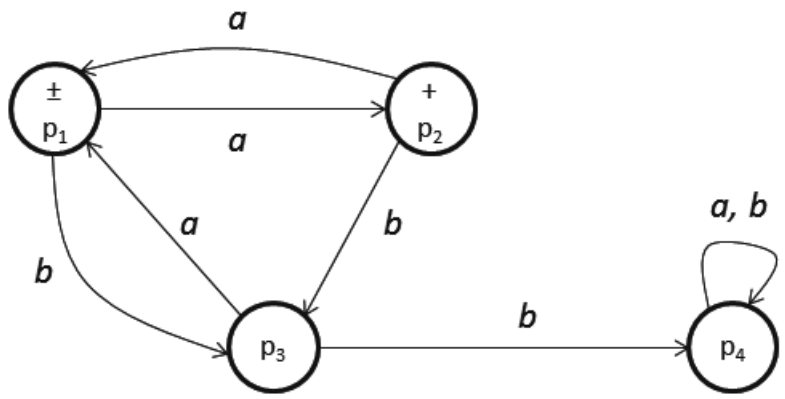
New state	Read an *a*	Read an *b*
$\pm z_1 = \pm p_1$ or $-s_1$	$+z_2$	$+z_3$
$+z_2 = +p_2$ or s_1	$+z_2$	$+z_4$
$+z_3 = +p_5$ or s_2	$+z_5$	$+z_3$
$+z_4 = +p_3$ or s_2	$+z_5$	$+z_4$
$+z_5 = p_4$ or $+s_3$	$+z_5$	$+z_5$

We see that also $(FA_1 + FA_2')'$ will have no final states.

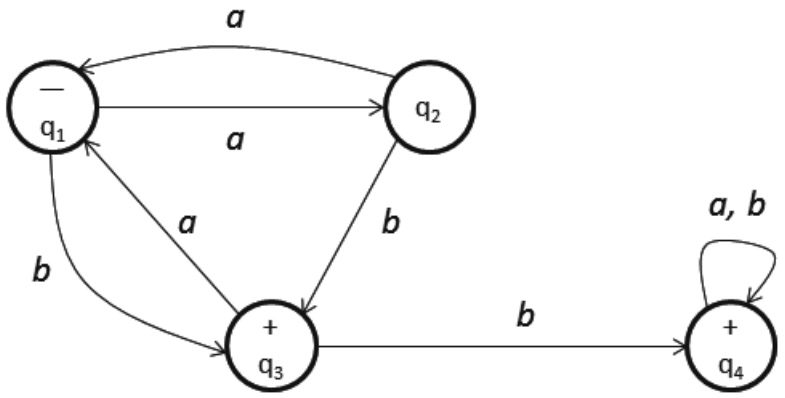
Thus $(FA_1' + FA_2)' + (FA_1 + FA_2')'$ accepts no words and that means that FA_1 and FA_2 are equivalent.

Problem 3

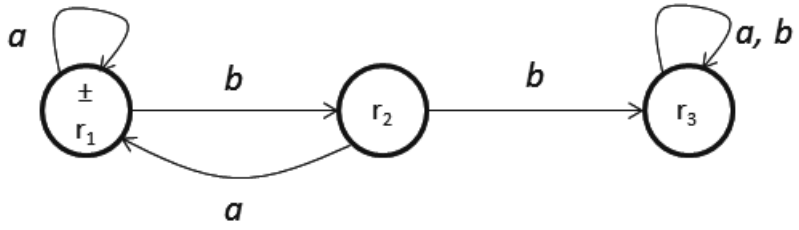
FA_1:



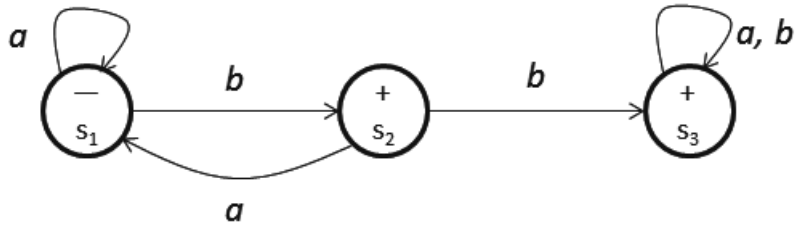
FA_1':



FA_2:



FA_2':



$(FA_1' + FA_2)$:

New state	Read an *a*	Read an *b*
$\pm z_1 = q_1$ or $+r_1$	$+z_2$	$+z_3$
$+z_2 = q_2$ or $+r_1$	$+z_1$	$+z_3$
$+z_3 = +q_3$ or r_2	$+z_1$	$+z_4$
$+z_4 = +q_4$ or r_3	$+z_4$	$+z_4$

The complement $(FA_1' + FA_2)'$ does not have any final states.

$(FA_1 + FA_2')$:

New state	Read an *a*	Read an *b*
$\pm z_1 = +p_1$ or s_1	$+z_2$	$+z_3$
$+z_2 = +p_2$ or s_1	$+z_1$	$+z_3$
$+z_3 = p_3$ or $+s_2$	$+z_1$	$+z_4$
$+z_4 = p_4$ or $+s_3$	$+z_4$	$+z_4$

The complement $(FA_1 + FA_2')'$ also has no final states.

This means that the union of the two complements will not accept any words, therefore FA_1 and FA_2 are equivalent.

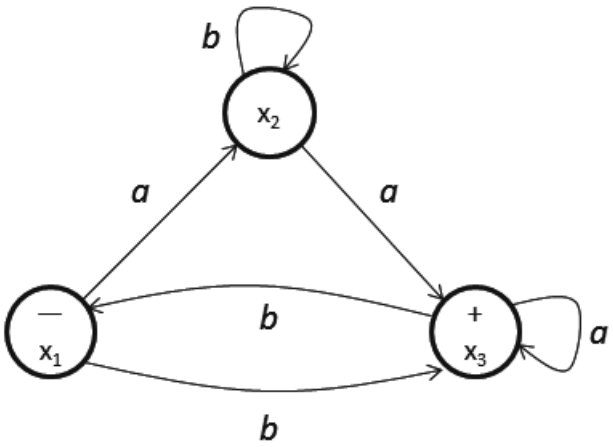
Recommended problems 11.2

Do the following problems to consolidate your knowledge of the work in this learning unit:
6 and 7 on page 218.

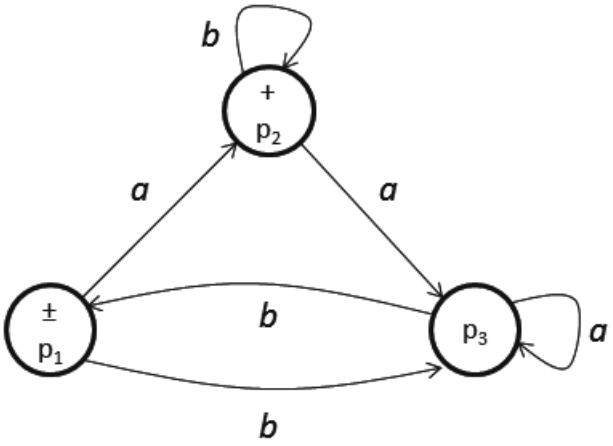
Recommended problems 11.2 – solutions

Problem 6

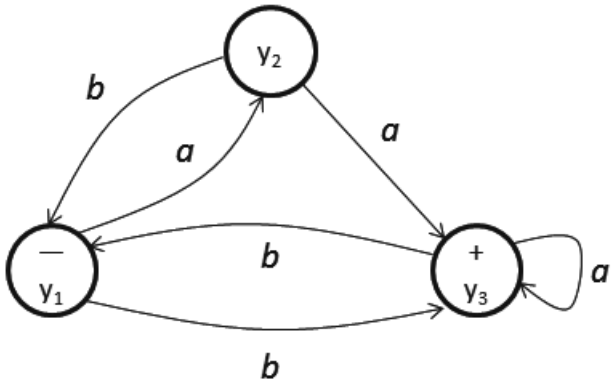
FA_1:



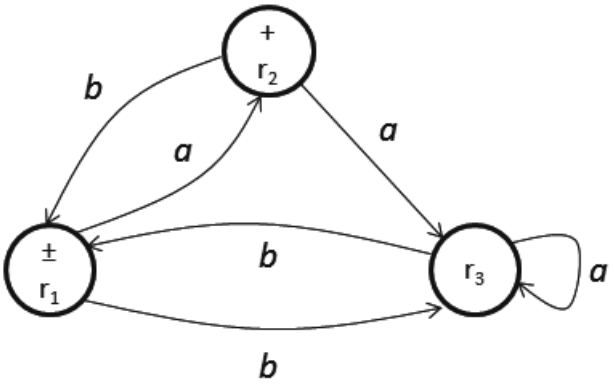
FA_1':



FA_2:



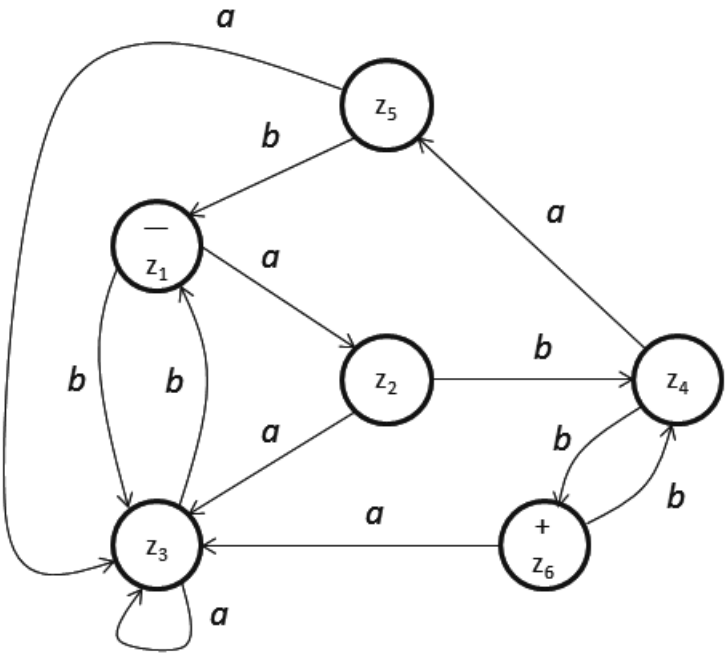
FA_2':



$(FA_1 + FA_2')$:

New state	Read an *a*	Read an *b*
$\pm z_1 = x_1$ or $+r_1$	$+z_2$	$+z_3$
$+z_2 = x_2$ or $+r_2$	$+z_3$	$+z_4$
$+z_3 = +x_3$ or r_3	$+z_3$	$+z_1$
$+z_4 = x_2$ or $+r_1$	$+z_5$	z_6
$+z_5 = +x_3$ or $+r_2$	$+z_3$	$+z_1$
$z_6 = x_2$ or r_3	$+z_3$	$+z_4$

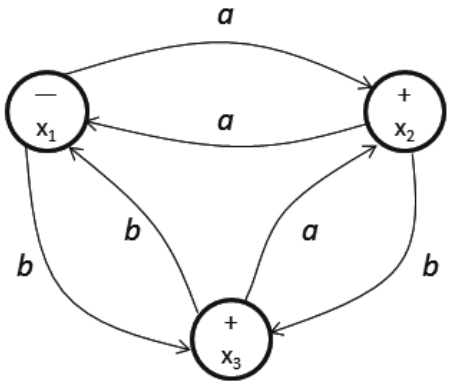
and the complement $(FA_1 + FA_2')'$ looks as follows:



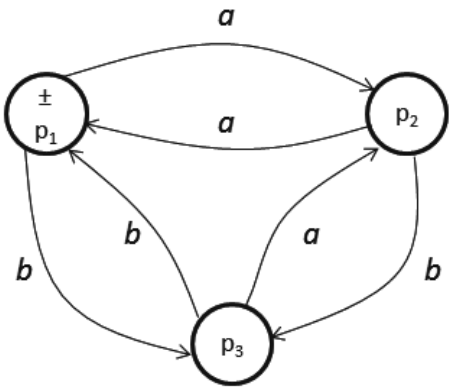
We see that we have a (reachable) final state namely z_6. There is therefore at least one word that is accepted (for example *abb*). This means that the union of this FA with $(FA_1' + FA_2)'$ will accept a word or words, therefore FA_1 and FA_2 are *not* equivalent and do *not* accept the same language. We need not even bother to build the machine $(FA_1' + FA_2)'$.

Problem 7

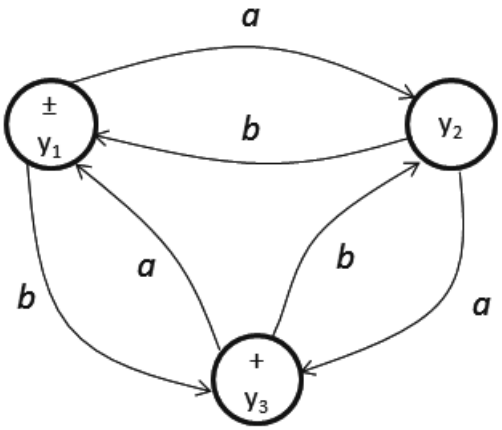
FA_1:



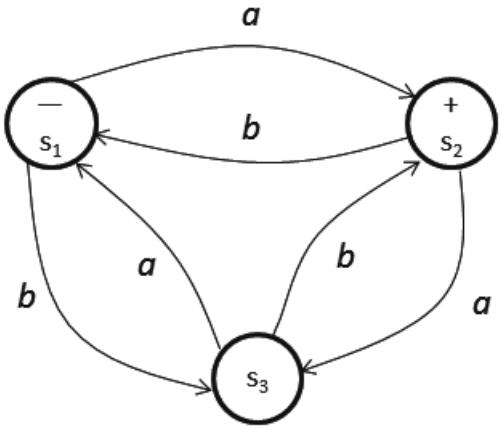
FA_1':



FA_2:



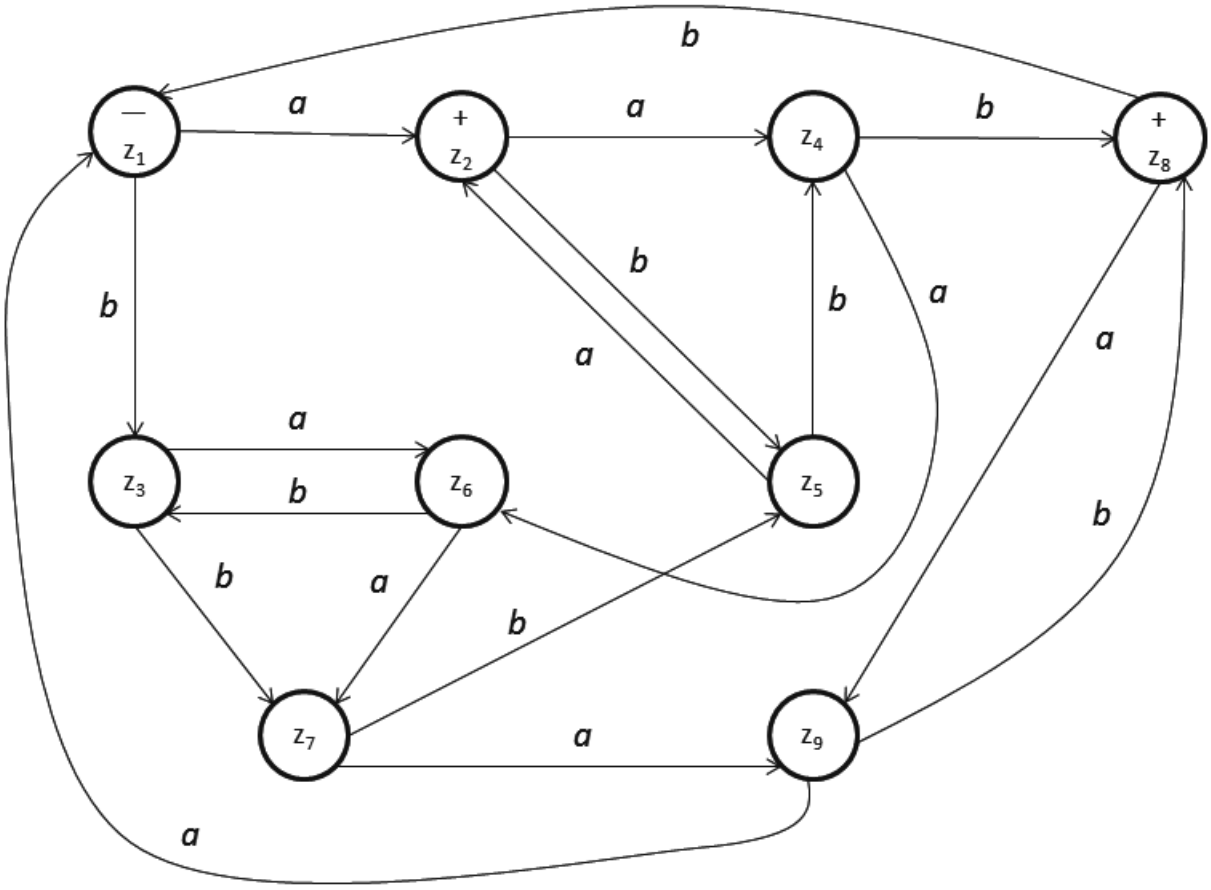
FA_2':



Let us first look at (FA_1' + FA_2):

New state	Read an *a*	Read an *b*
$\pm z_1 = +p_1$ or $+y_1$	z_2	$+z_3$
$z_2 = p_2$ or y_2	$+z_4$	$+z_5$
$+z_3 = p_3$ or $+y_3$	$+z_6$	$+z_7$
$+z_4 = +p_1$ or $+y_3$	$+z_6$	z_8
$+z_5 = p_3$ or $+y_1$	z_2	$+z_4$
$+z_6 = p_2$ or $+y_1$	$+z_7$	$+z_3$
$+z_7 = +p_1$ or y_2	$+z_9$	$+z_5$
$z_8 = p_3$ or y_2	$+z_9$	$\pm z_1$
$+z_9 = p_2$ or $+y_3$	$\pm z_1$	z_8

The complement $(FA_1' + FA_2)'$ looks as follows:



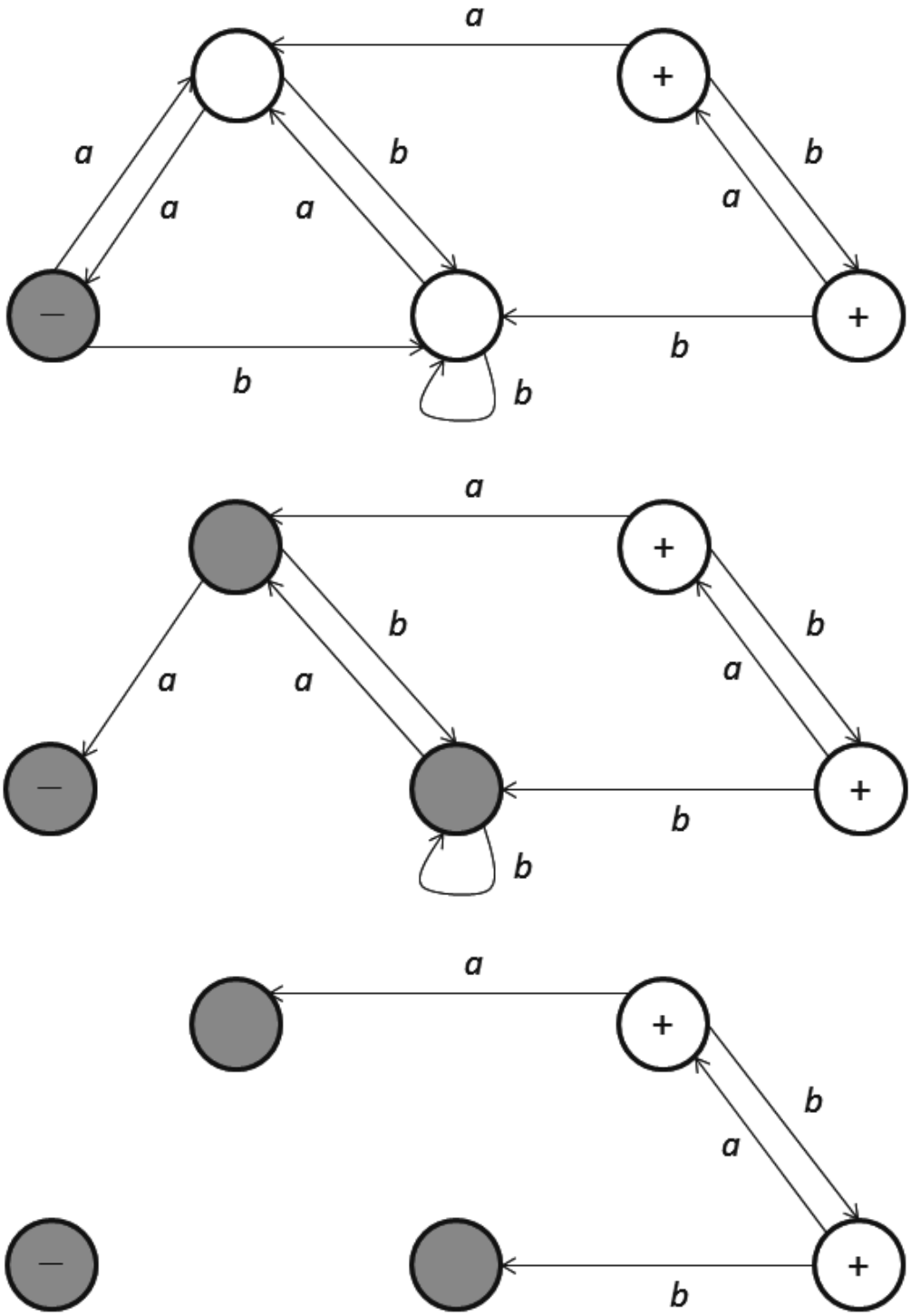
We have two final states, z_2 and z_8, that can be reached. This means that there are words that are accepted by this FA, for example *a* and *aab*. The union of this FA with $(FA_1 + FA_2')'$ will therefore also accept words and that means that FA_1 and FA_2 are not equivalent and therefore do not accept the same language.

Recommended problems 11.3

Do the following problems to consolidate your knowledge of the work in this learning unit:
12, 13(i), 13(ii), and 13(iv) on pages 219 and 220.

Recommended problems 11.3 – solutions

Problem 12



We are unable to paint any further state and the process stops. No final state has been painted blue. This FA does not accept any words.

Problem 13(i)

The FA has 5 states. We must test

$2^5 + 2^6 + 2^7 + 2^8 + 2^9$

words. By inspection, however, it is obvious that only the two words *b* and *ab* are accepted and nothing else. (Any other input string takes us to one of the two lower states and we cannot escape from them.) We conclude that a finite language is accepted.

Problem 13(ii)

3 states, so

$2^3 + 2^4 + 2^5 = 8 + 16 + 32 = 56$

words must be tested. But it is easy to see that only the word *a* is accepted - so we have a finite language.

Problem 13(iv)

4 states. We have to test all the words of length 4 and 5 and 6 and 7 until a word is accepted - there are

$2^4 + 2^5 + 2^6 + 2^7$ such words! Fortunately we find that the word *aaaa*

is accepted and we can say that the language is infinite (length(*aaaa*) = 4 and $4 \leq 4 < 8$).

Recommended problems 11.4

Do the following problems to consolidate your knowledge of the work in this learning unit:
15 and 17 on page 221.

Recommended problems 11.4 – solutions

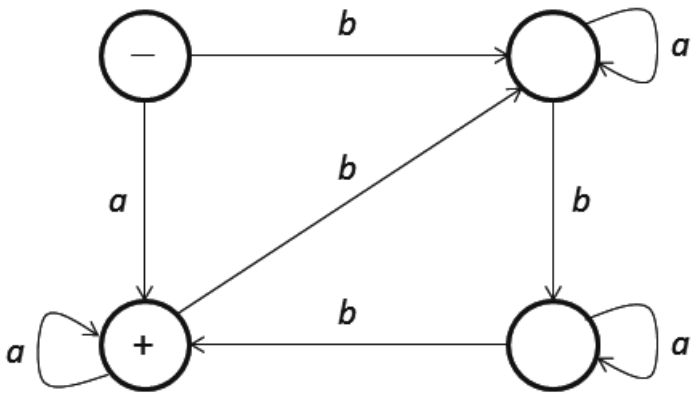
Problem 15

Step 1: Apply the blue paint algorithm for FAs. If no final state can be reached, stop and declare that the language is empty. If not, go to step 2.

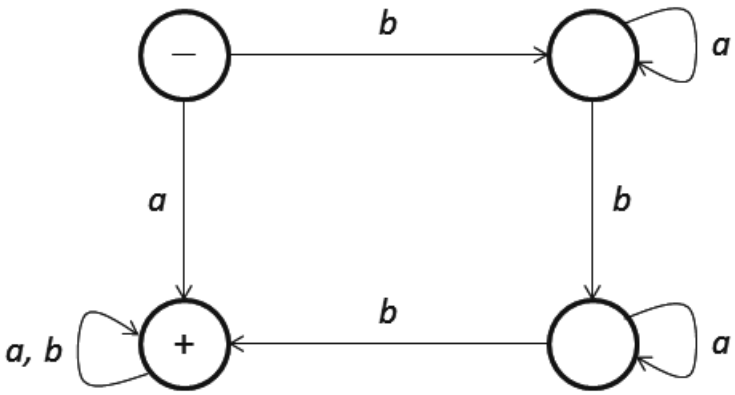
Step 2: Test all possible strings of length N to 2N-1 where N is the number of states. If any such string is accepted, we know that the language is infinite and if no such string is accepted, we know that the language is finite. *It is important to follow all possible paths when testing a string.*

Problem 17

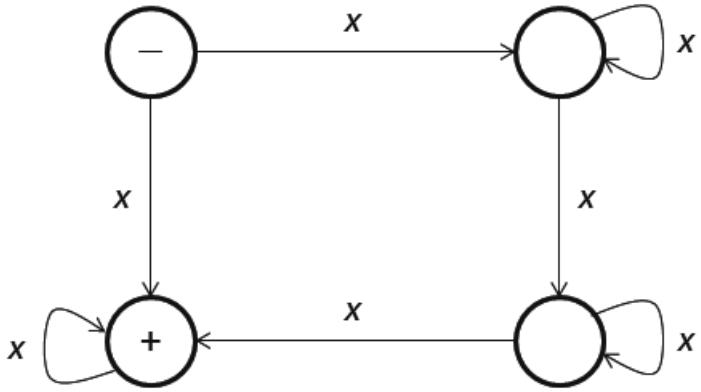
First let us illustrate it. Take the FA of problem 13(iv):



After step 1 we have:



After step 2 we have:



Step 3: It is obvious that the word $x^N = x^4 = xxxx$ is accepted. We may say that the FA accepts a non-empty language.

Does this always work? Yes:

Say the language is not empty. Consider the string x^N and assume that the NFA does not accept this string. This means that there is *no way* to get to a final state *in N or fewer steps.* But according to Theorem 17 an FA must accept at least one word with N or fewer letters if it accepts a non-empty language. The argument is identical for an NFA. Thus, if the language is not empty, x^N is accepted.

Assume x^N is accepted by the NFA. This means that we are able to get to a final state of the original FA (in N or fewer steps) with the correct replacement of all the edges with *x*'s. If we would have left this final state in the FA, we simply chop off the last part of the string.

Step 1 means that strings which are such that the FA gets to a final state and then moves out of this final state are now accepted. But the first part of the string is, of course, a permissible word - so the language does not become empty or non-empty because of step 1. Step 2 means that some combination of alphabet letters moves the FA to the final state under consideration and also does not change the empty or non-empty status. Step 3 is simply Theorem 17.

The procedure is effective because it is finite and works in all cases.

19 Self-test C

Self-test C scope

What is covered?
This self-test covers chapters 9 – 11 in Cohen and learning units 9 – 11.

Self-test submission
Please note that you should not submit this self-test to Unisa for marking. These assignments are intended for you to test yourself. Please contact your e-tutor or one of the lecturers if you need help.

Time allocated
You will need one week to complete this self-test together with Assignment 3.

Due date
Check Tutorial Letter 101 for the due date for this self-test.

Self-test C Questions

Question C.1
Problem 8 on page 185 in the 1997 edition.

Question C.2
Define the language L as follows:

$L = \{a^{n!}b^{n!}\} = \{ab\ aabb\ aaaaaabbbbbb\ldots\}$

Prove that L is nonregular by using the pumping lemma with length.

Question C.3
Use the pumping lemma with length to prove that the following language is nonregular:

$L = \{ab^{n+1}a^n$, with $n \in \{0, 1, 2, 3, ...\}\}$.

Question C.4
Problem 4 on page 217 in the 1997 edition.

Question C.5
Problem 13(iii) on page 220 in the 1997 edition.

Solution to Self-test C

The solutions will be presented in the following pages, one solution per page.

Question C.1
Problem 8 on page 185 in the 1997 edition.

Answer
(**Note**: We recommend that you follow the first shortcut method by applying the algorithm presented in the GOOD PROOF of Theorem 12 if you are asked a similar question in the examinations.)

Let L_1 be the language defined by the regular expression:

$r_1 = (\mathbf{b} + \mathbf{ab})^*(\mathbf{a} + \Lambda)$

and let L_2 be the language defined by the regular expression:

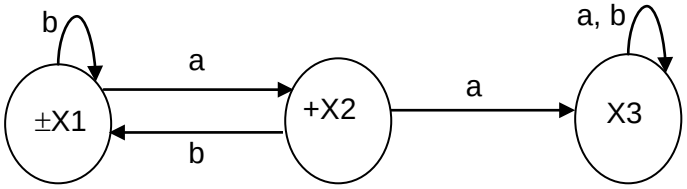
$r_2 = (\mathbf{b} + \mathbf{ab}^*\mathbf{a})^*\mathbf{ab}^*$

By evaluating r_1 we come to the conclusion that **L_1 is the language consisting of all words in which the letter *a* is never doubled** (i.e. the substring *aa* never occurs). The first factor (**b** + **ab**)* allows a

word to start with either an *a* or a *b*, and any generated string ends on a *b*. Note, the *aa*-substring cannot occur. The second factor (**a** + **Λ**) determines that a word can either end on an *a* or a Λ (in the case of Λ, the word will end on the last letter from the first factor which will always be a *b*). Thus words in L_1 will never contain the *aa*-substring and can end on either an *a* or a *b*. Also note that $\Lambda \in L_1$ because Λ can be generated by the first and second factors. Thus our FA should accept Λ.

A possible FA that will accept L_1 is given below:

FA_1:

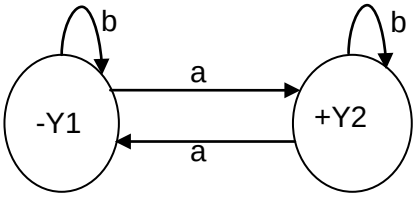


(Note, state x_1 allows Λ to be accepted.)

By evaluating r_2 = (**b** + **ab***a**)***ab*** we come to the conclusion that **L_2 is the language consisting of all words with an odd number of *a*'s**. The first factor (**b** + **ab***a**)* defines words that start with either an *a* or a *b*, and only an even number of *a*'s can be generated. The second factor **ab*** adds an *a* to the string, ensuring that an odd number of *a*'s appears in each generated word (a word can end on either an *a* or a *b*). From the first factor, a string of *b*'s (or Λ) can be generated, followed by a single *a* and then a string of *b*'s (or Λ) from the second factor. The smallest word is *a*.

A possible FA that will accept L_2 is given below:

FA_2:



There exist two alternative methods for obtaining $L_1 \cap L_2$.
Solution 1: We want to find $L_1 \cap L_2$:

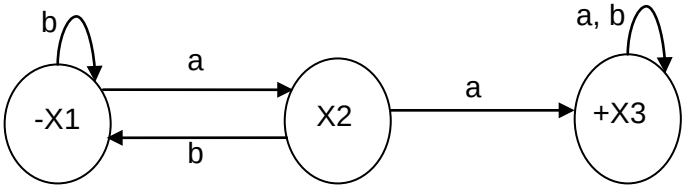
New name	Old states	Read an *a*	Read a *b*
$-z_1$	$\pm x_1$ or $-y_1$	$+z_2$	$-z_1$
$+z_2$	$+x_2$ or $+y_2$	z_3	$+z_4$
z_3	x_3 or $-y_1$	z_5	z_3
$+z_4$	$\pm x_1$ or $+y_2$	z_6	$+z_4$
z_5	x_3 or $+y_2$	z_3	z_5
z_6	$+x_2$ or $-y_1$	z_5	$-z_1$

Note: A z-state is a final state whenever both the x- and y-states are final states.

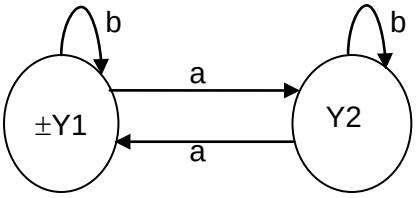
The FA for $L_1 \cap L_2$ is provided after solution 2.

Solution 2: We want to find $L_1 \cap L_2 = (L_1' + L_2')'$:

FA_1' for L_1':



FA_2' for L_2':



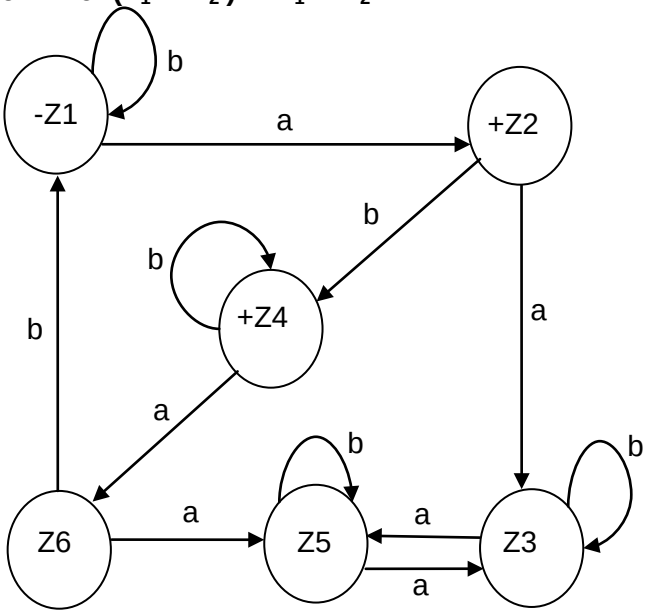
$L_1' + L_2'$:

New name	Old states	Read an *a*	Read a *b*
$\pm z_1$	$-x_1$ or $\pm y_1$	z_2	$\pm z_1$
z_2	x_2 or y_2	$+z_3$	z_4
$+z_3$	$+x_3$ or $\pm y_1$	$+z_5$	$+z_3$
z_4	$-x_1$ or y_2	$+z_6$	z_4
$+z_5$	$+x_3$ or y_2	$+z_3$	$+z_5$
$+z_6$	x_2 or $\pm y_1$	$+z_5$	$\pm z_1$

A z-state is a final state whenever one of the x- or y-states or both are final states.

Note: For $L_1 \cap L_2 = (L_1' + L_2')'$ the z-states become $-z_1$, $+z_2$, z_3, $+z_4$, z_5 and z_6.

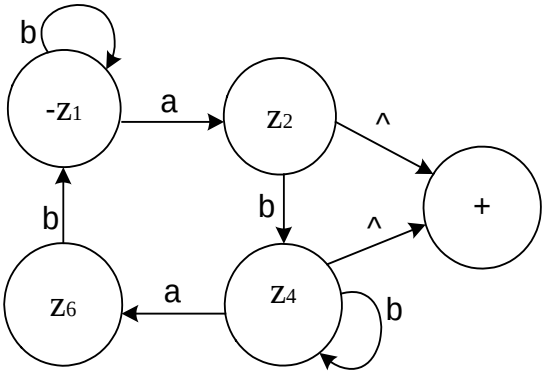
The FA for **$(L_1' + L_2')' = L_1 \cap L_2$:**



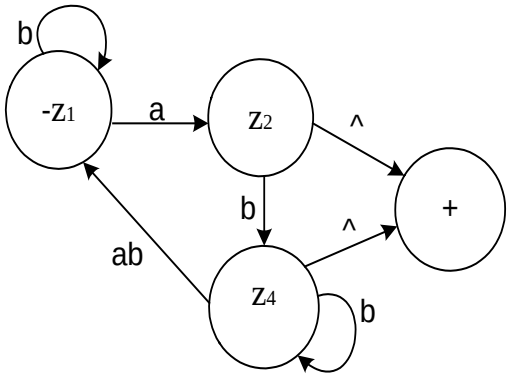
This FA accepts all words where the *aa*-substring does not occur and that have an odd number of *a*'s. We must find a regular expression for this language. (We will use the algorithm in Kleene's theorem provided on page 106 in Cohen.)
By examining the above FA, we note that whenever the states z_5 or z_3 are entered into, no words will be accepted. Therefore we can omit these two states from the machine (TG) from which we will deduce the regular expression.

In the process of obtaining a regular expression for the language, we should eliminate the states one by one. However, we must first create a unique final state and make sure that we have a unique start state.

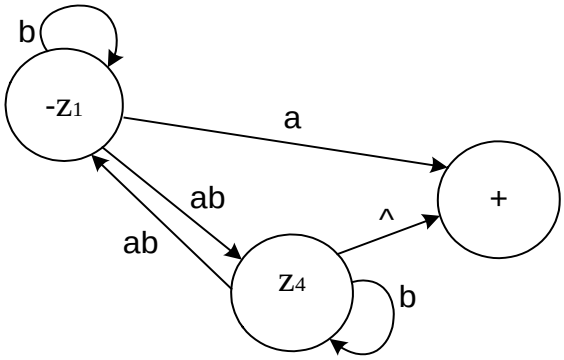
STEP 1: A unique final state is created.



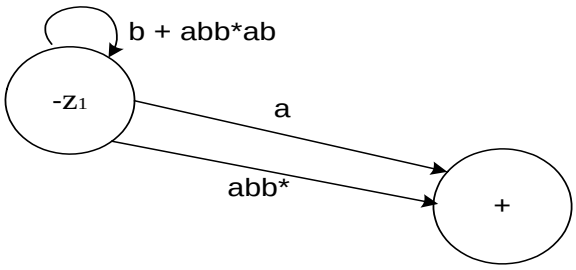
STEP 2: Eliminate state z_6.



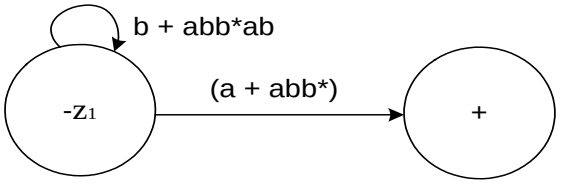
STEP 3: Eliminate state z_2. (Each incoming edge to z_2 must be connected to each outgoing edge.)



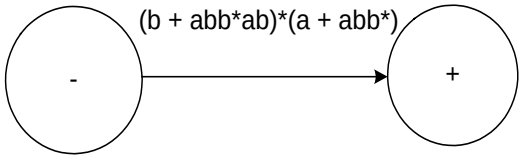
STEP 4: Eliminate state z_4. (There is a loop between z_1 and z_4. Each incoming edge to z_4 must be connected to each outgoing edge.)



STEP 5: Reduce the two outgoing edges from z_1 to the final state to one edge.



STEP 6: Remove the loop at z_1.



Thus the regular expression for the language that consists of all words with an odd number of single *a*'s is:

(**b** + **abb***ab**)*(**a** + **abb***).

Question C.2

Define the language L as follows:

L = {$a^{n!}b^{n!}$} = {*ab aabb aaaaaabbbbbb*…}

Prove that L is nonregular by using the pumping lemma **with** length.

Answer

Let L = {$a^{n!}b^{n!}$} = {*ab*, *aabb*, *aaaaaabbbbbb*, …}.

(Remember, n! = 1·2·3·4·5·····(n -1) · n.)

*A proof using the pumping lemma **with** length:*

Assume that L is a regular language, i.e. there exists an FA with, say *k* states that accepts L.

According to the definition of L, the word $w = a^{k!}b^{k!}$ is a word in L.

Clearly *w* has more than *k* letters, therefore, according to the pumping lemma **with** length, strings *x*, *y* and *z* exist where *w* = *xyz* and

length(*y*) > 0 and length(*x*) + length(*y*) ≤ *k*,

such that $w = xy^2z \in$ L.

The restriction on the lengths of the *x* and *y* substrings implies that the string *y* can only contain *a*'s from the first group of *k! a*'s. This means that the word xy^2z will contain more than *k! a*'s at the start of the word and still only *k! b*'s at the end of the word. Thus the word xy^2z is not a member of the language L. However, according to the pumping lemma with length, it must be a member of L.

This constitutes a contradiction. Therefore, our initial assumption that L is regular is incorrect. Thus, we retract it and conclude that L is non-regular.

Note: We strongly recommend that you use the pumping lemma **with** length in the examinations because the structure of the pumping lemma **with** length is more rigid than the structure of the pumping lemma without length and is easier to apply.

Question C.3

Use the pumping lemma **with** length to prove that the following language is nonregular:

L = {$ab^{n+1}a^n$, with n $\in$ {0, 1, 2, 3, ...} }.

Answer

Assume L = {$ab^{n+1}a^n$, where $n \in$ {0, 1, 2, 3, ...} } is regular.
Then there exists an FA with, say k states, that accepts L.

Let $w = ab^{k+1}a^k$ be a word in L.

According to the pumping lemma, w may be written as

$w = xyz$ such that
length(x) + length(y) ≤ k AND length(y) > 0

We consider the different possible choices for y:

i) y is the *a* in the beginning of w;
ii) y contains the *ab*-substring; or
iii) y consists of only *b*'s.

According to the pumping lemma, *xyyz* must also be an element of L. This is, however, not the case. We investigate the three cases:

i) If y is the *a* in the beginning of the word $w = xyz$, then the pumped word *xyyz* will have more than one *a*, followed by $k + 1$ *b*'s followed by only k *a*'s. This, however, is a contradiction – such a word does not belong to L.

ii) If y contains the *ab*-substring, then *xyyz* will contain more than one *ab*-substring, but this is a contradiction since L does not contain words with more than one *ab*-substring.

iii) If y consists of *b*'s then *xyyz* will contain more than $k + 1$ *b*'s and only k *a*'s. This is again a contradiction – such a word does not belong to L.

In all possible cases we found a contradiction. Our assumption that L is regular is incorrect and we conclude that L is not regular.

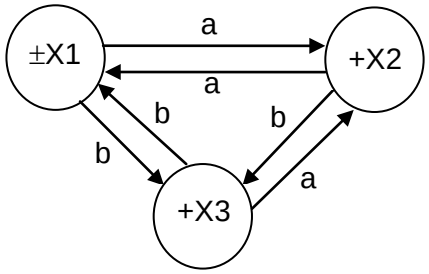
Question C.4

Problem 4 on page 217 in the 1997 edition.

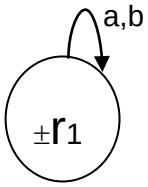
Answer

We have to decide whether or not FA_1 and FA_2 are equivalent. We provide the given FA's with numbered states:

FA_1:

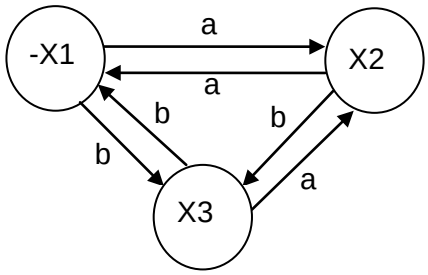


FA_2:

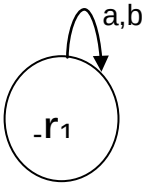


It is clear that these machines both accept the language (a + b)*. However, we have to follow the algorithm provided on pages 212-213 in Cohen.

FA_1':



FA_2'

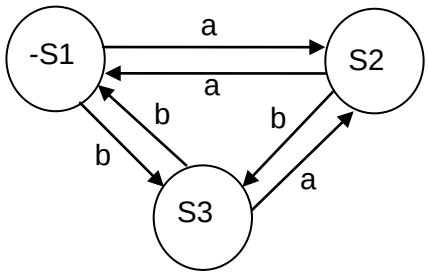


Assume FA_1 accepts L_1 and FA_2 accepts L_2. Now, in order to determine whether FA_1 and FA_2 are equivalent, we must determine whether $(L_1 \cap L_2') + (L_2 \cap L_1') = (L_1' + L_2)' + (L_2' + L_1)'$ accepts any words.

For $L_1' + L_2$ (use the states of FA_1' and FA_2):

New name	Old states	Read an *a*	Read a *b*
$\pm s_1$	$-x_1$ or $\pm r_1$	$+s_2$	$+s_3$
$+s_2$	x_2 or $\pm r_1$	$\pm s_1$	$+s_3$
$+s_3$	x_3 or $\pm r_1$	$+s_2$	$\pm s_1$

To obtain $(L_1' + L_2)'$ we have to change final into non-final and non-final into final states. So we have no final states in this machine.



For $L_2' + L_1$ (use the states of FA_2' and FA_1):

New name	Old states	Read an *a*	Read a *b*
$\pm s_1$	$\pm x_1$ or $-r_1$	$+s_2$	$+s_3$
$+s_2$	$+x_2$ or $-r_1$	$\pm s_1$	$+s_3$
$+s_3$	$+x_3$ or $-r_1$	$+s_2$	$\pm s_1$

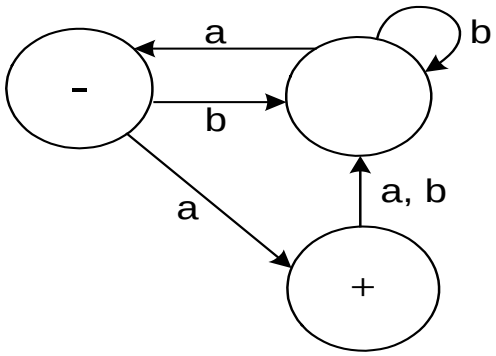
Since all the states are final states, the FA for $(L_2' + L_1)'$ has no final states and is identical to the FA for $(L_1' + L_2)'$. The FA's for $(L_1' + L_2)'$ and for $(L_2' + L_1)'$ accept no words.

Therefore the FA for $(L_1' + L_2)' + (L_2' + L_1)'$ will also not accept any words. Thus, the two given FA's are equivalent.

(Refer to the second last paragraph on page 213 in Cohen.)

Question C.5
Problem 13(iii) on page 220 in the 1997 edition.

Answer
The given FA:



The first part of Theorem 19, page 215 in Cohen states:

Let F be an FA with N states. If F accepts an input string *w* such that
$N \leq \text{length}(w) < 2N$
then F accepts an infinite language.

Therefore, to determine whether the given FA accepts an infinite language, we have to determine whether the machine accepts any word *w* of length 3, 4 or 5 (because N = 3, 2N = 6 and
$N \leq \text{length}(w) < 2N$).

We determine whether any words of length 3 can be accepted:

aaa: The FA does not accept the word *aaa*.
aab: The FA does not accept the word *aab*.
aba: The FA does not accept the word *aba*.
baa: The FA does accept the word *baa*.

We do not need to investigate other 3-character words. Because the FA accepts a word of length 3, we may conclude that the given FA accepts an infinite language.

20 Assignment 3

Assignment 3 scope

What is covered?
This assignment is a multiple-choice question (MCQ) assignment, and covers chapters 7 – 11 in Cohen.

Assignment submission
This assignment should be submitted either electronically via myUnisa (the preferred route), or by filling in a mark-reading sheet and submitting it via one of the regional centres.

Time allocated
You will need one week to complete this assignment together with self-test C.

Due date
Check Tutorial Letter 101 for the due date and unique assignment number for this assignment.

Assignment 3 questions

You can find the assignment question in tutorial letter 101.

Assignment 3 solutions

After the closing date, a discussion of the assignment will be posted to the Additional Resources page. You will be informed of this via an announcement on myUnisa.

21 Example Exam Paper

PDF version
You can get a PDF version of this example exam paper in Additional Resources.

Vodcasts
You can find videos of some of these being solved in the Videos folder in Additional Resources.

Example examination paper

QUESTION 1: Languages **[10]**

(a) Let S = {*a*, *bb*, *bab*, *abaab*}. For each of the following strings, state whether or not it is a word in S*:

(i) *abbabaabab*
(ii) *abaabbabbbaabb* (2)

(i)

(ii)

(b) Give an example of a set S such that S* only contains <u>all possible</u> strings of combinations of *a*'s and *b*'s that have length divisible by three. (4)

(c) Give an example of two sets S and T of strings such that
S* = T* but S $\not\subset$ T and T $\not\subset$ S. (4)

QUESTION 2: Regular expressions **[10]**

(a) Does the regular expression

[**b***+ (**abb***)* + (**aabb***)*]* **bbb** [**b*** + (**abb***)* + (**aabb***)*]*

define the language of *all* words in which the substring *bbb* appears at least once, but the substring *aaa* does not appear at all? If not, give a counter-example. (5)

(b) Give a regular expression generating the language consisting of all words containing exactly one occurrence of the substring *aa* and no occurrence of the substring *bb*. (5)

QUESTION 3: Recursive definitions **[10]**

A recursive definition for the language ODDAB should be compiled. Consider the alphabet $\sum = \{a, b\}$ and the language ODDAB, where ODDAB consists of all words of **odd** length that **contain the substring** *ab*. Provide

(i) an appropriate universal set, (1)
(ii) the generator(s) of ODDAB, (2)
(iii) an appropriate function on the universal set, and then (1)
(iv) use these concepts to write down a recursive definition for the language ODDAB. (6)

(i)

(ii)

(iii)

(iv)

QUESTION 4: Mathematical induction **[10]**

(i) Give a recursive definition of the set P of all positive integers greater than or equal to 5, (1)
(ii) formulate an appropriate induction principle, and (2)
(iii) then apply the induction principle to prove that
$2n - 3 \leq 2^{n-2}$ for all integers $n \geq 5$. (7)

(i) P is the

(ii) If a subset A of P is such that

(iii) Define $A \subseteq P$ as follows:

Show for n =

Assume

Required to prove

LHS
=

Thus

QUESTION 5: Finite automata **[10]**

Build an FA (finite automaton) that accepts the language of all words that satisfies both of the following conditions:

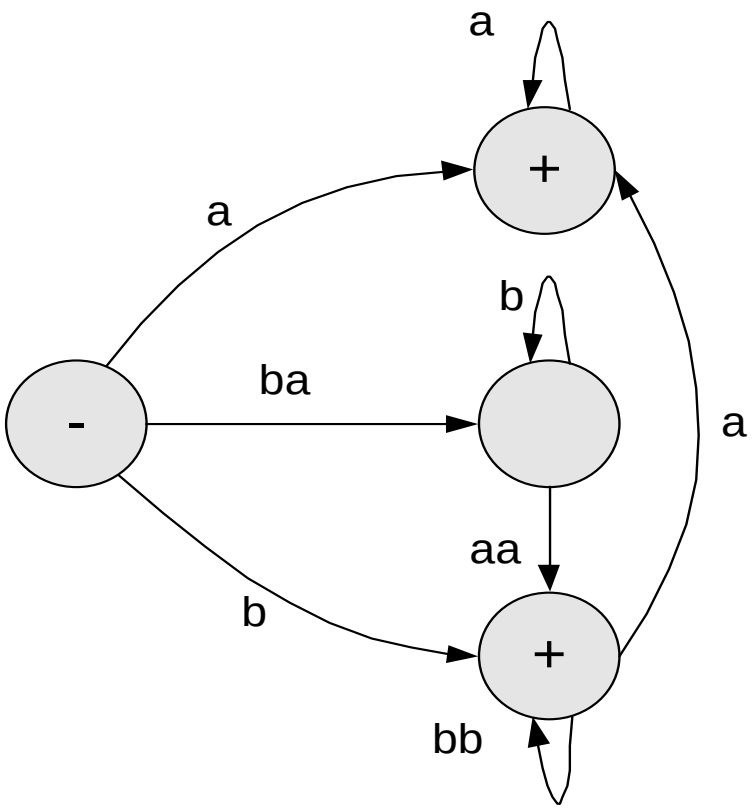
- NO word contains the substring *bba*, and
- ALL words end with a double letter, thus all words end with either *aa* or *bb*.

Note: Only one FA must be build.

NOTE: If you provide a nondeterministic finite automata (NFA) or transition graph (TG) for (c), the maximum mark you may be awarded is 2. Ensure that you build a deterministic FA.

QUESTION 6: Kleene's theorem (TG to RE) **[10]**

By using Kleene's theorem, find a regular expression that generates the language accepted by the following TG (transition graph):



Step 1 - Create a unique start state and a unique final state:

Step 2 - Eliminate state 3:

Step 3 - Eliminate state 4:

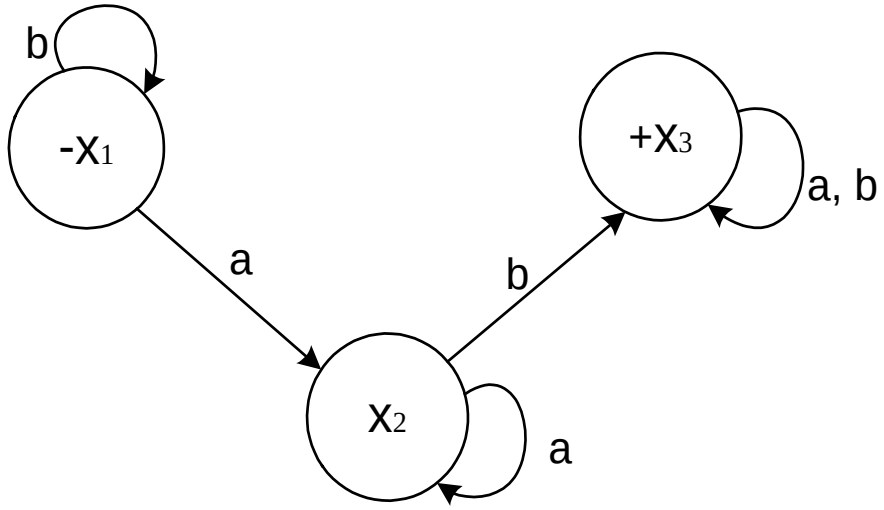
Step 4 - Eliminate state 2:

Step 5 – Eliminate state 1:

A possible regular expression is:

QUESTION 7: Kleene's theorem (RE to FA) **[10]**

Consider the following FA with the corresponding regular expression r_1:



Build an FA for the regular expression r_1* by applying Kleene's theorem. (Do not formulate any regular expression.)

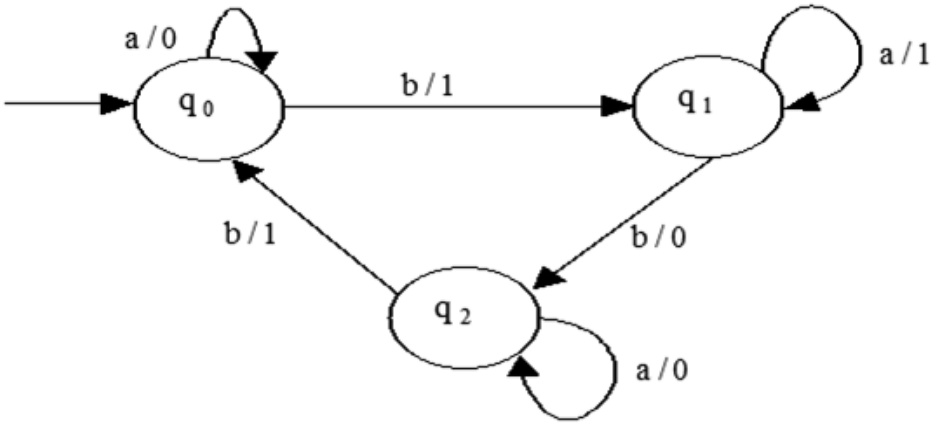
Use the table below to find your solution. Some z states have been provided for you, and they are not an indication of how many you will need. Remember to indicate start and final state(s).

New state	*a*	*b*	
z_1 =			
z_2 =			
z_3 =			

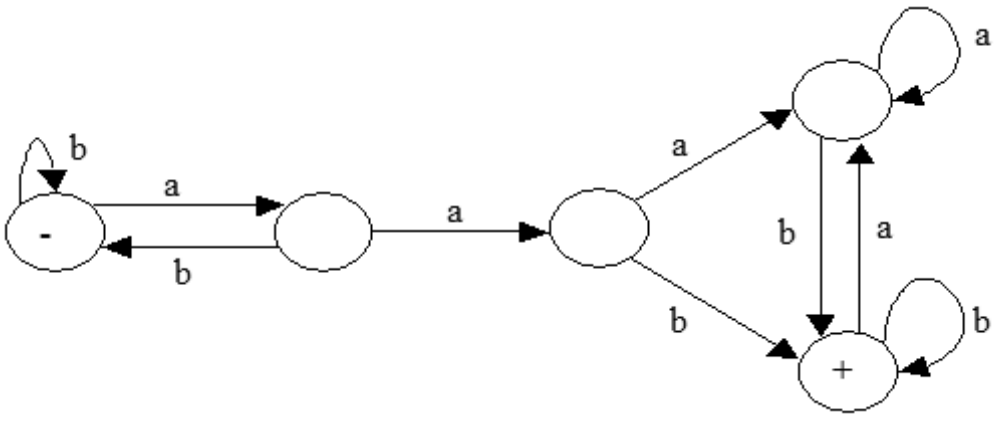
Now draw the new FA for **r**$_1$*

QUESTION 8: Regular language acceptors **[10]**

(a) Convert the following Mealy machine to a Moore machine: (5)



(b) Consider the following FA:



Find an NFA (non-deterministic FA) with four states that accepts the same language. (5)

QUESTION 9: Pumping lemma with length **[10]**

Use the Pumping Lemma **with** length to prove that the following language is nonregular:

L = {$a^n b^n a^m$, where $n \in \{0, 1, 2, 3, ...\}$ and $m \in \{0, 1, 2, 3, ...\}$}.

Use the prompts below to complete the proof.

Assume

Then there exists

We choose any word w =

Thus w may be written as

Then according to the pumping lemma with length,

There is/are __ possible choice(s) for y:

If y is pumped in each of the above case(s)

We conclude that

And,

We conclude that L is not regular.

QUESTION 10: Decidability **[10]**

Assume we have an alphabet $\sum = \{a, b\}$. Complete the following table:

	FA	TG	NFA
Number of start states			
Number of final states			
Permissible edge labels			
Number of lines leaving each state			
Deterministic or not			

ADDITIONAL QUESTION 10: DECIDABILITY **[10]**

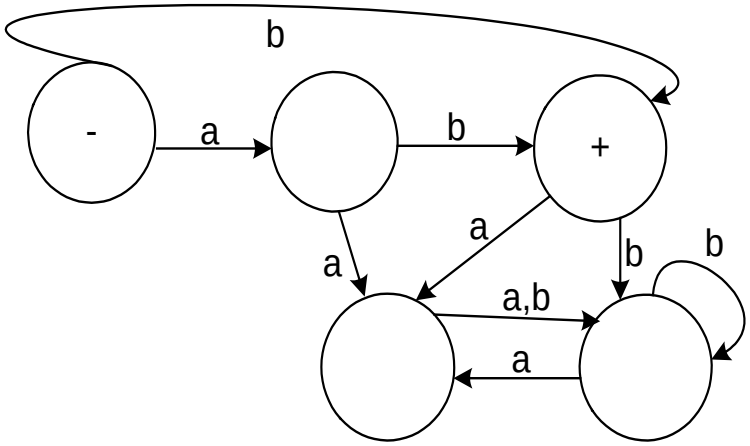
(a) Define a decision procedure.

(b) Use the definition in (a) to define decidability.

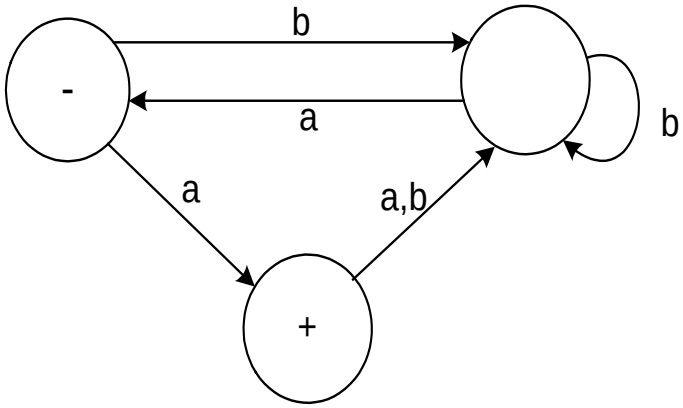
(c) Describe an effective procedure to decide whether a given FA accepts a finite or infinite language.

(d) Using the decision procedure described in (c) above, determine for each FA_1 and FA_2 below whether it accepts a finite or an infinite language.

FA_1



FA_2



In the case of FA_1:

In the case of FA_2:

Example examination paper solutions

QUESTION 1: Languages **[10]**

(a) Let S = {*a*, *bb*, *bab*, *abaab*}. For each of the following strings, state whether or not it is a word in S*:

(i) *abbabaabab*
(ii) *abaabbabbbaabb* (2)

(i) No, because there are no concatenations of *a, bb, bab* and *abaab* that will yield the word *abbabaabab*.

(ii) Yes, concatenations of *a, bb, bab* and *abaab* will yield the word: (*abaab)(bab)(bb)(aa)(bb)*.

(b) Give an example of a set S such that S* only contains <u>all possible</u> strings of combinations of *a*'s and *b*'s that have length divisible by three. (4)

There is only one possible set S: S = {*aaa, aab, aba, abb, baa, bab, bba, bbb*}. All the words in S* are multiples of three as all the words of S are of length exactly 3.

(c) Give an example of two sets S and T of strings such that S* = T* but S $\not\subset$ T and T $\not\subset$ S. (4)

We give two possible answers, there are many other possibilities:
S = {Λ, *a*} and T = {*a*, *aa*} or S = {*a*, *b*, *ab*} and T = {*a*, *b*, *ba*}.

QUESTION 2: Regular expressions **[10]**

(a) Does the regular expression

[**b***+ (**abb***)* + (**aabb***)*]* **bbb** [**b*** + (**abb***)* + (**aabb***)*]*

define the language of *all* words in which the substring *bbb* appears at least once, but the substring *aaa* does not appear at all? If not, give a counter-example. (5)

It is indeed the case that *all* words generated by the given regular expression contain the substring *bbb*. It is furthermore the case that *no* word generated by the given regular expression contains the substring *aaa*. However, not *all* such words (i.e. words containing the substring *bbb* but not the substring *aaa*) can be generated by the given regular expression. Some examples are:

babbb
aabbb
bbba.

The answer to this question is thus **no**, and the words above serve as counterexamples.

(b) Give a regular expression generating the language consisting of all words containing exactly one occurrence of the substring *aa* and no occurrence of the substring *bb*. (5)

The regular expression that we need to give, must make provision for words that start (and end) with either an *a* or a *b*. All occurrences of the letter *b* (except if it is the last letter of the word) must be followed by an *a*. Furthermore, only one *a* may be followed by another *a* - in order to ensure only one occurrence of the substring aa. A possible regular expression is:

(**b** + Λ)(**ab**)***aa**(**ba**)*(**b** + Λ)

This is of course not the only possibility.

QUESTION 3: Recursive definitions **[10]**

A recursive definition for the language ODDAB should be compiled. Consider the alphabet $\sum = \{a, b\}$ and the language ODDAB, where ODDAB consists of all words of **odd** length that **contain the substring** *ab*. Provide

(i) an appropriate universal set, (1)
(ii) the generator(s) of ODDAB, (2)
(iii) an appropriate function on the universal set, and then (1)
(iv) use these concepts to write down a recursive definition for the language ODDAB. (6)

(i) The set $\{a, b\}^*$ will be suitable because it contains, along with other words, all the words that are in the language ODDAB.

(ii) The generators should be the smallest words in ODDAB of odd length and should contain the substring *ab*. Thus,
aba, *aab*, *abb*, *bab*
will be suitable generators.

(iii) The function CONCAT as defined in the study guide will be suitable.

(iv) We give two possible recursive definitions. Note that all words in ODDAB should have an odd number of letters. The generators all have an odd number of letters; therefore, to keep the length of new words odd, we concatenate two letters at a time. Note, the generator(s) is/are always the smallest word(s) in a language.

ODDAB is the smallest subset of $\{a, b\}^*$ such that
$aba, aab, abb, bab \in$ ODDAB, and if $w \in$ ODDAB,
then also CONCAT(w,*aa*), CONCAT(w,*bb*), CONCAT(w,*ab*), CONCAT(w,*ba*),
CONCAT(*aa*,w), CONCAT(*bb*,w), CONCAT(*ab*,w), CONCAT(*ba*,w) $\in$ ODDAB.

Or an alternative definition:

Rule 1: $aba, aab, abb, bab \in$ ODDAB.

Rule 2: If $w \in$ ODDAB, then also CONCAT(w,*aa*), CONCAT(w,*bb*), CONCAT(w,*ab*), CONCAT(w,*ba*), CONCAT(*aa*,w), CONCAT(*bb*,w), CONCAT(*ab*,w), CONCAT(*ba*,w) $\in$ ODDAB.

Rule 3: Only words produced by rules 1 and 2 are in ODDAB.

QUESTION 4: Mathematical induction **[10]**

(i) Give a recursive definition of the set P of all positive integers greater than or equal to 5, (1)
(ii) formulate an appropriate induction principle, and (2)
(iii) then apply the induction principle to prove that
$2n - 3 \leq 2^{n-2}$ for all integers $n \geq 5$. (7)

(i) P is the smallest subset of Z (the set of integers) such that $5 \in$ P and if $n \in$ P, then also $n + 1 \in$ P.

(ii) If a subset A of P is such that $5 \in$ A and if $n \in$ A, then also $n + 1 \in$ A, then A = P.

(iii) Define $A \subseteq P$ as follows:
$A = \{n \in P \mid 2n - 3 \leq 2^{n-2}\}$

Show for n = 5
$2(5) - 3 \leq 2^{5-2}$
i.e. $7 \leq 8$
Thus, $5 \in A$ is true.

Assume $k \in A$, i.e. we assume that $2k - 3 \leq 2^{k-2}$

Required to prove that $k + 1 \in A$, i.e. $2(k + 1) - 3 \leq 2^{(k+1)-2}$.

LHS
$= 2(k + 1) - 3$
$= 2k + 2 - 3$
$= (2k - 3) + 2$
$\leq 2^{k-2} + 2$ (from induction assumption (1))
$\leq 2 \cdot 2^{k-2}$ (because the smallest value for 2^{k-2} with k = 5 is $2^{5-2} = 8$ and $2 \cdot 8 = 16 > 10 = 2^{5-2} + 2$)
$= 2^{1+k-2}$
$= 2^{(k+1)-2}$ = RHS

Thus $k + 1 \in A$
Hence, A = P so we conclude that $2n - 3 \leq 2^{n-2}$ for all integers $n \geq 5$.

QUESTION 5: Finite automata **[10]**

Build an FA (finite automaton) that accepts the language of all words that satisfies both of the following conditions:

- NO word contains the substring *bba*, and
- ALL words end with a double letter, thus all words end with either *aa* or *bb*.

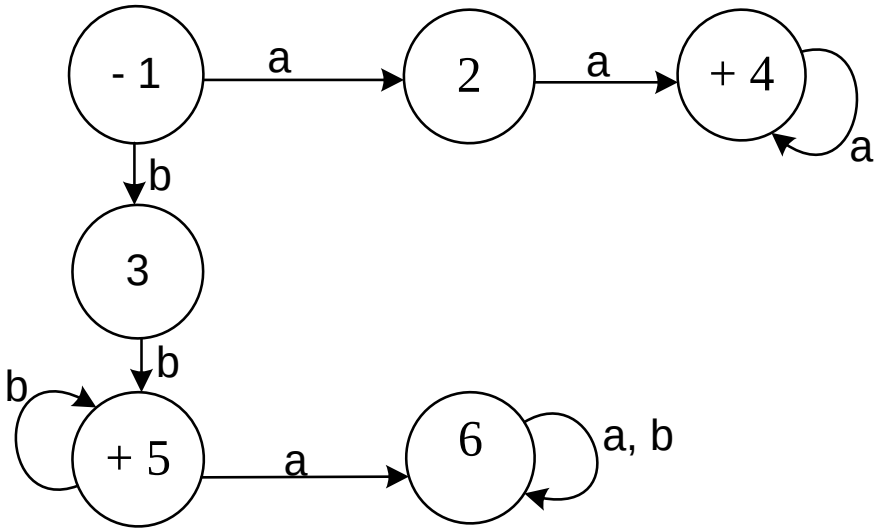
Note: Only one FA must be build.

NOTE: If you provide a nondeterministic finite automata (NFA) or transition graph (TG) for (c), the maximum mark you may be awarded is 2. Ensure that you build a deterministic FA.

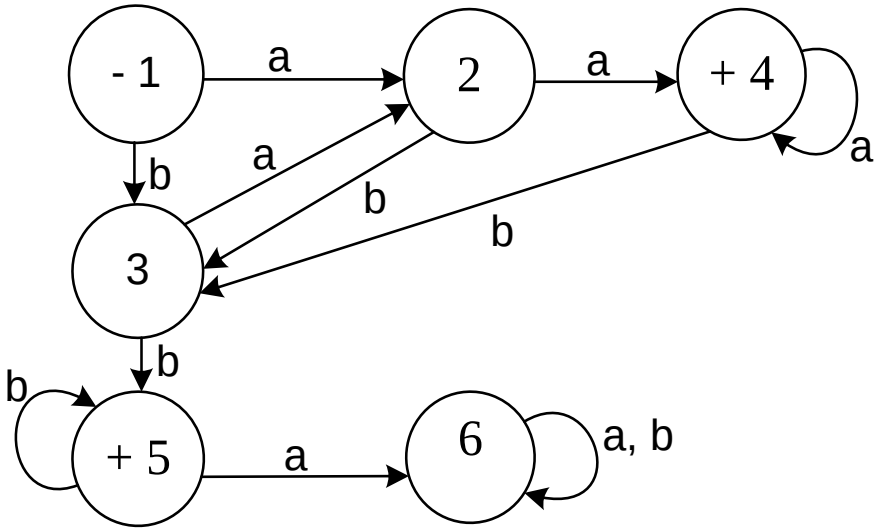
CAI tutorial: *Remember to use the CAI tutorial* Automata *available on CD or downloadable from the web. In preparation to this question, you can navigate through all the sections in the tutorial. Graphical illustrations are also provided. (Please refer to section 1 in this tutorial letter.)*

We do not need to keep track of the number of letters that have been read at any stage - even or odd is irrelevant for this question. A dead-end state is necessary to make provision for all words containing the *bba*-substring. You should further keep in mind that words ending on *aaa* or *bbb* are, of course, also permissible.

From the initial state 1, we move to state 2 with an *a* and to state 3 with a *b*. From state 2, we move to a final state 4 with an *a* and if we read an *a* in the final state, *we stay in the final state*, because words such as *aaa* and *aaaa* are permissible words in the language. From state 3, we move to a final state, state 5, with a *b* and if we read a *b* in the final state, *we stay in the final state*, because words such as *bbb* and *bbbb* are permissible words in the language. If, however, we read an *a* in state 5, we move to a dead-end state, state 6, from which we cannot leave because then the word contains the impermissible substring *bba*. The incomplete FA is given below:

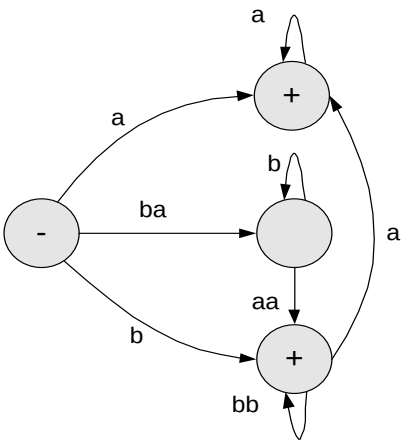


If we read a *b* in state 2, we go to state 3 and if we read an *a* in state 3, we move to state 2. What happens if a *b* is read in state 4? We move to state 3.

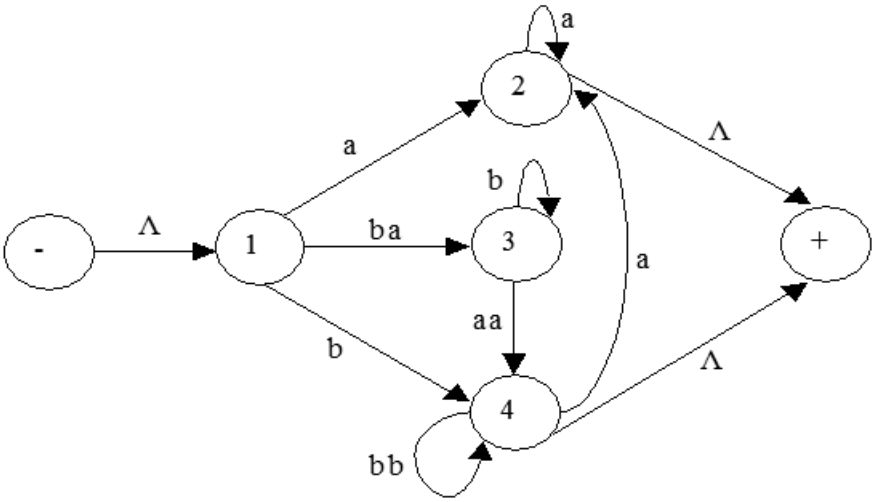


QUESTION 6: Kleene's theorem (TG to RE) **[10]**

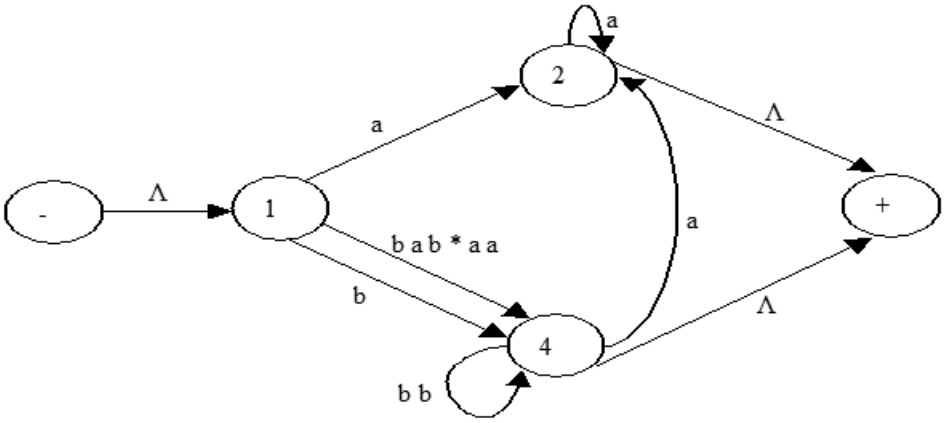
By using Kleene's theorem, find a regular expression that generates the language accepted by the following TG (transition graph):



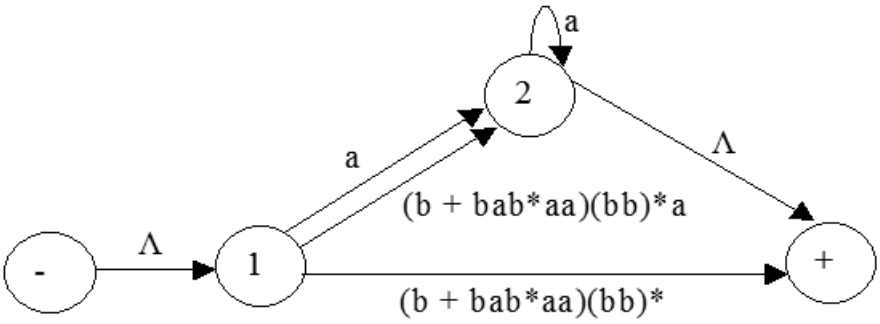
Step 1 - Create a unique start state **(note: this is actually redundant for this question because there is no incoming edges to the start state)** and a unique final state:



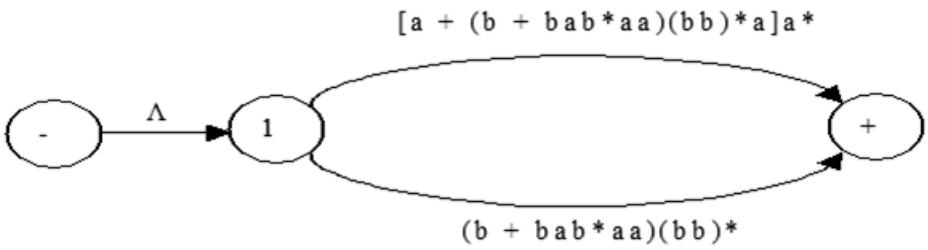
Step 2 - Eliminate state 3:



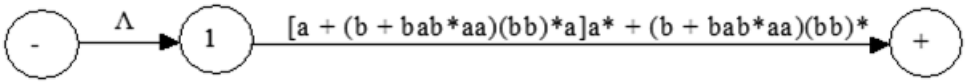
Step 3 - Eliminate state 4:



Step 4 - Eliminate state 2:



Step 5 – Eliminate state 1:

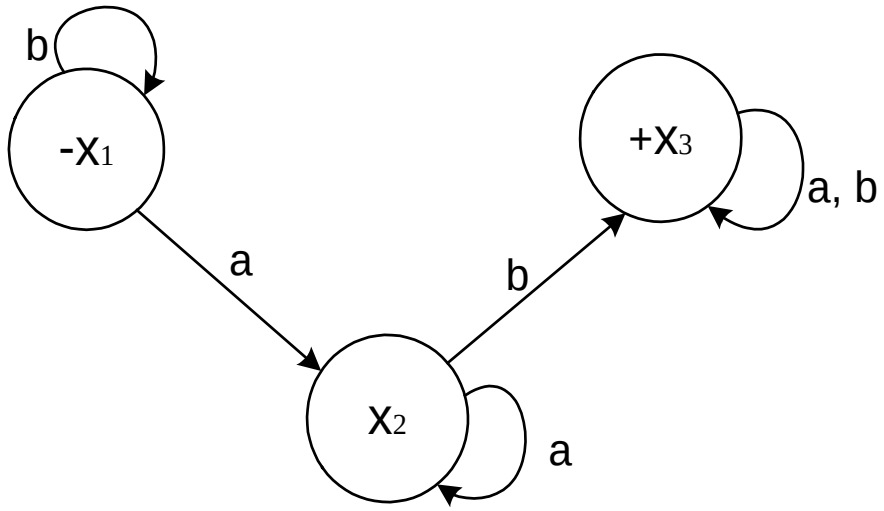


A possible regular expression is:

[**a** + (**b** + **bab***aa**)(**bb**)***a**]**a*** + (**b** + **bab***aa**)(**bb**)*

QUESTION 7: Kleene's theorem (RE to FA) **[10]**

Consider the following FA with the corresponding regular expression r_1:

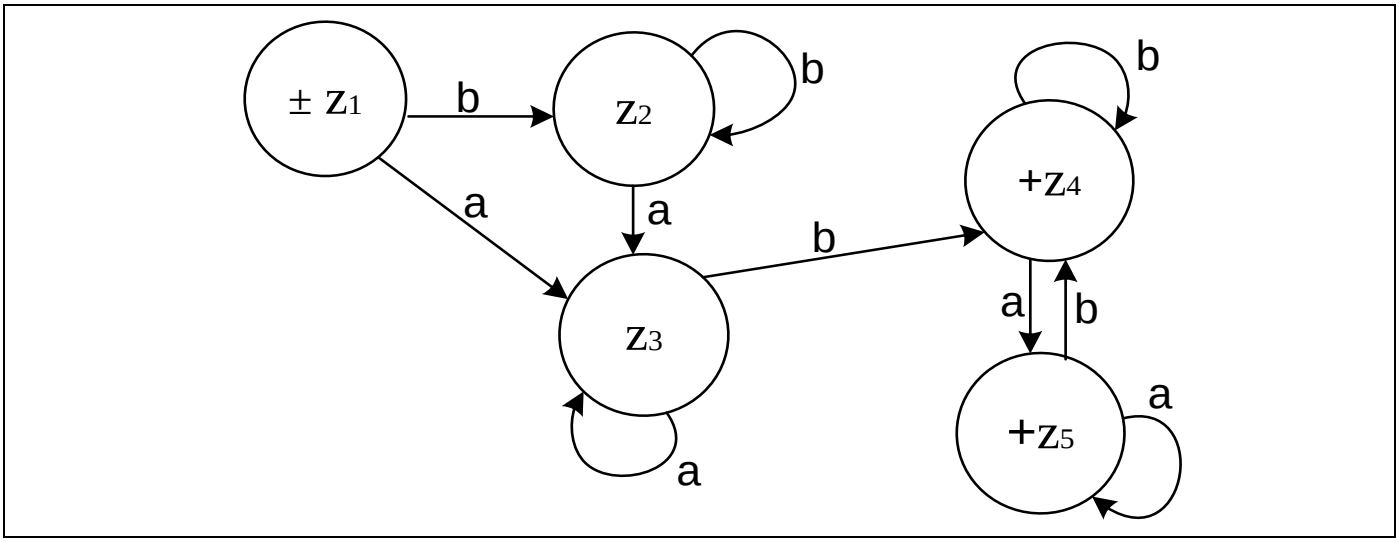


Build an FA for the regular expression r_1* by applying Kleene's theorem. (Do not formulate any regular expression.)

Use the table below to find your solution. Some z states have been provided for you, and they are not an indication of how many you will need. Remember to indicate start and final state(s).

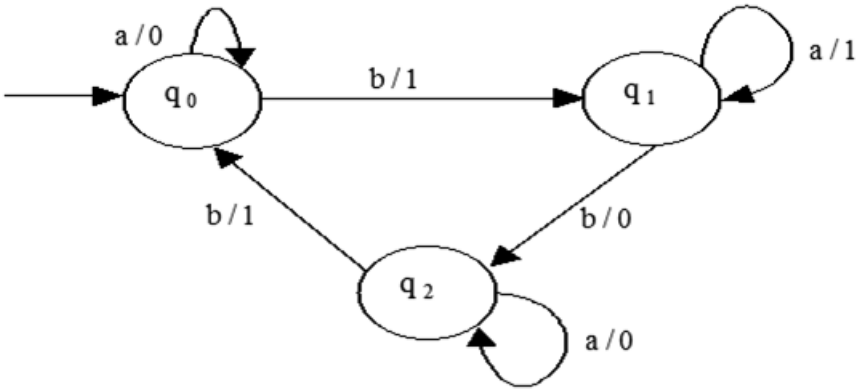
New state	*a*	*b*	
$\pm z_1 = x_1$	$x_2 = z_3$	$x_1 = z_2$	
$z_2 = x_1$	$x_2 = z_3$	$x_1 = z_2$	
$z_3 = x_2$	$x_2 = z_3$	x_1 or $x_3 = z_4$	
$+z_4 = x_1$ or x_3	x_2 or x_3 or $x_1 = z_5$	x_1 or $x_3 = z_4$	
$+z_5 = x_2$ or x_3 or x_1	x_2 or x_3 or $x_1 = z_5$	x_1 or $x_3 = z_4$	

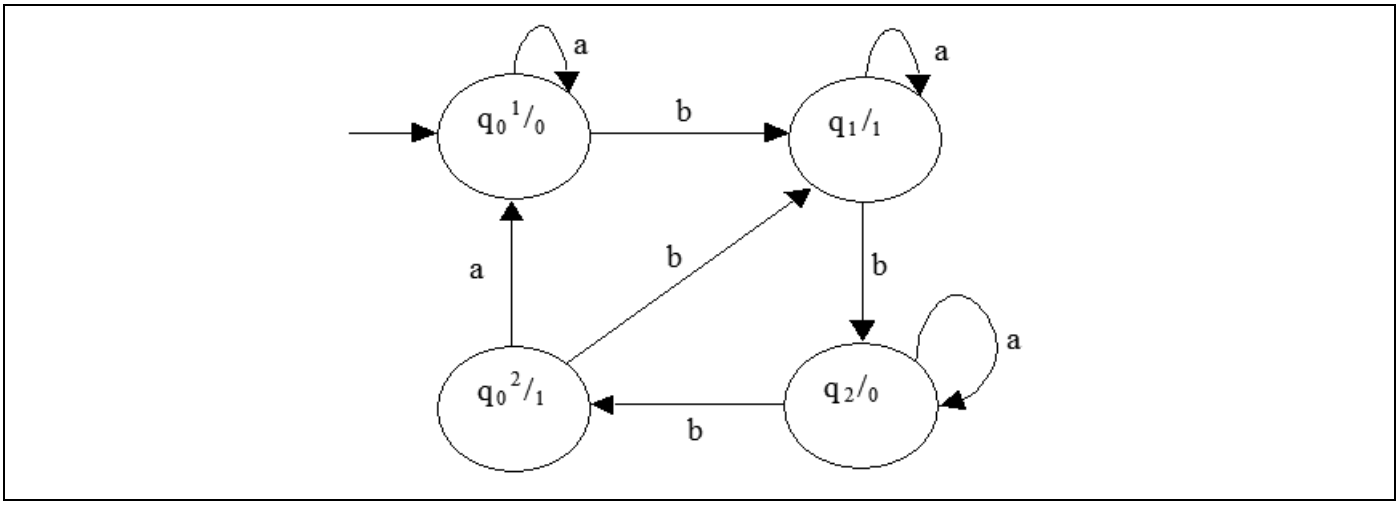
Now draw the new FA for **r**$_1$*



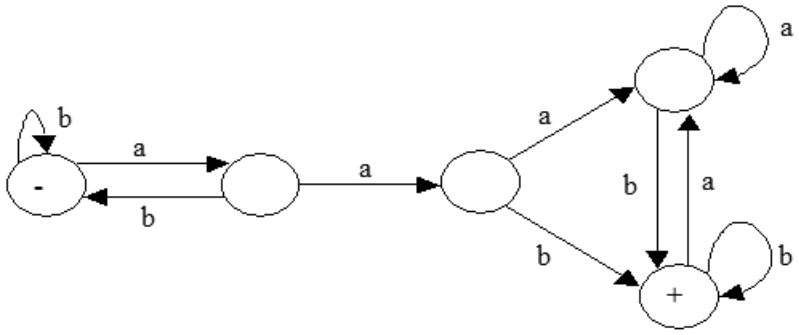
QUESTION 8: Regular language acceptors **[10]**

(a) Convert the following Mealy machine to a Moore machine: (5)

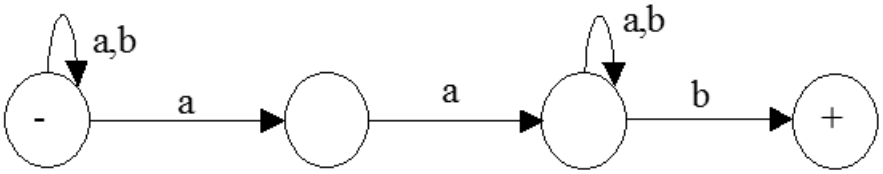




(b) Consider the following FA:



Find an NFA (non-deterministic FA) with four states that accepts the same language. (5)



QUESTION 9: Pumping lemma with length **[10]**

Use the Pumping Lemma **with** length to prove that the following language is nonregular:

L = {$a^n b^n a^m$, where $n \in$ {0, 1, 2, 3, ...} and $m \in$ {0, 1, 2, 3, ...}}.

Use the prompts below to complete the proof.

CAI tutorial: *Remember to use the CAI tutorial* Pumplemma *available on CD or downloadable from the web. In preparation of this question, you can navigate through all the sections in the tutorial. Graphical illustrations are also provided. (Please refer to section 1 in this tutorial letter.)*

Assume L = {$a^n b^n a^m$, where $n \in$ {0, 1, 2, 3, ...} and $m \in$ {0, 1, 2, 3, ...}} is regular.

Then there exists an FA with, say k states, that accepts L.

We choose any word $w = a^k b^k a^p$ be a word in L.

Thus w may be written as
$w = xyz$ such that
length(x) + length(y) ≤ k AND length(y) > 0

Then according to the pumping lemma with length, $xy^n z$ with $n \in$ {1, 2, 3, …} [or xyyz] also belongs to L.

There is/are 1 possible choice(s) for y:
This means that both the strings x and y can only consist of a's.

If y is pumped in each of the above case(s)
According to the pumping lemma, $xyyz$ must also be an element of L. This is, however, not the case because the pumped word, $xyyz$ has more a's (in the first group of a's) than b's.

We conclude that in each case the pumped word is not in L.

And we have found a contradiction. Our assumption that L is regular is incorrect.
We conclude that L is not regular.

QUESTION 10: Decidability **[10]**

Assume we have an alphabet ∑ = {*a*, *b*}. Complete the following table:

	FA	TG	NFA
Number of start states	1	1 or more	1
Number of final states	some or none	some or none	some or none
Permissible edge labels	letters from ∑	letters from ∑*	letters from ∑
Number of lines leaving each state	one edge for each letter in ∑	arbitrary	arbitrary
Deterministic or not	yes	no	no

ADDITIONAL QUESTION 10: DECIDABILITY **[10]**

(a) Define a decision procedure.

An effective solution to a problem with a *Yes* or *No* as answer is called a decision procedure.

(b) Use the definition in (a) to define decidability.

A problem for which a decision procedure can or may be found is called decidable.

(c) Describe an effective procedure to decide whether a given FA accepts a finite or infinite language.

Refer to Theorem 20, page 216 of Cohen:

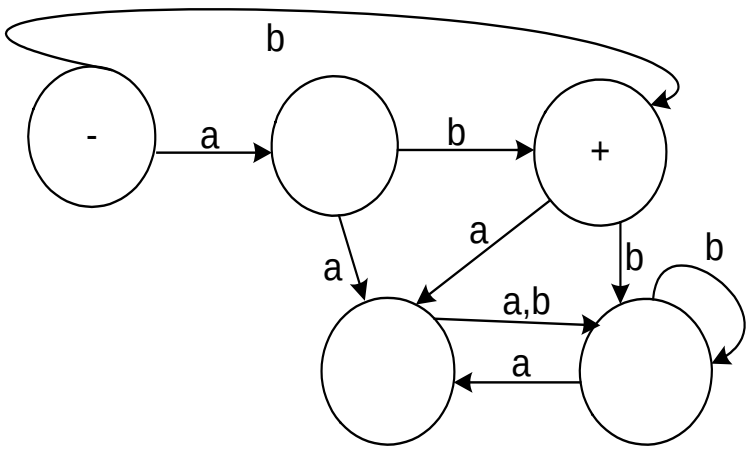
Assume a given FA has N states and there are m letters in the alphabet to be accepted, then there exist $m^N + m^{N+1} + m^{N+2} + \ldots + m^{2N-1}$
different input strings in the range N ≤ length of string < 2N.

All these words can be tested by executing the FA on them. If any of these words are accepted, then the language is infinite. If none of these words is accepted, then the language is finite.

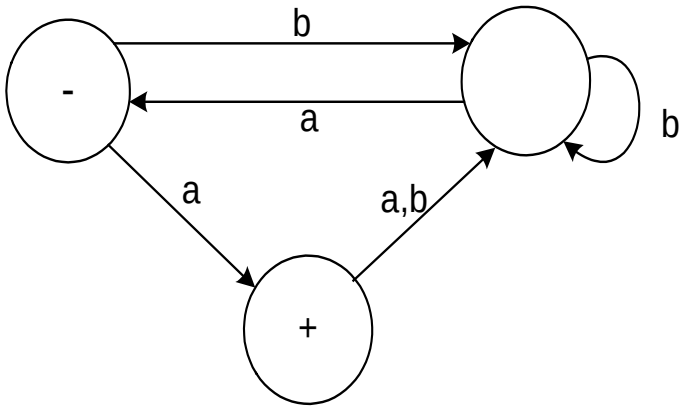
Note: The Blue Paint algorithm cannot be applied here.

(d) Using the decision procedure described in (c) above, determine for each FA_1 and FA_2 below whether it accepts a finite or an infinite language.

FA_1



FA_2



In the case of FA_1:

In this case N = 5. The only words accepted by FA_1 are *b* and *ab*. These words are of length 1 and 2. Words that start with *b* or *ab* and that contain other letters will leave the final state and will not return. Thus these longer words will not be accepted by FA_1.
Thus, words of
length $\geq$ N = 5 AND length < 2N = 2(5) = 10
i.e. 5 $\leq$ length of string < 10.
will not be accepted.
Therefore we can conclude that the language accepted is finite.

In the case of FA_2:

In this case N = 3. Now consider the word *bbaa*:
Length(*bbaa*) = 4 > N = 3 and *bbaa* is accepted by FA_2 where
N = 3 $\leq$ length(*bbaa*) = 4 < 2N = 6.
Hence the language accepted is infinite.

22 A Final Word

We trust that you have enjoyed this semester studying theoretical computer science in the context of automata theory, and that you will take you studies in this field further by enrolling for COS3701.