TOURING WITH TURING

by

Paul J. Kaiser

First Paperback Edition 2021
ISBN: XXXXXXXXXX

Printed in the United States of America

Please visit my personal website at:
http://www.cs.lewisu.edu/ kaiserpa/

Contents

Dedication

To my grandchildren:

Zoe
Aidan
Kamron
McKenna

So much potential.
So much energy.
So much love.

May you enjoy your life journey
to the fullest!

Grandpa

Preface

In the summer of ¡don't remember year!¿ I had the opportunity to visit Bletchley Park where the British code breakers spent World War II seeking to unravel the encryption techniques of the Axis powers.

As a teacher of both mathematics and of computer science I was familiar with the name Alan Turing, but I was only vaguely aware of his reputation in the fields of Computation Theory and Artificial Intelligence.

At Bletchley Park I learned there was so much more to the man and his accomplishments. What did he do before the war? He invented the modern computer. What did he do during the war? He deciphered the German Enigma code. What did he do after the war? He was one of the first to do research in bio-mathematics. Unfortunately, much of his story was kept quiet due to the War Secrets Act in England which extended 25 years from the end of the war. Even worse he was a homosexual who was convicted by the same British statute that was also applied to Oscar Wilde. Oscar Wilde went to prison; Alan Turing suffered chemical castration. He died in 1954 a few weeks shy of his 44th birthday after ingesting cyanide, possibly accidental.

It was also at Bletchley Park that I found the book **The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine**, in which Charles Petzold explains in detail the organization and reasoning required to understand the seminal publication. This book was published by Wiley Publishing, Inc., in 2008; it opened my eyes to an entirely new world in which mathematics and computer science intersect.

My personal math background was solidly grounded in real analysis and culminated in a doctorate in Optimal Control Theory. Along the way I had passing encounters with computer programming (FORTRAN and assembly language) and with some fundamental concepts (binary representation, arithmetic, circuitry). But I was totally unaware of Turing and his concept of a Universal Machine. Even when I returned to graduate school twelve years later to study computer science specifically, I encountered finite state machines and grammars while focusing on programming languages and compiler construction. Even at that time I was unaware

that finite state machines were Turing Machines having specialized names.

It was only after I read the book by Petzold and then read **Introduction to the Theory of Computation, 3rd ed**, by Michael Sipser, that this very intricate and challenging theory came together for me personally.

In many respects, the original definition of a Turing Machine varies significantly from the modern definitions for the various categories of finite automata. Modern machines are expected to stop in finite time; the original was not. Modern machines come in a variety of configurations – deterministic or nondeterministic, single-tape or multi-tape; the original was pretty vanilla by comparison. Over time researchers in mathematics and computer science have refined and expanded these topics into a very deep and conceptually beautiful theory.

Although there are many fine books available in the field of Computation Theory, Formal Languages, and Finite Automata, I believe that many of them fail to give sufficient recognition to the author of the seminal paper in the field and his creative insight. Seeking a solution to the Entſcheidungſproblem, Alan Turing developed the concept of a computing machine to calculate a specific numeric value and then expanded on that concept to envision a Universal Machine capable of simulating any other computing machine.

The following list summarizes the books I turned to familiarize myself with the key concepts. The following texts have all contributed elements I have incorporated into this text.

The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine,
Charles Petzold, Wiley Publishing, Inc., 2008.

Introduction to the Theory of Computation, 3rd ed,
Michael Sipser, Cengage Learning, 2013.

Introducing the Theory of Computation,
Wayne Goddard, Jones and Bartlett Publishers, 2010.

An Introduction to Formal Languages and Automata, 5th ed,
Peter Linz, Jones and Bartlett Publishers, 2011.

Introduction to Automata Theory, Languages, and Computation,

3rd ed,
John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson, 2012.

JFLAP: An Interactive Formal Languages and Automata Package, Susan H. Rodger and Thomas W. Finley, Jones and Bartlett Publishers, 2006.

Computers and Intractability: A guide to the Theory of NP-Completeness, Michael R. Garey and David S. Johnson, W.H. Freeman and Company, 1979.

No disrespect, but most of these books present finite state machines in increasing order of complexity: from **finite automata** to **push down automata** to **Turing Machines**. I completely understand this strategy and will be using it myself in the pages that follow! But I believe it is important both historically and theoretically to begin with the original paper **On Computable Numbers, with an Application to the** Entſcheidungſproblem.

With the help of Charles Petzold I was able to understand both the simplicity and the intricacy of Turing's arguments. Facility with binary representations is sufficient to begin the journey. But like so many important breakthroughs in mathematics, at some point there is unique and expected twist which either results in understanding or causes confusion.

Please feel free to share this book with others – either digitally or as a hard copy. My only request is that if you find this book beneficial to your progress that you make a $50.00 contribution to the Lewis University Computer and Mathematical Sciences (CAMS) Endowment Fund.

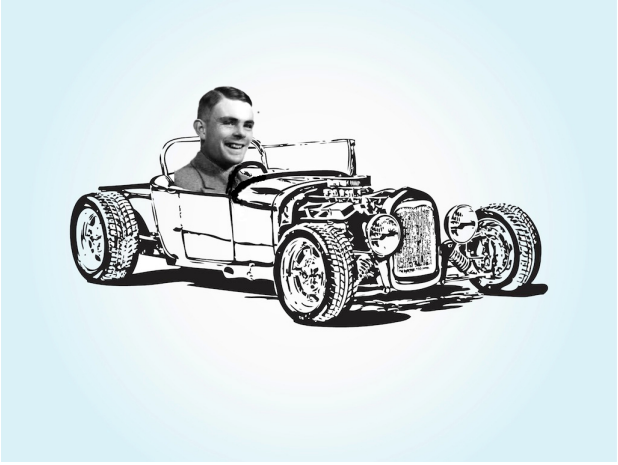
All of the source code files are available for download from:

www.cs.lewisu.edu/∼kaiserpa/

Lewis University
One University Parkway
Romeoville, Illinois 60446
(815) 838-0500

Chapter 1

Entscheidungsproblem



1.1 History

It is always best to start at the beginning: *What was Turing trying to accomplish when he wrote his paper in 1936?* He was attempting to answer a mathematical question called the Entſcheidungſproblem.

Entſcheidungſproblem is a German word meaning ***decision problem***. The form of the Entſcheidungſproblem has evolved over time.

In the 17th century Gottfried Leibniz had constructed a mechanical calculating machine when he asked the question:

> Can a machine manipulating symbols determine the truth values of mathematical statements?

David Hilbert in 1900 posed a set of problems for mathematicians to address at the turn of the twentieth century. His Tenth Problem was the following:

> Consider a diophantine equation, i.e., a polynomial in the variables $x_1, x_2, \ldots, x_n$ having integer coefficients. Devise a process which can determine in a finite number of steps whether or not the equation is solvable using integer values for the variables $x_1, x_2, \ldots, x_n$.

Hilbert and Ackerman developed a much more general form of the question in 1928:

> Is there an algorithm which takes as input statements of the first order logic (possibly augmented with additional axioms) and answers YES or NO according to whether the statement is universally valid, i.e., provable using rules of logic.

Turing approached this question by analyzing how a human might approach solving a particular problem. For our simple discussion here, let us consider the addition of two positive integers, e.g., 147 + 35:

- the solver is given the problem data (some form of input)
- the solver hopes to generate and display an answer (some form of output)
- the solver probably has some scratch paper nearby to assist with intermediate results

- the solver will proceed through a very specific list of relatively small steps

Now suppose the two positive integers are of the size

$$1,000,000,000,000,000^{1,000,000,000,000,000}!$$

- the solver can only focus on a small piece of the overall problem at any given time
- the solver must gradually work through many different repetitious phases to get to the final result

Turing proposed mimicking these attributes using the following building block components in an idealized computing machine:

- an infinitely long tape (infinite in both directions)
- a read-write head capable of reading / writing / erasing the contents of a single cell on the tape
- the read-write head is capable of moving left one cell / moving right one cell / remaining stationary over the current cell
- a finite set of potential input symbols and a finite set of potential output symbols
- a finite set of states to keep track of progress through the overall process and to allow the repetition of key components in the process by returning to a previous state
- the entire operation is controlled by the current state of the machine and the current symbol under the read-write head; if the current state / current symbol is meaningless to the machine, then it simply STOPS

It is important to note at this point that Turing really *did not want his machines to stop*! His focus was on computing numbers between 0 and 1 into their infinite binary expansions (which we will discuss shortly). As long as his machines were continuously computing the next binary digit, he was happy. What Turing did not want was for a machine to stop or for a machine to go off into an infinite loop of meaningless computations. Machines which did the latter he called ***circular***!

A subsequent addition to Turing's original computing machine was the new feature:

- some states may be designated as: a HALT state – either ACCEPT or REJECT

1.2 Formal Definition

The following is a *modern* definition of a Turing Machine.

Definition 1.2.1. A **Turing Machine** M is a 7-tuple (Q, Σ, Γ, δ, q_0, Q_{accept}, Q_{reject}) where Q, Σ, and Γ are all finite sets and

1. Q is a set of **states**
2. Σ is the **input alphabet** (non-blank symbols)
   a blank symbol will be represented with a $\square$
3. Γ is the **tape alphabet**
   $\square \in \Gamma, 0 \in \Gamma, 1 \in \Gamma$, and $\Sigma \subseteq \Gamma$
4. δ : Q x $\Gamma \rightarrow$ Q x Γ x {L,R,S} is a **transition function**
   i.e., δ (q,s) = (q',s',m) where
   q' = next state
   s' = write symbol (possibly a $\square$)
   m = left / right / stationary
5. q_0 is a unique **start state**
6. $Q_{accept} \subseteq$ Q, a set of **accept states** (may be empty $\emptyset$)
7. $Q_{reject} \subseteq$ Q, a set of **reject states** (may be empty $\emptyset$)

Comparison: Turing's Paper vs Modern Theory

TURING'S PAPER

cells divided into two categories:

F(igure) cells
□, 0, 1 only
0 and 1 **not** updatable

E(rasable) cells
any tape symbol

calculation of computable numbers

infinite binary expansions
$1/2 = 0.100000\ldots$
$2/3 = 0.101010\ldots$
⇒ HALTing bad!
⇒ circular bad!

infinite bi-directional tape

Turing used a @@ sentinel
to mark the start of the tape
never worked to the left of @@

MODERN THEORY

all cells identical

read / write any tape symbol
read / write any tape cell

more general computation

machines can HALT
ACCEPT and REJECT states

typically infinite bi-directional tape

also infinite one-directional tape
also multiple tapes

This chapter will initially focus on Turing's original definitions and his restrictions; as we progress, we will gradually transition toward some of the more modern concepts of finite state machines.

We conclude this section with two definitions that Alan Turing would have consider important.

Definition 1.2.2. A Turing Machine is considered **circle-free** provided it prints an infinite number of 0s and/or 1s into consecutive F cells, i.e., it generates an infinite binary expansion.

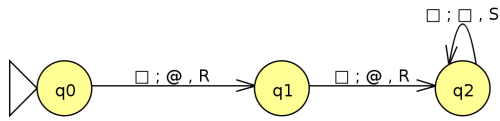
Definition 1.2.3. A Turing Machine is considered **circular** provided it prints only a finite number of 0s and/or 1s before it enters an infinite loop which generates no further F cell entries or it encounters an invalid state and STOPS.

Example

All of the examples found in this textbook will be presented and discussed using JFLAP software. Please refer to the back of the book (**Crash Course: JFLAP**) for information regarding this software.

The following example will be very true to Turing's ideas. The first step is to write the sentinel symbol in two consecutive cells (an F cell and an E cell). At that point the turing machine will simply go into an infinite loop doing nothing!

a circular Turing Machine



The transition function for this machine is obviously very simple:

current	symbol	print	move	next
$\triangleright q_0$	□	@	R	q_1
q_1	□	@	R	q_2
q_2	□	□	S	q_2

1.3 Computable Numbers

For the examples in this section, we will dispense with Turing's sentinel (@@) as the initial step in any computing machine and simply focus on the representation of binary expansions for numeric values However, for the time being I will continue to differentiate between F cells (which will contain 0s and 1s) and E cells (which serve as to hold temporary markings).

Turing focused on the binary expansion for numeric values between 0 and 1. All such numbers may be represented (calculated as an infinite binary expansion using only the two digits 0 and 1).

Just as natural numbers have a finite binary expansions, such as

$$45_{(10)} = 101101_{(2)} = 32 + 8 + 4 + 1$$

positive real numbers between 0 and 1 have an infinite binary expansions, such as

$$3/8_{(10)} = 0.011000\ldots_{(2)} = 1/4 + 1/8$$

or better still, such as

$$1/3_{(10} = 0.010101\ldots_{(2)} = 1/4 + 1/6 + 1/64 + \ldots$$

which is truly an infinite binary expansion!

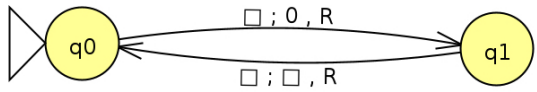
Definition 1.3.1. A **computable number** is any binary expansion which can be generated by a Turing Machine *plus or minus* a natural number.

The following pages contain several basic examples which illustrate the relationship between a computable number and a Turing machine.

Examples

computation of zero

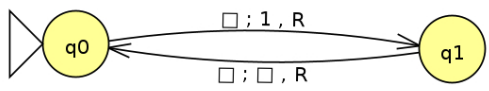
x = 0.000000 ...



current	symbol	print	move	next
$\triangleright q_0$	□	0	R	q_1
q_1	□	□	R	q_0

computation of one

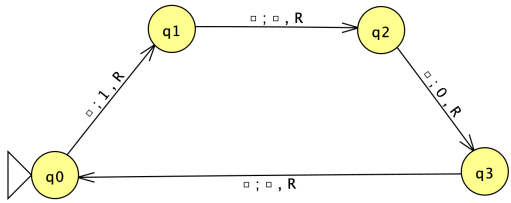
x = 0.111111 ...
x = 1/2 + 1/4 + 1/8 + 1/16 + ...



current	symbol	print	move	next
$\triangleright q_0$	□	1	R	q_1
q_1	□	□	R	q_0

computation of two-thirds

$$\mathbf{x = 0.101010\ldots}$$
$$\mathbf{x = 1/2 + 1/8 + 1/32 + 1/128 + \ldots}$$

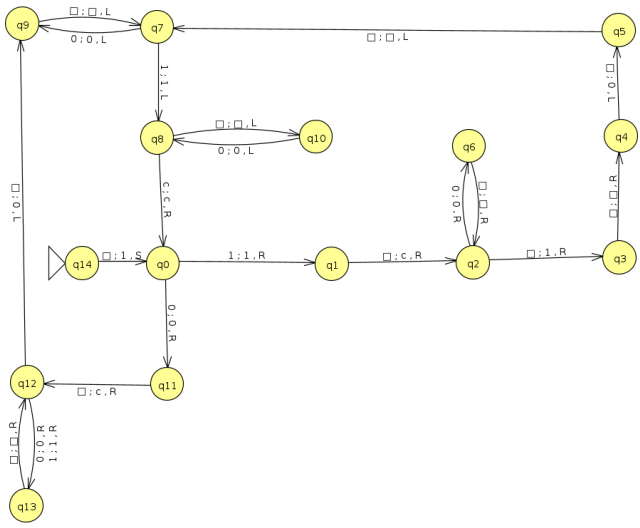


current	symbol	print	move	next
$\triangleright q_0$	□	1	R	q_1
q_1	□	□	R	q_2
q_2	□	0	R	q_3
q_3	□	□	R	q_0

computation of "flight of the conchords"

$$\mathbf{x = 0.1101001000100001000001000000100000001} \ldots$$

the sequence of 0s between consecutive 1s increases in length by one each time!
this is an example of a non-repeating and non-terminating binary expansion
i.e., an irrational number in binary form



My unusual name for this number is derived from the comedy group **Flight of the Conchords** and their song ***Binary Solo***.

Note: This is the first example in which I specifically make use of the E-cells in a calculation to create the correct number of 0s between consecutive 1s.

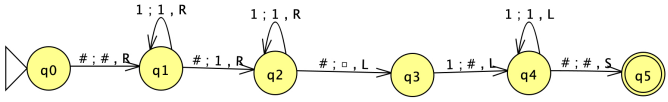
current	symbol	print	move	next
$\triangleright q_{14}$	$\square$	1	S	q_0
q_0	1	1	R	q_1
q_0	0	0	R	q_{11}
q_1	$\square$	c	R	q_2
q_2	$\square$	1	R	q_3
q_2	0	0	R	q_6
q_3	$\square$	$\square$	R	q_4
q_4	$\square$	0	L	q_5
q_5	$\square$	$\square$	L	q_7
q_6	$\square$	$\square$	R	q_2
q_7	1	1	L	q_8
q_7	0	0	L	q_9
q_8	c	c	R	q_0
q_8	$\square$	$\square$	L	q_{10}
q_9	$\square$	$\square$	L	q_7
q_{10}	0	0	L	q_8
q_{11}	$\square$	c	R	q_{12}
q_{12}	0	0	R	q_{13}
q_{12}	1	1	R	q_{13}
q_{12}	$\square$	0	R	q_9
q_{13}	$\square$	$\square$	R	q_{12}

At this point we escape from Turing's very restrictive structure and assume a more modern perspective. We want to be able to create computing machines that perform basic calculations. We present two Turing Machines that perform **simple integer addition**: the first does *unary* addition of positive integers, the second does *binary* addition of positive integers. Both will halt in an ACCEPT state if successful; both will leave the tape with the correct answer as its sole contents.

unary integer addition

$$\#111\#11111\# \;\rightarrow\; \#11111111\#$$
$$3 + 5 = 8$$

unary notation is like tic marks on a sheet of paper – one tic for each unit

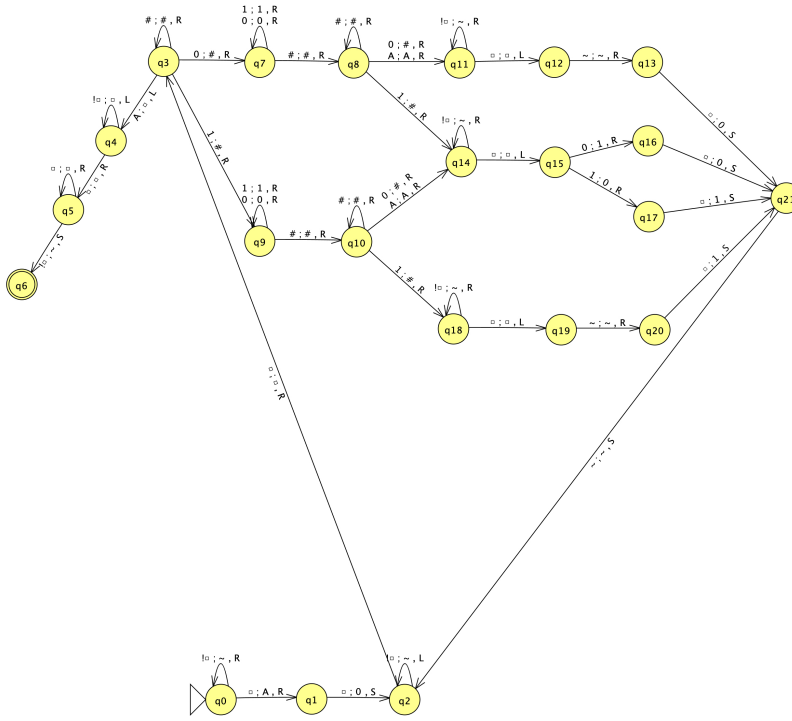


This machine cheats a bit to get the correct answer! It just replaces the symbol # in the middle with a 1 and then replaces the symbol pair 1# at the right end with the symbol pair #□. That does not really feel like we are doing addition, but the answer is correct. It is not that often that computing provides a nifty simple trick that does the job. Enjoy the few that come along!

binary integer addition

#101#100101# → #011101#
5 (1+4) + 41 (1+8+32) = 46 (2+4+8+32)

the most significant bits are toward the right hand side



This Turing Machine performs legitimate binary addition on each digit, keeping track of the first bit, the second bit, and also any carry bit. The various possibilities account for the large number of states in this machine.

Remember that this machine performs binary addition *from left to right* – the reverse of the more common representation.

1.3.1 Standard Descriptors and Numeric Descriptors

We now return our focus back to Turing's original paper and describe two representations for a specific machine – the first is symbolic and is called a **standard descriptor (SD)**; the second is numeric and is called a **numeric descriptor (ND)**.

Recall the basic elements of a Turing Machine:

- states: $q_0, q_1, q_2, \ldots, q_n$
- symbols: $s_0 = \square, s_1 = 0, s_2 = 1, s_3, s_4, \ldots, s_m$
- transition table 5-tuples have the form:

current	symbol	print	move	next
q_i	s_j	s_k	L / R / S	q_{next}

To generate a **standard descriptor**, convert each tuple in the transition table to a string of symbols:

- state $q_i \rightarrow$ DA...A (A is repeated i times)
- symbol $s_j \rightarrow$ DC...C (C is repeated j times)
- L / R / S remain as is: L / R / S
- at the end of each tuple, add a semicolon (;)
- concatenate in order all the generated strings

example

The tuple q_3 0 1 L q_5 becomes the string

DAAADCDCCLDAAAAA;

To generate a **numeric descriptor**, convert the standard descriptor to a *very large* integer:

- the letter A becomes the digit 1
- the letter C becomes the digit 2
- the letter D becomes the digit 3
- the letter L becomes the digit 4
- the letter R becomes the digit 5
- the letter S becomes the digit 6
- the semicolon (;) becomes the digit 7

Example

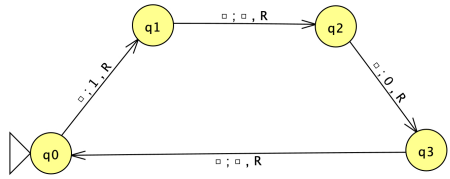
The standard descriptor for q_3 0 1 L q_5 is the string

DAAADCDCCLDAAAAA;

The numeric descriptor would therefore be the integer

31113232243111117.

Consider the Turing Machine **two-thirds**



current	symbol	print	move	next
$\triangleright q_0$	□	1	R	q_1
q_1	□	□	R	q_2
q_2	□	0	R	q_3
q_3	□	□	R	q_0

The **standard descriptor (SD)** for this Turing Machine would be:

DDDCCRDA;
DADDRDAA;
DAADDCRDAAA;
DAAADDRD;

The **numeric descriptor (ND)** for this Turing Machine would be:

3332253173133531173113325311173111335 37

Proposition 1.3.1. *Every computable number has* at least one *standard descriptor (SD) and* at least one *numeric descriptor (ND) associated with it!*

Comment. Multiple descriptors are possible for a single computable number! Two consecutive transitions might cancel each other out – e.g., move left and then immediately move right without changing any symbols. These could be removed and not affect the calculation. Or rearrange the tuples in the transition table – this will define an equivalent Turing Machine; the equivalent machines will each have a different set of descriptors.

Comment. A given natural number need **not** be a numeric descriptor for a computable number! There are no 8s, 9s, or 0s in a numeric descriptor. And even if a sequence of digits can be translated into a standard descriptor, the resulting SD might be gibberish; and even if the SD represented a Turing Machine, it might not be **circle-free**.

For this text I will use the symbol $\mathfrak{C}$ to represent the **set of all computable numbers**, which is equivalent to the **set of all circle-free Turing Machines**.

Lastly, if anyone is closely reading along in Turing's paper which is found in the Supplemental Materials, you have probably noticed that Turing designated states $q_1, q_2, \ldots$ and did not reference any state q_0. This eliminates any ambiguity regarding the meaning of the symbol **D** by itself – does it represent the state q_0 or does it represent the symbol $s_0 = \square$? Subsequent As and Cs distinguish states from symbols; however, the position of the encoding within the transition table also distinguishes states (first and fifth positions) from symbols (second and third positions).

I have chosen to allow this redundancy of meaning in my presentation to simplify implementing Turing Machines in the JFLAP software where the initial state created is q_0. And even though it changes standard descriptors and numeric descriptors ever so slightly, it does not impact the important conclusions derived by Turing.

Later, when we discuss Turing's Universal Machine, this little kludge will have *a very serious consequence.*

1.4 Number Systems and Cardinality

The following should be familiar number systems:

natural numbers	$\mathbb{N}$	$\{ 1, 2, 3, 4, \ldots\}$
integers	$\mathbb{Z}$	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
fractions	$\mathbb{Q}$	$\{ p/q \mid p,q \in \mathbb{Z} \text{ and } q \neq 0\}$

Fractions are also called *rational numbers*. If we carry out the division to generate a decimal expansion, we find that they will either terminate (i.e., divide evenly)

$$1/4 = 0.25$$

or they will repeat (i.e., return to a previously encountered division)

$$4/7 = 0.571428\mathbf{5714238}\ldots$$

These numbers make some sort of sense in their definition and their representation – they are *sensible* or *rational*!

On the other hand there are decimal expansions that are neither terminating nor repeating – they are *irrational*!

Examples

- **radical 2** = square root of 2 = $\sqrt{2} = 1.41421356237\ldots$
- **pi** = $\pi = 3.141592653589\ldots$
- **euler constant** = $e = 2.718281828459\ldots$
- x = $0.11010010001000010000010000001\ldots$

The first three examples above are (or should be) familiar to you as irrational numbers, but simply giving a finite listing of digits proves nothing about being rational or irrational. Most of the proofs for irrationality of a number require some serious thought. The fourth example above (the flight of the conchords) is obviously irrational, whether viewed as a binary expansion or as a decimal expansion.

The final number system that we commonly encounter is:

real numbers $\mathbb{R}$ { all possible decimal expansions } =
{ all rational numbers } ∪ { all irrational numbers }

There is the following relationship which obviously holds among these various number systems:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$$

Each set of numbers expands the set of numbers preceding it with the addition of new values. The collections are getting bigger ... and bigger ... and huger ... ***Or are they?***

Definition 1.4.1. We say that a set A is a **finite set** provided $A = \emptyset$ or there exists a natural number n and a **one-to-one and onto** function $f : \{1, 2, \ldots, n\} \to$ A.

Definition 1.4.2. We say that a set A is an **infinite set** provided that A is **not** a finite set.

Examples

A = { peter , paul, mary }	is a finite set	containing 3 items
B = { 1, 5, 10, 15, 25 }	is a finite set	containing 5 items
C = { } = ∅	is a finite set	containing 0 items
$\mathbb{N}$	is an infinite set	
$\mathbb{Z}$	is an infinite set	
$\mathbb{Q}$	is an infinite set	
$\mathbb{R}$	is an infinite set	

Definition 1.4.3. We say that a set A is a **countably infinite set** provided there exists a **one-to-one and onto** function $f : \mathbb{N} \to$ A.

Examples

$\mathbb{N}$	is countably infinite	$f(n) = n$
X = { 1, 1/2, 1/3, 1/4, ... }	is countably infinite	$f(n) = 1/n$
E = { 2, 4, 6, 8, ... }	is countably infinite	$f(n) = 2n$
O = { 1, 3, 5, 7, ... }	is countably infinite	$f(n) = 2n - 1$
$\mathbb{Z}$	is countably infinite	$f(n) = (-1)^n\ [n/2]$

The third and fourth example above illustrate the unusual characteristic that infinite sets may possess. Both sets **E** and **O** are proper subsets of the larger set $\mathbb{N}$ – and yet all three are the same size, countably infinite.

The last example above has a one-to-one and onto function which initially looks very strange! However, it simply starts with zero and then hops back and forth from a positive integer to its negative, the next positive integer to its negative, and so on.

Definition 1.4.4. We say that a set A is a **countable set** provided either A is a finite set or A is a countably infinite set.

Definition 1.4.5. We say that A is an **uncountable set** provided A is not a countable set.

Theorem 1.4.1. ***(Basic Properties)***

1. *If A is a countable set and $B \subseteq A$, then B is a countable set.*
2. *If both A and B are countable sets, then both $A \cup B$ and $A \cap B$ are countable sets.*
3. *If A_1, A_2, A_3, ... is a countable collection of countable sets, then $\cup_{i=1}^{\infty} A_i$ is a countable set.*

Proof. (Basic Properties)

Item 1
For each b ∈ B, identify the corresponding a ∈ A. Start listing the matched entries found in A from left to right. if the listing stops, then B is a finite set and is therefore countable; but if the listing continues without end, then B is a countably infinite set and is therefore countable.

Item 2
Alternate between the two sets A and B and listing only new items encountered (i.e., no duplicates). If the listing stops, then A ∪ B is a finite set and is therefore countable; but if the listing continues without end, then A ∪ B is a countably infinite set and is therefore countable.

A ∩ B ⊆ A. Hence A ∩ B is countable by *Item 1.*

Item 3
Consider the following listing of the sets under consideration:

$$\begin{array}{lllllll} \mathrm{A}_1 & = \{ & \mathrm{a}_{11} & \mathrm{a}_{12} & \mathrm{a}_{13} & \mathrm{a}_{14} & \ldots \ \} \\ \mathrm{A}_2 & = \{ & \mathrm{a}_{21} & \mathrm{a}_{22} & \mathrm{a}_{23} & \mathrm{a}_{24} & \ldots \ \} \\ \mathrm{A}_3 & = \{ & \mathrm{a}_{31} & \mathrm{a}_{32} & \mathrm{a}_{33} & \mathrm{a}_{34} & \ldots \ \} \\ \mathrm{A}_4 & = \{ & \mathrm{a}_{41} & \mathrm{a}_{42} & \mathrm{a}_{43} & \mathrm{a}_{44} & \ldots \ \} \\ \vdots & & & & & & \end{array}$$

Moving along each diagonal (from the *bottom left to the upper right*) list only new items encountered (i.e., no duplicates). If the listing stops, then $\cup_{i=1}^{\infty}\ \mathrm{A}_i$ is finite and therefore countable; but if the listing continues without end, then $\cup_{i=1}^{\infty}\ \mathrm{A}_i$ is countably infinite and therefore countable.

□

Corollary. *The set of fractions $\mathbb{Q}$ is a countable set.*

Proof. $\mathbb{Q} = \cup_{n=1}^{\infty}\mathbb{Q}_n$ where $\mathbb{Q}_n = \mathbb{Z}/n =$
$\{\ \ldots, -3/n, -2/n, -1/n, 0, 1/n, 2/n, 3/n, \ldots\}$
for $n = 1, 2, 3, \ldots$

Since $\{\ \mathbb{Q}_n\}$ is a countable collection and each set $\mathbb{Q}_n$ is a countable set, the union must also be a countable set. Hence the set of fractions $\mathbb{Q}$ is a countable set.

□

The last result in this section is the exception to the pattern we have been watching develop – every infinite set has ended up being countably infinite. **Georg Cantor** proved that such a pattern quickly fails to continue. The insightful technique in this elegant proof bears his name.

Theorem 1.4.2. ***(Cantor Diagonalization)***
The set of real numbers $\mathbb{R}$ *is an uncountable set.*

Proof. It is sufficient to show that the unit interval I = [0,1] is uncountable. If $\mathbb{R}$ were countable, then I would also have to be countable (since $\mathrm{I} \subseteq \mathbb{R}$). So if I is uncountable, then $\mathbb{R}$ must also be uncountable.

Let us suppose for argument that the unit interval I = [0,1] is in fact a countable set. We are using the method of *indirect proof*. The interval $\mathrm{I} = [0,1] = \{ x_1, x_2, x_3, \ldots\}$ and each value x_i is represented by a decimal expansion:

$$\begin{array}{rcl} x_1 & = & 0.\mathbf{d_{11}} d_{12} d_{13} d_{14} \ldots \\ x_2 & = & 0.d_{21} \mathbf{d_{22}} d_{23} d_{24} \ldots \\ x_3 & = & 0.d_{31} d_{32} \mathbf{d_{33}} d_{34} \ldots \\ x_4 & = & 0.d_{41} d_{42} d_{43} \mathbf{d_{44}} \ldots \\ \vdots & & \end{array}$$

Consider the following decimal expansion, which is surely in the interval [0,1]: $X = 0.D_1 D_2 D_3 D_4 \ldots$. where $D_j = 9 - d_{jj}$. Since X is in the interval $\mathrm{I} = [0,1] = \{ x_1, x_2, x_3, \ldots\}$, $X = x_k$ for some integer value k. *But which one?*

For all integers $k = 1, 2, 3, 4, \ldots$, X differs from x_k in the kth position

$$D_k = 9 - d_{kk} \neq d_{kk}$$

X cannot be equal to any of the decimal expansions that comprise the unit interval I. Hence our assumption that the unit interval is a countable set is incorrect.

□

1.5 Implications

Recall that every computable number in the interval [0,1] has at least one Turing Machine which computes it. Each Turing Machine has an associated standard descriptor (SD) which in turn has an associated numeric Descriptor (ND). Each ND is a natural number. A single Turing Machine may have several SDs and several NDs associated with it; but we can discard redundant items.

Let $\mathfrak{C}_0$ represent all the computable numbers in the interval [0,1]. And let $\mathfrak{C}_k$ represent all the computable numbers in the interval [k,k+1].

The set $\mathfrak{C}_0$ is equivalent to a collection of suitable NDs which in turn is a subset of the natural numbers . Hence $\mathfrak{C}_0$ is countable! Furthermore, it should be apparent that the following infinite collection

$$\{\ 1/2,\ 1/4,\ 1/8,\ 1/16,\ 1/32,\ 1/64,\ \ldots\ \}$$

is an obvious subset of $\mathfrak{C}_0$. Hence $\mathfrak{C}_0$ is countably infinite and so are the sets $\mathfrak{C}_k$!

Theorem 1.5.1. *The computable numbers* $\mathfrak{C}$ *are countably infinite.*

Proof. $\mathfrak{C} = \cup_{k=-\infty}^{+\infty} \mathfrak{C}_k$, each of which is a countably infinite set.

□

Recall that the real numbers are uncountable. There are a lot more real numbers than there are fractions, and there are a lot more real numbers than there are computable numbers. And just like there exist real numbers **r** which are not fractions, we find

Corollary. *There exist real numbers* ***r*** *which are not computable.*

Proof. If we assume that no such real number **r** exists, then $\mathbb{R} \subseteq \mathfrak{C}$ and $\mathbb{R}$ is countable. A contradiction!

□

Before we return to the issue of real numbers that are not computable, let us briefly identify several categories which are computable. Since the topic is not germane to the development of the theory of computing, we present this summary without proof.

Examples

1. the natural numbers $\mathbb{N}$
2. the integers $\mathbb{Z}$
3. the fractions $\mathbb{Q}$
4. the square roots $\sqrt{2}, \sqrt{3}, \sqrt{5}, \ldots$
5. the algebraic numbers $\mathfrak{A} =$
   $\{\text{roots to p(x)} | \text{p(x)} = a_n x^n + \cdots + a_1 x + a_0\}$
   $a_i \in \mathbb{Z}$ and $a_n \neq 0$
6. some important transcendental numbers,
   such as **pi** and **euler constant**

1.5.1 Two Real Numbers Which are Not Computable

We now identify two specific real numbers which are easy to define but are not computable. Both numbers are in the interval [0,1]. Consider once again the collection $\mathfrak{C}_0$ containing all the computable numbers in the interval [0,1] which we know to be countably infinite.

$\mathfrak{C}_0 = \{c_1, c_2, c_3, \ldots\}$ with

$$\begin{array}{rcl} c_1 & = & 0.\mathbf{d_{11}} d_{12} d_{13} d_{14} \ldots \\ c_2 & = & 0.d_{21}\mathbf{d_{22}} d_{23} d_{24} \ldots \\ c_3 & = & 0.d_{31} d_{32}\mathbf{d_{33}} d_{34} \ldots \\ c_4 & = & 0.d_{41} d_{42} d_{43}\mathbf{d_{44}} \ldots \\ \vdots & & \end{array}$$

The collection $\mathfrak{C}_0$ is listed in the following order:

- c_1 has the smallest numeric descriptor for any of the computable numbers
- c_2 has the smallest numeric descriptor for any of the remaining computable numbers
- etc.

In simple terms, we are listing the computable numbers in increasing order by their minimal numeric descriptor.

Focus on the diagonal digits (bits): $d_{11}, d_{22}, d_{33}, d_{44}, \ldots$

- Define the real number $\beta =$
  $0.d_{11} d_{22} d_{33} d_{44} \ldots$
- Define the real number $\beta' =$
  $0.(1 - d_{11})(1 - d_{22})(1 - d_{33})(1 - d_{44}) \ldots$

Both numbers are obviously in the interval [0,1] and look innocent enough. Why do they pose a problem to computability?

We consider the second number β' first. If β' is computable, then it appears in the listing for $\mathfrak{C}_0$, i.e., $\beta' = c_k$ for some natural number k. Consider the kth bit for $\beta' = c_k$:

kth bit for $\beta' \rightarrow (1 - d_{kk}) = d_{kk} \leftarrow$ kth bit for c_k

But this means $2d_{kk} = 1$ and $d_{kk} = 1/2$. Binary digits are either 0 or 1. A contradiction!

We next consider the first number β. If β is computable, then there exists a Turing Machine T which computes it. Using the transition table for T we can create a new Turing Machine T' which replaces all print 0 instructions with print 1 and vice-versa.

And what is the computable number that T' generates – β'! But this means that β' is also computable. A contradiction!

1.6 The Universal Machine

At this point in our discussion, Turing has developed a collection of simple machines, each designed to perform a specific computation. The important thing to remember is that **each** machine does a **single task**. It is at this point that Turing realized that the same basic components can be combined to simulate the operation of **any** simple Turing Machine. Turing's Universal Machine $\mathfrak{U}$ can perform any of the tasks that his individual machines can perform.

Terminology

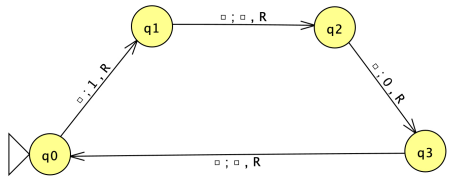
Definition 1.6.1. An **m-configuration** is the current state of the Turing Machine, i.e., a specific state from among the finite set Q: $q_0, q_1, \ldots, q_n$.

Definition 1.6.2. A **configuration** is a pair of values, the current state of the Turing Machine together with the current symbol under the read/write head, (q_i, s_j).

Definition 1.6.3. A **complete configuration** is a snapshot of the entire tape at a given moment in time, identifying the current state by inserting it *immediately to the left* of the current symbol.

example

Consider our Turing Machine two-thirds



m-configurations would include:

q_0, q_1, q_2, q_3

configurations would include:

$(q_0, \square), (q_1, \square), (q_2, \square), (q_3, \square)$

the tape is initially blank:

$\ldots\square\square\square\square\square\ldots$

complete configurations (listed in order) would include:

$q_0\square$	*initial tape*
$1q_1\square$	
$1\square q_2\square$	
$1\square 0q_3\square$	
$1\square 0\square q_0\square$	
$1\square 0\square 1q_1\square$	

Turing realized that he could simulate the behavior of any Turing Machine with a single tape Turing Machine which, rather than focusing on **numeric calculation**, would focus on **symbol manipulation** of the symbols appearing on the tape.

A basic Turing Machine T would have a numeric calculation transition table built in as part of its definition: the tape would be initially blank, F cells would be for actual output of binary digits, E cells would provide a scratch pad for calculations. Simple representations for this Turing Machine would be its Standard Descriptor (SD) and its Numeric Descriptor (ND).

Turings Universal Machine $\mathfrak{U}$ would have a symbol manipulation transition table built in as part of its definition. Turings Universal Machine $\mathfrak{U}$ would have an initial input tape containing the following (in order):

a sentinel @ @	first @ in initial F cell second @ in initial E cell
standard descriptor (SD)	for the Turing Machine T simulated stored in consecutive F cells
another sentinel ::	stored in the next F cell
initial complete configuration	stored in consecutive F cells
a colon (:)	used as a separator between complete configurations for balance of the tape

Example

Consider again our Turing Machine **two-thirds**. The following would be the initial contents of the input tape. In order to compress the representation and save space we display only the contents of the F cells and not the scratch pad E cells.

@DDDCCRDA;
DADDRDAA;
DAADDCRDAAA;
DAAADDRD;
::
DD

DD is the initial configuration $q_0\square$

The following is a listing of complete configurations as generated by the Universal Machine $\mathfrak{U}$.

	::DD:1:
DD → **DCCRDA**	**DCCDAD**
	::DD:1:
DAD → **DRDAA**	**DCCDAD:**
	DCCDDAAD
	::DD:1:
DAAD → **DCRDAAA**	**DCCDAD:**
	DCCDDAAD:0:
	DCCDDCDAAAD
	::DD:1:
DAAAD → **DRD**	**DCCDAD:**
	DCCDDAAD:0:
	DCCDDCDAAAD
	DCCDDCDDD
	::DD:1:
DD → **DCCRDA**	**DCCDAD:**
	DCCDDAAD:0:
	DCCDDCDAAAD
	DCCDDCDDD:1:
	DCCDDCDDCCDAD

Please note that I have already included a key feature of the simulation into the above example. In addition to using symbol manipulation to locate the proper transition for the current configuration and then updating the tape to the next complete configuration, if the Universal Machine $\mathfrak{U}$ recognizes the print character to be either a 0 or a 1 ***then it writes that numeric symbol to the tape prior to generating the next complete configuration***. As a result the solution to the given problem is interspersed among the symbolic representations as the Universal Machine U progresses through the simulation.

Comment. My desire to maintain consistency between Turing's paper and JFLAP's software creates a real problem within the complete configurations. The redundancy $q_0 = s_0 = D$ is not a problem in the standard descriptor; but it is a definite problem with complete configurations. There is exactly one position in a complete configuration that holds the current state. Any sequence of two or more Ds in a row becomes ambiguous.

e.g., $q_0\square\square$ $\square q_0\square$ $\square\square q_0$ $\square\square\square$ are all encoded

DDD

The ambiguity disappears if the start state was q1 rather than q0; all states would then contain at least one A in its encoding and would be uniquely identified within the complete configurations. That may have been Turing's intention all along!

Turing's Universal Machine $\mathfrak{U}$ is the modern digital computer! The Standard Descriptors and the Numeric Descriptors are precursors for future assembly languages and machine languages that we use today.

We have seen that Turing Machines can compute numeric values, perform arithmetic calculations, and perform string matching and manipulation. We have seen that Turing Machines can be very useful even when performing infinitely long processes; and they can be adapted to generate useful results in finite time.

So now we come full circle and return to the question we confronted in the very first section – the Entſcheidungſproblem! Is it possible to utilize the power of Turing Machines to answer this question?

1.7 Decidability

We now turn our attention to Turing Machines which are further enhanced with halting states: an ACCEPT state (or YES) and a REJECT state (or NO). Even with the addition of these two states, Turing Machines may still be circular or may crash when encountering an undefined configuration.

We are now going to consider problems that require a simple YES or NO answer. Such problems are called *decision problems*.

example

Rather than actually solve a problem:

e.g., solve $3x - 6 = 0$ for x.

we restate the problem as a question:

e.g., does the equation $3x - 6$ have an integer solution?

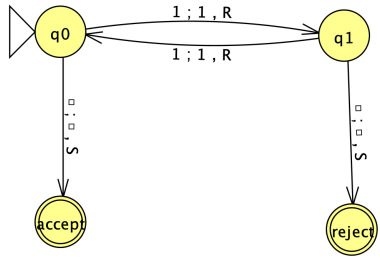
Definition 1.7.1. A Turing Machine with ACCEPT and REJECT states is called an **acceptor** or a **recognizer**.

Definition 1.7.2. A Turing Machine which is an acceptor/recognizer and satisfies the additional condition that **it will always halt** in either one of the ACCEPT / REJECT states is called a **decider**.

Example

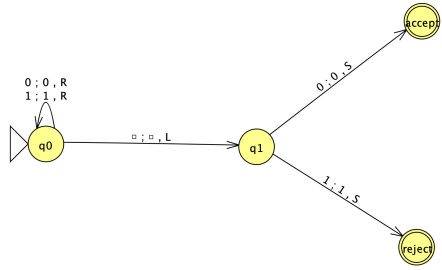
Question: Is the following positive integer even? YES or NO?

Turing Machine T1 *(unary data)*
input date: 1111111 = 7



Turing Machine T1 is a **decider**

Turing Machine T2 *(binary data)*
input date: 101011 = 43
most significant digits to the right



Turing Machine T2 is a **decider**

Turing Question: *Given a natural number* ***n****, is this number the Numeric Descriptor (ND) for a circle free Turing Machine which calculates a computable number?*

> This question is definitely appropriate to the discussion as presented in Turing's original paper.

And this question is provably **not** decidable!

Suppose this question were decidable. Then there exists a Turing Machine $\mathfrak{D}$ which decides this question. Let us use this $\mathfrak{D}$ to build another Turing Machine $\mathfrak{B}$:

1. set i = 0
2. generate (one at a time, in increasing order) $k = 1, 2, 3, \ldots$
   (a) for each k determine whether $\mathfrak{D}$ (k) accepts or rejects
      i. if $\mathfrak{D}$ rejects move to the next k
      ii. if $\mathfrak{D}$ accepts continue
   (b) increment i
   (c) running the Turing Machine with Numeric Descriptor k
      i. generate $c_{i1}, c_{i2}, c_{i3}, \ldots, c_{ii}$ for computable number C_i
      ii. stop at binary digit c_{ii} and print it
   (d) move to the next k

Observe that we have created a Turing Machine which computes the number β – a number which is not computable! A contradiction.

We must therefore conclude that no such Turing Machine $\mathfrak{D}$ exists. The question above is therefore undecidable.

Sipser Question: *Given the Standard Descriptor (SD) for a Turing Machine and given an input string ω of symbols, can we decide whether or not the Turing Machine defined by SD will accept the input string ω?*

> This question is appropriate to the modern theory of formal languages.

This question is also provably **not** decidable!

Suppose this question were decidable. Then there exists a Turing Machine $\mathfrak{D}$ which decides this question. Let us use this $\mathfrak{D}$ to build another Turing Machine $\mathfrak{X}$:

1. input is the Standard Descriptor (SD) for a Turing Machine
2. simulate the Turing Machine defined by <SD> on the string <SD>
   return the value **opposite of $\mathfrak{D}$ (<SD> , <SD>)**

In simple English, if the Turing Machine defined by the Standard Descriptor <SD> ACCEPTS the input string ω = <SD>, $\mathfrak{X}$ will REJECT and if the Turing Machine defined by the Standard Descriptor <SD> REJECTS the input string ω = <SD>, $\mathfrak{X}$ will ACCEPT.

Furthermore, $\mathfrak{X}$ is a Turing Machine and has its own SD (denoted <$SD_{\mathfrak{X}}$>) and its own ND (denoted <$ND_{\mathfrak{X}}$>).

Let us determine the value of $\mathfrak{X}$ (<$SD_{\mathfrak{X}}$ >).

$$\begin{aligned} \mathfrak{X} \text{ (<SD}_{\mathfrak{X}}\text{>) } &= \textbf{opposite of } \mathfrak{D} \text{ (<SD}_{\mathfrak{X}}\text{> , < SD}_{\mathfrak{X}}\text{>)} \\ &= \textbf{opposite } \mathfrak{X} \text{ (<SD}_{\mathfrak{X}}\text{>)} \end{aligned}$$

But this is impossible – $\mathfrak{X}$ can not simultaneously both ACCEPT and REJECT its own Standard Descriptor (<$SD_{\mathfrak{X}}$>).

We must therefore conclude that no such Turing Machine $\mathfrak{D}$ exists. The question above is therefore undecidable.

At this point we conclude our introductory chapter highlighting the portions of Turings paper that are relevant to the subsequent development of formal language theory.

The next two chapters will introduce two categories of Turing Machines – first the finite automata and second the push down automata. The chapter after that returns again to Turing Machines but in the context of formal languages and with a few more features to consider incorporating into our growing collection of definitions.

Chapter 2

Finite Automata

2.1 Formal Language Theory

The modern theory of computing has evolved into three areas of study:

- formal language theory and finite state machines (Turing Machines are divided into more specific categories according to the formal languages they accept)
- issues of decidability and undecidability
- space and time complexity

We have already encountered the fundamental Turing Machine and a handful of introductory problems regarding decidability. Some of our examples were decidable; others were not. We expand this study and identify several additional categories of problems within formal language theory. We then determine whether or not each of these categories is decidable.

If a problem is decidable, then the next obvious question would be: How long does it take to actually solve it? The third area of study focuses on the execution time and the space (storage) requirements necessary to solve a given problem.

Formal language theory covers a broad range of topics. At times formal languages will appear to be very similar to natural language and sentence structure. At times formal languages may appear to be nothing more than structured gibberish! However, formal languages play a very important role in computing – including pattern matching, lexical analysis, and programming language syntax.

Building Block Components

Definition 2.1.1. An **alphabet** Σ is a finite set of symbols.

examples

1. S = { 0 , 1 }
2. S = { a , b , c }
3. S = { a , b , c , . . . , z }
4. S = { boy , girl , a , the , wears , pants , skirt }

Definition 2.1.2. A **word** ω is a finite string of symbols taken from the underlying alphabet Σ.

examples

1. 0101 , 1000, 0 , 1 , ϵ = empty string!
2. abc , cba , abba , cab , baaaa
3. dog , cat , zebra , blahblahblah
4. the boy wears pants , a girl wears a skirt

Comment. Depending on the underlying alphabet, a word ω might look like any of the following:

- a string of bits, i.e., a binary number
- a word in the dictionary
- an English sentence
- a JAVA program

Definition 2.1.3. A **language** L is a collection of words ω built from the underlying alphabet Σ.

examples

1. L = { ω over {0,1} | #0 = #1 }
2. L = { ω over {a,b,c} | #c = #a + #b }
3. L = { ω over {a,b,...,z} | ω is a palindrome }
4. L = { syntactically correct JAVA program }

Obviously formal languages run a wide range of possibilities from the very simple to the very complex. As we continue our discussion of formal languages, we will fill in the details for *Chomsky's Hierarchy of Languages*:

level	**language name**	**finite state machine**
3	regular	finite automaton
2	context free	push down automaton
1	context sensitive	linearly bounded automaton
0	unrestricted	Turing Machine

As we progress through the subsequent chapters and sections, we will give meaning and substance to each of the terms above. At this point it is sufficient to understand that different categories of finite state machine are associated with the different levels of languages.

2.2 Definitions

Definition 2.2.1. A **finite automaton** M is a 5-tuple (Q, Σ, δ, q_0, F) where Q and Σ are both finite sets and

1. Q is a set of **states**
2. Σ is a set of **symbols** (alphabet)
3. δ is a **transition function**
   δ (q , s) = q_{next}
4. q_0 is a unique **start state**, $q_0 \in Q$
5. $F \subseteq Q$ is a set of **accept states**

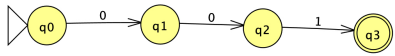
This scaled down Turing Machine satisfies the following conditions:

- input only, no output is generated
- processes input from left to right in single steps (no backtracking)
- machine stops when no more input is available
- if the final state is an element of F, then ACCEPT; if not, then REJECT
- transition function δ is hard-wired into the machine, same as a standard Turing Machine
- any undefined transition immediately halts and REJECTs

examples

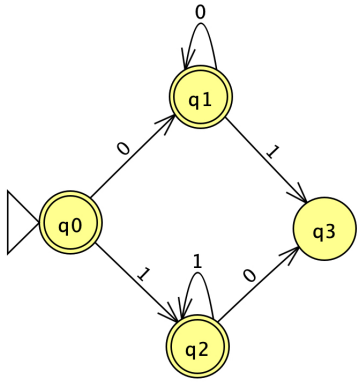
A finite automaton which recognizes a finite number of words (one to be specific!)

$$L = \{ 001 \}$$



A finite automaton which recognizes an infinite number of words

$$L = \{ \epsilon, 0, 1, 00, 11, 000, 111, 0000, 1111, 00000, 11111, \ldots \}$$



Definition 2.2.2. We say that a finite automaton M **accepts** the word ω provided:

1. $\omega = w_1\ w_2\ \ldots\ w_k$, with $w_j \in \Sigma$ for j = 1, 2, ..., k
2. there exists a sequence $r_0, r_1, \ldots, r_k \in Q$ such that

   - $r_0 = q_0$, the start state
   - $r_j = \delta\ (\ r_{j-1}\ ,\ w_j\ )$, for j = 1, 2, ..., k
   - $r_k \in F$, an accept state

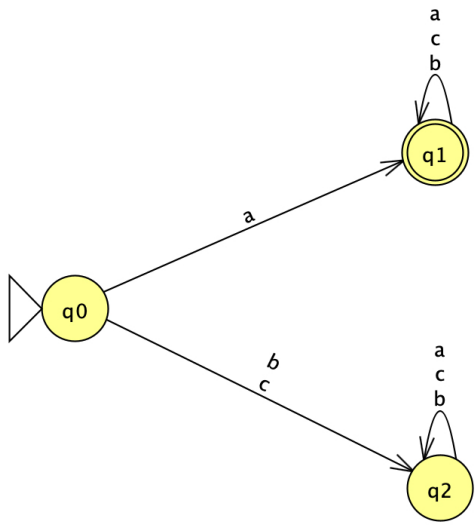
Definition 2.2.3. The language **recognized** by a finite automaton M is the collection

$$L\ (M) = \{\ \text{words}\ \omega \mid M\ \text{accepts}\ \omega\ \}$$

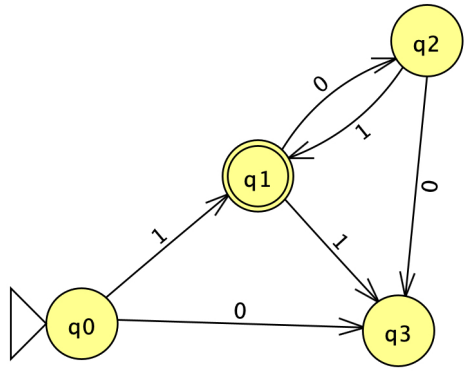
examples

easy example:

L (M) = { ω over {a,b,c} | ω begins with a }



harder example:
L (M) = { 1, 101, 10101, 1010101, 101010101, ... }



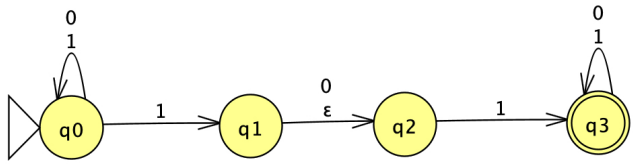
2.2.1 Nondeterminism

At this point in our discussion we come to a very important new topic – the differentiation between a **deterministic** machine and a **nondeterministic** machine.

The previous definition and examples should more precisely be referred to as deterministic finite automata (DFA rather than just FA) in that the transition function δ uniquely determines the next state. There are *no alternative options* and *no potential choices* for the machine to consider. The concept of nondeterminism introduces the possibility of the machine following multiple branches at any given configuration of state and symbol – a nondeterministic finite automata (NFA).

example

our first NFA:



- at state q_0: $\delta(q_0,1) = q_0$ and $\delta(q_0,1) = q_1$
  when I see a 1, which state should I move to?
  q_0 or q_1
- at state q_1: $\delta\ (q_1,0) = q_2$ and $\delta\ (q_1,\epsilon) = q_2$
  should I remove the input symbol 0
  or should I remove nothing?
  in either case, I move to state q_2

The special symbol ϵ represents the empty string (no characters). A transition incorporating the symbol ϵ (called an ϵ-transition) does not remove any symbol from the input line.

Nondeterministic machines are especially nice for hiding the complexity in a given task. Let the machine work out all the details generated by the various choices. We incorporate this new concept into another definition.

Please note that the updated definitions include the special symbol ϵ and that the transition function no longer defines a single state but rather a ***set*** of possible states.

Definition 2.2.4. A **nondeterministic finite automaton** M is a 5-tuple (Q, Σ_ϵ, δ, q_0, F) where Q and Σ_ϵ are both finite sets and

1. Q is a set of **states**
2. Σ_ϵ is a set of **symbols** (alphabet) augmented with the empty string ϵ = ""
3. δ is a **transition function** δ (q , s) = set of possible next states
4. q_0 is a unique **start state**, $q_0 \in Q$
5. $F \subseteq Q$ is a set of **accept states**

Definition 2.2.5. We say that a nondeterministic finite automaton M **accepts** the word ω provided:

1. $\omega = w_1\ w_2 \ldots w_k$, with $w_j \in \Sigma_\epsilon$, for j = 1, 2, ..., k
2. there exists a sequence $r_0, r_1, \ldots, r_k \in Q$ such that
   - $r_0 = q_0$, the start state
   - $r_j \in \delta$ (r_{j-1} , w_j), for j = 1, 2, ..., k
   - $r_k \in F$, an accept state

Definition 2.2.6. The language **recognized** by a nondeterministic finite automaton M is the collection

$$L\ (M) = \{\ \text{words } \omega \mid M \text{ accepts } \omega\ \}$$

Please note that a deterministic finite automaton (DFA) is automatically a nondeterministic finite automaton (NFA)! A transition function δ that identifies a single state automatically identifies a set of states (not a large set, just one). Hence

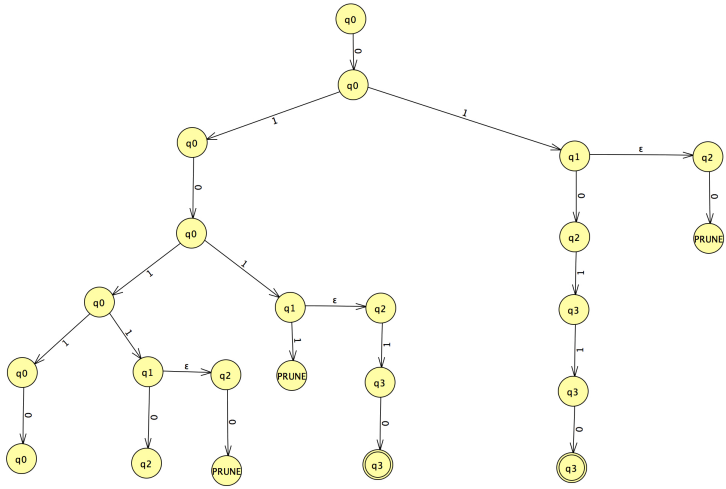
$$\{\ \text{DFAs}\ \} \subseteq \{\ \text{NFAs}\ \}$$

So what does nondeterminism look like in practice? How does a finite automaton keep track of the various choices as they are encountered? We illustrate the process in the following example using our first nondeterministic example.

example

Consider the following input string $\omega = 010110$. Does M ACCEPT or REJECT this word?

As we encounter choices, we will generate a tree structure; as we find undefined transitions, we will prune the tree structure. After considering the last symbol in the input string, we look to the various states that have not been pruned. If any one of them is an accept state, then the nondeterministic finite automaton ACCEPTs the string!



Comment.

an input symbol moves *vertically* down one level
an ϵ-transition moves *horizontally* right one position

Theorem 2.2.1. *Every nondeterministic finite automaton M has an equivalent deterministic finite automaton M', i.e., both machines recognize exactly the same language*

$$L\ (M) = L\ (M')$$

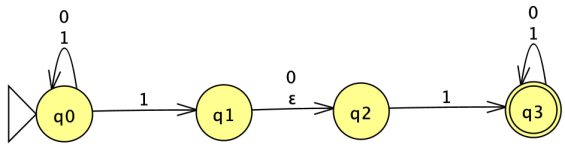
Proof. For any finite set A, the power set $\wp(\mathrm{A}) = \{\ \mathrm{S} \mid \mathrm{S} \subseteq \mathrm{A}\ \}$. If $\#(\mathrm{A}) = \mathrm{k}$, then $\#(\wp(\mathrm{A}) = 2^k$.

1. The original start state becomes the start state for the resulting deterministic finite automaton.
2. For every element in $\wp(\mathrm{Q})$ other than the empty set, determine its images via the transition function δ, excluding any ϵ-transitions. This image will itself be a set of states.
3. For every ϵ-transition, augment the image sets generated in step two. *Repeat* this step as needed.
4. Delete any element in $\wp(\mathrm{Q})$ having no incoming transition. *Repeat* this step as needed.
   the start state q_0 should be considered as having an incoming transition!
5. Any remaining element in $\wp(\mathrm{Q})$ containing an original accept state becomes an accept state for the resulting deterministic finite automaton.

□

example

Our original nondeterministic finite automaton again!



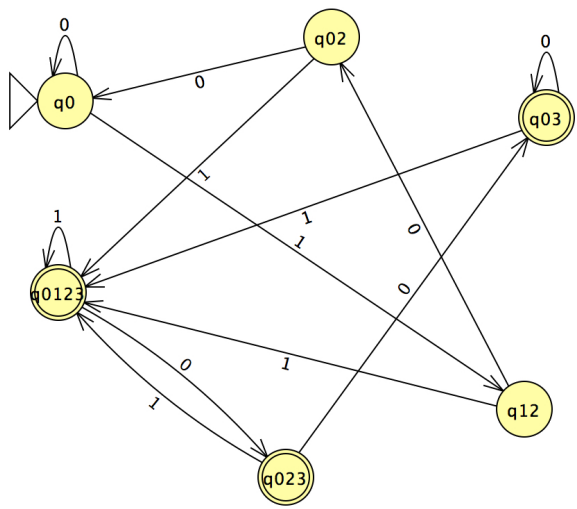
There are four states in our automaton. The number of possible subsets of four states would be sixteen.

$$\#\mathrm{Q} = 4 \Rightarrow \#\ \wp\ (\mathrm{Q}) = 2^4 = 16.$$

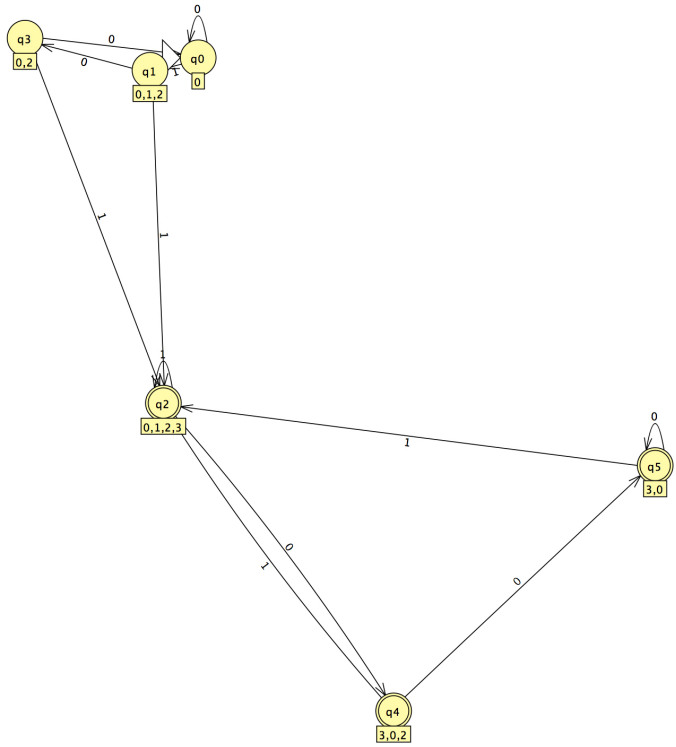
	subset	**0**	**1**	ϵ	
	▷ 0	0	0,1,2		
no incoming	1	2	–	**2**	
no incoming	2	–	3		
no incoming	3	3	3		
no incoming	0,1	0,2	0,1,**2**		←
	0,2	0	01,3,**2**		←
	0,3	0,3	0,1,3,**2**		←
no incoming	1,2	2	3		
no incoming	1,3	2,3	3		
no incoming	2,3	3	3		
	0,,12	0,2	0,1,3**2**		←
no incoming	0,1,3	0,2,3	0,1,3,**2**		←
	0,2,3	0,3	0,1,3,**2**		←
no incoming	1,2,3	2,3	3		
	0,1,2,3	0,2,3	0,1,3,**2**		←

So our remaining states would be: $q_0, q_{02}, q_{03}, q_{012}, q_{023}, q_{0123}$
initial state: q_0
accept states: $q_{03}, q_{023}, q_{0123}$

The above scratch work yields the following deterministic finite automaton:



Using JFLAP software, it is possible to do the conversion process with just a few clicks of the mouse button.



The layout for computer generated solution looks quite different from the solution on the previous page! But a careful examination of both should assure you that they are, in fact, the very same!

Question: Now that we have two flavors of finite automata, which one will we work with?

Answer: BOTH! The problem at hand will often determine which of the two is easier to work with.

- DFAs are easier to understand and implement due to their simplicity
- NFAs are easier to manipulate and combine due to their flexibility

2.3 Regular Languages & Regular Expressions

Definition 2.3.1. A **regular language** is a language L which is recognized by a finite automaton M, i.e.,

$$L = L\ (M)$$

Comment. M may be either deterministic or nondeterministic!

six fundamental operations with languages

union	$L_1 \cup L_2$
intersection	$L_1 \cap L_2$
complement	L^C
difference	$L_1 \sim L_2$
concatenation	$L_1 \circ L_2$
Kleene closure	L^*

where

$$L_1 \circ L_2 = \{\ \omega = \omega_1\ \omega_2 \mid \omega_1 \in L_1 \text{ and } \omega_2 \in L_2\ \}$$
$$L^* = \{\omega = \omega_1\ \omega_2\ \ldots\ \omega_k \mid \omega_j \in L \text{ and } k \geq 0\ \}$$

Theorem 2.3.1. *Regular Languages are closed under the six fundamental operations.*

Proof. **For union and intersection:** we will use the basic definitions

L_1 is recognized by $M_1 = (\ Q', \Sigma'\ ,\ \delta'\ ,\ q_0'\ ,\ F'\ )$
L_2 is recognized by $M_2 = (\ Q''\ ,\ \Sigma''\ ,\ \delta''\ ,\ q_0''\ ,\ F''\ )$

Define $M = (\ Q\ ,\ \Sigma\ ,\ \delta\ ,\ q_0\ ,\ F\ )$ as follows:

$Q = Q'$ x $Q'' = \{\ (q',q'') \mid q' \in Q' \text{ and } q'' \in Q''\ \}$
$\Sigma = \Sigma' \cup \Sigma''$
$q_0 = (\ q_0'\ ,\ q_0''\ )$
$\delta\ (\ (q',q'')\ ,\ s\ ) = (\ \delta'\ (q',s)\ ,\ \delta''\ (q'',s)\ )$
F will be defined shortly!

Comment. We are running both machines simultaneously!

For union
F = F′ x Q″ ∪ Q′ x F″, i.e., **either** machine may accept!

For intersection
F = F′ x F″, i.e., **both** machines must accept!

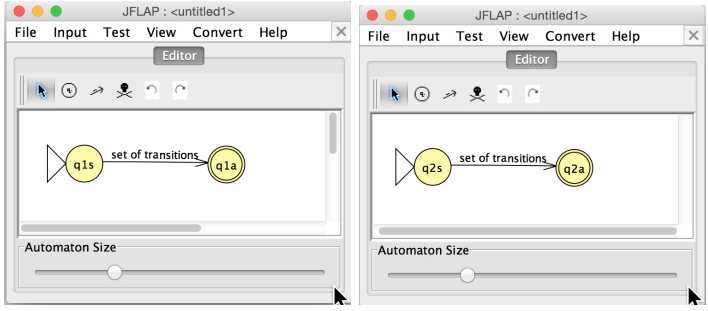
For complement and set difference: we will use some tricks from set theory

For complement
L is recognized by M = (Q , Σ , δ , q_0 , F)
L^C is recognized by M′ = (Q , Σ , δ , q_0 , F^C),
where F^C = Q ∼ F

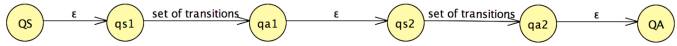
For set difference
$L_1 \sim L_2 = L_1 \cap L_2^C$
both complement and intersection are closed
as previously demonstrated

For concatenation and Kleene closure: we will use the flexibility of NFAs to our advantage

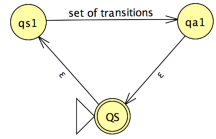
Language L_1 is recognized by a finite automaton M_1; language L_2 is recognized by a finite automaton M_2.



The following finite automaton will recognize the language $L_1 \circ L_2$.



The following finite automaton will recognize the language L_1^*.



□

Regular languages have been defined in terms of the finite automata that generate / recognize them. These machines can sometimes be quite cumbersome and confusing. An alternate approach to describing a language would be with a pattern or a template describing its words. This is the idea behind our next topic – regular expressions.

Definition 2.3.2. A **regular expression** R describing the words in a language L is defined inductively:

$\emptyset$ is a regular expression	the empty language
ϵ is a regular expression	the empty string
$s \in \Sigma$ is a regular expression	any single character
if R_1 and R_2 are regular expressions	then so is $R_1 \cup R_2$
if R_1 and R_2 are regular expressions	then so is $R_1 \circ R_2$
if R is a regular expression	then so is R^*

Notation

The notation for regular expressions may take several forms

$(a \cup b \cup c) \circ b^* \circ d \circ e$	traditional
$(a + b + c) \circ b^* \circ d \circ e$	arithmetic
$\{ a , b , c \}\ b^*\ d\ e$	implied concatenation

Remember:

$\emptyset$ = empty language $\Rightarrow$ no words
ϵ = empty string $\Rightarrow$ no characters, but is a word

so

$a \cup \emptyset = a$
$a \circ \emptyset = \emptyset$
$\emptyset^* = \epsilon$
$a \cup \epsilon = \{ a , \epsilon \}$
$a \circ \epsilon = a$
$\epsilon^* = \epsilon$

Definition 2.3.3. If R is a regular expression, then the language **generated** by R is

$$L = L(R) = \{ \text{ words } \omega \mid \omega \text{ matches the pattern defined by R } \}$$

Examples

$R = c \circ a \circ t$	$L(R) = \{ \text{ cat } \}$
$R = (a + b)^*$	$L(R) = \{ \omega \text{ over } \{ a, b \} \mid \lvert\omega\rvert \geq 0 \}$

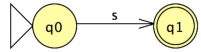
Lemma 2.3.2. *If R is a regular expression, the L (R) is a regular language.*

Proof. inductive proof modeled on the definition of a regular expression

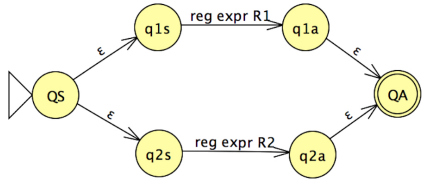
the empty language $\emptyset$ is a regular language

the empty string ϵ is a regular language

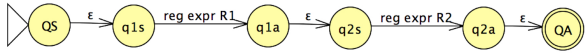
a single symbol s $\in \Sigma$ is a regular language



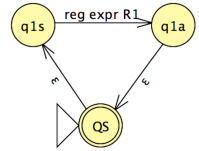
union



concatenation



Kleene closure



□

Lemma 2.3.3. *If L is a regular language, then there exists a regular expression R such that*

$$L = L\ (\ R\ ).$$

Proof. The proof of this lemma requires one more type of finite automaton a generalized nondeterministic finite automaton (GNFA).

Definition 2.3.4. A generalized nondeterministic finite automaton (GNFA) M is a 5-tuple (Q , Σ , δ , q_s , q_a) where Q and Σ are both finite sets and

1. Q is a set of **states**
2. Σ is a set of **symbols** (alphabet)
3. δ is a **transition function**
   δ : (Q$\sim\{q_a\}$) x (Q$\sim\{q_s\}$) : $\rightarrow \Re$ (regular expressions)
   transitions are represented by regular expressions
   rather than individual symbols in Σ_ϵ
4. q_s is a unique **start state**, $q_s \in$ Q, with no incoming arcs
5. q_a is a unique **accept state**, $q_a \in$ Q, with no outgoing arcs

Let L be the regular language under consideration and let M represent the deterministic finite automaton such that L = L (M). If necessary, using nondeterministic techniques, augments M

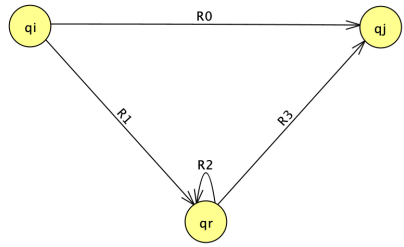
- with a unique start state q_s having no incoming arcs
- with a unique accept state q_a have no outgoing arcs

A single transition from state q_i to state q_j
is represented by the regular expression of a single symbol.
Multiple transitions from state q_i to state q_j
is represented by the union of single symbol regular expressions.
If q_i and q_j are two states with no transition symbol,
then represent a transition from state q_i to state q_j
by the regular expression $\emptyset$.

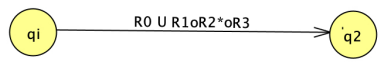
We have now converted a DFA M with k states into a GNFA M' with k+2 states. Furthermore, transitions in M' are all represented by a single regular expression, albeit many having the value $\emptyset$.

Our goal now is to reduce our GNFA having k+2 states to an equivalent GNFA having exactly 2 states – q_s and q_a. The regular expression representing the transition from q_s to q_a will be the desired generator for the language. We do this reduction by eliminating states one at a time using a surprisingly simple process.

Suppose we wish to remove state q_r from the GNFA. Consider any pair of states q_i and q_j. The following diagram illustrates the regular expressions appropriate to the three states.



The direct path (regular expression R_0) can be augmented by the regular expression passing through q_r ($R_1 \circ R_2^* \circ R_3$.



□

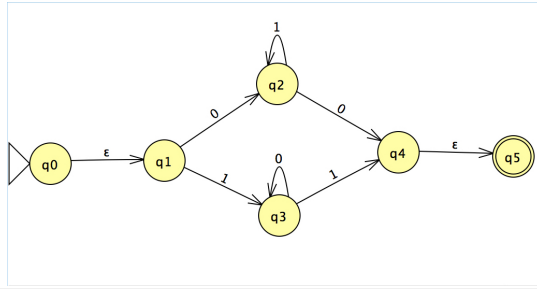
Before we present an example of this process, we summarize all our results regarding regular languages in the following theorem.

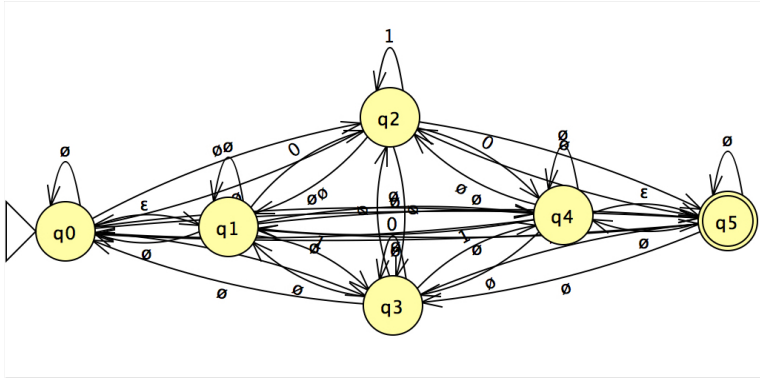
Theorem 2.3.4. *The following five statements are all equivalent:*

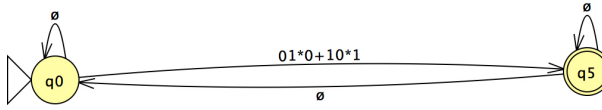
1. *L is a regular language.*
2. *there exists a DFA M such that $L = L(M)$.*
3. *there exists a NFA M such that $L = L(M)$.*
4. *there exists a GNFA M such that $L = L(M)$.*
5. *there exists a regular expression R such that $L = L(R)$.*

examples

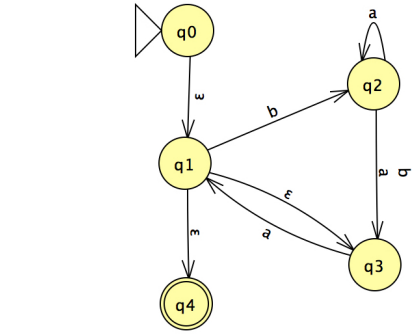
***example 1*: DFA → GNFA → reg expr**

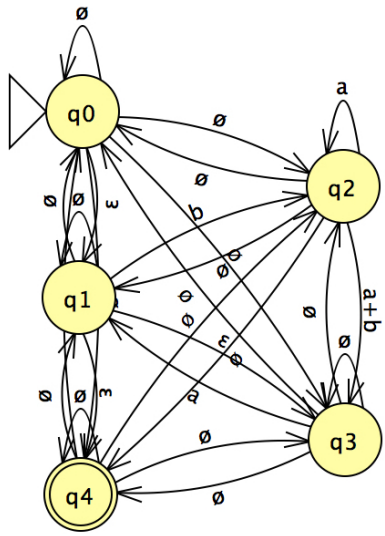


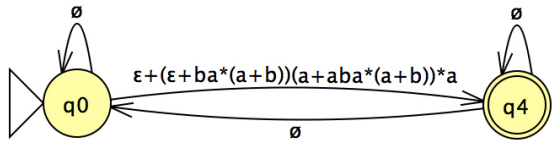




***example 2*: DFA → GNFA → reg expr**







2.4 The Pumping Lemma

In this section we present the pumping lemma which describes a very important property which belongs to all regular languages. It is interesting that the primary use of the property is not in recognizing words within a regular language but rather in proving a given language cannot be regular!

Theorem 2.4.1. ***(The Pumping Lemma)*** *If L is a regular language, then there exists an integer* ***p*** *(called the* ***pumping length****) such that if ωis any string in L with* $|\omega| \geq p$ *then ω may be divided into three substrings* $\omega = x\ y\ z$ *with*

1. $|\ y\ | > 0$ *i.e., y is not* ϵ
2. $|\ x\ y\ | \leq p$
3. $x\ y^k\ z \in L$ *for k = 0, 1, 2, ...*

Proof. Let p = #Q = number of states.

Let $\omega = s_1\ s_2 \ldots s_k$ with $|\ \omega\ | = k \geq p = \#Q$.

$$q_0 \text{ -}(s_1)\text{→ } q_1 \text{ -}(s_2)\text{→ } q_2 \text{ -}\ldots\text{→ } q_k$$

Since $k \geq p = \#Q$, the *pigeon hole principle* tells us that there must be at least one pair of duplicate states in the sequence! Find the first such pair:

$$0 \leq i < j \leq p.$$

Set $x = s_1\ s_2 \ldots s_i$, $y = s_{i+1}\ s_{i+2} \ldots s_j$, and $z = s_{j+1}\ s_{j+2} \ldots s_k$

Observe

1. $|\ y\ | = j - i > 0$
2. $|\ x\ y\ | = j \leq p$
3. q_0 -x→ q_i -y→ q_j -z→ q_k, with $q_i = q_j$
   $\Rightarrow x\ y^k\ z \in L$, for k = 0, 1, 2, ...

□

Examples

Remember that the Pumping Lemma is typically used not to identify strings that belong to regular languages, but rather to identify languages which can not be regular!

L = { $\omega = 0^n 1^n$ | n ≥ 0 } =
{ ϵ, 01, 0011, 000111, 00001111, ... }

Assume that L is a regular language and that p is its pumping length.

Consider the word $\omega = 0^p 1^p \in$ L.
According to the Pumping Lemma, ω = x y z.

| y | > 0 and | x y | ≤ p implies that y = 0 ... 0 = 0^m, m ≥ 1.
So x y^2 z = $0^{m+p}1^p \in$ L. However, the two exponents are different and x y^2 z can not be a word in L. *A contradiction.*

Therefore L is not a regular language.

L = { $\omega \in$ {0,1}* | #0 = #1 } =
{ ϵ, 01,10, 0011, 0101, 0110, 1100, 1010, 1001, ... }

Assume that L is a regular language and that p is its pumping length.

Once again, consider the word $\omega = 0^p 1^p \in$ L.
According to the Pumping Lemma, ω = x y z.

| y | > 0 and | x y | ≤ p implies that y = 0 ... 0 = 0^m, m ≥ 1.
So x y^2 z = $0^{m+p}1^p \in$ L. However, #0 and #1 are different and x y^2 z can not be a word in L. *A contradiction.*

Therefore L is not a regular language.

$$\mathbf{L = \{\ \omega = 0^i 1^j \mid i > j \geq 0\ \} =}$$
$$\mathbf{\{\ 0,\ 00,\ 001,\ 000,\ 0001,\ 00011,\ \ldots\ \}}$$

Assume that L is a regular language and that p is its pumping length.

Consider the word $\omega = 0^{p+1}1^p \in$ L.
According to the Pumping Lemma, $\omega =$ x y z.

$|$ y $| > 0$ and $|$ x y $| \leq$ p implies that y $= 0 \ldots 0 = 0^m$, m ≥ 1.
So x z = x y^0 z $= 0^{p-m+1}1^p \in$ L. However, $\#1 \geq \#0$ and x y^0 z can not be a word in L. *A contradiction.*

Therefore L is not a regular language.

2.5 Applications

UNIX TOOLS: SED, GREP, and AWK

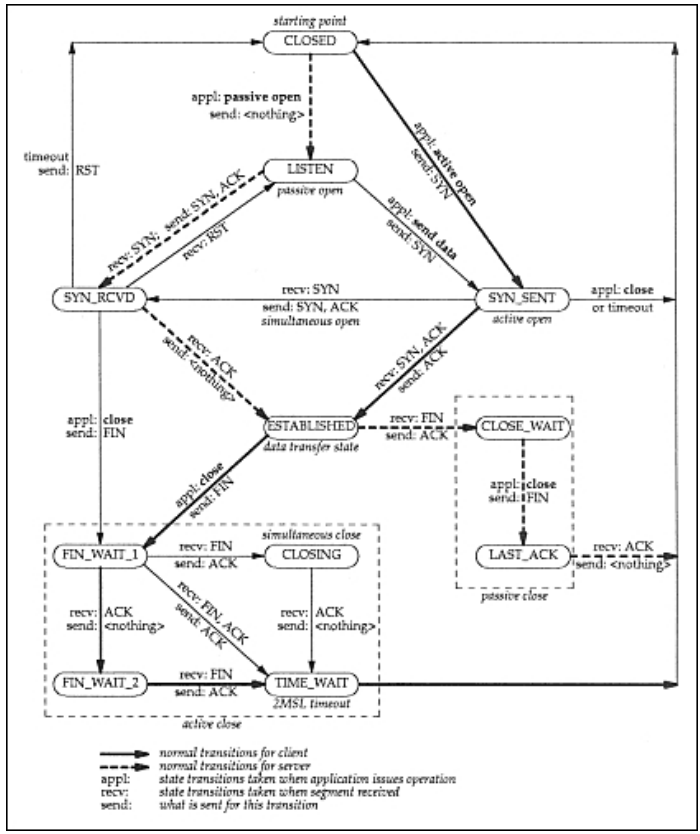
Sed, grep, and awk are true Unix tools, known for their awkward names and equally awkward syntax. **sed** is an abbreviation for *string editor*; **grep** comes from the **ed** command *g/re/p* (globally search a regular expression and print); and **awk** has its name derived from its authors (*Aho, Weinberger, and Kernighan*).

They represent the most immediate access to working with regular expressions. Even their most recent competitor, **Perl**, is known for producing very powerful but incomprehensible code. Though I acknowledge their awkward natures, their usefulness cannot be ignored, and learning how to use each will aid you in your ascension to line processing supremacy. Each is best used in the following manner:

grep	pattern matching
sed	replacement and line manipulation
awk	advanced line processing

TRANSITION DIAGRAMS FOR INTERNET PROTOCOLS

Transition diagrams are a useful graphical representation that can be used to describe internet protocols. The alphabet for these finite automata are the various types of packet being sent between computers.



LEXICAL ANALYSIS

Identifying the basic components of a programming language within an input stream of ASCII characters is an important first step in building a compiler for a programming language. This is the role of a **lexical analyzer**.

Definition 2.5.1. A **token** is a (syntactic element – value) pair; the value of a token is called a **lexeme**.

program source file fragment:

```
if (a > 0)
   then b = c + a
   else b = 7;
```

becomes the following token stream:

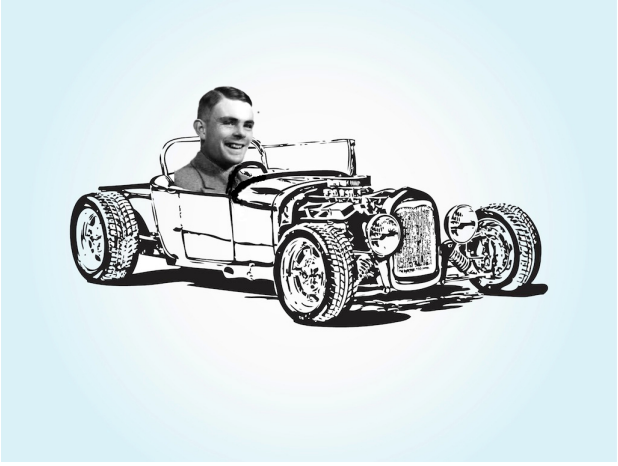
IF	if
LPAREN	(
ID	a
GT	>
INTEGER	0
RPAREN	)
THEN	then
ID	b
ASSIGN	=
ID	c
ADDOP	+
ID	a
ELSE	else
ID	b
ASSIGN	=
INTEGER	7
SEMICOLON	;

Token definitions are typically given by regular expressions!

INTEGER = {+,-,ϵ} {1,2,3,. . .,9} {0,1,2,3,. . .,9}*
IDENTIFIER = {A,. . ., Z,a,. . .,z} {A,. . .,Z,a,. . .,z,0,. . .,9}*
ADDOP = {+,-}
IF = if
THEN = then
ELSE = else

Chapter 3

Push Down Automata



3.1 Definitions

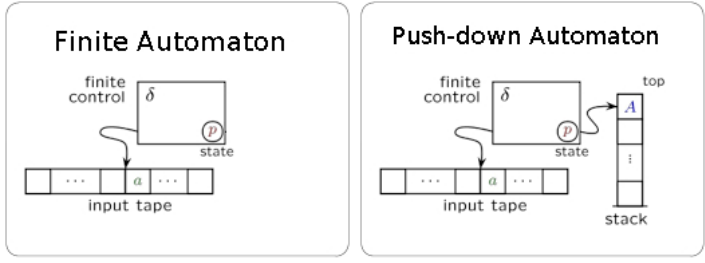
Finite automata are a nice category of machines – simple to understand, deterministic and nondeterministic versions are equivalent in power, and the languages recognized by them are closed under the six basic operations.

finite automaton ⇔ regular language

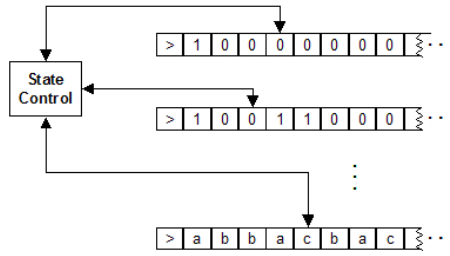
But not every language is a regular language! The Pumping Lemma demonstrated that non-regular languages exist.

The major flaw with finite automata is that they do not have any memory, any recall, any counting capability. Even the most recently encountered input symbol is immediately forgotten – the universe for a finite automaton is only the current state and the current input symbol!

So, what would be the next level of finite state machine? A machine capable of recall for recent events. Can we augment a finite automaton by providing a LIFO stack structure for short term memory? And can we provide the usual stack operations of push, pop, and top for the proper functioning of the stack?



This new machine augments the simplest Turing Machine with a stack (a second I/O device). This would be like a second tape on a Turing Machine. Which opens up a whole set of questions: if we add a second tape, what about a third tape ... and what about a fourth tape ...?



Multiple Tape/Head Turing Machines

Note: The Transitions function would have to be modified to account for the multiple tapes:

$$\delta\ (\ \mathbf{q}_i\ ,\ \mathbf{s}_1\ ,\ \mathbf{s}_2\ ,\ \ldots\ ,\ \mathbf{s}_n\ ) =$$
$$(\ \mathbf{q}_j\ ,\ \mathbf{s}_1\text{'}\ ,\ \mathbf{LRS}_1\ ,\ \mathbf{s}_2\text{'}\ ,\ \mathbf{LRS}_2\ ,\ \ldots\ ,\ \mathbf{s}_n\text{'}\ ,\ \mathbf{LRS}_n\ )$$

Each additional source of input/output would seemingly provide more options and possibly more power to our simple single tape Turing Machine.

But does it really?

Theorem 3.1.1. *All finite tape Turing Machines are equivalent in power!*

Proof. (finite tape machines)

($\Rightarrow$)

- a 1-tape TM is a 2-tape TM (which does not use the 2nd tape!)
- a 2-tape TM is a 3-tape TM (which does not use the 3rd tape!)
- a 3-tape TM is a 4-tape TM (which does not use the 4th tape!)

($\Leftarrow$)

a k-tape TM is equivalent to a 1-tape TM
(just interleave the tapes using k-cell groupings)

□

Definition 3.1.1. A **push down automaton** M is a 6-tuple (V, Σ, Γ, δ, q_0, F) where V, Σ, and Γ are finite sets and

1. V is a set of **vertices** or **states**
2. Σ is a set of **input symbols** or **alphabet**
3. Γ is a set of **stack symbols**
   $\Sigma \subseteq \Gamma$
   $s \in \Sigma$ is called a **terminal symbol**
   $s \in \Gamma \sim \Sigma$ is called a **nonterminal symbol**
4. δ : Q x Σ_ϵ x $\Gamma_\epsilon \rightarrow \wp$ (Q x Γ_ϵ)
5. $q_0 \in$ V is the **start state**
6. F $\subseteq$ V is a set of **accept states**

Comment. The above definition allows for choice in the definition of the transition function δ. Hence, the definition is for a *nondeterministic* push down automaton.

The transition function δ may be represented in a variety of ways.

δ (q , s , t) = (q_{next} , t'),
where s is input symbol,
t is *popped* from the stack,
t' is *pushed* onto the stack

For state diagrams found in many textbooks, an arrow ($\rightarrow$) is used to represent the pop / push combination:

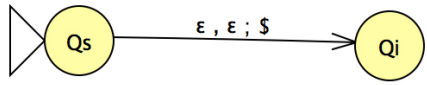
s , t $\rightarrow$ t'

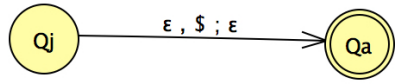
For state diagrams created in JFLAP, the arrow is typically replaced with a semicolon (;):

s , t ; t'

Any of the three symbols s, t, or t' may be replaced with ϵ, i.e., no input symbol considered, no pop off the stack, or no push onto the stack.

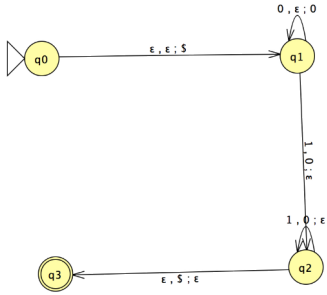
Furthermore, stacks typically have an empty or nonempty boolean function associated with them. Our examples incorporate the use a sentinel symbol ($) as part of the definition of the push down automaton instead.

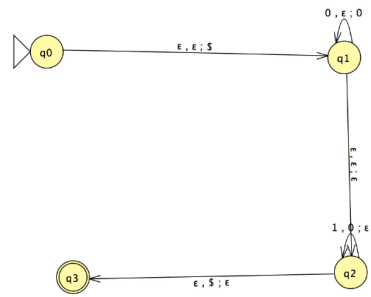




examples

The following two examples consider the language L = { $\omega \mid \omega = 0^k 1^k$ } which (in the last chapter) the Pumping Lemma verified is **not** a regular language. The example on the left considers L when k $\geq$ 1, i.e., the empty string is not in the language; the example on the right considers L when k $\geq$ 0, i.e., the empty string is in the language.





Observe the subtle difference between the two push down automata.

The only variance between the two is an ϵ-transition from state q_1 to state q_2 in the second. This is to allow acceptance of the empty string! The first PDA is deterministic – at each step of the way the machine knows exactly what it must do depending on whether the input is a 0 or a 1. The second PDA is nondeterministic – it must guess when it has seen all the 0s and begins looking for 1s.

The difference between determinism and nondeterminism in a push down automaton is that a *single stack* can not respond to multiple requests. Consider an old 45-rpm record player with spindle – what happens when five people attempt to play a record at the same time!

For push down automata, deterministic and nondeterministic versions are not equivalent in power. All of our push down automata in this chapter will be assumed nondeterministic!

Definition 3.1.2. We say that a push down automaton M **accepts** the word $\omega = w_1\ w_2 \ldots w_k$ with $w_j \in \Sigma$ provided:

there exists a sequence $r_0, r_1, \ldots, r_k$ V and a sequence $s_0, s_1, \ldots, s_k \in \Gamma^*$ of stack contents such that

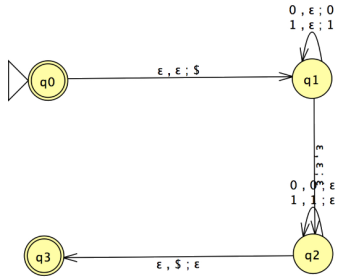
- $r_0 = q_0$, start state
- $(r_j , b) = \delta (r_{j-1}, w_j, a)$, j = 1, 2, k
  top (s_{j-1}) = a, i.e., pop a
  top (s_j) = b, i.e., push b
- $r_k \in F$, an accept state

Definition 3.1.3. The language **recognized** by a finite push down automaton M is the collection

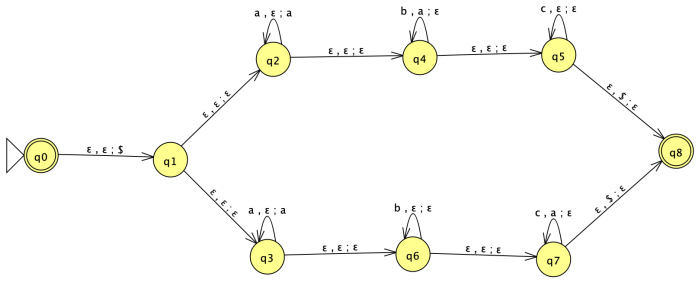
$$L (M) = \{ \text{ words } \omega \mid M \text{ accepts } \omega \}.$$

examples

L = { $\omega\omega^R$ | $\omega \in$ {0,1}* }



L = { $a^i b^j c^k$ | i, j, k $\geq$ 0 and ((i = j) or (i = k)) }



3.2 Context Free Grammars

In the last chapter we discussed finite automata and the languages that they recognized. As a result of our considerations we realized that these languages could also be recognized using pattern matching techniques and regular expressions.

So far in this chapter we have discussed nondeterministic push down automata and several examples of languages recognized by such machines. We are about to begin a search for an alternate way of characterizing the languages recognized by nondeterministic push down automata.

Definition 3.2.1. A **context free grammar** G is a four-tuple (Σ , V , S , P) where:

1. Σ is a finite nonempty set of **terminals** or **alphabet**
2. V is a finite nonempty set of **nonterminals** or **variables**
3. $S \in V$ is a distinguished nonterminal called the **start symbol**
4. P is a set of **production rules** of the form: $a \rightarrow b$
   $a \in V$
   $b \in (\Sigma \cup V)^*$

examples

$\mathbf{L = \{ (0\ 1)^*\ 1\ 1 \}}$

$S \rightarrow 0\ A$
$S \rightarrow 1\ C$
$A \rightarrow 1\ B$
$B \rightarrow 0\ A$
$B \rightarrow 1\ C$
$C \rightarrow 1$

$\mathbf{L = \{ 0^n 1^n \mid n \geq 0 \}}$

$S \rightarrow 0\ S\ 1$
$S \rightarrow \epsilon$

L = { simple arithmetic: expressions, terms, factors }

S → E
E → T
E → T + E
T → F
T → F * T
F → a
F → b
F → c
F → (E)

Note: JFLAP does not like ***parentheses*** ( ),
so JFLAP uses ***brackets*** [ ] instead!

L = { very simple English }

<sentence> → <subject> <predicate>
<subject> → <noun clause>
<predicate> → <verb clause>
<noun clause> → <noun> | <adjective> <noun>
<verb clause> → <verb> | <verb> <noun clause>
<noun> → alice | ice cream
<verb> → loves
<adjective> → cold

3.2.1 Important Grammar Concepts

Definition 3.2.2. A **sentential form** γ is any string of terminals and nonterminals, i.e., $\gamma \in (\Sigma \cup V)^*$.

Definition 3.2.3. We say that a sentential form γ_1 **directly derives** a second sentential form γ_2 provided there exists a production rule in P whose application converts γ_1 into γ_2, denoted symbolically

$$\gamma_1 \Rightarrow \gamma_2$$

Definition 3.2.4. We say that a sentential form γ_1 **derives** a second sentential form γ_2 provided there exists a finite sequence of sentential forms $s_0, s_1, \ldots, sk$ such that

$$\gamma_1 = s_0 \Rightarrow s_1 \Rightarrow \ldots \Rightarrow s_k = \gamma_2$$

denoted symbolically

$$\gamma_1 \Rightarrow^* \gamma_2$$

Definition 3.2.5. If $G = (\Sigma , V , S , P)$ is a context free grammar, then the language L **generated** by G is

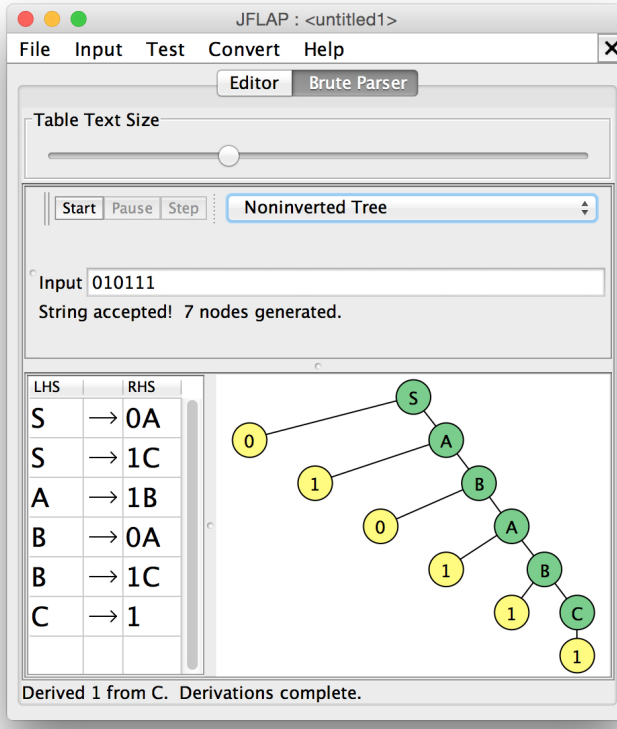
$$L = L (G) = \{ x \in \Sigma^* | \ S \Rightarrow^* x \}$$

Comment. The words generated by a regular expression R typically look quite a bit like words! The words generated by a context free grammar G sometimes look like words, but more often they look like sentences! Hence, the partial derivations of words in L (G) are referred to as ***sentential forms*** and the words in L (G) are also referred to as ***sentences***.

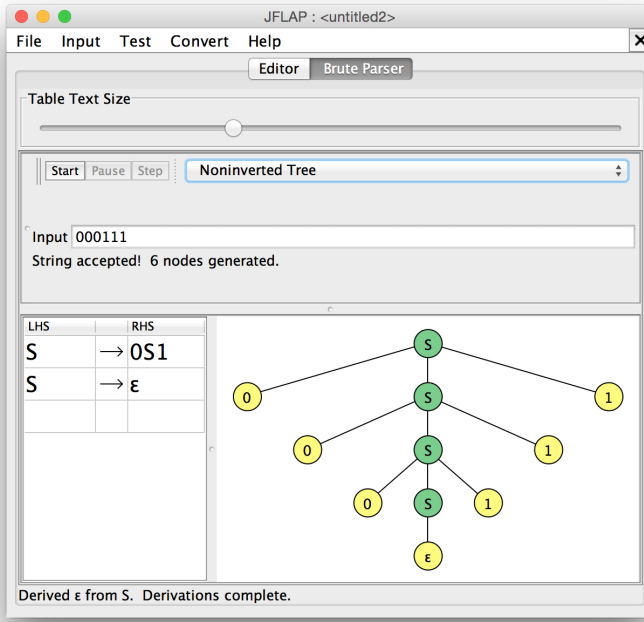
A derivation tree is a graphical representation of how the start symbol S derives a sentence. Some samples immediately follow.

examples

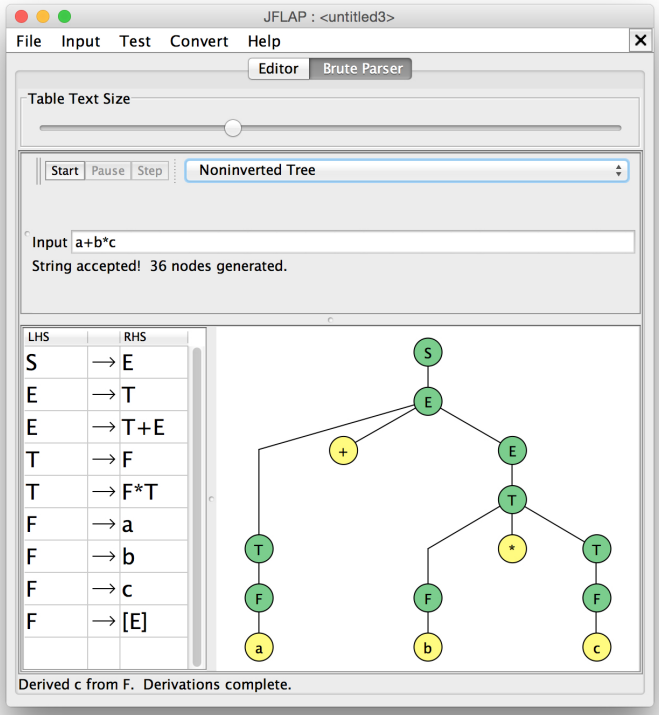
L = { (0 1)* 1 1 }
input: 010111



$\mathbf{L = \{\ 0^n 1^n \mid n \geq 0\ \}}$
input: 000111



L = { simple arithmetic }
input: a + b * c



Derivations typically involve some guesswork to identify which production rule to try at any given point. However, we can bring some method to the madness!

Definition 3.2.6. A **leftmost derivation** always uses a production rule in P for the leftmost variable found in a sentential form.

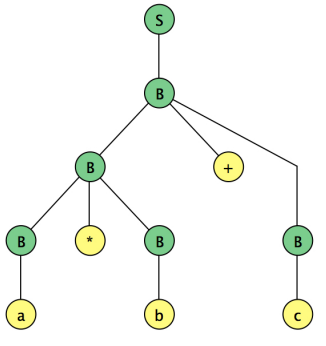
Definition 3.2.7. A **rightmost derivation** always uses a production rule in P for the rightmost variable found in a sentential form.

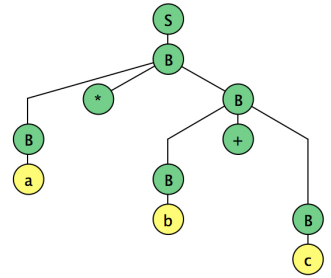
Comment. Multiple derivations of the same sentence often, but not always, yield the same derivation tree.

example

L = { simple arithmetic }
a variation on previous grammar!
input: a * b + c

S → B
B → B + B
B → B * B
B → (B)
B → a
B → b
B → c





Definition 3.2.8. A grammar G is said to be **ambiguous** if there exist at least two different derivation trees for the same sentence.

Grammars primarily focus on the **syntax** (structure) of a language, i.e., what is the proper form for a correct sentence. Grammars do not particularly concern themselves with the **semantics** (meaning) of a language, i.e., what is the precise understanding of the sentence.

However, syntax and semantics have a very fuzzy boundary:

- a grammar may be used to implicitly define the *priority* of mathematical operators
  exponentiation is highest
  multiplication and division are in the middle
  addition and subtraction are the lowest
- an ambiguous grammar may use semantics to clarify a preferred usage
  the *dangling else problem*
  is common to many programming languages

But this discussion is more appropriate for a text on programming languages and compiler construction where consideration of semantics and syntax are both essential. Our focus here is limited to the recognition of words / sentences in a particular formal language.

3.3 Chomsky Hierarchy of Grammars

In the last section we introduced the concept of a context free grammar. Recall that there is a hierarchy of formal languages, based on varying degrees of complexity. The general definition for a grammar (which follows below) is a slight generalization from the more specific definition of a *context free grammar* found in Section 3.2.

Definition 3.3.1. A **grammar** G is a four-tuple (Σ , V , S , P) where:

1. Σ is a finite nonempty set of **terminals** or **alphabet**
2. V is a finite nonempty set of **nonterminals** or **variables**
3. $S \in V$ is a distinguished nonterminal called the **start symbol**
4. P is a set of **production rules** of the form: $a \rightarrow b$
   $a \in (\Sigma \cup V)^* \; V \; (\Sigma \cup V)^*$
   $b \in (\Sigma \cup V)^*$

With this more general definition of a grammar, we can now subdivide grammars into four distinct categories – a hierarchy of grammars!

Hierarchy

Type 0 unrestricted grammar

production rules of the form: $a \rightarrow b$ as defined above
$a \in (\Sigma \cup V)^* \; V \; (\Sigma \cup V)^*$
$b \in (\Sigma \cup V)^*$

Type 1 context sensitive grammar

production rules of the form: $a \rightarrow b$
$| a | \leq | b |$
$S \rightarrow \epsilon$ allowed only if $S \notin$ right hand side of P
for any production rule in P

Type 2 context free grammar (as defined earlier)

production rules of the form: a $\rightarrow$ b
| a | = 1
left hand side of P is single nonterminal
for any production rule in P

Type 3 regular (linear) grammar

production rules of the form: a $\rightarrow$ b
| a | = 1
b is one of four types: ϵ, c, B, or c B

Comment. A regular grammar is automatically a context free grammar!

Theorem 3.3.1. *L is a regular language if and only if there is a regular grammar G so that L = L (G).*

Proof. (regular language)

($\Rightarrow$)

Suppose L is a regular language. Then there is a nondeterministic finite automaton M such that L (M) = L.

conversion algorithm NFA $\rightarrow$ RG

- states q_i become *nonterminals*
- symbols on transition arcs become *terminals*
- start state q_0 becomes *start symbol* S item accept states q_a become *production rules* of the form $q_a \rightarrow \epsilon$

($\Leftarrow$)

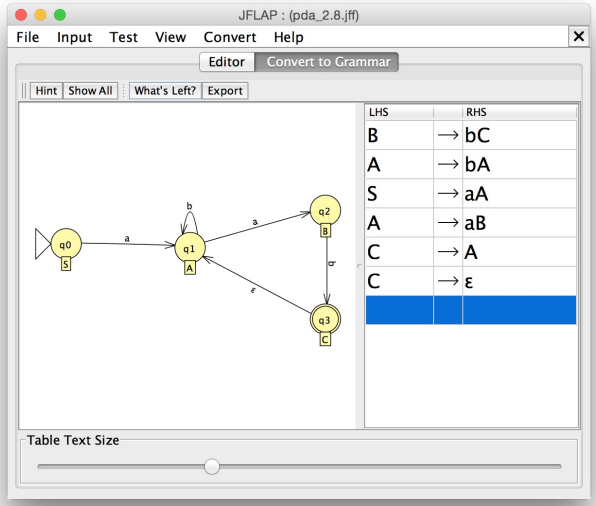
Suppose G is a regular grammar.

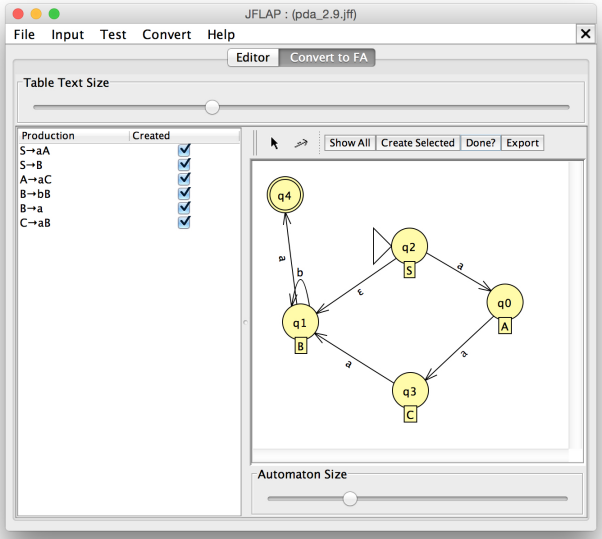
conversion algorithm RG $\rightarrow$ NFA

- nonterminals become *states*
- terminals become *symbols on transition arcs*
- start symbol S becomes *start state* q_0
- create a unique *accept state* q_a
- q_a is a destination state for *any* production rule which does not have a nonterminal on its right hand side

□

examples


JFLAP : (pda_2.8.jff)
File Input Test View Convert Help
Editor Convert to Grammar
Hint Show All What's Left? Export
LHS RHS
B → bC
A → bA
S → aA
A → aB
C → A
C → ε
q0 S
q1 A
q2 B
q3 C
Table Text Size


JFLAP : (pda_2.9.jff)
File Input Test Convert Help
Editor Convert to FA
Table Text Size
Production Created
S→aA
S→B
A→aC
B→bB
B→a
C→aB
Show All Create Selected Done? Export
q4
q2 S
q0 A
q1 B
q3 C
Automaton Size

3.3.1 Chomsky Normal Form

Theorem 3.3.2. *If L is a context free language, then L can be generated by a grammar in Chomsky Normal Form (CNF), i.e., every production rule is of the form:*

- $A \to B\ C$ *(two nonterminals, neither being S)*
- $A \to a$ *(single terminal)*
- *no* ϵ *rules, except possibly* $S \to \epsilon$

Proof. Suppose G is a context free grammar.

conversion algorithm CFG → CNF

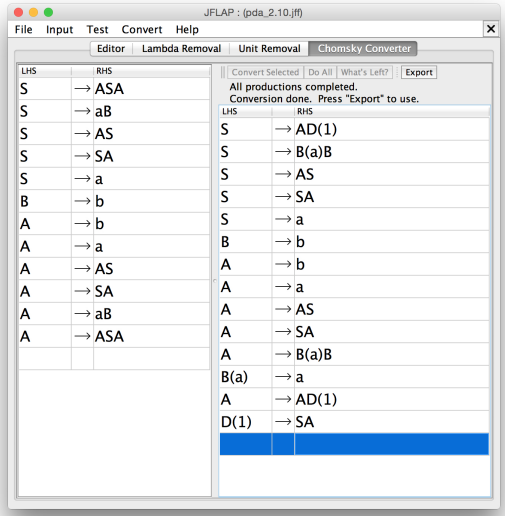
1. add a new start symbol S_0 with production rule $S_0 \to S$
2. focus on ϵ production rules, i.e., $A \to \epsilon$
   for every occurrence of A
   in the right hand side of a production rule,
   generate two new production rules:
   one with A in place and one with A removed
   repeat this step as necessary
3. focus on production rules $A \to B$
   for any production rule $B \to x$
   substitute $A \to x$
   repeat this step as necessary
4. focus on production rules $A \to B_1\ B_2 \ldots B_k$ with $k \geq 3$
   if B_j happens to be a terminal symbol b_j,
   replace it with a new nonterminal symbol
   and add the appropriate production $B_j \to b_j$
   replace the single production rule $A \to B_1\ B_2 \ldots B_k$
   with collection of simpler production rules

$$
\begin{aligned}
A &\to B_1\ C_1 \\
C_1 &\to B_2\ C_2 \\
C_2 &\to B_3\ C_3 \\
&\vdots \\
C_{k-2} &\to B_{k-1}\ B_k
\end{aligned}
$$

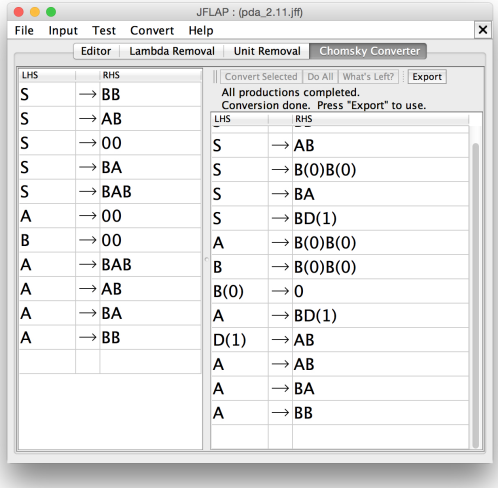
□

examples

CFG → CNF



CFG → CNF



3.4 CFG ≡ NPDA

In the last chapter we spent a great deal of time proving that the following statements are all equivalent to one another:

1. L is a regular language
2. L = L (M) where M is a deterministic finite automaton
3. L = L (M) where M is a nondeterministic finite automaton
4. L = L (M) where M is a generalized nondeterministic finite automaton
5. L = L (R) where R is a regular expression
6. L = L (G) where G is a regular grammar

The last entry above was part of our discussion in the preceding section. We also saw that regular grammars are a special case of context free grammars; both grammars have only one nonterminal on the left hand side of any production rule, but regular grammars have more restrictive limitations on the right hand side.

In this chapter we prove that the following four statements are all equivalent to one another:

1. L is a context free language
2. L = L (G) where G is a context free grammar
3. L = L (G) where G is a context free grammar in Chomsky Normal Form
4. L = L (M) where M is a nondeterministic push down automaton

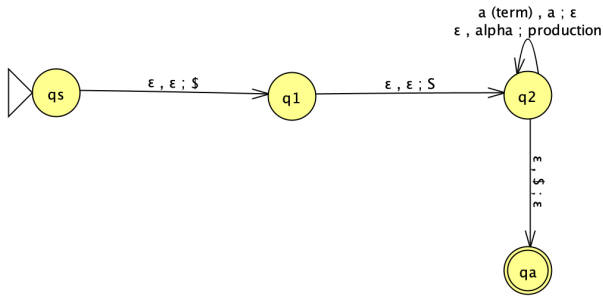
We arrive at these equivalencies by proving the following two lemmas.

Lemma 3.4.1. *If L = L (G) where G is a context free grammar, then there exists a nondeterministic push down automaton M such that L = L (G) = L (M).*

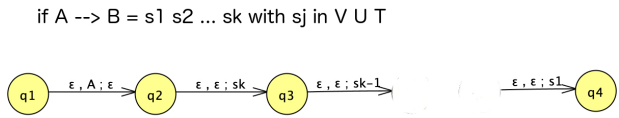
Proof. (G is context free grammar)

L = L (G) where G is a context free grammar having production rules
{ a → b | | a | = 1 }.

The corresponding nondeterministic push down automaton would be:



It is important that the symbols on the right hand side of the production rule be pushed onto the stack in reverse order. Symbol s_1 should be at the top of the stack when done.



The nondeterministic character of the push down automaton determines which is the correct production rule to apply at any given time. If a correct derivation exists, the process will recognize it!

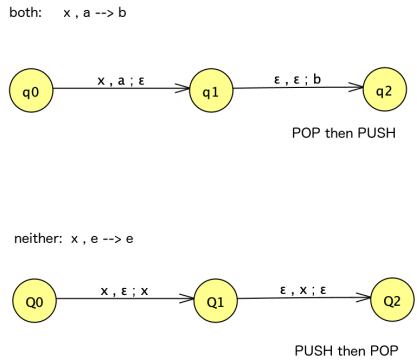
□

Lemma 3.4.2. *If $L = L\ (\ M\ )$ where M is a nondeterministic push down automaton, then there exists a context free grammar G such that $L = L\ (\ M\ ) = L\ (\ G\ )$.*

Proof. (M is a nondeterministic push down automaton)

We may assume that the nondeterministic push down automaton M satisfies the following three important properties:

1. there exists a single start state q_s
   there exists a single accept state q_a
2. NPDA inserts stack sentinel ($) when moving from start state q_s
   NPDA deletes stack sentinel ($) when moving into accept state q_a
3. every transition involves either a simple push or a simple pop



overall strategy for constructing the grammar

The variables V = { A_{pq} | p,q ∈ Q }, are based on **pairs of states** in M.

The start symbol is S = $A_{q_s q_a}$.

The production rules are constructed based on **pairs of states** in M which recognize various sentential forms:

1. ∀ p ∈ Q,
   add a production rule $A_{pp} \rightarrow \epsilon$ to grammar G
2. ∀ p, q, r ∈ Q,
   add a production rule $A_{pq} \rightarrow A_{pr}\ A_{rq}$ to grammar G
3. ∀ p, q, r, s ∈ Q, ∀ u ∈ Γ, and ∀ a, b ∈ Σ_ϵ with
   (r , u) ∈ δ (p , a , ϵ) and (q , ϵ) ∈ δ (s , b , u)
   add a production rule $A_{pq} \rightarrow$ a A_{rs} b to grammar G

More specifically, the variables A_{pq} serve as variables in G that represent transitions in M that move

- from a state p and an empty stack
- to a state q and once again an empty stack

or alternatively

- from a state p with a nonempty stack
- to a state q returning to the original stack contents

The first step in such a process must be a push
(upon reading input symbol a).
The last step in such a process must be a pop
(upon reading input symbol b).

Two possibilities may occur along the way:

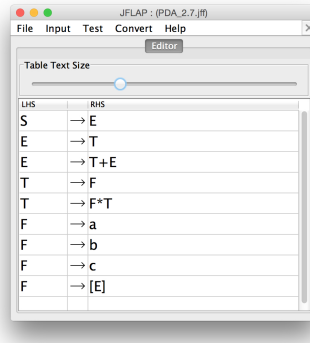
- the stack becomes empty at some intervening state r
  hence the production rule $A_{pq} \rightarrow A_{pr}\ A_{rq}$
- the stack remains nonempty throughout
  hence the production rule $A_{pq} \rightarrow$ a A_{rs} b

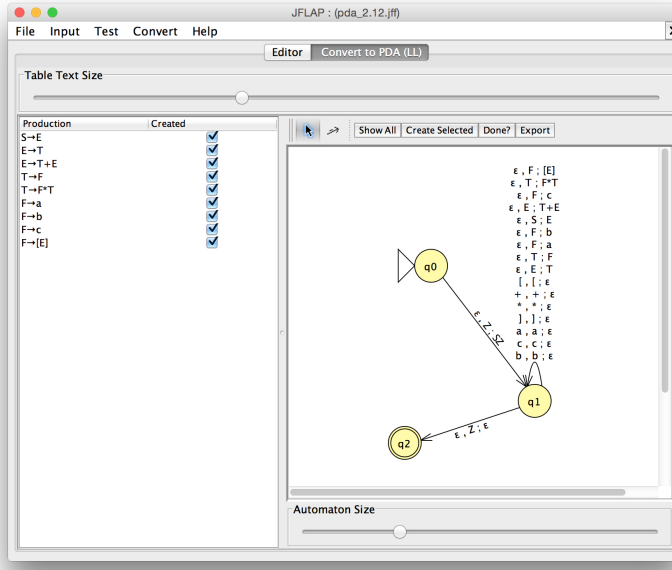
□

examples

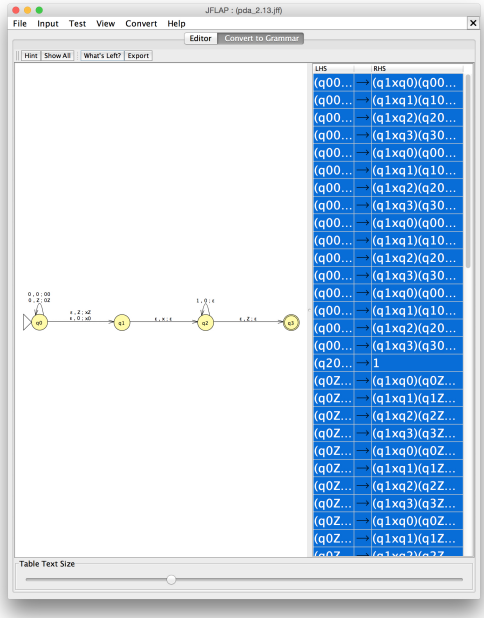
L = { simple arithmetic }

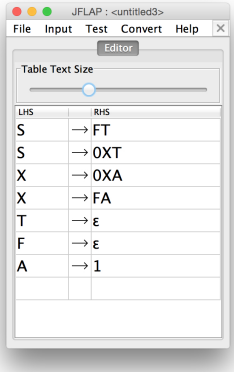
The nondeterministic feature of the push down automata under consideration make this conversion trivial!!





$$\mathbf{L = \{\ 0^n 1^n \mid n \geq 0\ \}}$$
$$\mathbf{G = \{\ S \rightarrow 0\ S\ 1 \mid \epsilon\ \}}$$





This algorithm likes to generate a lot of production rules!! Even after trimming what it believes are redundant rules, the end result is a grammar significantly larger than the original model for the push down automaton.

3.5 The Pumping Lemma

Push Down Automata and Context Free Languages have a parallel result to the pumping lemma for Finite Automata and Regular Languages.

Theorem 3.5.1. ***(The Pumping Lemma)***
If L is a context free language, then there exists an integer ***p*** *(called the* ***pumping length****) such that if w is any string in L with* $\mid \omega \mid \geq \mathbf{p}$ *then* ω *may be divided into five substrings* $\omega = x\ u\ y\ v\ z$ *with*

1. $\mid u\ v \mid > 0$, *either u is not* ϵ *or v is not* ϵ
2. $\mid u\ y\ v \mid \leq p$
3. $x\ u^k\ y\ v^k\ z \in L$ *for* $k = 0, 1, 2, \ldots$

Proof. L is a context free language

- there exists G, a context free grammar,
  such that L = L (G)
- there exists G', a context free grammar in CNF,
  such that L = L (G')
- a derivation tree for a string ω would be a binary tree!

If a binary tree T has a longest path of length k, then the number of terminal nodes must be less than or equal to 2^k.

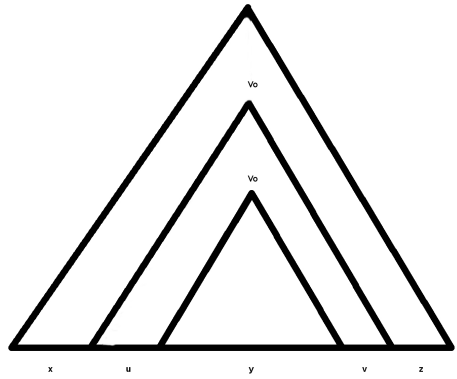
or equivalently

If a binary tree T has more than 2^k *terminal nodes, then the longest path has length greater than k.*

The grammar G'(in CNF) contains nonterminals (variables) V.
Define $p = 2^{|V|+1}$.

If $\mid \omega \mid \geq p = 2^{|V|+1}$, then the longest path in any derivation tree T for ω must be at least $\mid V \mid + 1$. The string ω may have several derivation trees T, so we choose the one having the *smallest number of nodes.*

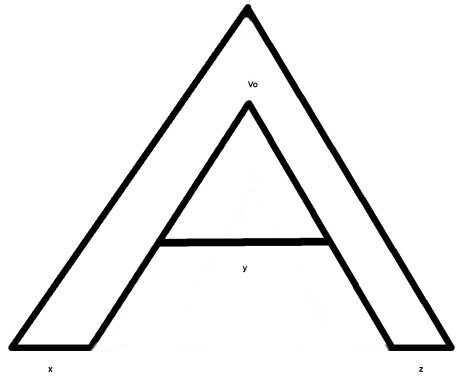
The number of nodes along the longest path in the derivation tree T for ω must be at least $\mid V \mid + 1$. Using the *pigeon hole principle* once again, there must be at least one repeated pair of nodes. Choose a repeated pair which is closest to the bottom (repeated node designated v_0).



Since the repeated pair of nodes are closest to the bottom of the derivation tree T, the subtree T anchored at the top v_0 must have its longest path length at most $|\ V\ | + 1$. Therefore

$$|\ u\ y\ v\ | \leq 2^{|V|+1} = p \qquad \text{(property \#2)}$$

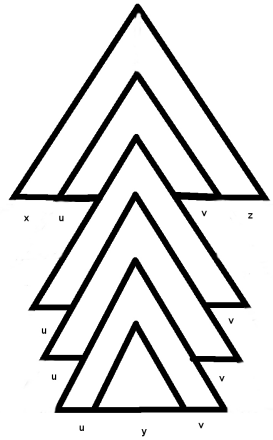
If $|\ u\ v\ | = 0$, i.e., both $u = \epsilon$ and $v = \epsilon$, then replacing the taller derivation tree T anchored at v_0 with the shorter derivation tree T anchored at v_0 yields a derivation tree for ω having a smaller number of nodes.



A contradiction! Therefore

$$|\ u\ v\ | > 0 \qquad \text{(property \#1)}$$

Lastly, at any occurrence of v_0, we may substitute the taller derivation tree for the shorter. Hence the following is possible. Therefore



$$x\ u^k\ y\ v^k\ z \in L\ ,\ \text{for } k = 0,\ 1,\ 2,\ \ldots \qquad \text{(property \#3)}$$

□

As we saw after the pumping lemma for regular languages, not every language is regular! Context free languages have pushed the envelope further and have provided additional tools for recognition of language categories. However, not every language is context free!

As before, the pumping lemma is more often used to prove that a given language is not context free than to help identify words within a context free language. We see this in the next two examples.

examples

L = { $\omega = a^n b^n c^n$ | n $\geq$ 0 } is not context free

Assume that L is a context free language and that p is its pumping length.

Consider the word $\omega = a^p b^p c^p \in L$.
According to the Pumping Lemma, ω = x u y v z.

| u v | > 0 and | u y v | $\leq$ p implies that
u y v may contain only a's and b's or may contain only b's and c's
but u y v may not contain a's and b's and c's!

Hence, x u^2 y v^2 z increases only two of the three exponents; the three exponents are no longer equal. x u^2 y v^2 z can not be a word in L. A contradiction.

Therefore L is not a context free language.

$\mathbf{L = \{\ \omega\ \omega \mid \omega \in \{0,1\}^*\ \}}$ is not context free

Assume that L is a context free language and that p is its pumping length.

Consider the word $\omega = 0^p1^p0^p1^p \in$ L.
According to the Pumping Lemma, ω = x u y v z.

| u v | > 0 and | u y v | ≤ p implies that
u y v must straddle the midpoint of the string ω

- if in left half, x u^2 y v^2 z would push 1s into front of resulting right half
- if in right half, x u^2 y v^2 z would push 0s into tail of resulting left half

If u y v straddles the midpoint of the string ω, 1s are on the left and 0s are on the right. Hence, x y z = x u^0 y v^0 z would have the form $0^p1^i0^j1^p$, with i < p and j < p. x y z can not be a word in L. A contradiction.

Therefore L is not a context free language.

3.6 Closure Properties

Recall that regular languages satisfy the closure property for all six basic operations on formal languages: union, intersection, complement, difference, concatenation, and Kleene closure.

Unfortunately, life is not so perfect with context free languages!

Lemma 3.6.1. *The following basic operations satisfy the closure property for context free languages:*

1. *union*
2. *concatenation*
3. *Kleene closure*

Proof. (closure properties)

Suppose S_1 is the start symbol for the grammar G_1 and S_2 is the start symbol for the grammar G_2.

The production rules for the union would be:

$G = G_1 \cup G_2$ augmented by the additional rule $S \rightarrow S_1 \mid S_2$

The production rules for the concatenation would be:

$G = G_1 \cup G_2$ augmented by the additional rule $S \rightarrow S_1\ S_2$

Suppose S_0 is the start symbol for the grammar G.

The production rules for the Kleene closure would be:

G augmented by the additional rule $S \rightarrow \epsilon \mid S_0\ S$

□

Lemma 3.6.2. *The following basic operations do not satisfy the closure property for context free languages:*

1. *intersection*
2. *complement*
3. *difference*

Proof. (failure of closure properties)

intersection fails to be closed

$L_1 = \{ a^i b^j c^k \mid i,j,k>0, i=j \}$ is a context free language
$L_2 = \{ a^i b^j c^k \mid i,j,k>0, j=k \}$ is context free language

But $L_1 \cap L_2 = \{ a^i b^j c^k \mid i,j,k>0, i=j=k \}$ is not context free language

complement fails to be closed

if complement were closed, then so would be intersection!
$L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$

difference fails to be closed

if difference were closed, then so would be intersection!
$L_1 \cap L_2 = L_1 \sim (L_1 \sim L_2)$

□

3.7 Applications

We saw earlier that regular expressions are an exceedingly useful tool in string manipulation and pattern matching. They are ideal for implementing low level aspects in compiler construction – lexical analysis or token recognition.

Context free grammars are likewise very valuable in the overall implementation of compilers – not merely with syntax issues but also with translation into executable code. However, we have seen that deterministic and nondeterministic flavors for push down automata are not equivalent in power.

Deterministic machines are generally faster; nondeterministic machines can be frustratingly slow. Courses like *Introduction to Programming Languages* and *Compiler Construction* seek to identify deterministic techniques which can be exploited to enhance performance. Two of the more common deterministic approaches include:

- LL (1) grammars
  left-to-right scan of input
  generating a left derivation tree
  using a one symbol look ahead
  top down approach
  easily implemented – recursive descent
- LR (1) grammars
  left-to-right scan of input
  generating a right derivation tree
  using a one symbol look ahead
  bottom up approach
  implemented using shift / reduction techniques

Since the details of this topic are best left to a course in compiler construction, we have simply highlighted the importance of the topic and return back to the study of formal languages. We are returning back to our original topic of Turing Machines, but we will view them in a new and different light.

We conclude this section by referencing a context free grammar for a scaled down version of the programming language *Pascal* by Niklaus Wirth.

Syntax of Mini-Pascal (Welsh & McKeag, 1980)

<program> **::=** **program** *<identifier>* **;** *<block>* **.**

<block> **::=** *<variable declaration part>*
<statement part>

<variable declaration part> **::=** *<empty>* |
var *<variable declaration>* **;**
{ *<variable declaration>* **;** }

<variable declaration> **::=** *<identifier >* { **,** *<identifier>* } **:** *<type>*

<type> **::=** *<simple type>* | *<array type>*

<array type> **::=** **array [** *<index range>* **] of** *<simple type>*

<index range> **::=** *<integer constant>* **..** *<integer constant>*

<simple type> **::=** **char | integer | boolean**

<type identifier> **::=** *<identifier>*

<statement part> **::=** *<compound statement>*

<compound statement> **::=** **begin** *<statement>*{ **;** *<statement>* } **end**

<statement> **::=** *<simple statement>* | *<structured statement>*

<simple statement> **::=** *<assignment statement>* | *<read statement>* | *<write statement>*

<assignment statement> **::=** *<variable>* **:=** *<expression>*

<read statement> **::=** **read (** *<variable>* { **,** *<variable>* } **)**

<write statement> **: : =** **write (** *<variable>* { **,** *<variable>* } **)**

<structured statement> **: : =** *<compound statement>* | *<if statement>* |
<while statement>

<if statement> **: : =** **if** *<expression>* **then** *<statement>* |
if *<expression>* **then** *<statement>* **else** *<statement>*

<while statement> **: : =** **while** *<expression>* **do** *<statement>*

<expression> **: : =** *<simple expression>* |
<simple expression> *<relational operator>* *<simple expression>*

<simple expression> **: : =** *<sign>* *<term>* { *<adding operator>* *<term>* }

<term> **: : =** *<factor>* { *<multiplying operator>* *<factor>* }

<factor> **: : =** *<variable>* | *<constant>* | (*<expression>*) | **not** *<factor>*

<relational operator> **: : =** **=** | **<>** | **<** | **<=** | **>=** | **>** | **or** | **and**

<sign> **: : =** + | − | *<empty>*

<adding operator> **: : =** + | −

<multiplying operator> **: : =** * | **div**

<variable> **: : =** *<entire variable>* | *<indexed variable>*

<indexed variable> **: : =** *<array variable>* **[** *<expression>* **]**

<array variable> **: : =** *<entire variable>*

<entire variable> **: : =** *<variable identifier>*

<*variable identifier*> **::=** <*identifier*>

Lexical grammar

<*constant*> **::=** <*integer constant*> | <*character constant*> | <*constant identifier*>

<*constant identifier*> **::=** <*identifier*>

<*identifier*> **::=** <*letter*> { <*letter or digit*> }

<*letter or digit*> **::=** <*letter*> | <*digit*>

<*integer constant*> **::=** <*digit*> { <*digit*> }

<*character constant*> **::=** '< letter or digit >' | "< letter or digit > {< letter or digit >}"

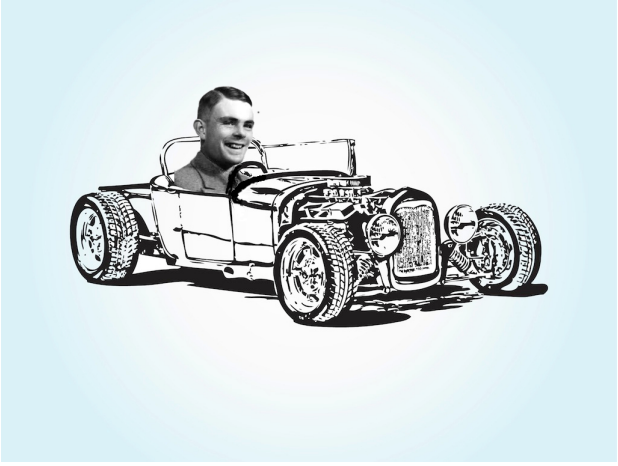
<*letter*> **::=** a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
p | q | r | s | t | u | v | w | x | y | z | A | B | C |
D | E | F | G | H | I | J | K | L | M | N | O | P
| Q | R | S | T | W | V | W | X | Y | Z

<*digit*> **::=** 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*special symbol*> **::=** + | - | * | = | <> | < | > | <= | >= |
(|) | [|] | **:=** | . | , | ; | : | .. | **div** | **or** | **and** | **not** | **if** | **then** | **else** | **of** |
while | **do** | **begin** | **end** | **read** | **write** | **var** | **array** | **function** |
procedure | **program** | **true** | **false** | **char** | **integer** | **boolean**

Chapter 4

Turing Machines Redux



4.1 Back to Square One

Our journey through formal languages and finite state machines returns back to the point where we started:

Turing Machines → finite automata →
push down automata → Turing Machines

But along the way we have encountered several options that allow for some variation:

- single input tape
- multiple tapes – one for input and one for stack storage
- deterministic finite state machines
- nondeterministic finite state machines

As a result we now have a variety of options available for our Turing Machines:

- single tape – both input and output
- single input tape and / or single output tape
- scratchpad tape
- deterministic / nondeterministic
- temporary storage – stack
- temporary storage – queue
  to hold configurations for nondeterministic machines

Do all these alternative allow for greater computational power over and above the capabilities of the original Turing Machine?

NO!

Theorem 4.1.1. *All these variations on the basic Turing Machine are equivalent in computational power to the original.*

Proof. (variations) All the above variations can be implemented by multiple tapes. Multiple tapes can be interleaved into a single tape!

□

What sort of questions or problems have finite state machines been able to address:

- generating binary expansions – infinite sequences of 0s and 1s
- computable numbers
- symbol manipulation – interpreting SDs for Turing Machines
- Universal Machine
- Turing Machines that generate a simple ACCEPT / REJECT response
- several categories of language recognition
- formal languages / Chomsky hierarchy

So now we return to this Chomsky hierarchy and its relationship to various Turing Machines. We also confront more directly the underlying quirk in Turing Machines – some of them go south (circular machines) on input data!

In this chapter we focus our attention on Turing Machines that utilize ACCEPT and REJECT states. Their purpose is to operate on input strings ω to determine whether or not that string is ACCEPTed or REJECTed.

Definition 4.1.1. We say that a language L is **accepter / recognized** by a Turing Machine M provided

$$L = L(M) = \{ \omega \mid M \text{ accepts } \omega \}.$$

The Turing Machine M is called an **acceptor** or a **recognizer**. The language L is called a **Turing-recognizable language** or a **recursively enumerable language**.

Comment. An acceptor / recognizer may not stop on some input!!!

Definition 4.1.2. We say that a language L is **decided** by a Turing Machine M provided

1. M halts for all input strings ω
2. L = L (M) = { ω | M accepts ω }.

The Turing Machine M is called **decider**. The language L is called a **Turing-decidable language** or a **recursive language**.

Since we have been very precise in the two definitions above, it seems likely that there should be a Turing-recognizable language that is not Turing-decidable! In fact we discussed one in the last section of Chapter One.

Sipser Question:
(appropriate to the modern theory of formal languages)

Given the Standard Descriptor (SD) for a Turing Machine and given an input string ω of symbols, can we decide whether or not the Turing Machine defined by SD will accept the input string ω?

In formal language theory, the Sipser Question is called an **acceptance problem**. We will shortly discuss several other categories of similar problems – some decidable and some not. But first we need to round out the theory of formal languages using the acceptance problem as an illustrative example.

Notation: We will use the notation <M> to represent the standard descriptor for a Turing Machine M.

Lemma 4.1.2. *The language* $\boldsymbol{ACCEPT}_{TM}$ *(<M>,ω) = { (<M>,ω) | Turing Machine M accepts string ω } is recursively enumerable, i.e., Turing-recognizable.*

Proof. (Turing-recognizable)

Define a Turing Machine $\mathfrak{R}$ which acts on input (<M>,ω) as follows:

1. simulate the Turing Machine M acting on input ω
2. if M ACCEPTs, then so does $\mathfrak{R}$

Comment. Note that $\mathfrak{R}$ need not stop on all input; hence $\mathfrak{R}$ is a recognizer.

□

Lemma 4.1.3. *The language* $\boldsymbol{ACCEPT}_{TM}$ *(<M>,ω) = { (<M>,ω) | Turing Machine M accepts string ω } is not recursive, i.e., Turing-decidable.*

Proof. (not Turing-decidable)

Please refer to the last section of Chapter One. If the acceptance problem were in fact decidable, then a decider for the problem creates an interesting paradox!

□

We will now digress briefly to present some additional theory regarding formal languages.

Theorem 4.1.4. *A language L is recursive if and only if both languages L and L^C are recursively enumerable.*

Proof. (recursive languages)

($\Rightarrow$)

Suppose M is a decider for the language L, i.e., L = L (M). Since M is a decider it is also an acceptor (which always halts)! Hence, the language L is recursively enumerable.

Define a new Turing Machine M^C which is identical in its structure to the Turing Machine M except that the ACCEPT states and the REJECT states are reversed. Since M is a decider for L, M^C is also a decider (and an acceptor), but for the language L^C! Hence, the language L^C is recursively enumerable.

($\Leftarrow$)

Suppose M_1 is an acceptor for the language L, i.e., L = L (M_1). Suppose M_2 is an acceptor for the language L^C, i.e., L^C = L (M_2).

Define a new Turing Machine M which steps through an input string ω utilizing both M_1 and M_2 in parallel. Since $L \cup L^C$ exhaust all the possibilities, either M_1 or M_2 must ACCEPT.

if M_1 ACCEPTs, then M ACCEPTs if M_2 ACCEPTs, then M REJECTs

The Turing Machine M is a decider for the language L = L (M) = L (M_1). Hence the language L is recursive.

□

Summary of Possibilities

- If L **is** recursively enumerable and L^C **is** recursively enumerable,
  then L is called ***recursive***
- If L **is** recursively enumerable and L^C **is not** recursively enumerable,
  then L is called ***recursively enumerable***
- If L **is not** recursively enumerable and L^C **is** recursively enumerable,
  then L is called ***co-recursively enumerable***
- If L **is not** recursively enumerable and L^C **is not** recursively enumerable,
  then L is called ***just plain weird***!

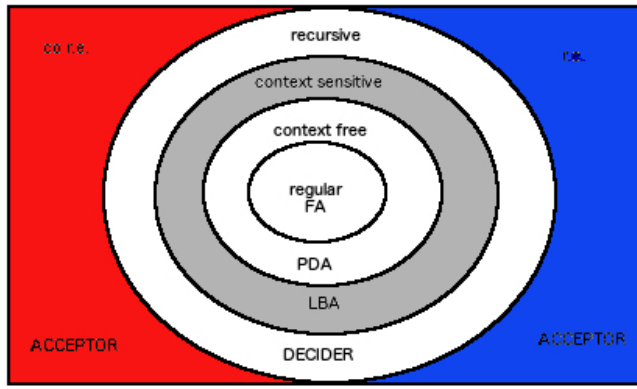
Lemma 4.1.5. *The language* $\boldsymbol{ACCEPT}_{TM}$ *(<M>,ω) =* { *(<M>,ω) | Turing Machine M accepts string ω* } *is not co-recursively enumerable.*

Proof. (not co-recursively enumerable)

We already know $\mathbf{ACCEPT}_{TM}$ (<M>,ω) is recursively enumerable. If $\mathbf{ACCEPT}_{TM}$ (<M>,ω) were co-recursively enumerable, then by the theorem above $\mathbf{ACCEPT}_{TM}$ (<M>,ω) would be recursive, i.e., decidable. A contradiction!

□

The diagram below illustrates the relationship between all the various finite state machines and the grammars / formal languages that they recognize. Included in the diagram are context sensitive grammars which are found between context free grammars and unrestricted grammars in the Chomsky Hierarchy.



The following properties for context sensitive grammars are stated ***without proof***:

Definition 4.1.3. A **linear bounded automaton** M is a Turing Machine where the length of the output may not exceed the size of the input.

Theorem 4.1.6. *The following three statements are all equivalent to one another:*

1. *L is a context sensitive language*
2. *L = L (G) where G is a context sensitive grammar*
3. *L = L (M) where M is a linear bounded automaton*

Theorem 4.1.7. *If a Turing Machine M is a linear bounded automaton, then M is a decider.*

Corollary. *If a language L is a context sensitive language, then language L is a recursive language.*

Consider for a moment a fixed finite alphabet Σ.

Lemma 4.1.8. *The set of all words over the alphabet* Σ *is a countably infinite set,* Σ^*.

Proof. (uncountable) Let $\Sigma_n = \{\ \omega$ over $\Sigma \mid |\omega| = \text{n}\ \}$. Σ_n is a finite set. $\Sigma^* = \cup\ \Sigma_n$ is therefore a countably infinite set.

□

Lemma 4.1.9. *The set* $\wp\ (\ \Sigma^*\ )$ *is an uncountable set.*

Proof. Σ^* is a countable set.
Thus we can list it in a sequence! $\{\ \omega_1, \omega_2, \omega_3\ , \ldots\ \}$

Any set S in the power set $(\ \Sigma^*\ )$ can be represented by its characteristic function:

$$\text{X}_S\ (\text{k}) = 1 \text{ if } \omega_k \in \text{S or } 0 \text{ if } \omega_k \notin \text{S}$$

For example, a characteristic function of all 0's represents the empty set $\emptyset$ and a characteristic function of all 1's represents the entire set Σ^*.

Let us suppose $\wp\ (\ \Sigma^*\ )$ is a countable set. Then we can list the subsets (i.e., their characteristic functions) in a sequence:

$$\begin{aligned}
\text{X}_1 &= \mathbf{c_{11}} c_{12} c_{13} \ldots \\
\text{X}_2 &= c_{21} \mathbf{c_{22}} c_{23} \ldots \\
\text{X}_3 &= c_{31} c_{32} \mathbf{c_{33}} \ldots \\
&\vdots
\end{aligned}$$

Define a new subset (i.e., a new characteristic function) by

$$\text{X}_S\ (\text{k}) = 1 \text{ - } c_{kk}$$

X_S is not found in the listing above. A contradiction.

□

In our diagram of the Chomsky Hierarchy above, the weird languages were those outside the boundaries, that is, they were neither recursively enumerable (L recognized by an acceptor) nor co recursively enumerable (L^C recognized by an acceptor).

We conclude this section with the following corollary to our work above.

Corollary. *There exists a language $L_W \in \wp\ (\ \Sigma^*\ )$ that is neither recursively enumerable nor co recursively enumerable.*

Proof. The set $\wp\ (\ \Sigma^*\ )$ is an uncountable set while the collection of Turing Machines is only a countable set!

□

4.2 Closure Properties & Turing Machines

Lemma 4.2.1. *Turing decidable (recursive) languages* ***are closed*** *under:*

- *union*
- *intersection*
- *set difference*
- *complement*
- *concatenation*
- *Kleene closure*

Turing recognizable (recursively enumerable) languages ***are closed*** *under:*

- *union*
- *intersection*
- *concatenation*
- *Kleene closure*

but ***are not closed*** *under:*

- *set difference*
- *complement*

Proof. (closure properties)

(union and intersection)

Let M_1 be the decider or the recognizer for language L_1; let M_2 be the decider or the recognizer for language L_2. Run both machines in parallel on the string ω.

> For union, if either machine ACCEPTs, then ACCEPT.
> For intersection, if both machines ACCEPT, then ACCEPT.

(set difference and complement)

If M is a decider for language L, then M can be easily modified into M^C a decider for language L^C by interchanging the ACCEPT and REJECT states.

However, if a language L is ***only*** Turing recognizable and not Turing decidable, then L^C can not also be Turing recognizable or that would imply that L is Turing decidable by our work in the previous section.

Since both Turing decidable languages and Turing recognizable languages are closed under intersection, closure under complement is equivalent to closure under set difference.

$$L^C = \Sigma^* \sim L \qquad \text{and} \qquad L_1 \sim L_2 = L_1 \cap L_2^C$$

(concatenation and Kleene closure)

Let M_1 be the decider or the recognizer for language L_1; let M_2 be the decider or the recognizer for language L_2.

For an input string ω, for each of the $|\omega| + 1$ way to divide $\omega = x\ y$, run M_1 on x and M_2 on y.

If M_1 ACCEPTs x and M_2 ACCEPTs y, then ACCEPT.

Let M be the decider or the recognizer for language L.

For an input string ω, if $\omega = \epsilon$ then ACCEPT.

Otherwise, for each of the $2^{|\omega|+1}$ ways to divide $\omega = x_1 \ldots x_k$ ($x_i \neq \epsilon$),
and run M on x_i for each i.

If M ACCEPTs x_i for all i, then ACCEPT.

□

SUMMARY
Closure Properties

hierarchy	grammar	machine	union	intersect	concat	Kleene	diff	comp
3	regular	FA	yes	yes	yes	yes	yes	yes
2	context free	NPDA	yes	NO	yes	yes	NO	NO
1	context sensitive	LBA	yes	yes	yes	yes	yes	yes
	recursive / decidable	TM accept/reject	yes	yes	yes	yes	yes	yes
0	recursively enumerable	TM accept/reject/loop	yes	yes	yes	yes	NO	NO

4.3 Decidability / Undecidability

We now discuss the question of decidability in more detail than we did in our previous discussions at the end of Chapter One and at the beginning of Chapter Four.

For the remainder of this chapter we discuss problems in the context of recognizing various properties of formal languages based on their ***representations***:

regular languages	as represented by their finite automaton (DFA)
context free languages	as represented by their grammar (CFG)
other languages	as represented by a Turing Machine (TM)

and also in the context of the following ***four categories of questions***:

ACCEPT (<M>,ω)	does M accept ω?
EMPTY (<M>)	is L (M) = $\emptyset$?
EQUAL (<M_1>,<M_2>)	is L (M_1) = L (M_2)?
ALL (<M>)	is L (M) = Σ^*?

With three categories of formal languages and four basic questions possible for each category, we arrive at twelve fundamental questions:

QUESTION	DFA	CFG	TM
ACCEPT	YES	YES	NO
EMPTY	YES	YES	NO
EQUAL	YES	NO	NO
ALL	YES	NO	NO

ACCEPT (<M> , ω)

ACCEPT$_{TM}$ (<M> , ω) is not decidable.

We have already discussed this problem in Chapter One and then again in the first section of this chapter. We will now consider the acceptance problem for two other categories: regular languages and context free languages.

ACCEPT$_{DFA}$ (<M> , ω) is decidable.

Define the Turing Machine $\mathfrak{D}$ to run on input (<M>,ω) by simulating the deterministic finite automaton M on the string ω.

> If M ACCEPTs ω, then $\mathfrak{D}$ ACCEPTs;
> if M REJECTs ω, then $\mathfrak{D}$ REJECTs.

Unlike Turing Machines which may loop indefinitely, deterministic finite automata always halt (when the input string ω is exhausted). Hence $\mathfrak{D}$ is a decider.

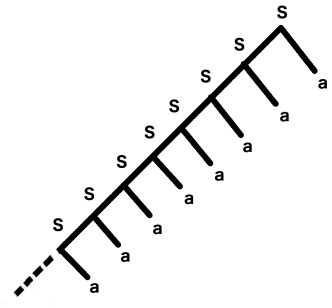
ACCEPT$_{CFG}$ (<G> , ω) is decidable.

Greater care must be taken with derivation trees for an arbitrary grammar, however. Consider the following simple grammar G

$S \rightarrow S\ a \mid \epsilon$

$L\ (G) = \{\ a^n \mid n \geq 0\ \}$

Even though the grammar is relatively simple, its derivation trees may get unboundedly large in size. Hence, brute force methods would be an inappropriate strategy.



Recall, every context free grammar G has an equivalent Chomsky Normal Form grammar G' and when working with a CNF grammar G' if an input string ω has $|\omega| = n$ then its derivation requires exactly $2n - 1$ steps.

Define the Turing Machine $\mathfrak{D}$ to run on input (<G>,ω) as follows:

1. $\mathfrak{D}$ first converts G to an equivalent grammar G' in CNF
2. $\mathfrak{D}$ then generates all derivation trees $\boldsymbol{t}$ having less than or equal $2n - 1$ steps
   for each $\boldsymbol{t}$, test $\boldsymbol{t} = \omega$
   if true, then ACCEPT
3. REJECT

EMPTY (<M>)

***EMPTY**$_{DFA}$ (<M>) is decidable.*

In order for a regular language to be nonempty there must exist at least one path from the start state q_s to an accept state q_a. Define the Turing Machine $\mathfrak{D}$ to run on input <M> as follows:

1. $\mathfrak{D}$ marks q_s as **reachable**
2. repeat until no new states are marked
   if q_i is reachable and there is a transition from q_i to q_j,
   then mark q_j as reachable
3. if reachable states $\cap$ accept states $= \emptyset$, then ACCEPT else REJECT

The fact that Q contains only a finite number of states guarantees that step 2 will terminate. Hence, $\mathfrak{D}$ is a decider.

***EMPTY**$_{CFG}$ (<G>) is decidable.*

In order for a context free language to be nonempty there must exist at least one derivation rooted at the start symbol S.

Define the Turing Machine $\mathfrak{D}$ to run on input <G> as follows:

1. $\mathfrak{D}$ marks all the terminal symbols as **productive**
2. repeat until no new nonterminal symbols are marked
   if $A \rightarrow s_1\ s_2, \ldots, s_k$ is a production
   and all the symbols s_i are marked as productive,
   then mark the nonterminal A as productive
3. if the start symbol S is marked as productive, then REJECT else ACCEPT

The fact that there are only a finite number of nonterminal symbols guarantees that step 2 will terminate. Hence $\mathfrak{D}$ is a decider.

***EMPTY**$_{TM}$ (<M>) is not decidable.*

Suppose there is a Turing Machine $\mathfrak{D}$ which is a decider for the problem **EMPTY**$_{TM}$. We will use this Turing Machine $\mathfrak{D}$ to create a decider for the problem **ACCEPT**$_{TM}$!

1. Define a Turing Machine M_ω having as its input an arbitrary string $\boldsymbol{t}$
   if $\boldsymbol{t} \neq \omega$, then REJECT
   if $\boldsymbol{t} = \omega$, simulate Turing Machine M (input to $\mathbf{EMPTY}_{TM}$) on ω
   if M ACCEPTs ω, then ACCEPT
2. Define the Turing Machine $\mathfrak{D}$' having as input (<M>,ω)
   use <M> and ω to create the encoding for <M_w>
   simulate $\mathfrak{D}$ on <M_ω>
   if $\mathfrak{D}$ ACCEPTs, then REJECT else ACCEPT

Note that $\mathfrak{D}$ halts on all input strings; hence $\mathfrak{D}$' halts on all input strings! Furthermore, $\mathfrak{D}$' ACCEPTs (<M>,ω) if and only if $\mathfrak{D}$ REJECTS (<M_ω>). We have created a decider for $\mathbf{ACCEPT}_{TM}$. A contradiction.

Please note the technique in this last demonstration. We have morphed our original problem $\mathbf{EMPTY}_{TM}$ into a previously solved problem $\mathbf{ACCEPT}_{TM}$. Remember this fact! We shall see it again shortly.

EQUAL (<M_1, M_2>)

***EQUAL**$_{DFA}$ (<M_1> , <M_2>) is decidable.*

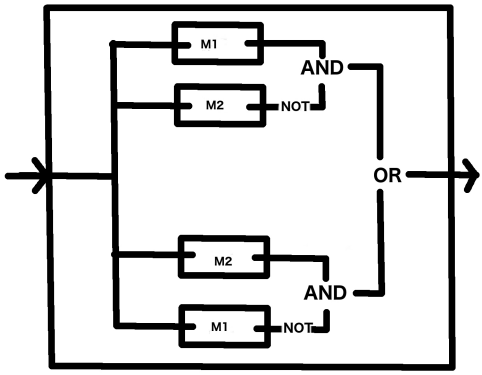
For any two formal languages L_1 and L_2 we have:

$$L_1 = L_2 \text{ if and only if } (L_1 \cap L_2^C) \cup (L_1^C \cap L_2) = \emptyset$$

Suppose M_1 is a deterministic finite automaton which recognizes the language L_1 and suppose M_2 is a deterministic finite automaton which recognizes the language L_2. Define the deterministic finite automaton M as follows:

Let $\mathfrak{D}$ be a decider for **EMPTY**$_{DFA}$. Define $\mathfrak{D}$' for input (<M_1>,<M_2>) as follows:

1. $\mathfrak{D}$' determines the encoding <M> for the deterministic finite automaton pictured below
2. simulate $\mathfrak{D}$ on <M>
   if $\mathfrak{D}$ ACCEPTs, then ACCEPT else REJECT



Since $\mathfrak{D}$ stops on all input with either ACCEPT or REJECT, so does $\mathfrak{D}$'.

***EQUAL**$_{CFG}$ (<**G**$_1$> , <**G**$_2$>) is not decidable!*

But, you might ask, would not a similar argument to the one presented above work here as well?

The answer is NO . . . because context free languages are **not closed** under intersection and **not closed** under complement.

We will have to investigate this further. And we postpone that discussion until later!

***EQUAL**$_{TM}$ (<**M**$_1$> , <**M**$_2$>) is not decidable!*

We should have been expecting this – nothing has been decidable with Turing Machines to date.

But suppose **EQUAL**$_{TM}$ is decidable and Turing Machine $\mathfrak{D}$ is a decider for **EQUAL**$_{TM}$. Let <M> be the encoding for an arbitrary Turing Machine, and let <M$_\emptyset$> be the encoding for a Turing Machine with L (M$_\emptyset$) = $\emptyset$. Note: M$_\emptyset$ rejects EVERYTHING!

Define $\mathfrak{D}$' for input (<M>) as follows:

1. $\mathfrak{D}$' determines the encoding <M$_\emptyset$>
2. simulate $\mathfrak{D}$ on (<M>,<M$_\emptyset$>)
   if $\mathfrak{D}$ ACCEPTs, then ACCEPT else REJECT

Since $\mathfrak{D}$ stops on all input with either ACCEPT or REJECT, so does $\mathfrak{D}$'. $\mathfrak{D}$' is a decider for **EMPTY**$_{TM}$. A contradiction.

Please note the technique in this last demonstration. We have morphed our original problem **EQUAL**$_{TM}$ into a previously solved problem **EMPTY**$_{TM}$. Remember this fact! We shall see it again shortly.

ALL (<M>)

***ALL**$_{DFA}$ (<M>) is decidable.*

You should try your hand at this problem. Consider using the decider $\mathfrak{D}$ for either **EMPTY**$_{DFA}$ or **EQUAL**$_{DFA}$ as part of your solution.

***ALL**$_{CFG}$ (<G>) is not decidable.*

You might be a bit unsure about which way this result is going to turn out.

We postpone the verification of this statement until later!

***ALL**$_{TM}$ (<M>) is not decidable.*

You should no longer be at all surprised at this result! Turing Machines have consistently failed to be decidable!

We postpone the verification of this statement until later!

HALT

***HALT**$_{TM}$ (<M>)*
= { <M> | M is a Turing Machine
that will HALT on any input }
is not decidable.

Suppose **HALT**$_{TM}$ is decidable by a Turing Machine $\mathfrak{D}$. Define a Turing Machine $\mathfrak{D}$' for the problem **ACCEPT**$_{TM}$ on (<M>,ω) as follows:

1. $\mathfrak{D}$' simulates $\mathfrak{D}$ on input <M>
   if $\mathfrak{D}$ REJECTs <M>, then $\mathfrak{D}$' REJECTs
2. $\mathfrak{D}$' simulates M on ω
   if <M> ACCEPTs ω, then $\mathfrak{D}$' ACCEPTs else $\mathfrak{D}$' REJECTs

Since $\mathfrak{D}$ stops on all input with either ACCEPT or REJECT, so does $\mathfrak{D}$'. $\mathfrak{D}$' is a decider for **ACCEPT**$_{TM}$. A contradiction.

Please note the technique in this last demonstration. We have once again morphed our original problem **HALT**$_{TM}$ into a previously solved problem **ACCEPT**$_{TM}$. Remember this fact! We shall see it again shortly.

We have completed our first pass at the original dozen decidability questions we proposed, plus we also analyzed the Halting Problem to make it a baker's dozen. However, there still remain some gaps in our analysis which we need to address. Our focus in the coming sections will be on a technique called ***reduction*** – converting a new problem into a previous problem which we have already solved.

4.4 Reducibility

Recall that in the last section we highlighted that several of our verifications to solve the stated problem *morphed* into another previously solved problem.

$\mathbf{EMPTY}_{TM}$ morphed into $\mathbf{ACCEPT}_{TM}$
$\mathbf{EQUAL}_{TM}$ morphed into $\mathbf{EMPTY}_{TM}$
$\mathbf{HALT}_{TM}$ morphed into $\mathbf{ACCEPT}_{TM}$

However, the word *morph* is not a proper mathematical term! So let us make it one with some new definitions.

Definition 4.4.1. We say that $f : \Sigma^* \to \Sigma^*$ is a **computable function** provided there exists a Turing Machine M such that

- M halts on all input ω and
- the resultant output is $f(\omega)$

Definition 4.4.2. We say that a language A is **mapping reducible** to a language B, denoted $A \leq B$, provided there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that

$\omega \in A$ if and only if $f(\omega) \in B$

The computable function f is called a **reduction** from A to B.

examples

1. $\mathbf{ACCEPT}_{TM} \leq \mathbf{EMPTY}_{TM}$
2. $\mathbf{EMPTY}_{TM} \leq \mathbf{EQUAL}_{TM}$
3. $\mathbf{ACCEPT}_{TM} \leq \mathbf{HALT}_{TM}$

Theorem 4.4.1. *(Properties of Reducibility)*

1. *$A \leq B$ if and only if $A^C \leq B^C$.*
2. *If $A \leq B$ and $B \leq C$, then $A \leq C$.*
3. *If $A \leq B$ and B is decidable (recursive), then A is decidable (recursive).*
4. *If $A \leq B$ and B is Turing recognizable (recursively enumerable), then A is Turing recognizable (recursively enumerable).*
5. *If $A \leq B$ and A is not decidable, then B is not decidable.*
6. *If $A \leq B$ and A is not Turing recognizable, then B is not Turing recognizable.*

Proof. Six components:

1. $\mathrm{A} \leq \mathrm{B} \Leftrightarrow$ there exists f s/t $\omega \in \mathrm{A}$ if and only if $f(\omega) \in \mathrm{B}$
   $\Leftrightarrow$ there exists f s/t $\omega \in \mathrm{A}^C$ if and only if $f(\omega) \in \mathrm{B}^C$
   $\Leftrightarrow \mathrm{A}^C \leq \mathrm{B}^C$
2. $\mathrm{A} \leq \mathrm{B}$ and $\mathrm{B} \leq \mathrm{C}$
   $\Rightarrow$ there exists f s/t $\omega \in \mathrm{A}$ if and only if $f(\omega) \in \mathrm{B}$
   and there exists g stt $\omega \in \mathrm{B}$ if and only if $g(\omega) \in \mathrm{C}$
   $\Rightarrow \omega \in \mathrm{A}$ if and only if $g \circ f(\omega) in$ C
   $\Rightarrow \mathrm{A} \leq \mathrm{C}$
3. together with
4. $\mathrm{A} \leq \mathrm{B} \Rightarrow$ there exists f stt $\omega \in \mathrm{A}$ if and only if $f(\omega) \in \mathrm{B}$.
   Suppose B is decided or recognized by the Turing Machine $\mathfrak{X}$.
   Define the Turing Machine $\mathfrak{X}$' on A as follows:
   $$\mathfrak{X}'(\omega) = \mathfrak{X}(f(\omega)).$$
   $\mathfrak{X}$' is a decider or recognizer for A.
5. together with
6. are the simple contrapositive statements of (3) and (4)!

□

Recall from Section 1:

- $\mathbf{ACCEPT}_{TM}$ is not decidable (recursive)
- $\mathbf{ACCEPT}_{TM}$ is Turing recognizable (recursively enumerable)
- $\mathbf{ACCEPT}_{TM}^{C}$ is not Turing recognizable
- $\mathbf{ACCEPT}_{TM}$ is not co-Turing recognizable

- $\mathbf{EQUAL}_{TM}$ is not decidable (recursive)

Two Questions

Question: How does one show that a given language B **IS** Turing recognizable?

Answer: find a recognizer $\mathfrak{R}$ for the language B!

Question: How does one show that a given language B **IS NOT** Turing recognizable?

Answer: start with a language A
which is not Turing recognizable

e.g., $\mathbf{ACCEPT}_{TM}^{C}$

and show that A $\leq$ B!!

examples

EQUAL$_{TM}$ is not Turing recognizable

ACCEPT$^{C}_{TM} \leq$ **EQUAL**$_{TM}$ if and only if **ACCEPT**$_{TM} \leq$ **EQUAL**$^{C}_{TM}$

Define Turing Machine $M_{\emptyset}$ which rejects everything, i.e., L ($M_{\emptyset}$) = $\emptyset$.

Define a second Turing Machine M' which for any input string **t** simulates Turing Machine M on input string ω

Observe that **ACCEPT**$_{TM}$ (<M>,ω) is true if and only if **EQUAL**$^{C}_{TM}$ (<M'>,<$M_{\emptyset}$>) is true.

EQUAL$_{TM}$ is not co-Turing recognizable

ACCEPT$^{C}_{TM} \leq$ **EQUAL**$^{C}_{TM}$ if and only if **ACCEPT**$_{TM} \leq$ **EQUAL**$_{TM}$

Define Turing Machine M_{Σ} which accepts everything, i.e., L (M_{Σ}) = Σ^*.

Define a second Turing Machine M' which for any input string **t** simulates Turing Machine M on input string ω

Observe that **ACCEPT**$_{TM}$ (<M>,ω) is true if and only if **EQUAL**$_{TM}$ (<M'>,<M_{Σ}>) is true.

Corollary. *There exists a language L which is neither Turing recognizable nor co-Turing recognizable – namely* ***EQUAL***$_{TM}$*!*

We had previously demonstrated this using two theoretical facts: the collection of all languages is an uncountably infinite set and the collection of all Turing Machines is a countably infinite set. Here we have actually presented a specific concrete example of the exception!

4.5 Computation Histories & Rice's Theorem

Reductions Using Computation Histories

A computation history for a Turing Machine M on an input string ω is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the computation of this machine.

Recall for Turing's Universal Machine that, while it is executing its commands, immediately following the standard descriptor for the simulated machine will be a sequence of configurations:

$$\#\mathrm{C}_1\#\mathrm{C}_2\# \ldots \#\mathrm{C}_k\#$$

Computation histories are finite sequences. Deterministic machines have exactly one computation history on any given input; nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches.

The use of computation histories is a technique most often employed to show that $\mathbf{ACCEPT}_{TM}$ is reducible to another formal language.

example

ALL_{CFG} is not decidable.

Suppose that $\mathbf{ALL}_{CFG}$ is decidable. Then there exists a decider $\mathfrak{D}$ for this language.

Consider a computation history for an arbitrary Turing Machine M running on an arbitrary input string ω. If this is an *accepting* computation history:

1. C_1 is a starting configuration
2. C_k is an accepting configuration
3. for every configuration C_j the next configuration $Cj + 1$ must naturally follow according to the rules of M

If the Turing Machine M **accepts** the input string ω, then there exists *at least one accepting computation history*; if the Turing Machine M **does not accept** the input string ω, then there is *no accepting computation history.* In analyzing $\mathbf{ACCEPT}_{TM}$, we will consider the behavior of its computation histories.

Given an input string (a possible computation history) **t**, a Turing Machine $\mathfrak{X}$ that would REJECT any accepting computation history performs as follows:

1. $\mathfrak{X}$ would check whether C_1 is a starting configuration for <M> and ω
   if it IS NOT, then ACCEPT
2. $\mathfrak{X}$ would check whether C_k is an accepting configuration for <M> and ω
   if it IS NOT, then ACCEPT
3. $\mathfrak{X}$ would scan the input string **t** and selects each of the component C_js
   $\mathfrak{X}$ pushes C_j onto stack
   $\mathfrak{X}$ compares C_j with C_{j+1} by popping C_j off stack one symbol at a time
   if the two configurations (Cj and C_{j+1}) DO NOT match a transition in <M>, then ACCEPT, else continue to next Cj

Observe that M does not accept ω if and only if L = L ($\mathfrak{X}$) = Σ^*.

Hence $\mathbf{ACCEPT}_{TM}$ (<M>,ω) $\leq$ $\mathbf{ALL}_{CFG}$ (<$\mathfrak{X}$>) and $\mathbf{ACCEPT}_{TM}$ must be decidable. A contradiction.

Comment. There is one subtle twist that is now introduced into this argument. Comparing $C)_j$ (on the stack) with C_{j+1} (in the input stream) is complicated by the fact that popping the stack recovers data in reverse order! To eliminate this problem and simplify the comparisons, we may assume that the input strings are not in the original form:

$$\#C_1\#C_2\# \ldots \#C_k\#$$

but rather

$$\#C_1\#C_2^R\#C_3\#C_4^R\# \ldots \#C_k\#$$

with all *even subscripted* configurations being reversed!

General Undecidability Criterion for Turing Machines

Every time we have addressed a decidability question regarding Turing Machines we have found that the question is undecidable. Rather than doing this in perpetuity, let us prove this statement once and for all! This is the aim of the important result known as Rice's Theorem.

Definition 4.5.1. A subset P of all possible languages, i.e., $P \in \wp(\Sigma^*)$, is called a **property** for Turing Machines.

Definition 4.5.2. A property P is said to be a **nontrivial property** of Turing Machines provided

- there exists a Turing Machine M_Y such that $L(M_Y) \in P$ and
- there exists a Turing Machine M_N such that $L(M_N) \notin P$

examples

nontrivial property

$P_{empty} = \{ \emptyset \}$ $\qquad\qquad$ $\emptyset \in P_{empty}$ and $\Sigma^* \notin P_{empty}$

trivial property

$P_{recognizable} = \{ L \mid L$ is Turing recognizable $\}$
is a property shared by all Turing Machines!
all Turing Machines are recognizers!

Theorem 4.5.1. *(Rice's Theorem) Suppose P is a nontrivial property of Turing Machines. Then the language*

$$\boldsymbol{L}_P = \{ <M> \mid L(<M>) \in P \}$$

is undecidable.

Proof. (Rice's Theorem)

We will show that $\mathbf{ACCEPT}_{TM} \leq \mathbf{L}_P$.

Consider the input stream (<M>,ω).

case one: $\emptyset \notin \mathbf{P}$

Choose a Turing Machine M_Y such that L (M_Y) $\in$ P.

Define another Turing Machine M_ω which acts on an input string **t**

1. simulate M on ω
2. if M ACCEPTs, then
   simulate M_Y on input string **t**
   if M_Y ACCEPTs, then ACCEPT

M_ω accepts **t** if and only if M accepts ω and M_Y accepts **t**. So

L (M_ω) = L (M_Y) if M accepts ω
L (M_ω) = $\emptyset$ if M does not accept w

But L (M_Y) $\in$ P and $\emptyset \notin$ P implies that
L (M_ω) $\in$ P if and only if M accepts ω.

Thus $\mathbf{ACCEPT}_{TM} \leq \mathbf{L}_P$.

case two: $\emptyset \in \mathbf{P}$

Then $\emptyset \notin P^C$ and $\mathbf{L}_P = (\ \mathbf{L}_P{}^C\ )^C$.

□

We are now in a position to wrap up all the loose ends we have left scattered across this chapter!

All our unanswered remaining problems are in fact all undecidable!

$\mathbf{ALL}_{CFG}$ (<G>) is not decidable!

We proved this using computational histories!

$\mathbf{ALL}_{TM}$ is not decidable!

This is a simple corollary of Rice's Theorem, using $\mathrm{P}_{all} = \{ \Sigma^* \}$.

$\mathbf{EQUAL}_{CFG}$ is not decidable!

$\mathbf{ALL}_{CFG}$ (<M>) $\leq$ $\mathbf{EQUAL}_{CFG}$ (<M>,<M_{Σ}>)

So, if $\mathbf{EQUAL}_{CFG}$ is decidable, then so is $\mathbf{ALL}_{CFG}$, which contradicts the very first statement above.

4.6 Other Problems in Decidability

This section is neither a thorough presentation nor a comprehensive study of other problems in decidability. Rather this section is to provide an awareness that decidability is any problem statement which can be massaged into an ACCEPT / REJECT form. Two important categories follow.

Mathematical Foundations

Propositional logic focuses on simple statements (**propositions**), their truth values (either **true** or **false**),and the four operators **not**, **and**, **or**, and **if-then**. Truth tables are an essential tool for determining the truth value for more complicated formulas built up from simpler components.

Predicate calculus builds on propositional logic by incorporating the concepts of **variables**, **quantifiers** $\exists$ (existential) and $\forall$ (universal), and **well-formed formulas**.

Theorem 4.6.1. *The propositional logic is decidable.*

Theorem 4.6.2. *The predicate calculus is not decidable.*

Number Theory

Number theory is one of the oldest branches of mathematics and also one of its most difficult. General statements regarding number theory are typically very difficult to verify. For example,

$\forall\, x\, \exists\, y\, [\, x + x = y\,]$ example of **true** statement

$\exists\, y\, \forall\, x\, [\, x + x = y\,]$ example of **false** statement

Alonzo Church, building on the work of Kurt Gödel, showed that within a given number system certain fundamental questions may be undecidable!

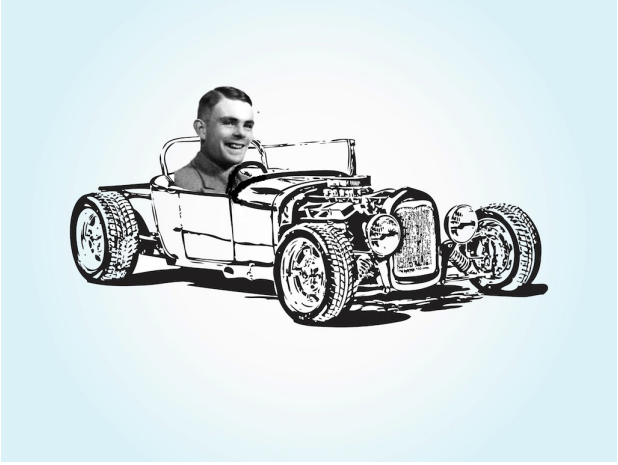
Theorem 4.6.3. *Theory ($\mathbb{N}$, +), natural numbers with only addition, is decidable.*

Theorem 4.6.4. *Theory ($\mathbb{N}$, + , $\times$), natural numbers with both addition and multiplication, is not decidable.*

Please refer to Introduction to the Theory of Computation, 3rd edition, by Michael Sipser, Cengage Learning, 2013 for a much better introduction to this material. I have merely highlighted a handful of key results.

Chapter 5

Basic Terminology



5.1 Landau Notation

Decidability focuses on the single question:

Can this problem be solved – YES or NO?

Time complexity considers only decidable problems and focuses on the question:

How much time (steps)
will I need to determine a solution?

Space complexity considers decidable problems and focuses on the question:

How much scratch paper (memory)
will I need to determine a solution?

During the previous discussions we have encountered several flavors of Turing Machines: single tape versus multi-tape machines and deterministic versus nondeterministic machines. For the last two chapters of this text we will focus primarily on **single tape** Turing Machines but we will consider separately **deterministic** and **nondeterministic** machines.

Time will be measured in the *number of steps* (i.e., transitions) necessary to solve the problem using a single tape Turing Machine.

Space will be measured in the *number of cells* (i.e., tape positions) necessary to solve the problem using a single tape Turing Machine.

In order to have some consistent notation for speed and storage requirements, we will use a specialized shorthand – called **Landau Notation**. Given a Turing Machine operating on an input tape of length n (tape cells), both speed and space requirements will be measured by identifying the ***most dominant contributing term*** in the speed and space formula.

Landau Notation

Landau Notation is a mathematical shorthand designed to quickly compare the rate of growth for functions $f : \mathbb{N} \to \mathbb{N}$ with $f(n) > 0$.

Definition 5.1.1. We say that $f(n)$ is $\bigcirc(g(n))$, **big oh**, provided there exists $C_2 > 0$ and $n_0 \in \mathbb{N}$ such that

$$f(n) \leq C_2 g(n) \text{ for all } n \geq n_0.$$

Definition 5.1.2. We say that $f(n)$ is $\Omega(g(n))$, **big omega**, provided there exists $C_1 > 0$ and $n_0 \in \mathbb{N}$ such that

$$C_1 g(n) \leq f(n) \text{ for all } n \geq n_0.$$

Definition 5.1.3. We say that $f(n)$ is $\Theta(g(n))$, **big theta**, provided there exists $\mathrm{C}_1, C_2 > 0$ and $n_0 \in \mathbb{N}$ such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \text{ for all } n \geq n_0.$$

Lemma 5.1.1. *(Relationships between Properties)*

1. *$f(n)$ is $\bigcirc(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.*
2. *$f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $\bigcirc(g(n))$ and $f(n)$ is $\Omega(g(n))$.*
3. *All three properties $\bigcirc$, Ω, and Θ are transitive properties.*

Proof. All three proofs are trivial!

□

Definition 5.1.4. We say that $f(n)$ is $\circ(g(n))$, **little oh**, provided

$$\lim_{n\to\infty} f(n)/g(n) = 0.$$

Definition 5.1.5. We say that $f(n)$ is $\omega(g(n))$, **little omega**, provided

$$\lim_{n\to\infty} f(n)/g(n) = \infty.$$

Lemma 5.1.2. *(Relationships between Properties)*

1. *$f(n)$ is $\circ(g(n))$ if and only if $g(n)$ is $\omega(f(n))$.*
2. *if $f(n)$ is $\circ(g(n))$, then $f(n)$ is $\bigcirc(g(n))$.*
3. *if $f(n)$ is $\omega(g(n))$, then $f(n)$ is $\Omega(g(n))$.*
4. *if $\lim_{n\to\infty} f(n)/g(n) = L > 0$, then $f(n)$ is $\Theta(g(n))$.*
5. *Both properties $\circ$ and ω are transitive properties.*

Proof. All five proofs are trivial!

□

examples

$\log_2(n)$ is $\circ(\sqrt{n})$
$\Rightarrow \log_2(n)$ is $\bigcirc(\sqrt{n})$

$$\lim_{n\to\infty} \frac{\log_2(n)}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{1}{n\ln(2)}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{\ln(2)n} = \lim_{n\to\infty} \frac{2}{\ln(2)\sqrt{n}} = 0$$

n^k is $\circ(2^n)$
$\Rightarrow n^k$ is $\bigcirc(2^n)$

$$\lim_{n\to\infty} \frac{n^k}{2^n} = \lim_{n\to\infty} \frac{kn^{k-1}}{\ln(2)2^n} = \ldots = \lim_{n\to\infty} \frac{k!}{\ln(2)^k 2^n} = 0$$

Growth Chart for Basic Functions

Growth Chart for Basic Functions

	1	log2 (n)	sqrt (n)	n	n^2	n^3	2^n	2^(n^2)	2^(n^3)
1	1	0	1	1	1	1	2	2	2
2	1	1	1.41421	2	4	8	4	16	256
4	1	2	2	4	16	64	16	65536	1.84467E+19
8	1	3	2.82843	8	64	512	256	1.84467E+19	1.3408E+154
16	1	4	4	16	256	4096	65536	1.15792E+77	#NUM!
32	1	5	5.65685	32	1024	32768	4294967296	#NUM!	#NUM!
64	1	6	8	64	4096	262144	1.84467E+19	#NUM!	#NUM!
128	1	7	11.31371	128	16384	2097152	3.40282E+38	#NUM!	#NUM!
256	1	8	16	256	65536	16777216	1.15792E+77	#NUM!	#NUM!
512	1	9	22.62742	512	262144	134217728	1.3408E+154	#NUM!	#NUM!
1024	1	10	32	1024	1048576	1073741824	#NUM!	#NUM!	#NUM!

5.2 Deterministic Time and Space

Definition 5.2.1. The class **DTIME (t(n))** is the collection of deterministic Turing Machines M such that

- M is a decider
- for an input tape of length **n**
  M terminates in at most **t(n)** steps

example

Consider the language L = { 0^k1^k | k ≥ 0 }

Define a decider $\mathfrak{D}$ as follows:

1. scan across the tape:
   if a 0 follows a 1,
   then REJECT
2. return to the start of the tape
3. repeat until either no 0's or no 1's remain
   scan across the tape
   crossing off the first 0 and the first 1 encountered
   return to start of the tape
4. if any 0's remain or any 1's remain,
   then REJECT
   else ACCEPT

Number of steps required:
n (scan) + n (return) + n (repeat) {n (scan) + n (return)} + n (scan)
$= 2n^2 + 3n$
$= \bigcirc(n^2)$

Thus $\mathfrak{D} \in$ DTIME (n^2).

Definition 5.2.2. The class **DSPACE (s(n))** is the collection of deterministic Turing Machines M such that

- M is a decider
- for an input tape of length **n**
  M terminates having used at most **s(n)** tape cells

examples

Consider the language L = { 0^k1^k | k ≥ 0 }

Consider the same decider $\mathfrak{D}$ as defined in the previous example. Observe that $\mathfrak{D}$ repeatedly criss-crossed the original input tape and even modified most tape cells, but *no new cells were used*!

Number of cells required:
n = $\bigcirc(n)$

Thus $\mathfrak{D}$ ∈ DSPACE (n).

Consider the language L = { 0^k1^k | k ≥ 0 }

But consider an alternative decider $\mathfrak{D}$' which is a minor modification of $\mathfrak{D}$:

Replace step (3) above with the following
which is repeated until either no 0's or no 1's remain

- scan across the tape to see if the total number of 0's and 1's is odd
  if so REJECT
  if not return to start of the tape
- cross off the first 0 and every other alternate 0
- cross off the first 1 and every other alternate 1
- return to start of the tape

Observe that $\mathfrak{D}$' is significantly faster than the original $\mathfrak{D}$. It cancels out half of the symbols on each scan, requiring only $\log_2(n)$ scans.

Thus $\mathfrak{D}$' ∈ DTIME (n $\log_2$(n))
But $\mathfrak{D}$' ∈ DSPACE (n) (no change)

Lemma 5.2.1. *(Basic Properties of Time and Space)*

1. *If $f(n)$ is $\bigcirc(g(n))$, then DTIME (f(n)) $\subseteq$ DTIME (g(n)).*
2. *If $f(n)$ is $\bigcirc(g(n))$, then DSPACE (f(n)) $\subseteq$ DSPACE (g(n)).*

Proof. Both statements are trivial!

□

Definition 5.2.3. The class $\mathbf{P} = \cup_k$ DTIME (n^k) is referred to as **polynomial time**.

Definition 5.2.4. The class **PSPACE** $= \cup_k$ DSPACE (n^k) is referred to as **polynomial space**.

Definition 5.2.5. The class **EXP** $= \cup_k$ DTIME $(2^{(n^k)})$ is referred to as **exponential time**.

Definition 5.2.6. The class **EXPSPACE** $= \cup_k$ DSPACE $(2^{(n^k)})$ is referred to as **exponential space**.

Theorem 5.2.2. *(Basic Properties of Time and Space)*

1. *P $\subseteq$ EXP*
2. *PSPACE $\subseteq$ EXPSPACE*

Proof. Use the lemma above!

□

In English:

- The class P represents problems that can be solved in reasonable, albeit possibly lengthy, time.
- The class EXP represents problems that typically exceed normal waiting times.

- The class PSPACE represents problems that require reasonable amounts of memory.
- The class EXPSPACE represents problems that require exorbitant amounts of memory.

Lemma 5.2.3. *For any given function* $f : \mathbb{N} \to \mathbb{N}$ *with* $f(n) > 0$,

$$DTIME\ (f(n)) \subseteq DSPACE\ (f(n)).$$

Proof. If a Turing Machine $\mathfrak{D}$ terminates in at most f(n) steps, then $\mathfrak{D}$ visits at most f(n) tape cells – and most likely significantly fewer!

Hence DTIME (f(n)) $\subseteq$ DSPACE (f(n)).

□

Theorem 5.2.4. *(Basic Properties of Time and Space)*

1. $P \subseteq PSPACE$
2. $EXP \subseteq EXPSPACE$

Proof. Use the lemma above!

□

Theorem 5.2.5. *(Summary of All Basic Properties)*

1. $P \subseteq EXP$
2. $PSPACE \subseteq EXPSPACE$
3. $P \subseteq PSPACE$
4. $EXP \subseteq EXPSPACE$

Proof. Theorem 5.2.2 and Theorem 5.2.4.

□

5.3 Nondeterministic Time and Space

Recall that deterministic machines have only one choice at any given step in a calculation; nondeterministic machines may choose from several options.

Definition 5.3.1. The class **NTIME (t(n))** is the collection of nondeterministic Turing Machines M such that

- M is a decider
- for an input tape of length **n**
  M terminates with its longest path
  along any of its branches
  having at most **t(n)** steps.

example

Consider the language SAT =
{ boolean expressions ϕ | ϕ is satisfiable }

ϕ is an expression comprised of

- variables: $x_1, x_2, \ldots, x_k$
- operators $\wedge$ (and/conjunction), $\vee$ (or/disjunction), $\neg$ (not/negation)
- parentheses (and)

Comment. We will later define a more structured form for the boolean expression ϕ, but this general form is sufficient for our current discussion.

If the input tape contains the boolean expression ϕ and has length **n**, then the number of variables **k** is less than or equal to **n**. The total number of possible truth value combinations would be $\mathbf{2}^k = \bigcirc(2^n)$.

Define a decider $\mathfrak{D}$ as follows:

1. generate the set of possible truth values
   for each truth value combination, evaluate ϕ
   for each variable ($k \leq n$),
   scan the input line (length n)
   substitute its truth value into the expressions
   and evaluate any subexpressions that have been determined
   if ϕ is true, then ACCEPT else continue
2. REJECT

deterministic time would be $2^n \bigcirc (n^2)$ which is $\bigcirc(2^n)$

$\mathfrak{D} \in$ DTIME (2^n)

nondeterministic time would be $1 \bigcirc(n^2)$ which is $\bigcirc(n^2)$

$\mathfrak{D} \in$ NTIME (n^2)

Definition 5.3.2. The class **NSPACE (s(n))** is the collection of nondeterministic Turing Machines M such that

- M is a decider
- for an input tape of length **n**
  M terminates along any of its branches
  having used at most **s(n)** tape cells

example

Consider the language SAT again
= { boolean expressions ϕ | ϕ is satisfiable }

Consider the same decider $\mathfrak{D}$ as above. In addition to the actual input symbols, the tape would also have to hold the truth values for the variables (less than or equal to **n** items) and the temporary storage for the stack (also less than or equal to **n** items).

deterministic space would be 3 $\bigcirc(n)$ which is $\bigcirc(n)$

D $\in$ DSPACE (n)

nondeterministic space would be 3 $\bigcirc(n)$ which is $\bigcirc(n)$

D $\in$ NSPACE (n)

The difference between deterministic Turing Machines and nondeterministic Turing Machines is that the ability to choose is built into the nondeterministic version. It considers all possible choices *simultaneously.* The deterministic version is capable of solving the same problems; however, it considers all possible choices *sequentially.*

Obviously this takes more time! But it does not require more space.

Space is reusable; time is not.

Lemma 5.3.1. *(Basic Properties of Time and Space)*

1. *If $f(n)$ is $\bigcirc(g(n))$, then NTIME (f(n)) $\subseteq$ NTIME (g(n)).*
2. *If $f(n)$ is $\bigcirc(g(n))$, then NSPACE (f(n)) $\subseteq$ NSPACE (g(n)).*

Proof. Both statements are trivial!

□

Definition 5.3.3. The class **NP** $= \cup_k$ NTIME (n^k) is referred to as **nondeterministic polynomial time**.

Definition 5.3.4. The class **NPSPACE** $= \cup_k$ NSPACE (n^k) is referred to as **nondeterministic polynomial space**.

Definition 5.3.5. The class **NEXP** $= \cup_k$ NTIME $(2^{(n^k)})$ is referred to as **nondeterministic exponential time**.

Definition 5.3.6. The class **NEXPSPACE** $= \cup_k$ NSPACE $(2^{(n^k)})$ is referred to as **nondeterministic exponential space**.

Theorem 5.3.2. *(Basic Properties of Time and Space)*

1. *NP $\subseteq$ NEXP*
2. *NPSPACE $\subseteq$ NEXPSPACE*

Proof. Use the lemma above!

□

Lemma 5.3.3. *For any given function $f : \mathbb{N} \to \mathbb{N}$ with $f(n) > 0$,*

NTIME (f(n)) $\subseteq$ NSPACE (f(n)).

Proof. If a Turing Machine $\mathfrak{D}$ terminates in at most f(n) steps, then $\mathfrak{D}$ visits at most f(n) tape cells – and most likely significantly fewer!

Hence DTIME (f(n)) $\subseteq$ DSPACE (f(n)).

□

Theorem 5.3.4. *(Basic Properties of Time and Space)*

1. $NP \subseteq NPSPACE$
2. $NEXP \subseteq NEXPSPACE$

Proof. Use the lemma above! □

Theorem 5.3.5. *(Summary of All Basic Properties)*

1. $NP \subseteq NEXP$
2. $NPSPACE \subseteq NEXPSPACE$
3. $NP \subseteq NPSPACE$
4. $NEXP \subseteq NEXPSPACE$

Proof. Theorem 5.3.2 and Theorem 5.3.4.

□

5.4 Important Theorems

Theorem 5.4.1. *(Deterministic Multi-Tape to Deterministic Single-Tape) If M is a deterministic multi-tape Turing Machine and M ∈ DTIME (t(n)), then M has an equivalent deterministic single-tape Turing Machine M' with M' ∈ DTIME (t(n)^2).*

Proof. We briefly consider multi-tape Turing Machines in our discussion as a stepping stone in our understanding of time and space complexity.

Suppose M is a deterministic multi-tape Turing Machine. We can interleave the k tapes and symbol indicating current head positions on each tape into a single tape. Each step in the Turing Machine M requires that we scan the entire interleaved tape and update the interleaving as appropriate.

The time to scan the tape is less than or equal to t(n); the number of steps to simulate is less than or equal to t(n). Hence, the total number of steps is less than or equal to t(n)^2.

□

Corollary. *If $M \in P_{multi-tape}$, then its equivalent M' $\in P_{single-tape}$.*

Comment. Converting a multi-tape Turing Machine to an equivalent single-tape Turing Machine incurs a **polynomial cost**!

Theorem 5.4.2. *(Nondeterministic Single-Tape to Deterministic Single-Tape) If M is a nondeterministic single-tape Turing Machine and M ∈ NTIME (t(n)), then M has an equivalent deterministic single-tape Turing Machine M' with M' ∈ DTIME ($2^{O(t(n))}$).*

Proof. Suppose M is a nondeterministic single-tape Turing Machine. The longest branch found within the computation tree has length t(n). This implies that the number of leaves in the computation tree is less than or equal to $b^{t(n)}$. Hence the total number of nodes in the computation tree is less than or equal to

$$1 + b + b^2 + b^3 + \ldots + b^{t(n)} \leq 2\ b^{t(n)}$$

The time it takes to travel from the root to a given node in the computation tree is less than or equal to t(n). Hence, the total running time is less than or equal to

$$t(n)\ 2^{bt(n)}$$

We can simulate M easily using a deterministic 3-tape Turing Machine: the first tape for input, the second tape as a scratch tape, and the third tape is a queue for holding the next nodes to be studied in a ***breadth first search***. Note here that we have again introduced a multi-tape Turing Machine into our discussion.

This deterministic 3-tape Turing Machine may be simulated using a deterministic single-tape Turing Machine M' with a running time less than or equal to

$$[\ t(n)\ 2^{bt(n)}\ ]^{\ 2} = t(n)^2\ 2^{2bt(n)} = 2^{O(t(n))}$$

according to Theorem One. □

Corollary. *If $M \in NP_{single-tape}$, then its equivalent $M' \in EXP_{single-tape}$.*

Corollary.

$$NP \subseteq EXP$$

Comment. Converting a nondeterministic single-tape Turing Machine to an equivalent deterministic single-tape Turing Machine incurs an **exponential cost**!

Theorem 5.4.3. *(Deterministic Space to Deterministic Time) If M is a deterministic single-tape Turing Machine and M ∈ DSPACE (s(n)), then M ∈ DTIME* ($2^{\bigcirc(s(n))}$).

Proof. Suppose M is a deterministic single-tape Turing Machine and length of its output tape is s(n). Suppose M has **q** states and **g** symbols in its alphabet. Each configuration is a snapshot of the tape contents plus the location of the location of the head and its current state.

- tape contents: $g^{s(n)}$
- location of head: 1, 2, . . . , s(n)
- current state: 1, 2, . . . , q

Total possible unique configurations would be

$$\mathbf{q\ s(n)\ g}^{s(n)} = \mathbf{2}^{\bigcirc(s(n))}.$$

Therefore, if M is not circular, then the above estimate is the maximum number of states that M could conceivably go through before terminating.

□

Corollary.

$$PSPACE \subseteq EXP$$

Theorem 5.4.4. *(Savitch's Theorem) If M is a nondeterministic single-tape Turing Machine and M ∈ NSPACE (s(n)), then M has an equivalent deterministic single-tape Turing Machine M' with M' ∈ DSPACE (s(n)^2).*

Proof. Suppose M is a nondeterministic single-tape Turing Machine and the length of its output tape is always less than or equal to s(n). Suppose M has **q** states and **g** symbols in its alphabet. Following the same counting techinques as in the previous theorem, the total possible unique configurations would be

$$\mathbf{q\ s(n)\ g}^{s(n)} = \mathbf{2}^{\bigcirc(s(n))} \leq \mathbf{2}^{d\ s(n)}$$

We first define a recursive procedure CANYIELD which given two configurations c_1 and c_2 and number of steps **t** will determine whether or not the Turing Machine M can move from configuration c_1 to configuration c_2 in *at most* **t** steps.

```
if t = 0
    then if c1 = c2 ACCEPT
else if t = 1
    then if M can take a single step from c1 to c2 ACCEPT
else
    for all possible configurations cm
        if CANYIELD (c1,cm,t/2) and CANYIELD (cm,c2,t/2)
            both ACCEPT
                then ACCEPT
REJECT
```

We now define a deterministic single-tape Turing Machine M' which is equivalent to M and whose output tape never exceeds length s(n)^2. We first modify M slightly so that when it finally ACCEPTs, it

1. clears the length of the tape
2. returns to its start position (leftmost cell)

Let us designate the starting configuration as c_{start} and the final configuration c_{accept}.

Our final deterministic single-tape Turing Machine M is defined as follows:

On input string ω,
output the result of CANYIELD (c_{start},c_{accept},$2^{d\ s(n)}$)

Each recursive call of CANYIELD pushes two configurations onto the stack together with an integer value – each push requires $\bigcirc$ (s(n)) extra space.

Furthermore, the depth of recursion would be
$\log_2 (2^{d\ s(n)}) = d\ s(n) = \bigcirc$ (s(n)).

Hence the stack would never exceed $\bigcirc$ (s(n)) $\bigcirc$ (s(n)) = $\bigcirc$ ($s(n)^2$).

□

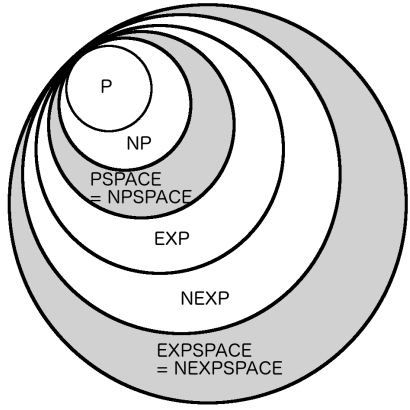
Corollary.

1. *PSPACE = NPSPACE*
2. *EXPSPACE = NEXPSPACE*

We conclude this section and this chapter by summarizing all the results we have accumulated to date!

Theorem 5.4.5.

$$P \subseteq NP \subseteq NPSPACE = PSPACE \subseteq EXP \subseteq NEXP \subseteq NEXPSPACE = EXPSPACE$$



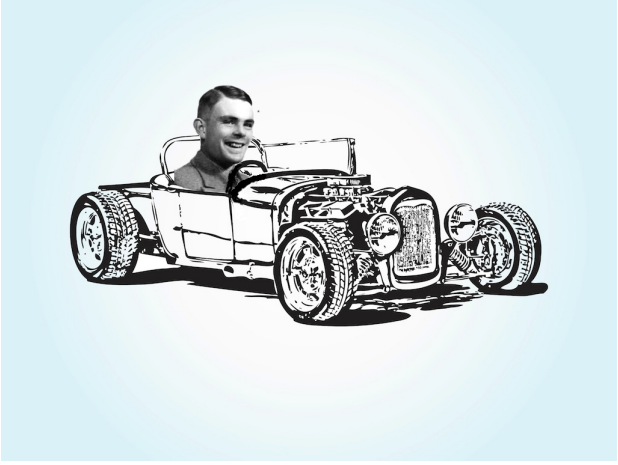
Most researchers in the field of studying the foundations of computing believe that all of the above set inclusions are proper, for example,

$$\mathrm{P} \subsetneq \mathrm{NP}$$

but these are just conjectures and have yet to be proven.

Chapter 6

P versus NP



6.1 Statement of the Problem

P is the class of problems solvable in deterministic polynomial time, i.e., a computer program will yield results in a "reasonable time".

EXP is the class of problems solvable by brute force techniques in deterministic exponential time, i.e., a computer program will yield results but may not be in a "reasonable time". It might take days, weeks, months, years, ...

And sandwiched between these two classes is the class NP, problems solvable in nondeterministic polynomial time.

$$P \subseteq NP \subseteq EXP$$

In the hierarchy of language classes, the class NP falls between being solvable and not being solvable in reasonable time. We might ask ourselves: does the class NP more closely resemble the class P or does the class NP more closely resemble EXP?

We recently proved that a deterministic simulation of a Turing Machine M in NP is a Turing Machine M in EXP, i.e., a conversion from a nondeterministic algorithm to a deterministic algorithm comes at an exponential cost. That would seem to move NP closer to EXP in the hierarchy. However, we have only considered fairly simple deterministic techniques for calculating permutations, combinations, possible truth tables, etc. These techniques do, in fact, incur an exponential penalty. But we have not ruled out the possibility that a deterministic polynomial time technique might exist that accomplishes the same result! And if such a deterministic polynomial time technique were to exist, NP would not only move closer to P, but NP would in fact be equal to P.

This chapter explores two important issues concerning P and NP:

1. Just because an algorithm is in P does not mean it is fast and can not be improved. Obviously, some algorithms execute more quickly and others more slowly. We look more closely at the Landau Notation to rank algorithms in terms of efficiency, especially as the size n of the problem becomes increasingly large. This is the focus of Section 2: Analysis of Algorithms.

2. We identify several of the hardest problems that belong to the class NP. And we put forth the relatively startling fact: If any one of these problems has a deterministic polynomial time solution, then the entire class NP = P! This is the focus of Section 3: NP Completeness.

In this last chapter of the text, we move to a much higher level of abstraction when talking about Turing Machines. We have previously studied the formal definition of a Turing Machine. We have come to see its capabilities and the power of the Universal Machine. The standard descriptor (SD) for a given Turing Machine is essentially the machine language or the assembly language for the Universal Machine.

And just as programmers might choose to move away from assembly language coding to a higher level language, so too will we move away from SDs and vague textual descriptions to algorithms using a higher level language – a pseudocode similar to C or JAVA.

Our problems will still be stated in the form of decidability and/or recognition of formal languages, and our algorithms will look more like modern programs.

6.2 Analysis of Algorithms

In this section we will discuss the efficiency of several algorithms. Our examples are meant to be illustrative and not comprehensive. Normally, detailed information regarding this topic is found in textbooks with titles such as

- Data Structures
- Analysis of Algorithms

In the deterministic world, the choice of one algorithm over another may depend heavily on the exponent for **n** in its efficiency estimate

$$\bigcirc(n^k)$$

Replacing a single factor of **n** in the efficiency estimate with either $\sqrt{n}$ or $\log_2(n)$ will also provide better performance.

examples

SEQUENTIAL SEARCH

A is an array [1 . . . n] of data
t is a target item to locate

SEQUENTIAL SEARCH (A , t)

1. for i = 1 to n do
   if (A[i] == t)
   then FOUND at index i
2. NOT FOUND

Since the time efficiency for each comparison (A[i] == t) is the same, the overall efficiency is determined by the number of repetitions (n) executed.

SEQUENTIAL SEARCH is $\bigcirc(n)$

BINARY SEARCH

A is an array [1 ... n] of data with A[1] $\leq$ A[2] $\leq \ldots \leq$ A[n]
t is a target item to locate

BINARY SEARCH (A , t)

1. lb = 1; ub = n
2. while (lb $\leq$ ub) do
   mid = (lb + ub)/2
   if (A[mid] == t)
   then FOUND at index mid
   else if (t < A[mid])
   then ub = mid − 1
   else // (t > A[mid])
   lb = mid + 1
3. NOT FOUND

Since the time efficiency for each comparison (A[i] == t) is the same, the overall efficiency is determined by the number of repetitions executed, which is at most $\log_2(n)$.

BINARY SEARCH is $\bigcirc(\log_2(n))$

SELECTION SORT

A is an array [1 ... n] of data in random order
rearrange the items so that:
$A[1] \leq A[2] \leq \ldots \leq A[n]$

```
SELECTION SORT ( A )

1. for i = 1 to n - 1 do
       least = A[i]; loc = i
       for j = i + 1 to n do
           if (A[j] < least)
               then { least = A[j]; loc = j }
       if (loc != i)
           then { temp = A[i]; A[i] = A[loc]; A[loc] = temp }
```

Since the time efficiency for each comparison (A[i] == t) is the same and the time efficiency for each interchange of data at the last step is the same, the overall efficiency is determined by the nested repetitions (each of order **n**) executed.

SELECTION SORT is $\bigcirc(n^2)$

MATRIX MULTIPLICATION

A is an array [1 ... n,1 ... n] of data
B is an array [1 ... n,1 ... n] of data
C is an array [1 ... n,1 ... n] of data
calculate the matrix product of A x B, storing the result in the matrix C

MATRIX MULTIPLICATION (A, B, C)

```
1. for i = 1 to n do
       for j = 1 to n do
           C[i,j] = 0
           for k = 1 to n do
               C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

The overall efficiency is determined by the nested repetitions (each of order **n**) executed.

MATRIX MULTIPLICATION is $\bigcirc(n^3)$

PATH

G is a directed graph with **n** nodes
G is represented by an adjacency matrix A[1 . . . n,1 . . . n]

$$\text{where A[i,j]} = \begin{cases} 1 & \text{if an arc exists from node i to node j} \\ 0 & \text{if such an arc does not exist} \end{cases}$$

s is a specified start node
t is a specified terminal node

Does there exist a path in the graph that connects node **s** to node **t**?

```
PATH (A, s, t)

1. for i = 1 to n do
       M[i] = 0, i.e., all nodes are unmarked

2. M[s] = 1

3. repeat until no additional nodes are marked
       for i = 1 to n do
           if (M[i] == 1)
               then for j = 1 to n do
                   if ((A[i,j] ==1)
                       then M[j] = 1

4. if (M[t] == 1)
       then YES
       else NO
```

The overall efficiency is determined by the nested repetitions (each of order **n**) executed.

$$\text{PATH is } \bigcirc(n^3)$$

SAT (traditional form)

The language SAT = { boolean expressions ϕ | ϕ is satisfiable } was first discussed in Section 3 of the last chapter. We return to it now in Chapter Six to restate the problem in its more traditional form.

ϕ is an expression comprised of

- variables: $x_1, x_2, \ldots, x_k$
- operators $\wedge$ (and/conjunction), $\vee$ (or/disjunction), $\neg$ (not/negation)
- parentheses (and)

axioms for boolean algebra

$$\begin{array}{lll} a \vee T = T & a \vee a = a & a \vee (a \wedge b) = a \\ a \vee F = a & a \wedge a = a & a \wedge (a \vee b) = a \\ a \wedge T = a & a \vee \neg a = T & \\ a \wedge F = F & a \wedge \neg a = F & \neg(\neg a) = a \end{array}$$

$$\begin{array}{cc} a \vee b = b \vee a & a \wedge b = b \wedge a \\ a \vee (b \vee c) = (a \vee b) \vee c & a \wedge (b \wedge c) = (a \wedge b) \wedge c \\ \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) & a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \\ \neg(a \vee b) = \neg a \wedge \neg b & \neg(a \wedge b) = \neg a \vee \neg b \end{array}$$

some additional terminology

Definition 6.2.1. A **literal** is either a simple variable **x** or the negation of a simple variable $\neg$**x**.

Definition 6.2.2. A **clause** is the or / disjunction of a finite number of literals.

Definition 6.2.3. A boolean expression is said to be in **conjunctive normal form** (CNF) if it is the and / conjunction of a finite number of clauses.

Theorem 6.2.1. *For every boolean expression ϕ of the propositional logic, there exists an equivalent boolean expression ψ in conjunctive normal form.*

Proof. This theorem is the result of repeated applications of the axioms, particularly de Morgan's laws and the distributive laws. □

SAT (ϕ)

ϕ is a boolean expression in conjunctive normal form
determine whether or not a set of true/false assignments
exists which satisfies ϕ

Let n = $|\phi|$ (total number of symbols)
let k = number of variables x_i
$\Rightarrow k \leq n$

1. for x_1 = 0,1 do
   for x_2 = 0,1 do
   :
   for x_k = 0,1 do
   plug in $x_1, x_2, \ldots, x_k$ into ϕ
   if ϕ is true then YES (recall this step is $\bigcirc(n^2)$)
2. NO

The nested loops to generate $x_1, x_2, \ldots, x_k$ requires 2^k steps in deterministic time, but is constant in nondeterministic time!

SAT is $\bigcirc(2^n)$ deterministic time
SAT is $\bigcirc(n^2)$ nondeterministic time

SAT $\in$ DTIME $(2^n) \subseteq$ EXP
SAT $\in$ NTIME $(n^2) \subseteq$ NP

6.3 NP Completeness

So the question remains:

$$\text{Is P} \subsetneq \text{NP?}$$
$$\text{or}$$
$$\text{Is P} = \text{NP?}$$

If we wanted to prove the latter we would want to consider the most difficult problem of its kind – all other problems in NP pale in comparison. If we can show that such a problem is solvable in deterministic polynomial time, then all the rest of the problems in NP come along for free!

But we quickly realize that we have to recognize such a problem. Which returns us to a topic previously discussed – reducibility. Recall a definition from the recent past:

Definition 6.3.1. We say that $f : \Sigma^* \to \Sigma^*$ is a **computable function** provided there exists a Turing Machine M such that

- M halts on all input ω and
- the resultant output is $f(\omega)$

Definition 6.3.2. We say that a language A is **mapping reducible** to a language B, denoted $A \leq B$, provided there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that

$$\omega \in A \text{ if and only if } f(\omega) \in B$$

The computable function f is called a **reduction** from A to B.

For decidability, determinism versus nondeterminism is irrelevant, as is the question of efficiency. However, these issues are most relevant for questions concerning P and NP.

Definition 6.3.3. We say that a language A is **polynomial time reducible** to a language B, denoted $A \leq_P B$, provided there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that

- $f \in P$
- $\omega \in A$ if and only if $f(\omega) \in B$

Lemma 6.3.1. *(Properties Polynomial Time Reducibility)*

1. *If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$*
2. *If $A \leq_P B$ and $B \in P$, then $A \in P$*
3. *If $A \leq_P B$ and $A \notin P$, then $B \notin P$*

Proof. Three Components:

1. Define $h = g \circ f : \Sigma^* \to \Sigma^*$.
   $\omega \in$ A if and only if $f(\omega) \in$ B if and only if $g(f(\omega)) \in$ C
2. Suppose M $\in$ P and M decides B.
   Define N on the input string ω as follows:
   (a) compute $f(\omega)$
   (b) simulate M on the string $f(\omega)$
   Then N $\in$ P and N decides A.
3. This is just the contrapositive to item 2.

□

Definition 6.3.4. B is called **NP complete** provided

1. B $\in$ NP
2. $\forall$ A $\in$ NP, A $\leq_P$ B

Every other problem in NP can be reduced to B! Such a problem would truly be the hardest to solve.

Theorem 6.3.2. *(Properties NP Completeness)*

1. *If B is NP complete and $B \in P$, then $P = NP$.*
2. *If B is NP complete and $B \leq_P C$, then C is NP complete.*

Proof. Use the previous Lemma!

□

All the above looks really good, but unfortunately it would be meaningless unless there really exists an NP complete problem! That is why the following theorem is so important.

Theorem 6.3.3. *(Cook – Levin Theorem)*

***SAT** is NP complete.*

Proof. In Section 2 we demonstrated that SAT $\in$ NP. So it only remains to verify that if A $\in$ NP then A $\leq_P$ SAT.

So let us assume A $\in$ NP and that the nondeterministic Turing Machine M decides A in polynomial time, $\bigcirc(n^k)$ for k sufficiently large.

Furthermore, A also utilizes polynomial space, $\bigcirc(n^k)$.

Recall that when we previously discussed the concept of reducibility, we concluded by introducing the technique of computation histories, i.e., the sequence of configurations that the output tape goes through to arrive at an accept state.

$$\#C_1\#C_2\#\ldots\#C_k\#$$

The length of any individual configuration is $\bigcirc(n^k)$.
And the number of configurations leading to an accepting state is $\bigcirc(n^k)$.
So we may store an accepting sequence in a $n^k \times n^k$ matrix T.

first row	$\mathbf{q_s}$	$\mathbf{w_1}$	w_2	$\ldots$	$\ldots$	w_n	$\square$	$\ldots$	$\ldots$	$\square$
:										
middle rows	s_1	$\ldots$	s_i	$\mathbf{q_k}$	$\mathbf{s_{i+1}}$	$\ldots$	$s_{n'}$	$\square$	$\ldots$	$\square$
:										
last row	t_1	$\ldots$	t_j	$\mathbf{q_a}$	$\mathbf{t_{j+1}}$	$\ldots$	$t_{n"}$	$\square$	$\ldots$	$\square$

All the entries in our matrix T are elements in the set Q (states) $\cup$ Γ (tape symbols).

Consider now Turing Machine A running on an input string ω. A decides ω if and only if there exists a complete configuration beginning with the starting state $q_s$and terminating in an accepting state q_a.

We are going to build a boolean expression ϕ which represents in propositional logic the $n^k \times n^k$matrix T above! The boolean variables for our expression ϕ will be $x_{i,j,s}$ where:

$$x_{i,j,s} = \begin{cases} 1 & \text{if } T[i,j] = s \\ 0 & \text{if not} \end{cases}$$

We will build our complete boolean expression ϕ in four pieces:

$$\phi = \phi_{start} \wedge \phi_{accept} \wedge \phi_{cell} \wedge \phi_{step}$$

$$\phi_{start} = x_{1,1,q_s} \wedge x_{1,1,\omega_1} \wedge \ldots \wedge x_{1,n+1,\omega_n} \wedge x_{1,n+2,\square} \wedge \ldots \wedge x_{1,n^k,\square}$$

$$\phi_{accept} = \vee_{1 \le i,j \le n^k}\ x_{i,j,q_a}$$

Note that both ϕ_{start} and ϕ_{accept} are conjunctive normal form! ϕ_{start} is the conjunction of numerous single clauses. ϕ_{accept} is just one very long clause.

$$\phi_{cell} = \wedge_{1 \le i,j \le n^k}\ \{(\vee_{s \in Q \cup \Gamma}\ x_{i,j,s}) \wedge\ (\wedge_{s \ne t}\ (\neg x_{i,j,s} \vee \neg x_{i,j,t}))\}$$

Note that ϕ_{cell} is stating that exactly one symbol appears in any given cell. And although it is a bit of an optical illusion, ϕ_{cell} is also conjunctive normal form.

We now come to the crux of the issue in creating the last boolean expression – identifying valid transitions from one configuration to the next in the matrix T.

$\phi_{step} = \wedge_{1 \le i < n^k, 1 < j < n^{k-2}}$ legal 2×3window into T at location (i,j)

Legal windows describe valid transitions from one line in the matrix T to the next. The vast majority of 2×3 windows would be legal windows of the form:

$$\begin{array}{ccc} a & b & c \\ a & b & c \end{array}$$

If no state symbol appears nearby on the first line, then no changes will occur in that area of the tape!

So let us consider the possibilities for 2×3 windows where a state symbol is nearby. We augment the first line with an ***additional*** tape cell on the left and on the right:

$$x \quad a \quad b \quad c \quad y$$

If x is a state symbol, then
$\delta(x, a) = a'Rq$ generates a second row $q \quad b \quad c$

If a is a state symbol, then
$\delta(a, b) = b'Lq$ generates a second row $x \quad b' \quad c$
$\delta(a, b) = b'Rq$ generates a second row $b' \quad q \quad c$
$\delta(a, b) = b'Sq$ generates a second row $q \quad b' \quad c$

If b is a state symbol, then
$\delta(b, c) = c'Lq$ generates a second row $q \quad a \quad c'$
$\delta(b, c) = c'Rq$ generates a second row $a \quad c' \quad q$
$\delta(b, c) = c'Sq$ generates a second row $a \quad q \quad c'$

If c is a state symbol, then
$\delta(c, y) = y'Lq$ generates a second row $a \quad q \quad b$
$\delta(c, y) = y'Rq$ generates a second row $a \quad b \quad y'$
$\delta(c, y) = y'Sq$ generates a second row $a \quad b \quad q$

If y is a state symbol, then
$\delta(y, *) = *Lq$ generates a second row $a \quad b \quad q$

If the following is a 2×3 window into the matrix T located at (i,j) and it is a valid transition,

$$\begin{matrix} a & b & c \\ d & e & f \end{matrix}$$

Then it is represented by the boolean expression:

$$x_{i,j,a} \wedge x_{i,j+1,b} \wedge x_{i,j+2,c} \wedge x_{i+1,j,d} \wedge x_{i+1,j+1,e} \wedge x_{i+1,j+2,f}$$

Thus we come to the following:

$$\phi_{step} = \wedge_{1 \leq i < n^k, 1 < j < n^{k-2}}$$
$$\{x_{i,j,a} \wedge x_{i,j+1,b} \wedge x_{i,j+2,c} \wedge x_{i+1,j,d} \wedge x_{i+1,j+1,e} \wedge x_{i+1,j+2,f}\}$$

Note that ϕ_{step} is conjunctive normal form! ϕ_{step} is the conjunction of numerous single clauses.

Since

$$\phi = \phi_{start} \wedge \phi_{accept} \wedge \phi_{cell} \wedge \phi_{step}$$

and each component is in conjunctive normal form, the boolean expression ϕ is in conjunctive normal form.

Furthermore,

ϕ_{start}	requires $\bigcirc(n^k)$ steps
ϕ_{accept}	requires $\bigcirc(n^{2k})$ steps
ϕ_{cell}	requires $\bigcirc(n^{4k})$ steps
ϕ_{step}	requires $\bigcirc(n^{2k})$ steps

Thus our reduction from A $\in$ NP to SAT is done in polynomial time!

Hence

$$\text{A} \leq_P \text{SAT}$$

□

6.4 Additional NP Complete Problems

We concluded the last section by showing that SAT is NP complete. So why on earth would we be interested in finding other NP complete problems? The answer is relatively simple – the more problems that are NP complete, the greater the opportunities are that one of them might prove to be deterministic polynomial time.

So the research community in the field of computer science continues to look for new NP complete problems. This chapter will certainly not provide an exhaustive list of such problems, but it will provide some illustrative examples and verify that they are also NP complete.

3_SAT

3_SAT is SAT with one additional restriction: each clause must be a 3-clause, i.e., each clause must contain exactly three literals.

As with SAT we are confronted with generating the possible truth values for the variables in the expression. Hence, 3_SAT $\in$ NP.

We will now show that SAT $\leq_P$ 3_SAT. For each clause in a SAT boolean expression, substitute the boolean expression comprised of 3-clauses:

SAT expression	**equivalent 3_SAT expression**
a	$(a \vee a \vee a)$
$(a \vee b)$	$(a \vee a \vee b)$
$(a \vee b \vee c)$	$(a \vee b \vee c)$
$(a \vee b \vee c \vee d)$	$(a \vee b \vee x) \wedge (\neg x \vee c \vee d)$
$(a \vee b \vee c \vee d \vee e)$	$(a \vee b \vee x) \wedge (\neg x \vee c \vee y) \wedge (\neg y \vee d \vee e)$
if $k > 3$	then $k - 2$ 3-clauses!

All these substitutions are simple and straight-forward, done easily in polynomial time.

It should be readily apparent that if ϕ is conjunctive normal form and ϕ' is its equivalent 3-clause form, then ϕ is satisfiable if and only if ϕ' is satisfiable.

Therefore SAT $\leq_P$ 3_SAT.

3_SAT is an especially useful problem to prove other problems are NP complete – as we shall see in our remaining examples.

CLIQUE

Suppose G is an undirected graph, i.e., G is a collection of nodes (vertices) , { $n_1, n_2, \ldots n_p$ }, together with a collection of edges (arcs), { e_{ij} connecting n_i and n_j }. Since the graph is undirected, e_{ij} may be traversed in either direction (node n_i to node n_j or node n_j to node n_i).

Definition 6.4.1. A **k-clique** in the graph G is a subset of nodes C containing exactly k nodes such that for any two nodes n_i and n_j in C there is an edge joining the two.

CLIQUE

Given an undirected graph G
    does there exist an integer k such that G contains a k-clique?

Since a potential k-clique C is a subset of nodes in the set $\{n_1, n_2, \ldots n_p\}$. There are 2^p possible subsets to consider. Hence, CLIQUE $\in$ NP.

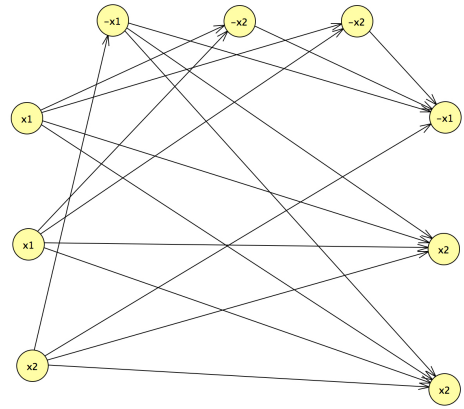
We will now show that 3_SAT $\leq_P$ CLIQUE.

For any boolean expression in 3_SAT each clause has exactly 3 literals. Suppose there are k clauses.

For each of the 3 literals in each clause, we create a single node **n** in the graph G. For each pair of nodes in the graph G, we create an undirected edge **e** with these exceptions:

- no edges between nodes created from the same clause
- no edges between a node representing a literal and a node representing its negative

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$



The transformation described above is a polynomial time reduction from 3_SAT to CLIQUE.

($\Rightarrow$)

If the boolean expression $\phi \in$ 3_SAT, then at least one literal is true in each of its clauses. Choose one of them! The k nodes you have selected form a k-clique for the graph G.

($\Leftarrow$)

If the graph G created by the above technique has a k-clique, then no two nodes in the k-clique come from the same clause! Hence, no edge can be between nodes from the same clause. For each node in the k-clique consider its underlying literal:

- if the literal is x_i, set x_i to TRUE
- if the literal is $\neg x_i$, set x_i to FALSE

Remember, no edges exist between a node representing a literal and a node representing its negative!

Hence $\phi \in$ 3_SAT if and only if G $\in$ CLIQUE.

Therefore 3_SAT $\leq_P$ CLIQUE.

VERTEX_COVER

Suppose G is an undirected graph, i.e., G is a collection of nodes (vertices) , { $n_1, n_2, \ldots n_p$ }, together with a collection of edges (arcs), { e_{ij} connecting n_i and n_j }. Since the graph is undirected, e_{ij} may be traversed in either direction (node n_i to node n_j or node n_j to node n_i).

Definition 6.4.2. A **vertex cover** for the graph G is a subset of nodes C such that if nodes in C are removed from the graph G together with any arc connecting to C, no edges remain in the resulting subgraph.

VERTEX_COVER

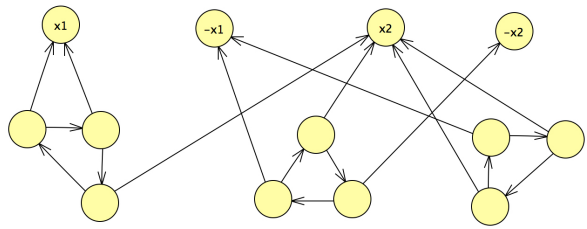
Given an undirected graph G
    does G contain a vertex cover?

Since a potential vertex cover C is a subset of nodes in the set $\{n_1, n_2, \ldots n_p\}$. There are 2^p possible subsets to consider. Hence, VERTEX_COVER $\in$ NP.

We will now show that 3_SAT $\leq_P$ VERTEX_COVER. For any boolean expression in 3_SAT each clause has exactly 3 literals. Suppose there are **p** distinct variables and **q** clauses.

1. For each of the **p** variables, we create a pair of nodes in the graph G, the first representing the literal x_i and the second representing the literal $\neg x_i$. Each such pair is joined with an edge.
2. For each of the **q** clauses, we create a triple of nodes in the graph G, each representing one of the three literals in the clause. Each such triple has three edges between all possible pairs.
3. For each node that was created in 2 connect it to the appropriate node that was created in 1
4. Lastly, set k = p + 2 q.

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$



The transformation described above is a polynomial time reduction from 3_SAT to VERTEX_COVER.

($\Rightarrow$)

If the boolean expression $\phi \in$ 3_SAT, then at least one literal is true in each of its clauses. Choose one of them! Put the variable node that matches our selected literal and was created in (1) into our vertex cover. Put the other two nodes in the triple nodes created in (2) into our vertex cover. Then our vertex cover obviously works and k = p + 2q.

($\Leftarrow$)

If the graph G created by the above technique has a vertex cover with k nodes. Considering the simplicity of our graph construction, the vertex cover must include exactly one variable node (1) and two of every three triple nodes (2). This accounts for all the nodes in the vertex cover.

For each variable node in the vertex cover consider its underlying literal:

- if the literal is x_i, set x_i to TRUE
- if the literal is $\neg x_i$, set x_i to FALSE

Hence $\phi \in$ 3_SAT if and only if G $\in$ VERTEX_COVER.

Therefore 3_SAT $\leq_P$ VERTEX_COVER.

SUBSET_SUM

SUBSET_SUM

Given a collection S of integer values $\{ x_1, x_2, \ldots, x_k \}$ and an integer target **t**,

is it possible to choose a subset of S so that their sum is the value **t**?

Comment. The collection S is not a set in the usual sense (i.e., no duplicate entries), but rather what is called a *multi-set* in which duplicate values are permitted but considered separate items.

Since a potential set of choices is a subset of $\{ x_1, x_2, \ldots, x_k \}$. There are 2^k possible subsets to consider. Hence, SUBSET_SUM $\in$ NP.

We will now show that 3_SAT $\leq_P$ SUBSET_SUM. However, we will generalize the problem from simple integer values to integer tuples of a fixed size:

$$x = (x_1, x_2, \ldots, x_p)$$

And our target will also be an integer tuple of the same size:

$$t = (t_1, t_2, \ldots, t_p)$$

For any boolean expression in 3_SAT each clause has exactly 3 literals. Suppose there are **p** distinct variables and **q** clauses. We construct a 2 p + 2 q + 1 set of (p + q) tuples as represented in the following matrix.

- The upper left hand corner of the matrix represents the truth values for the **p** variables – hence 2 p rows.
- The lower right hand corner of the matrix represents slack variables – 2 for each clause. Remember a 3 clause may contain repeated literals!
- The lower left hand corner of the matrix simply contains all zeros.
- The upper right hand corner of the matrix represents the literals in each clause – distinguish between the literal x_i and

$\neg x_i$; identify one, two, or three distinct literals as appropriate to the clause.

- The target row at the bottom of the matrix indicates (1) each variable is either TRUE or FALSE and (2) each clause contains three literals.

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

	x1	x2	c1	c2	c3
x1	1	0	1	0	0
¬x1	1	0	0	1	1
x2	0	1	1	0	1
¬x2	0	1	0	1	0
c1	0		1	0	0
c1'			1	0	0
c2			0	1	0
c2'			0	1	0
c3			0	0	1
c3'			0	0	1
t	1	1	3	3	3

The transformation described above is a polynomial time reduction from 3_SAT to SUBSET_SUM.

($\Rightarrow$)

If the boolean expression $\phi \in$ 3_SAT, then for each variable if x_i is TRUE we select the x_i tuple or if x_i is FALSE we select the $\neg x_i$ tuple. So far your clause total will be between one and three! Select the necessary slack variable tuples to bring the total to three.

We have selected the appropriate tuples from the collection.

($\Leftarrow$)

If a subset T of the tuples in the above collection S add up to the target t at the bottom, then its underlying boolean expression ϕ is satisfiable:

- If the tuple row for x_i is in the subset T, then set x_i to TRUE otherwise set x_i to FALSE; the column total for each variable is exactly one, so this assignment is well-defined
- Since there are only two slack variables in each clause column and the column total for each clause is exactly three, at least one literal is true by our TRUE / FALSE assignment strategy above

Hence $\phi \in$ 3_SAT if and only if G $\in$ SUBSET_SUM.

Therefore 3_SAT $\leq_P$ SUBSET_SUM.

Closing Comments

We have come to the end of our tour! We started this class with a question: *What was Turing trying to accomplish when he wrote his paper in 1936?*

A natural consideration at the end of our journey would be to ask ourselves another question: *What did Turing actually accomplish?*

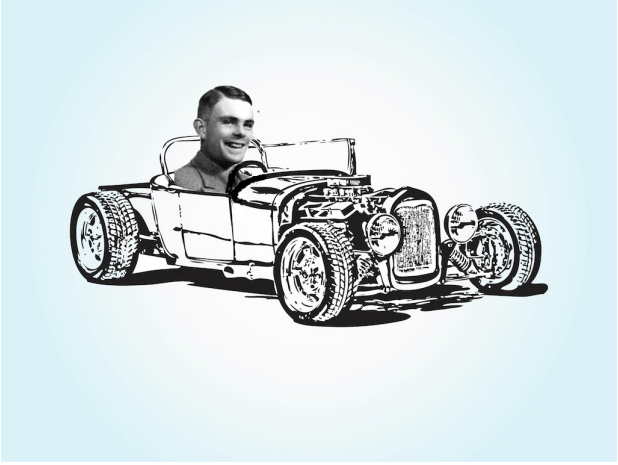
Every topic we have discussed in this class is a direct descendant from Turing's seminal paper. Many contributors have expanded the germ of an idea first envisioned by a young mathematician and have broadened its scope and content into topics such as formal languages, finite automata, regular expressions, grammars, time complexity, and space complexity. And these only scratch the surface! **Chapter 7: Topics for Further Study** lists even more areas of study that comprise the field of Computation Theory.

If I have not convinced you that Alan Turing is one of the most significant individuals during the twentieth century, then let me also mention that a second paper written in 1950 made him the *Father of Artificial Intelligence*. And between the writing of these two papers, Turing worked with the code breakers at Bletchley Park, where he and his colleagues built the *bombe*s that broke the German Enigma Machine!

So, I am hopeful that your personal Tour with Turing is only just beginning. I might suggest you continue by reading the biography *Alan Turing: The Enigma*, by Andrew Hodges and published by Princeton University Press.

Chapter 7

Topics for Further Study



The following items are suggested as they might be of interest to readers after completing these notes. They build on the foundation presented here and delve into new topics that either broaden one's perspective or investigate deeper into the details.

7.1 Finite Automata Regular Languages

Finite Automata augmented with an output device:

- **Moore Machines**
  output generated when executing a particular transition
- **Mealy Machines**
  output generated when entering a particular state

7.2 Push Down Automata Context Free Grammars

Deterministic techniques to facilitate the parsing of grammars:

- **LL (k)**
  left-to-right processing
  left-most derivation
  k symbol look-ahead
  top-down analysis
- **LR (k)**
  left-to-right processing
  right-most derivation
  k symbol look-ahead
  bottom-up analysis

7.3 Turing Machines

Turing Machines incorporating additional features:

- **Enumerators**
  Turing Machines augmented with a printer
- **Linear Bounded Automata / Context Sensitive Languages**
  Turing Machines that execute within the space of the input tape

7.4 Space Complexity

Turing Machines with an input tape and an output tape.
Space complexity for functions $s(n) \leq n$:

- **class L**
  $= \text{DSPACE}(\log_2(n))$
- **class NL**
  $= \text{NSPACE}(\log_2(n))$

7.5 Other Completeness Categories

Class NP is not the only context in which **completeness** can be discussed. The concept of the **hardest problem** in a given class is useful across a wide spectrum.

7.6 Lambda Calculus

The lambda calculus is a formal system for expressing computation by way of variable binding and substitution.

First formulated by Alonzo Church in the 1930s, the lambda calculus was used to answer Hilbert's Entscheidungsproblem.

7.7 Unlimited Register Machines

An unlimited register machine has an infinite number of locations, called registers, which can store natural numbers.

Any given program on a URM may make use of only a finite number of such registers. However,

- there is no upper bound to the number of registers a program may use
- there is no uper bound to the size of the natural numbers that can be stored

7.8 Recursive Functions

μ-recursive functions are the functions that can be computed by Turing Machines

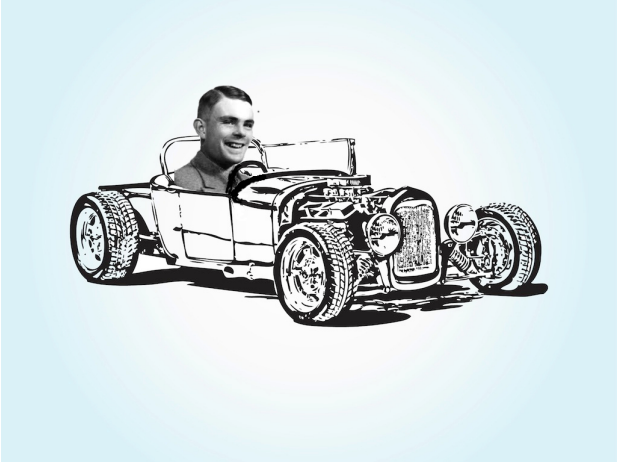
μ-recursive functions build upon the concept of primitive recursive functions

The most famous μ-recursive function which is not a primitive recursive function is the Ackermann function

$$A(x,y) = \begin{cases} y+1 & if x = 0 \\ A(x-1,1) & if y = 0 \\ A(x-1, A(x, y-1)) & otherwise \end{cases}$$

Chapter 8

Check Your Understanding



Chapter One

Question 1:

For each of the binary expansions, below, determine the *numeric value* for the expansion as a fraction **and** define a Turing Machine using JFLAP which *computes* that number:

- x = 0.001000000 ... terminates after 3 digits
- x = 0.100100000 ... terminates after 4 digits
- x = 0.100100100 ... repeating group of three
- x = 0.110110110 ... repeating group of three

For this problem, define your Turing Machine as Turing would have done! F-cells and E-cells and the two-character sentinel at the start of the tape. Generate the binary expansion to the right of the binary point (start cell). Remember to write 0s and 1s only to F-cells.

Question 2:

For the third item in Question 1 above:

- determine the standard descriptor (SD) for the Turing Machine
- determine the numeric descriptor (ND) for the Turing Machine

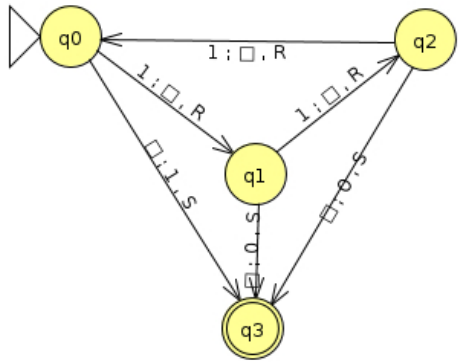
Question 3:

Again for the third item in Question 1 above:

beginning with the initial tape configuration being $q_0\square$, determine the increasingly long tape contents resulting from **each** of the first ten (10) steps of the Turing Machine. List the eleven tape configuration in order from first to last.

Question 4:

Consider the following state diagram for a simple Turing Machine.



- Determine what this Turing Machine is doing?
  consider input strings 111 and 11111
- Write out the transition table δ for this Turing Machine

Question 5:

For the Turing Machine in Question 4 above:

- determine the standard descriptor (SD)
- determine the numeric descriptor (ND)

Chapter Two

Question 6:

Using JFLAP, find the state diagram for the deterministic finite automaton M defined by:

- Q = q_1, q_2, q_3, q_4, q_4
- Σ = u, d
- $q_0 = q_3$
- F = q_3

(continued next page)

-

δ	u	d
q_1	q_0	q_2
q_2	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_5
q_5	q_4	q_5

Question 7:

Using JFLAP, find the state diagram for the deterministic finite automaton M that accepts/recognizes the language

L = { 01101 } a single word

Question 8:

Using JFLAP, find the state diagram for the deterministic finite automaton M that accepts/recognizes the language

L = { *any* string of 0s and 1s | even number of 0s }

Question 9:

Using JFLAP, find the state diagram for the deterministic finite automaton M that accepts/recognizes the language

L = { *any non-empty* string of 0s and 1s | even number of 0s }

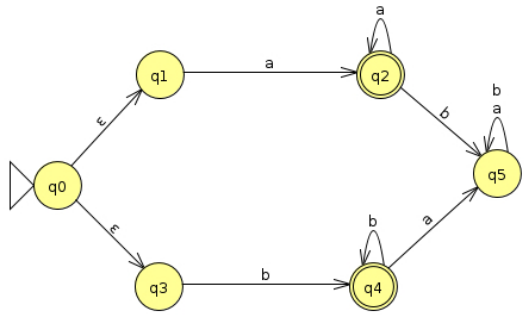
Question 10:

For the state diagram in Question 9 above:

Write out the formal definition for this deterministic finite automaton as a 5-tuple M = (Q, Σ, δ, q_0, F).

Question 11:

The following graphic is the state diagram for a nondeterministic finite automaton M.



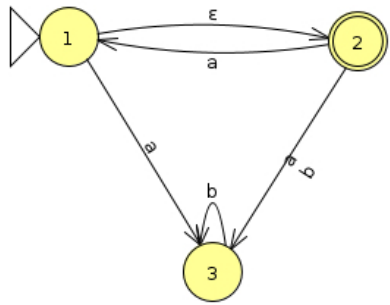
Determine the language L = L(M).

Question 12:

Prove that every nondeterministic finite automaton M which has ***multiple accept states*** can be easily converted to an equivalent nondeterministic finite automaton M' which has only a ***single accept state***.

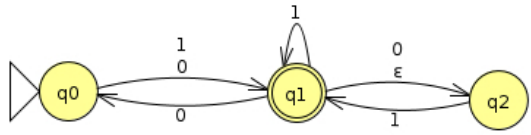
Question 13:

Using JFLAP, convert the following state diagram for the given nondeterministic finite automaton M to the state diagram for an equivalent deterministic finite automaton M'.



Question 14:

Using JFLAP, convert the following state diagram for the given nondeterministic finite automaton M to the state diagram for an equivalent deterministic finite automaton M'.



Question 15:

Suppose L is a language. We can define a related language

$$L^R = \{ \omega^R \mid \omega \in L \}$$

where ω^R represents the original word ω in ***reverse order***. For example,

- the word **cat** $\in L^R$ if and only if the word **tac** $\in L$
- the word **abra** $\in L$ if and only if the word **arba** $\in L^R$

Prove the following two statements:

If L is a regular language, then L^R is also a regular language.

If L^R is a regular language, then L is also a regular language.

Note: This is a very easy proof if you see the trick!

Question 16:

If L is a regular language, then L^C is also a regular language.

We proved the above statement by **reversing** the roles of the accept and reject states in the deterministic finite automaton accepting or recognizing the language L.

Find a nondeterministic finite automaton M where this simple technique of **reversing** accept and reject states **does not** accept the language $L(M)^C$

Question 17:

Suppose L is a regular language. Then L = L(M) where M is some deterministic finite automaton. Consider the language

$$L^E = \{ \omega \in L \text{ and the length of } \omega \text{ is even } \}$$

Prove that the language L^E is also a regular language.

Question 18:

Suppose L is a regular language. Then L = L(M) where M is some deterministic finite automaton. Consider the language

$$\text{PREFIX(L)} = \{ x \mid \exists \text{ y such that x y } in \text{ L } \}$$

Prove that the language PREFIX(L) is also a regular language.

Question 19:

Determine a nondeterministic finite automaton M which accepts or recognizes the language L associated with the following regular expression:

$$0 (01 + 001)^* 1$$

Question 20:

For **EACH** of the following regular expressions, determine two words that **ARE** in the language of that regular expression and determine two words that **ARE NOT** in the language.

- a* b*
- a (ba)* b
- (ϵ + a) b

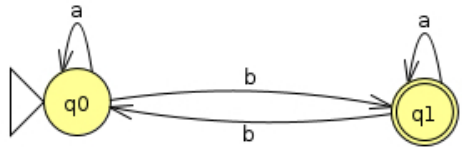
Question 21:

For **each** of the following regular expressions, determine an equivalent (but much simpler) regular expression which accepts/recognizes the same language.

- (r + ϵ)*
- s s* + ϵ
- (ϵ + r) (r + s)* (s + ϵ)
- (r + s + rs + sr)*

Question 22:

Determine the regular expression which defines the language accepted/recognized by the follow deterministic finite automaton:



Question 23:

Prove that the language

$$L = \{ \omega \in \{0,1\}^* \mid \omega \text{ is a palindrome } \}$$

is **not** a regular language.

Question 24:

Prove that the language

$$L = \{ \omega\omega \mid \omega \in \{0,1\}^* \}$$

is **not** a regular language.

Question 25:

Is the following language regular? YES or NO

$$L = \{ \omega \in \{0,1\}^* \mid \omega \text{ is } \textbf{not} \text{ a palindrome} \}$$

Question 26:

Prove that the language $L = \{ \omega = a\ a\ \ldots\ a = a^p \mid p \text{ is a prime number} \}$ is **not** a regular language.

Chapter Three

Question 27:

Using JFLAP, describe the language L(G) defined by the following grammar:

- S → a S a
- S → b S b
- S → c S c
- S → a
- S → b
- S → c
- S → ϵ

Question 28:

Using JFLAP, describe the language L(G) defined by the following grammar:

- S → 0 S 1
- S → 1 S 0
- S → S S this will cause JFLAP to go into an infinite loop! Why?
- S → ϵ

Question 29:

Using JFLAP, convert the following **context free grammar** into an equivalent grammar in **Chomsky Normal Form**:

- S → A S a | a B
- A → B | S
- B → b | ϵ

Question 30:

Using JFLAP, convert the following **context free grammar** into an equivalent grammar in **Chomsky Normal Form**:

- S $\rightarrow$ a X b X
- X $\rightarrow$ a Y | b Y | ϵ
- Y $\rightarrow$ X | c

Question 31:

G is a context free grammar which is in **Chomsky Normal Form**. Prove the following statement using ***mathematical induction*** on the length of the input string ω, $|\omega|$ = n.

If $\omega \in$ L(G) and the length of ω is n $\geq$ 0, then *exactly* 2n - 1 steps are required for any derivation of ω.

Note: This question is more mathematical in nature and requires significantly more thought.

Question 32:

Consider the language

$$L = \{ \omega \in \{0,1\}^* \mid \omega \text{ contains at least 3 1s } \}$$

- Determine a context free grammar for the language L.
- Determine a push down automaton which accepts/recognizes the language L.

Question 33:

Consider the language

$$L = \{ \omega \in \{0,1\}^* \mid |\omega| \text{ is odd and middle symbol is 0 } \}$$

- Determine a context free grammar for the language L.
- Determine a push down automaton which accepts/recognizes the language L.

Question 34:

Consider the language

$$L = \{ \omega = a^m\ b^n\ c^k \mid m,n,k \geq 0 \text{ and } k = m+n \}$$

- Determine a context free grammar for the language L.
- Determine a push down automaton which accepts/recognizes the language L.

Question 35:

Convert the **context free grammar** G found in the file ***QUESTION_35.JFF*** into a **nondeterministic push down automaton**.

Remember: There are multiple options to convert a CFG to a NPDA. Use the option *Convert CFG to PDA (LL1)* to answer this question. Also, **look** at the result! The actual grammar should be an obvious component in the resulting NPDA.

Question 36:

Convert the **nondeterministic push down automaton** M found in the file **QUESTION_36.JFF** into a **context free grammar**.

Remember: The option *Convert to Grammar* typically requires that the NPDA is modified slightly.

- create a new final node with incoming transition ϵ , $Z \rightarrow \epsilon$
- JFLAP requires that for each transition, one element is POPped off the stack
  and either 0 or 2 elements are PUSHed onto the stack

Question 37:

Prove that the language

$$L = \{ a^n\ b^{2n}\ c^{3n} \mid n \geq 0 \}$$

is **not** context free.

Question 38:

Prove that the language

$$\mathrm{L} = \{\ \mathrm{a}^n\ \mathrm{b}^{2n}\ \mathrm{a}^n \mid \mathrm{n} \geq 0\ \}$$

is **not** context free.

Question 39:

Suppose L_1 is a **regular language** and L_2 is a **context free grammar**.

Prove that $\mathrm{L}_1 \cap \mathrm{L}_2$ is a **context free language**.

Hint: What is the basic difference between a simple finite automaton and a push down automaton?

Question 40:

Suppose L is a **context free language**. Consider the language

$$\mathrm{L}^R = \{\ \omega^R \mid \omega \in \mathrm{L}\ \}$$

generated by *reversing* all the words in L.

Hint: First, visualize the grammar for the language L. Then, try to visualize the grammar for the language L^R.

Chapter Four

Question 41:

Using JFLAP, develop a state diagram for a Turing Machine M which accepts the language

$$L = \{ \omega = 0^n \ 1^n \ 2^n \mid n \geq 0 \}$$

Test your Turing Machine using *Input: Multiple Run.*

Question 42:

Using JFLAP, develop a state diagram for a Turing Machine M which accepts the language

$$L = \{ \omega \in \{0,1\}^* \mid |\omega| \text{ is odd middle symbol is } 0 \}$$

Test your Turing Machine using *Input: Multiple Run.*

Question 42:

Using JFLAP, develop a state diagram for a Turing Machine M which **shifts** a **binary** input string **one position to the right** but NEVER moves the print head toward the **left** in the process.

Test your Turing Machine using *Input: Step.* The Turing Machine in question is performing a specific task; the end result is an output string and **not** an ACCEPT/REJECT state.

Question 44:

Using JFLAP, develop a state diagram for a Turing Machine M which ***reverses*** a **binary** input string. The resulting binary output string need **not** be in the same location as the original input. The original input *may be* modified and/or erased in the process.

Test your Turing Machine using *Input: Step.* The Turing Machine in question is performing a specific task; the end result is an output string and **not** an ACCEPT/REJECT state.

Question 45:

Consider a **deterministic** Turing Machine with the standard *single tape* for input and output. In very general terms, describe how such a machine might handle the following problem:

A simple input tape contains a unary number followed by a pound sign (#). The Turing Machine is to convert the input into its binary number equivalent (generated left-to-right from least significant bit to most significant bit. The resulting binary pattern should be the output.

For example, 1111111111 (unary) $\longrightarrow$ 0101 (binary).

Now, consider a deterministic Turing Machine which has *two tapes*: one for input and another for output. Again, in very general terms, describe how such a machine might handle the same problem.

Hint: Can you develop a simple set of rules to increment a binary number by one, remembering to process them from left to right!

$$\begin{aligned} 00101 &\longrightarrow 10101 \\ 11101 &\longrightarrow 00011 \\ 01011 &\longrightarrow 11011 \\ 10101 &\longrightarrow 01101 \end{aligned}$$

Question 46:

Consider a **nondeterministic** Turing Machine with the standard *single tape* for input and output. In very general terms, describe how such a machine might handle the following problem:

A simple input tape contains two finite strings from the alphabet $\Sigma = \{0,1\}$ separated by a pound sign (#) and terminated by a pound sign (#). The Turing Machine is to determine whether the first string is a substring of the second.

ACCEPT indicates that it is; REJECT indicates that it is not.

Now, consider a nondeterministic Turing Machine with *two tapes*, both for input. Again, in very general terms, describe how such a machine might handle the same problem.

Question 47:

A ***nervous*** Turing Machine is one which must move either LEFT or RIGHT at every transition; it may **not** remain STATIONARY!

Prove that nervous Turing Machines are equivalent in power to ordinary (calm?) Turing Machines.

Question 48:

Consider a deterministic finite automaton M. A state **q** is said to be ***useful*** provided there is an input string ω such that M actually enters state **q** while processing the string ω.

Show that the language

$$\mathbf{USEFUL}_{DFA} = \{ \text{<M,q>} \mid \text{M is a DFA, q is useful} \}$$

is **decidable**.

Question 49:

Consider a deterministic finite automaton M. The language L(M) that it accepts can be either finite or infinite.

Show that the language

$$\mathbf{INFINITE}_{DFA} = \{ \text{<M>} \mid \text{L(M) is infinite} \}$$

is **decidable**.

Question 50:

Prove: If **A** is recursively enumerable and $\mathbf{A} \leq \mathbf{A}^C$, then **A** is recursive.

Question 51:

Prove: **A** is recursively enumerable if and only if $\mathbf{A} \leq \mathbf{ACCEPT}_T M$.

Question 52:

Prove: **A** is decidable if and only if $\mathbf{A} \leq \mathbf{0^*\ 1^*}$.

Chapter Five

Question 53:

For each of the following six items, determine whether each of the six statements is TRUE or FALSE.

$2n = O(n)$	$n \log n = O(n^2)$
$n^2 = O(n)$	$3^n = 2^{O(n)}$
$n^2 = O(n \log^2 n)$	$n^{1/2} = O(n^{1/3})$

Question 54:

For each of the following six items, determine whether each of the six statements is TRUE or FALSE.

$n = o(n^2)$	$1 = o(n)$
$2n = o(n^2)$	$n = o(\log n)$
$2^n = o(3^n)$	$1 = o(1/n)$

Question 55:

Suppose f(n) is $O(n^3)$ and g(n) is $O(n^7)$. What is the order of growth for each of the following the following five functions:

addition	f(n) + g(n)
subtraction	f(n) - g(n)
multiplication	f(n) * g(n)
division	f(n) / g(n)
composition	fog(n) = f(g(n))

Question 56:

Is the following boolean formula satisfiable? Justify your answer!

$$\phi = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$$

If you were to delete any **one** of the parenthesized clauses above, would the formula then be satisfiable? Justify your answer!

Question 57:

Is the following boolean formula satisfiable?

$$\phi = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg w \vee x \vee \neg y) \wedge (\neg w \vee \neg x \vee z)$$

Determine the number of satisfying assignment possibilities.

Hint: 4 variables implies $2^4 = 16$ true/false combinations.
Use a truth table!

Question 58:

Provide a clause C' in **three conjunctive normal form** that is equivalent to the following:

$$C = a \vee \neg b \vee c \vee \neg d \vee \neg e \vee f$$

Provide a clause C' in **three conjunctive normal form** that is equivalent to the following:

$$C = g \vee \neg h$$

Question 59:

A **triangle** in an undirected graph is a **3-clique**. Consider the formal language

TRIANGLE = { <G> | G is an undirected graph and G contains a triangle }

Prove that **TRIANGLE** $\in$ P.

Question 60:

Two directed graphs G and H are said to be **isomorphic** if the nodes of G may be reordered/relabled to match the nodes of H and the edges of G also match the resulting edges of H. Consider the formal language

ISO = { <G,H> | G and H are directed graphs and G and H are isomorphic }

Prove that **ISO** $\in$ NP.

Question 61:

A **ladder** is a sequence of strings, s_1, s_2, ..., s_k, such that every string differs from the preceding string in exactly one position. Ffor example, the following is a ladder of English words starting with "head" and ending with "free".

head, hea**r**, **n**ear, **f**ear, **b**ear, be**e**r, **d**eer, dee**d**, **f**eed, fee**t**, f**r**et, fre**e**

Consider the formal language

LADDER$_{DFA}$ = { <M,s,t> | M is a D and L(M) contains a ladder of strings starting with s and ending with t }

Prove that **LADDER**$_{DFA} \in$ PSPACE.

Question 62:

Verify that PSPACE is closed under the following operations:

- union
- concatenation
- Kleene closure

Chapter Six

Question 63:

Euclid's algorithm is one of the oldest algorithms! Euclid's algorithm determines the greatest common divisor of two natural numbers.

GCD (A,B)

- while (B > 0)
  temp = B
  B = A mod B (the remainder of A / B)
  B = temp
- GCD = A

Calculate the greatest common divisor for 120 and 63.

Calculate the greatest common divisor for 1542 and 2963.

Estimate the running time for this algorithm.

Question 64:

Search the Internet for information regarding the **QUICKSORT algorithm** for sorting an array A [1 ... n] into increasing order.

- Write out the algorithm for QUICKSORT.
- What is the running time for this algorithm?

Question 65:

Search the Internet for information regarding the **WARSHALL'S algorithm** for determining the transitive closure of an adjacency matrix for a directed graph.

- Write out the algorithm for WARSHALL.
- What is the running time for this algorithm?

Question 66:

Convert the following boolean expression ϕ in **3_SAT** to its corresponding graph in **CLIQUE**.

$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

Question 67:

Convert the following boolean expression ϕ in **3_SAT** to its corresponding graph in **VERTEX_COVER**.

$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

Question 68:

Convert the following boolean expression ϕ in **3_SAT** to its corresponding tuple representation (i.e., table) in **SUBSET_SUM**.

$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

Question 69:

A graph G is comprised of vertices (nodes) V and edges (arcs) E. In an undirected graph the edges are **bidirectional**; in a directed graph, the edges have a specific starting vertex and a specific ending vertex (often denoted by an arrow). Consider an **undirected** graph G.

A set D is called a **dominating set** in the graph G provided any vertex v in E $\sim$ D is adjacent to at least one node in D.

Consider the formal language

$$\text{DOMINATION} = \{ \langle G \rangle \mid \exists \text{ dominating set D} \}$$

Prove that **DOMINATION** $\in$ NP

Question 70:

Prove that **3_SAT** $\leq$ **DOMINATION**.

Hint: Clauses c become nodes in a graph. Variables v become triangles of three nodes (v,¬ v, v'), where v' is a meaningless dummy item to create the triple.

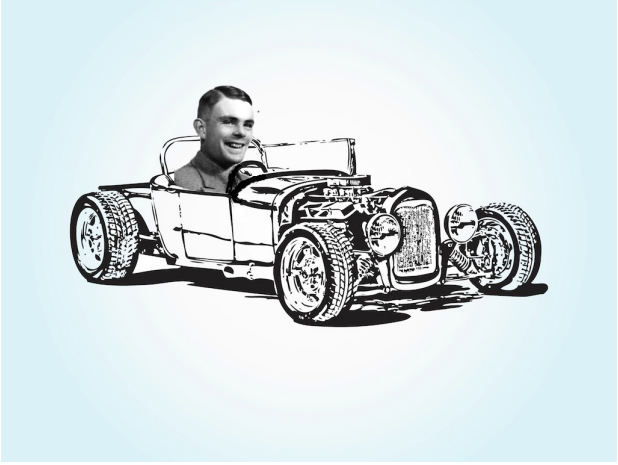
Question 71:

Using the technique above, convert the following boolean expression ϕ in **3_SAT** to its graph in **DOMINATION**.

$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

Chapter 9

Turing's Original Paper



ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

have valuable applications. In particular, it is shown (§ 11) that the Hilbertian Entscheidungsproblem can have no solution.

In a recent paper Alonzo Church† has introduced an idea of "effective calculability", which is equivalent to my "computability", but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem‡. The proof of equivalence between "computability" and "effective calculability" is outlined in an appendix to the present paper.

1. *Computing machines.*

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach § 9. For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called "m-configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the r-th, bearing the symbol $\mathfrak{S}(r)$ which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration q_n and the scanned symbol $\mathfrak{S}(r)$. This pair $q_n, \mathfrak{S}(r)$ will be called the "configuration": thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (*i.e.* bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed. Some of the symbols written down

† Alonzo Church, "An unsolvable problem of elementary number theory", *American J. of Math.*, 58 (1936), 345–363.

‡ Alonzo Church, "A note on the Entscheidungsproblem", *J. of Symbolic Logic*, 1 (1936), 40–41.

will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure.

It is my contention that these operations include all those which are used in the computation of a number. The defence of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by "machine", "tape", "scanned", etc.

2. *Definitions.*

Automatic machines.

If at each stage the motion of a machine (in the sense of § 1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*-machine).

For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in § 1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems. In this paper I deal only with automatic machines, and will therefore often omit the prefix *a*-.

Computing machines.

If an *a*-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m*-configuration, the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine*. The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the *number computed by the machine*.

At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the *m*-configuration will be said to describe the *complete configuration* at that stage. The changes of the machine and tape between successive complete configurations will be called the *moves* of the machine.

Circular and circle-free machines.

If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called *circular*. Otherwise it is said to be *circle-free*.

A machine will be circular if it reaches a configuration from which there is no possible move, or if it goes on moving, and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind. The significance of the term "circular" will be explained in § 8.

Computable sequences and numbers.

A sequence is said to be computable if it can be computed by a circle-free machine. A number is computable if it differs by an integer from the number computed by a circle-free machine.

We shall avoid confusion by speaking more often of computable sequences than of computable numbers.

3. *Examples of computing machines.*

I. A machine can be constructed to compute the sequence 010101 .... The machine is to have the four m-configurations "$\mathfrak{b}$", "$\mathfrak{c}$", "$\mathfrak{f}$", "$\mathfrak{e}$" and is capable of printing "0" and "1". The behaviour of the machine is described in the following table in which "R" means "the machine moves so that it scans the square immediately on the right of the one it was scanning previously". Similarly for "L". "E" means "the scanned symbol is erased" and "P" stands for "prints". This table (and all succeeding tables of the same kind) is to be understood to mean that for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the m-configuration described in the last column. When the second column is left blank, it is understood that the behaviour of the third and fourth columns applies for any symbol and for no symbol. The machine starts in the m-configuration $\mathfrak{b}$ with a blank tape.

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{b}$	None	$P0$, R	$\mathfrak{c}$
$\mathfrak{c}$	None	R	$\mathfrak{e}$
$\mathfrak{e}$	None	$P1$, R	$\mathfrak{f}$
$\mathfrak{f}$	None	R	$\mathfrak{b}$

If (contrary to the description in § 1) we allow the letters L, R to appear more than once in the operations column we can simplify the table considerably.

m-config.	*symbol*	*operations*	*final m-config.*
𝔟	None	$P0$	𝔟
	0	$R, R, P1$	𝔟
	1	$R, R, P0$	𝔟

II. As a slightly more difficult example we can construct a machine to compute the sequence 00101101111011111011111 .... The machine is to be capable of five m-configurations, viz. "𝔬", "𝔮", "𝔭", "𝔣", "𝔟" and of printing "ə", "x", "0", "1". The first three symbols on the tape will be "ə ə 0"; the other figures follow on alternate squares. On the intermediate squares we never print anything but "x". These letters serve to "keep the place" for us and are erased when we have finished with them. We also arrange that in the sequence of figures on alternate squares there shall be no blanks.

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
𝔟		$Pə, R, Pə, R, P0, R, R, P0, L, L$	𝔬
𝔬	1	R, Px, L, L, L	𝔬
	0		𝔮
𝔮	Any (0 or 1)	R, R	𝔮
	None	$P1, L$	𝔭
𝔭	x	E, R	𝔮
	ə	R	𝔣
	None	L, L	𝔭
𝔣	Any	R, R	𝔣
	None	$P0, L, L$	𝔬

To illustrate the working of this machine a table is given below of the first few complete configurations. These complete configurations are described by writing down the sequence of symbols which are on the tape,

with the m-configuration written below the scanned symbol. The successive complete configurations are separated by colons.

: ə ə 0 0 : ə ə 0 0 : ə ə 0 0 : ə ə 0 0 : ə ə 0 0 1 :
ɓ ɒ q q q p
ə ə 0 0 1 : ə ə 0 0 1 : ə ə 0 0 1 : ə ə 0 0 1 :
p p f f
ə ə 0 0 1 : ə ə 0 0 1 : ə ə 0 0 1 0 :
f f ɒ
ə ə 0 0 1 x 0 : ....
ɒ

This table could also be written in the form

ɓ : ə ə ɔ 0 0 : ə ə q 0 0 : ..., (C)

in which a space has been made on the left of the scanned symbol and the m-configuration written in this space. This form is less easy to follow, but we shall make use of it later for theoretical purposes.

The convention of writing the figures only on alternate squares is very useful: I shall always make use of it. I shall call the one sequence of alternate squares F-squares and the other sequence E-squares. The symbols on E-squares will be liable to erasure. The symbols on F-squares form a continuous sequence. There are no blanks until the end is reached. There is no need to have more than one E-square between each pair of F-squares: an apparent need of more E-squares can be satisfied by having a sufficiently rich variety of symbols capable of being printed on E-squares. If a symbol β is on an F-square S and a symbol α is on the E-square next on the right of S, then S and β will be said to be *marked* with α. The process of printing this α will be called marking β (or S) with α.

4. *Abbreviated tables.*

There are certain types of process used by nearly all machines, and these, in some machines, are used in many connections. These processes include copying down sequences of symbols, comparing sequences, erasing all symbols of a given form, etc. Where such processes are concerned we can abbreviate the tables for the m-configurations considerably by the use of "skeleton tables". In skeleton tables there appear capital German letters and small Greek letters. These are of the nature of "variables". By replacing each capital German letter throughout by an m-configuration

and each small Greek letter by a symbol, we obtain the table for an m-configuration.

The skeleton tables are to be regarded as nothing but abbreviations: they are not essential. So long as the reader understands how to obtain the complete tables from the skeleton tables, there is no need to give any exact definitions in this connection.

Let us consider an example:

m-config.	*Symbol*	*Behaviour*	*Final m-config.*
$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$	$\mathfrak{d}$	L	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	not $\mathfrak{d}$	L	$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$
$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$	α		$\mathfrak{C}$
	not α	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	None	R	$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$
$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$	α		$\mathfrak{C}$
	not α	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	None	R	$\mathfrak{B}$

From the m-configuration $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$ the machine finds the symbol of form α which is farthest to the left (the "first α") and the m-configuration then becomes $\mathfrak{C}$. If there is no α then the m-configuration becomes $\mathfrak{B}$.

If we were to replace $\mathfrak{C}$ throughout by $\mathfrak{q}$ (say), $\mathfrak{B}$ by $\mathfrak{r}$, and α by x, we should have a complete table for the m-configuration $\mathfrak{f}(\mathfrak{q}, \mathfrak{r}, x)$. $\mathfrak{f}$ is called an "m-configuration function" or "m-function".

The only expressions which are admissible for substitution in an m-function are the m-configurations and symbols of the machine. These have to be enumerated more or less explicitly: they may include expressions such as $\mathfrak{p}(\mathfrak{e}, x)$; indeed they must if there are any m-functions used at all. If we did not insist on this explicit enumeration, but simply stated that the machine had certain m-configurations (enumerated) and all m-configurations obtainable by substitution of m-configurations in certain m-functions, we should usually get an infinity of m-configurations; *e.g.*, we might say that the machine was to have the m-configuration $\mathfrak{q}$ and all m-configurations obtainable by substituting an m-configuration for $\mathfrak{C}$ in $\mathfrak{p}(\mathfrak{C})$. Then it would have $\mathfrak{q}$, $\mathfrak{p}(\mathfrak{q})$, $\mathfrak{p}\big(\mathfrak{p}(\mathfrak{q})\big)$, $\mathfrak{p}\Big(\mathfrak{p}\big(\mathfrak{p}(\mathfrak{q})\big)\Big)$, ... as m-configurations.

Our interpretation rule then is this. We are given the names of the m-configurations of the machine, mostly expressed in terms of m-functions. We are also given skeleton tables. All we want is the complete table for the m-configurations of the machine. This is obtained by repeated substitution in the skeleton tables.

Further examples.

(In the explanations the symbol "→" is used to signify "the machine goes into the m-configuration. . . .")

$\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$		$\mathfrak{f}\big(\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha), \mathfrak{B}, \alpha\big)$	From $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ the first α is erased and $\rightarrow \mathfrak{C}$. If there is no $\alpha \rightarrow \mathfrak{B}$.
$\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$	E	$\mathfrak{C}$	
$\mathfrak{e}(\mathfrak{B}, \alpha)$		$\mathfrak{e}\big(\mathfrak{e}(\mathfrak{B}, \alpha), \mathfrak{B}, \alpha\big)$	From $\mathfrak{e}(\mathfrak{B}, \alpha)$ all letters α are erased and $\rightarrow \mathfrak{B}$.

The last example seems somewhat more difficult to interpret than most. Let us suppose that in the list of m-configurations of some machine there appears $\mathfrak{e}(\mathfrak{b}, x)$ $(= \mathfrak{q}$, say). The table is

	$\mathfrak{e}(\mathfrak{b}, x)$	$\mathfrak{e}\big(\mathfrak{e}(\mathfrak{b}, x), \mathfrak{b}, x\big)$
or	$\mathfrak{q}$	$\mathfrak{e}(\mathfrak{q}, \mathfrak{b}, x)$.

Or, in greater detail:

	$\mathfrak{q}$	$\mathfrak{e}(\mathfrak{q}, \mathfrak{b}, x)$
	$\mathfrak{e}(\mathfrak{q}, \mathfrak{b}, x)$	$\mathfrak{f}\big(\mathfrak{e}_1(\mathfrak{q}, \mathfrak{b}, x), \mathfrak{b}, x\big)$
	$\mathfrak{e}_1(\mathfrak{q}, \mathfrak{b}, x)$ E	$\mathfrak{q}$.

In this we could replace $\mathfrak{e}_1(\mathfrak{q}, \mathfrak{b}, x)$ by $\mathfrak{q}'$ and then give the table for $\mathfrak{f}$ (with the right substitutions) and eventually reach a table in which no m-functions appeared.

$\mathfrak{pe}(\mathfrak{C}, \beta)$			$\mathfrak{f}\big(\mathfrak{pe}_1(\mathfrak{C}, \beta), \mathfrak{C}, \mathfrak{a}\big)$	From $\mathfrak{pe}(\mathfrak{C}, \beta)$ the machine prints β at the end of the sequence of symbols and $\rightarrow \mathfrak{C}$.
$\mathfrak{pe}_1(\mathfrak{C}, \beta)$	Any	R, R	$\mathfrak{pe}_1(\mathfrak{C}, \beta)$	
	None	$P\beta$	$\mathfrak{C}$	
$\mathfrak{l}(\mathfrak{C})$		L	$\mathfrak{C}$	From $\mathfrak{f}'(\mathfrak{C}, \mathfrak{B}, \alpha)$ it does the same as for $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$ but moves to the left before $\rightarrow \mathfrak{C}$.
$\mathfrak{r}(\mathfrak{C})$		R	$\mathfrak{C}$	
$\mathfrak{f}'(\mathfrak{C}, \mathfrak{B}, \alpha)$			$\mathfrak{f}\big(\mathfrak{l}(\mathfrak{C}), \mathfrak{B}, \alpha\big)$	
$\mathfrak{f}''(\mathfrak{C}, \mathfrak{B}, \alpha)$			$\mathfrak{f}\big(\mathfrak{r}(\mathfrak{C}), \mathfrak{B}, \alpha\big)$	
$\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$			$\mathfrak{f}'\big(\mathfrak{c}_1(\mathfrak{C}), \mathfrak{B}, \alpha\big)$	$\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$. The machine writes at the end the first symbol marked α and $\rightarrow \mathfrak{C}$.
$\mathfrak{c}_1(\mathfrak{C})$	β		$\mathfrak{pe}(\mathfrak{C}, \beta)$	

The last line stands for the totality of lines obtainable from it by replacing β by any symbol which may occur on the tape of the machine concerned.

$ce(\mathfrak{C}, \mathfrak{B}, \alpha)$ $\quad c\left(e(\mathfrak{C}, \mathfrak{B}, \alpha), \mathfrak{B}, \alpha\right)$

$ce(\mathfrak{B}, \alpha)$ $\quad ce\left(ce(\mathfrak{B}, \alpha), \mathfrak{B}, \alpha\right)$

$ce(\mathfrak{B}, \alpha)$. The machine copies down in order at the end all symbols marked α and erases the letters α; $\rightarrow \mathfrak{B}$.

$re(\mathfrak{C}, \mathfrak{B}, \alpha, \beta)$ $\quad f\left(re_1(\mathfrak{C}, \mathfrak{B}, \alpha, \beta), \mathfrak{B}, \alpha\right)$

$re_1(\mathfrak{C}, \mathfrak{B}, \alpha, \beta)$ $\quad E, P\beta$ $\quad \mathfrak{C}$

$re(\mathfrak{B}, \alpha, \beta)$ $\quad re\left(re(\mathfrak{B}, \alpha, \beta), \mathfrak{B}, \alpha, \beta\right)$

$re(\mathfrak{C}, \mathfrak{B}, \alpha, \beta)$. The machine replaces the first α by β and $\rightarrow \mathfrak{C} \rightarrow \mathfrak{B}$ if there is no α.

$re(\mathfrak{B}, \alpha, \beta)$. The machine replaces all letters α by β; $\rightarrow \mathfrak{B}$.

$cr(\mathfrak{C}, \mathfrak{B}, \alpha)$ $\quad c\left(re(\mathfrak{C}, \mathfrak{B}, \alpha, \alpha), \mathfrak{B}, \alpha\right)$

$cr(\mathfrak{B}, \alpha)$ $\quad cr\left(cr(\mathfrak{B}, \alpha), re(\mathfrak{B}, \alpha, \alpha), \alpha\right)$

$cr(\mathfrak{B}, \alpha)$ differs from $ce(\mathfrak{B}, \alpha)$ only in that the letters α are not erased. The m-configuration $cr(\mathfrak{B}, \alpha)$ is taken up when no letters "α" are on the tape.

$cp(\mathfrak{C}, \mathfrak{A}, \mathfrak{E}, \alpha, \beta)$ $\quad f'\left(cp_1(\mathfrak{C}_1 \mathfrak{A}, \beta), f(\mathfrak{A}, \mathfrak{E}, \beta), \alpha\right)$

$cp_1(\mathfrak{C}, \mathfrak{A}, \beta)$ $\quad \gamma$ $\quad f'\left(cp_2(\mathfrak{C}, \mathfrak{A}, \gamma), \mathfrak{A}, \beta\right)$

$cp_2(\mathfrak{C}, \mathfrak{A}, \gamma)$ $\quad \begin{cases} \gamma & \mathfrak{C} \\ \text{not } \gamma & \mathfrak{A}. \end{cases}$

The first symbol marked α and the first marked β are compared. If there is neither α nor β, $\rightarrow \mathfrak{E}$. If there are both and the symbols are alike, $\rightarrow \mathfrak{C}$. Otherwise $\rightarrow \mathfrak{A}$.

$$cpe(\mathfrak{C}, \mathfrak{A}, \mathfrak{E}, \alpha, \beta) \qquad cp\left(e\left(e(\mathfrak{C}, \mathfrak{C}, \beta), \mathfrak{C}, \alpha\right), \mathfrak{A}, \mathfrak{E}, \alpha, \beta\right)$$

$cpe(\mathfrak{C}, \mathfrak{A}, \mathfrak{E}, \alpha, \beta)$ differs from $cp(\mathfrak{C}, \mathfrak{A}, \mathfrak{E}, \alpha, \beta)$ in that in the case when there is similarity the first α and β are erased.

$$cpe(\mathfrak{A}, \mathfrak{E}, \alpha, \beta) \qquad cpe\left(cpe(\mathfrak{A}, \mathfrak{E}, \alpha, \beta), \mathfrak{A}, \mathfrak{E}, \alpha, \beta\right).$$

$cpe(\mathfrak{A}, \mathfrak{E}, \alpha, \beta)$. The sequence of symbols marked α is compared with the sequence marked β. $\rightarrow \mathfrak{E}$ if they are similar. Otherwise $\rightarrow \mathfrak{A}$. Some of the symbols α and β are erased.

$\mathfrak{q}(\mathfrak{C})$	Any	R	$\mathfrak{q}(\mathfrak{C})$	$\mathfrak{q}(\mathfrak{C}, \alpha)$. The machine finds the last symbol of form α. $\rightarrow \mathfrak{C}$.
	None	R	$\mathfrak{q}_1(\mathfrak{C})$	
$\mathfrak{q}_1(\mathfrak{C})$	Any	R	$\mathfrak{q}(\mathfrak{C})$	
	None		$\mathfrak{C}$	
$\mathfrak{q}(\mathfrak{C}, \alpha)$			$\mathfrak{q}\left(\mathfrak{q}_1(\mathfrak{C}, \alpha)\right)$	
$\mathfrak{q}_1(\mathfrak{C}, \alpha)$	α		$\mathfrak{C}$	
	not α	L	$\mathfrak{q}_1(\mathfrak{C}, \alpha)$	
$\mathfrak{pe}_2(\mathfrak{C}, \alpha, \beta)$			$\mathfrak{pe}\left(\mathfrak{pe}(\mathfrak{C}, \beta), \alpha\right)$	$\mathfrak{pe}_2(\mathfrak{C}, \alpha, \beta)$. The machine prints $\alpha\ \beta$ at the end.
$\mathfrak{ce}_2(\mathfrak{B}, \alpha, \beta)$			$\mathfrak{ce}\left(\mathfrak{ce}(\mathfrak{B}, \beta), \alpha\right)$	$\mathfrak{ce}_3(\mathfrak{B}, \alpha, \beta, \gamma)$. The machine copies down at the end first the symbols marked α, then those marked β, and finally those marked γ; it erases the symbols α, β, γ.
$\mathfrak{ce}_3(\mathfrak{B}, \alpha, \beta, \gamma)$			$\mathfrak{ce}\left(\mathfrak{ce}_2(\mathfrak{B}, \beta, \gamma), \alpha\right)$	
$\mathfrak{e}(\mathfrak{C})$	ə	R	$\mathfrak{e}_1(\mathfrak{C})$	From $\mathfrak{e}(\mathfrak{C})$ the marks are erased from all marked symbols. $\rightarrow \mathfrak{C}$.
	Not ə	L	$\mathfrak{e}(\mathfrak{C})$	
$\mathfrak{e}_1(\mathfrak{C})$	Any	R, E, R	$\mathfrak{e}_1(\mathfrak{C})$	
	None		$\mathfrak{C}$	

5. *Enumeration of computable sequences.*

A computable sequence γ is determined by a description of a machine which computes γ. Thus the sequence 001011011101111... is determined by the table on p. 234, and, in fact, any computable sequence is capable of being described in terms of such a table.

It will be useful to put these tables into a kind of standard form. In the first place let us suppose that the table is given in the same form as the first table, for example, I on p. 233. That is to say, that the entry in the operations column is always of one of the forms $E : E, R : E, L : P\alpha : P\alpha, R : P\alpha, L : R : L :$ or no entry at all. The table can always be put into this form by introducing more m-configurations. Now let us give numbers to the m-configurations, calling them $q_1, \ldots, q_R$, as in § 1. The initial m-configuration is always to be called q_1. We also give numbers to the symbols $S_1, \ldots, S_m$

and, in particular, blank $= S_0$, $0 = S_1$, $1 = S_2$. The lines of the table are now of form

m-config.	*Symbol*	*Operations*	*Final m-config.*	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

Lines such as

q_i	S_j	E, R	q_m	

are to be written as

q_i	S_j	PS_0, R	q_m	

and lines such as

q_i	S_j	R	q_m	

to be written as

q_i	S_j	PS_j, R	q_m	

In this way we reduce each line of the table to a line of one of the forms (N_1), (N_2), (N_3).

From each line of form (N_1) let us form an expression $q_i S_j S_k L q_m$; from each line of form (N_2) we form an expression $q_i S_j S_k R q_m$; and from each line of form (N_3) we form an expression $q_i S_j S_k N q_m$.

Let us write down all expressions so formed from the table for the machine and separate them by semi-colons. In this way we obtain a complete description of the machine. In this description we shall replace q_i by the letter "D" followed by the letter "A" repeated i times, and S_j by "D" followed by "C" repeated j times. This new description of the machine may be called the *standard description* (S.D). It is made up entirely from the letters "A", "C", "D", "L", "R", "N", and from ";".

If finally we replace "A" by "1", "C" by "2", "D" by "3", "L" by "4", "R" by "5", "N" by "6", and ";" by "7" we shall have a description of the machine in the form of an arabic numeral. The integer represented by this numeral may be called a *description number* (D.N) of the machine. The D.N determine the S.D and the structure of the

machine uniquely. The machine whose D.N is n may be described as $\mathcal{M}(n)$.

To each computable sequence there corresponds at least one description number, while to no description number does there correspond more than one computable sequence. The computable sequences and numbers are therefore enumerable.

Let us find a description number for the machine I of § 3. When we rename the m-configurations its table becomes:

q_1	S_0	PS_1, R	q_2
q_2	S_0	PS_0, R	q_3
q_3	S_0	PS_2, R	q_4
q_4	S_0	PS_0, R	q_1

Other tables could be obtained by adding irrelevant lines such as

q_1	S_1	PS_1, R	q_2

Our first standard form would be

$$q_1 S_0 S_1 R q_2;\ q_2 S_0 S_0 R q_3;\ q_3 S_0 S_2 R q_4;\ q_4 S_0 S_0 R q_1;.$$

The standard description is

DADDCRDAA;DAADDRDAAA;

DAAADDCCRDAAAA;DAAAADDRDA;

A description number is

3133253117311335311173111332253111173111133531 7

and so is

313325311731133531117311133225311117311113353173132325311 7

A number which is a description number of a circle-free machine will be called a *satisfactory* number. In § 8 it is shown that there can be no general process for determining whether a given number is satisfactory or not.

6. *The universal computing machine.*

It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$,

then $\mathfrak{U}$ will compute the same sequence as $\mathcal{M}$. In this section I explain in outline the behaviour of the machine. The next section is devoted to giving the complete table for $\mathfrak{U}$.

Let us first suppose that we have a machine $\mathcal{M}'$ which will write down on the F-squares the successive complete configurations of $\mathcal{M}$. These might be expressed in the same form as on p. 235, using the second description, (C), with all symbols on one line. Or, better, we could transform this description (as in § 5) by replacing each m-configuration by "D" followed by "A" repeated the appropriate number of times, and by replacing each symbol by "D" followed by "C" repeated the appropriate number of times. The numbers of letters "A" and "C" are to agree with the numbers chosen in § 5, so that, in particular, "0" is replaced by "DC", "1" by "DCC", and the blanks by "D". These substitutions are to be made after the complete configurations have been put together, as in (C). Difficulties arise if we do the substitution first. In each complete configuration the blanks would all have to be replaced by "D", so that the complete configuration would not be expressed as a finite sequence of symbols.

If in the description of the machine II of § 3 we replace "ə" by "DAA", "ɘ" by "$DCCC$", "q" by "$DAAA$", then the sequence (C) becomes:

$$DA:DCCCDCCCDAADCDDC:DCCCDCCCDAAADCDDC:\ldots \quad (C_1)$$

(This is the sequence of symbols on F-squares.)

It is not difficult to see that if $\mathcal{M}$ can be constructed, then so can $\mathcal{M}'$. The manner of operation of $\mathcal{M}'$ could be made to depend on having the rules of operation (*i.e.*, the S.D) of $\mathcal{M}$ written somewhere within itself (*i.e.* within $\mathcal{M}'$); each step could be carried out by referring to these rules. We have only to regard the rules as being capable of being taken out and exchanged for others and we have something very akin to the universal machine.

One thing is lacking: at present the machine $\mathcal{M}'$ prints no figures. We may correct this by printing between each successive pair of complete configurations the figures which appear in the new configuration but not in the old. Then (C_1) becomes

$$DDA:0:0:DCCCDCCCDAADCDDC:DCCC.\ldots \quad (C_2)$$

It is not altogether obvious that the E-squares leave enough room for the necessary "rough work", but this is, in fact, the case.

The sequences of letters between the colons in expressions such as (C_1) may be used as standard descriptions of the complete configurations. When the letters are replaced by figures, as in § 5, we shall have a numerical

description of the complete configuration, which may be called its description number.

7. *Detailed description of the universal machine.*

A table is given below of the behaviour of this universal machine. The m-configurations of which the machine is capable are all those occurring in the first and last columns of the table, together with all those which occur when we write out the unabbreviated tables of those which appear in the table in the form of m-functions. *E.g.*, $\mathfrak{e}(\mathfrak{anf})$ appears in the table and is an m-function. Its unabbreviated table is (see p. 239)

$\mathfrak{e}(\mathfrak{anf})$	$\mathfrak{d}$	R	$\mathfrak{e}_1(\mathfrak{anf})$
	not $\mathfrak{d}$	L	$\mathfrak{e}(\mathfrak{anf})$
$\mathfrak{e}_1(\mathfrak{anf})$	Any	R, E, R	$\mathfrak{e}_1(\mathfrak{anf})$
	None		$\mathfrak{anf}$

Consequently $\mathfrak{e}_1(\mathfrak{anf})$ is an m-configuration of $\mathfrak{U}$.

When $\mathfrak{U}$ is ready to start work the tape running through it bears on it the symbol $\mathfrak{d}$ on an F-square and again $\mathfrak{d}$ on the next E-square; after this, on F-squares only, comes the S.D of the machine followed by a double colon "::" (a single symbol, on an F-square). The S.D consists of a number of instructions, separated by semi-colons.

Each instruction consists of five consecutive parts

(i) "D" followed by a sequence of letters "A". This describes the relevant m-configuration.

(ii) "D" followed by a sequence of letters "C". This describes the scanned symbol.

(iii) "D" followed by another sequence of letters "C". This describes the symbol into which the scanned symbol is to be changed.

(iv) "L", "R", or "N", describing whether the machine is to move to left, right, or not at all.

(v) "D" followed by a sequence of letters "A". This describes the final m-configuration.

The machine $\mathfrak{U}$ is to be capable of printing "A", "C", "D", "0", "1", "u", "v", "w", "x", "y", "z". The S.D is formed from ";", "A", "C", "D", "L", "R", "N".

Subsidiary skeleton table.

$\mathfrak{con}(\mathfrak{C}, \alpha)$	Not A	R, R	$\mathfrak{con}(\mathfrak{C}, \alpha)$
	A	$L, P\alpha, R$	$\mathfrak{con}_1(\mathfrak{C}, \alpha)$
$\mathfrak{con}_1(\mathfrak{C}, \alpha)$	A	$R, P\alpha, R$	$\mathfrak{con}_1(\mathfrak{C}, \alpha)$
	D	$R, P\alpha, R$	$\mathfrak{con}_2(\mathfrak{C}, \alpha)$
$\mathfrak{con}_2(\mathfrak{C}, \alpha)$	C	$R, P\alpha, R$	$\mathfrak{con}_2(\mathfrak{C}, \alpha)$
	Not C	R, R	$\mathfrak{C}$

$\mathfrak{con}(\mathfrak{C}, \alpha)$. Starting from an F-square, S say, the sequence C of symbols describing a configuration closest on the right of S is marked out with letters α. $\rightarrow \mathfrak{C}$.

$\mathfrak{con}(\mathfrak{C},)$. In the final configuration the machine is scanning the square which is four squares to the right of the last square of C. C is left unmarked.

The table for $\mathfrak{U}$.

$\mathfrak{b}$			$\mathfrak{f}(\mathfrak{b}_1, \mathfrak{b}_1, ::)$
$\mathfrak{b}_1$	$R, R, P:, R, R, PD, R, R, PA$		$\mathfrak{anf}$
$\mathfrak{anf}$			$\mathfrak{g}(\mathfrak{anf}_1, :)$
$\mathfrak{anf}_1$			$\mathfrak{con}(\mathfrak{fom}, y)$
$\mathfrak{fom}$	;	R, Pz, L	$\mathfrak{con}(\mathfrak{fmp}, x)$
	z	L, L	$\mathfrak{fom}$
	not z nor ;	L	$\mathfrak{fom}$
$\mathfrak{fmp}$			$\mathfrak{cpe}\big(\mathfrak{e}(\mathfrak{fom}, x, y), \mathfrak{sim}, x, y\big)$

$\mathfrak{b}$. The machine prints $:DA$ on the F-squares after $:: \rightarrow \mathfrak{anf}$.

$\mathfrak{anf}$. The machine marks the configuration in the last complete configuration with y. $\rightarrow \mathfrak{fom}$.

$\mathfrak{fom}$. The machine finds the last semi-colon not marked with z. It marks this semi-colon with z and the configuration following it with x.

$\mathfrak{fmp}$. The machine compares the sequences marked x and y. It erases all letters x and y. $\rightarrow \mathfrak{sim}$ if they are alike. Otherwise $\rightarrow \mathfrak{fom}$.

$\mathfrak{anf}$. Taking the long view, the last instruction relevant to the last configuration is found. It can be recognised afterwards as the instruction following the last semi-colon marked z. $\rightarrow \mathfrak{sim}$.

$\mathfrak{sim}$ $\quad$ $\mathfrak{f}'(\mathfrak{sim}_1, \mathfrak{sim}_1, z)$

$\mathfrak{sim}_1$ $\quad$ $\mathrm{con}(\mathfrak{sim}_2,)$

$\mathfrak{sim}_2$

A		$\mathfrak{sim}_3$
not A	R, Pu, R, R, R	$\mathfrak{sim}_2$

$\mathfrak{sim}_3$

not A	L, Py	$\mathfrak{e}(\mathfrak{m}^{\mathfrak{k}}, z)$
A	L, Py, R, R, R	$\mathfrak{sim}_3$

$\mathfrak{sim}$. The machine marks out the instructions. That part of the instructions which refers to operations to be carried out is marked with u, and the final m-configuration with y. The letters z are erased.

$\mathfrak{m}^{\mathfrak{k}}$ $\quad$ $\mathfrak{g}(\mathfrak{m}^{\mathfrak{k}}_p, :)$

$\mathfrak{m}^{\mathfrak{k}}_1$

not A	R, R	$\mathfrak{m}^{\mathfrak{k}}_1$
A	L, L, L, L	$\mathfrak{m}^{\mathfrak{k}}_2$

$\mathfrak{m}^{\mathfrak{k}}_2$

C	R, Px, L, L, L	$\mathfrak{m}^{\mathfrak{k}}_2$
:		$\mathfrak{m}^{\mathfrak{k}}_4$
D	R, Px, L, L, L	$\mathfrak{m}^{\mathfrak{k}}_3$

$\mathfrak{m}^{\mathfrak{k}}_3$

not :	R, Pv, L, L, L	$\mathfrak{m}^{\mathfrak{k}}_3$
:		$\mathfrak{m}^{\mathfrak{k}}_4$

$\mathfrak{m}^{\mathfrak{k}}_4$ $\quad$ $\mathrm{con}\Big(\mathfrak{l}\big(\mathfrak{l}(\mathfrak{m}^{\mathfrak{k}}_5)\big), \Big)$

$\mathfrak{m}^{\mathfrak{k}}_5$

Any	R, Pw, R	$\mathfrak{m}^{\mathfrak{k}}_5$
None	$P:$	$\mathfrak{sh}$

$\mathfrak{m}^{\mathfrak{k}}$. The last complete configuration is marked out into four sections. The configuration is left unmarked. The symbol directly preceding it is marked with x. The remainder of the complete configuration is divided into two parts, of which the first is marked with v and the last with w. A colon is printed after the whole. $\rightarrow \mathfrak{sh}$.

$\mathfrak{sh}$ $\quad$ $\mathfrak{f}(\mathfrak{sh}_1, \mathfrak{inst}, u)$

$\mathfrak{sh}_1$ $\quad$ L, L, L $\quad$ $\mathfrak{sh}_2$

$\mathfrak{sh}_2$

D	R, R, R, R	$\mathfrak{sh}_2$
not D		$\mathfrak{inst}$

$\mathfrak{sh}_3$

C	R, R	$\mathfrak{sh}_4$
not C		$\mathfrak{inst}$

$\mathfrak{sh}_4$

C	R, R	$\mathfrak{sh}_5$
not C		$\mathfrak{pe}_2(\mathfrak{inst}, 0, :)$

$\mathfrak{sh}_5$

C		$\mathfrak{inst}$
not C		$\mathfrak{pe}_2(\mathfrak{inst}, 1, :)$

$\mathfrak{sh}$. The instructions (marked u) are examined. If it is found that they involve "Print 0" or "Print 1", then 0: or 1: is printed at the end.

inst			$\mathfrak{g}\left(\mathfrak{f}(\mathfrak{inst}_1), u\right)$
$\mathfrak{inst}_1$	α	R, E	$\mathfrak{inst}_1(\alpha)$
$\mathfrak{inst}_1(L)$			$\mathfrak{ce}_5(\mathfrak{ov}, v, y, x, u, w)$
$\mathfrak{inst}_1(R)$			$\mathfrak{ce}_5(\mathfrak{ov}, v, x, u, y, w)$
$\mathfrak{inst}_1(N)$			$\mathfrak{ce}_5(\mathfrak{ov}, v, x, y, u, w)$
$\mathfrak{ov}$			$\mathfrak{e}(\mathfrak{anf})$

inst. The next complete configuration is written down, carrying out the marked instructions. The letters u, v, w, x, y are erased. $\rightarrow$anf.

8. *Application of the diagonal process.*

It may be thought that arguments which prove that the real numbers are not enumerable would also prove that the computable numbers and sequences cannot be enumerable*. It might, for instance, be thought that the limit of a sequence of computable numbers must be computable. This is clearly only true if the sequence of computable numbers is defined by some rule.

Or we might apply the diagonal process. "If the computable sequences are enumerable, let α_n be the n-th computable sequence, and let $\phi_n(m)$ be the m-th figure in α_n. Let β be the sequence with $1-\phi_n(n)$ as its n-th figure. Since β is computable, there exists a number K such that $1-\phi_n(n) = \phi_K(n)$ all n. Putting $n = K$, we have $1 = 2\phi_K(K)$, *i.e.* 1 is even. This is impossible. The computable sequences are therefore not enumerable".

The fallacy in this argument lies in the assumption that β is computable. It would be true if we could enumerate the computable sequences by finite means, but the problem of enumerating computable sequences is equivalent to the problem of finding out whether a given number is the D.N of a circle-free machine, and we have no general process for doing this in a finite number of steps. In fact, by applying the diagonal process argument correctly, we can show that there cannot be any such general process.

The simplest and most direct proof of this is by showing that, if this general process exists, then there is a machine which computes β. This proof, although perfectly sound, has the disadvantage that it may leave the reader with a feeling that "there must be something wrong". The proof which I shall give has not this disadvantage, and gives a certain insight into the significance of the idea "circle-free". It depends not on constructing β, but on constructing β', whose n-th figure is $\phi_n(n)$.

* *Cf.* Hobson, *Theory of functions of a real variable* (2nd ed., 1921), 87, 88.

Let us suppose that there is such a process; that is to say, that we can invent a machine $\mathfrak{D}$ which, when supplied with the S.D of any computing machine $\mathfrak{M}$ will test this S.D and if $\mathfrak{M}$ is circular will mark the S.D with the symbol "u" and if it is circle-free will mark it with "s". By combining the machines $\mathfrak{D}$ and $\mathfrak{U}$ we could construct a machine $\mathfrak{H}$ to compute the sequence β'. The machine $\mathfrak{D}$ may require a tape. We may suppose that it uses the E-squares beyond all symbols on F-squares, and that when it has reached its verdict all the rough work done by $\mathfrak{D}$ is erased.

The machine $\mathfrak{H}$ has its motion divided into sections. In the first $N-1$ sections, among other things, the integers 1, 2, ..., $N-1$ have been written down and tested by the machine $\mathfrak{D}$. A certain number, say $R(N-1)$, of them have been found to be the D.N's of circle-free machines. In the N-th section the machine $\mathfrak{D}$ tests the number N. If N is satisfactory, *i.e.*, if it is the D.N of a circle-free machine, then $R(N) = 1 + R(N-1)$ and the first $R(N)$ figures of the sequence of which a D.N is N are calculated. The $R(N)$-th figure of this sequence is written down as one of the figures of the sequence β' computed by $\mathfrak{H}$. If N is not satisfactory, then $R(N) = R(N-1)$ and the machine goes on to the $(N+1)$-th section of its motion.

From the construction of $\mathfrak{H}$ we can see that $\mathfrak{H}$ is circle-free. Each section of the motion of $\mathfrak{H}$ comes to an end after a finite number of steps. For, by our assumption about $\mathfrak{D}$, the decision as to whether N is satisfactory is reached in a finite number of steps. If N is not satisfactory, then the N-th section is finished. If N is satisfactory, this means that the machine $\mathfrak{M}(N)$ whose D.N is N is circle-free, and therefore its $R(N)$-th figure can be calculated in a finite number of steps. When this figure has been calculated and written down as the $R(N)$-th figure of β', the N-th section is finished. Hence $\mathfrak{H}$ is circle-free.

Now let K be the D.N of $\mathfrak{H}$. What does $\mathfrak{H}$ do in the K-th section of its motion? It must test whether K is satisfactory, giving a verdict "s" or "u". Since K is the D.N of $\mathfrak{H}$ and since $\mathfrak{H}$ is circle-free, the verdict cannot be "u". On the other hand the verdict cannot be "s". For if it were, then in the K-th section of its motion $\mathfrak{H}$ would be bound to compute the first $R(K-1)+1 = R(K)$ figures of the sequence computed by the machine with K as its D.N and to write down the $R(K)$-th as a figure of the sequence computed by $\mathfrak{H}$. The computation of the first $R(K)-1$ figures would be carried out all right, but the instructions for calculating the $R(K)$-th would amount to "calculate the first $R(K)$ figures computed by H and write down the $R(K)$-th". This $R(K)$-th figure would never be found. *I.e.*, $\mathfrak{H}$ is circular, contrary both to what we have found in the last paragraph and to the verdict "s". Thus both verdicts are impossible and we conclude that there can be no machine $\mathfrak{D}$.

We can show further that *there can be no machine* $\mathfrak{E}$ *which, when supplied with the S.D of an arbitrary machine* $\mathcal{M}$, *will determine whether* $\mathcal{M}$ *ever prints a given symbol* (0 *say*).

We will first show that, if there is a machine $\mathfrak{E}$, then there is a general process for determining whether a given machine $\mathcal{M}$ prints 0 infinitely often. Let $\mathcal{M}_1$ be a machine which prints the same sequence as $\mathcal{M}$, except that in the position where the first 0 printed by $\mathcal{M}$ stands, $\mathcal{M}_1$ prints $\overline{0}$. $\mathcal{M}_2$ is to have the first two symbols 0 replaced by $\overline{0}$, and so on. Thus, if $\mathcal{M}$ were to print

$$A\,B\,A\,0\,1\,A\,A\,B\,0\,0\,1\,0\,A\,B\ldots,$$

then $\mathcal{M}_1$ would print

$$A\,B\,A\,\overline{0}\,1\,A\,A\,B\,0\,0\,1\,0\,A\,B\ldots$$

and $\mathcal{M}_2$ would print

$$A\,B\,A\,\overline{0}\,1\,A\,A\,B\,\overline{0}\,0\,1\,0\,A\,B\ldots.$$

Now let $\mathfrak{F}$ be a machine which, when supplied with the S.D of $\mathcal{M}$, will write down successively the S.D of $\mathcal{M}$, of $\mathcal{M}_1$, of $\mathcal{M}_2$, ... (there is such a machine). We combine $\mathfrak{F}$ with $\mathfrak{E}$ and obtain a new machine, $\mathcal{G}$. In the motion of $\mathcal{G}$ first $\mathfrak{F}$ is used to write down the S.D of $\mathcal{M}$, and then $\mathfrak{E}$ tests it, : 0 : is written if it is found that $\mathcal{M}$ never prints 0; then $\mathfrak{F}$ writes the S.D of $\mathcal{M}_1$, and this is tested, : 0 : being printed if and only if $\mathcal{M}_1$ never prints 0, and so on. Now let us test $\mathcal{G}$ with $\mathfrak{E}$. If it is found that $\mathcal{G}$ never prints 0, then $\mathcal{M}$ prints 0 infinitely often; if $\mathcal{G}$ prints 0 sometimes, then $\mathcal{M}$ does not print 0 infinitely often.

Similarly there is a general process for determining whether $\mathcal{M}$ prints 1 infinitely often. By a combination of these processes we have a process for determining whether $\mathcal{M}$ prints an infinity of figures, *i.e.* we have a process for determining whether $\mathcal{M}$ is circle-free. There can therefore be no machine $\mathfrak{E}$.

The expression "there is a general process for determining ..." has been used throughout this section as equivalent to "there is a machine which will determine ...". This usage can be justified if and only if we can justify our definition of "computable". For each of these "general process" problems can be expressed as a problem concerning a general process for determining whether a given integer n has a property $G(n)$ [*e.g.* $G(n)$ might mean "n is satisfactory" or "n is the Gödel representation of a provable formula"], and this is equivalent to computing a number whose n-th figure is 1 if $G(n)$ is true and 0 if it is false.

9. *The extent of the computable numbers.*

No attempt has yet been made to show that the "computable" numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in computing a number?"

The arguments which I shall use are of three kinds.

(*a*) A direct appeal to intuition.

(*b*) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(*c*) Giving examples of large classes of numbers which are computable.

Once it is granted that computable numbers are all "computable", several other propositions of the same character follow. In particular, it follows that, if there is a general process for determining whether a formula of the Hilbert function calculus is provable, then the determination can be carried out by a machine.

I. [Type (*a*)]. This argument is only an elaboration of the ideas of § 1.

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent†. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as

† If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. If these sets are restricted to be measurable, we can define the "distance" between two symbols as the cost of transforming one symbol into the other if the cost of moving unit area of printer's ink unit distance is unity, and there is an infinite supply of ink at $x = 2$, $y = 0$. With this topology the symbols form a conditionally compact space.

17 or 99999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always "observed" squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with "immediate recognisability", it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as imme-

diately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find "... hence (applying Theorem 157767733443477) we have ...". In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. This idea is developed in III below.

The simple operations must therefore include:

(*a*) Changes of the symbol on one of the observed squares.

(*b*) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(*A*) A possible change (*a*) of symbol together with a possible change of state of mind.

(*B*) A possible change (*b*) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an "m-configuration" of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned

squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the m-configuration. The machines just described do not differ very essentially from computing machines as defined in § 2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer.

II. [Type (*b*)].

If the notation of the Hilbert functional calculus† is modified so as to be systematic, and so as to involve only a finite number of symbols, it becomes possible to construct an automatic‡ machine $\mathfrak{K}$, which will find all the provable formulae of the calculus§.

Now let α be a sequence, and let us denote by $G_\alpha(x)$ the proposition "The x-th figure of α is 1", so that‖ $-G_\alpha(x)$ means "The x-th figure of α is 0". Suppose further that we can find a set of properties which define the sequence α and which can be expressed in terms of $G_\alpha(x)$ and of the propositional functions $N(x)$ meaning "x is a non-negative integer" and $F(x, y)$ meaning "$y = x+1$". When we join all these formulae together conjunctively, we shall have a formula, $\mathfrak{A}$ say, which defines α. The terms of $\mathfrak{A}$ must include the necessary parts of the Peano axioms, viz.,

$$(\exists u)\, N(u) \,\&\, (x)\left(N(x) \rightarrow (\exists y)\, F(x, y)\right) \,\&\, \left(F(x, y) \rightarrow N(y)\right),$$

which we will abbreviate to P.

When we say "$\mathfrak{A}$ defines α", we mean that $-\mathfrak{A}$ is not a provable formula, and also that, for each n, one of the following formulae (A_n) or (B_n) is provable.

$$\mathfrak{A} \,\&\, F^{(n)} \rightarrow G_\alpha(u^{(n)}), \qquad (A_n)\P$$

$$\mathfrak{A} \,\&\, F^{(n)} \rightarrow \left(-G_\alpha(u^{(n)})\right), \qquad (B_n),$$

where $F^{(n)}$ stands for $F(u, u') \,\&\, F(u', u'') \,\&\, \ldots\, F(u^{(n-1)}, u^{(n)})$.

---

† The expression "the functional calculus" is used throughout to mean the *restricted* Hilbert functional calculus.

‡ It is most natural to construct first a choice machine (§ 2) to do this. But it is then easy to construct the required automatic machine. We can suppose that the choices are always choices between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices $i_1, i_2, \ldots, i_n$ ($i_1 = 0$ or 1, $i_2 = 0$ or 1, ..., $i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + \ldots + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ...

§ The author has found a description of such a machine.

‖ The negation sign is written before an expression and not over it.

¶ A sequence of r primes is denoted by $^{(r)}$.

I say that α is then a computable sequence: a machine $\mathfrak{K}_\alpha$ to compute α can be obtained by a fairly simple modification of $\mathfrak{K}$.

We divide the motion of $\mathfrak{K}_\alpha$ into sections. The n-th section is devoted to finding the n-th figure of α. After the $(n-1)$-th section is finished a double colon :: is printed after all the symbols, and the succeeding work is done wholly on the squares to the right of this double colon. The first step is to write the letter "A" followed by the formula (A_n) and then "B" followed by (B_n). The machine $\mathfrak{K}_\alpha$ then starts to do the work of $\mathfrak{K}$, but whenever a provable formula is found, this formula is compared with (A_n) and with (B_n). If it is the same formula as (A_n), then the figure "1" is printed, and the n-th section is finished. If it is (B_n), then "0" is printed and the section is finished. If it is different from both, then the work of $\mathfrak{K}$ is continued from the point at which it had been abandoned. Sooner or later one of the formulae (A_n) or (B_n) is reached; this follows from our hypotheses about α and $\mathfrak{A}$, and the known nature of $\mathfrak{K}$. Hence the n-th section will eventually be finished. $\mathfrak{K}_\alpha$ is circle-free; α is computable.

It can also be shown that the numbers α definable in this way by the use of axioms include all the computable numbers. This is done by describing computing machines in terms of the function calculus.

It must be remembered that we have attached rather a special meaning to the phrase "$\mathfrak{A}$ defines α". The computable numbers do not include all (in the ordinary sense) definable numbers. Let δ be a sequence whose n-th figure is 1 or 0 according as n is or is not satisfactory. It is an immediate consequence of the theorem of § 8 that δ is not computable. It is (so far as we know at present) possible that any assigned number of figures of δ can be calculated, but not by a uniform process. When sufficiently many figures of δ have been calculated, an essentially new method is necessary in order to obtain more figures.

III. This may be regarded as a modification of I or as a corollary of II.

We suppose, as in I, that the computation is carried out on a tape; but we avoid introducing the "state of mind" by considering a more physical and definite counterpart of it. It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. This note is the counterpart of the "state of mind". We will suppose that the computer works in such a desultory manner that he never does more than one step at a sitting. The note of instructions must enable him to carry out one step and write the next note. Thus the state of progress of the computation at any stage is completely determined by the note of

instructions and the symbols on the tape. That is, the state of the system may be described by a single expression (sequence of symbols), consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression may be called the "state formula". We know that the state formula at any given stage is determined by the state formula before the last step was made, and we assume that the relation of these two formulae is expressible in the functional calculus. In other words, we assume that there is an axiom $\mathfrak{A}$ which expresses the rules governing the behaviour of the computer, in terms of the relation of the state formula at any stage to the state formula at the preceding stage. If this is so, we can construct a machine to write down the successive state formulae, and hence to compute the required number.

10. *Examples of large classes of numbers which are computable.*

It will be useful to begin with definitions of a computable function of an integral variable and of a computable variable, etc. There are many equivalent ways of defining a computable function of an integral variable. The simplest is, possibly, as follows. If γ is a computable sequence in which 0 appears infinitely† often, and n is an integer, then let us define $\xi(\gamma, n)$ to be the number of figures 1 between the n-th and the $(n+1)$-th figure 0 in γ. Then $\phi(n)$ is computable if, for all n and some γ, $\phi(n) = \xi(\gamma, n)$. An equivalent definition is this. Let $H(x, y)$ mean $\phi(x) = y$. Then, if we can find a contradiction-free axiom $\mathfrak{A}_\phi$, such that $\mathfrak{A}_\phi \rightarrow P$, and if for each integer n there exists an integer N, such that

$$\mathfrak{A}_\phi \ \& \ F^{(N)} \rightarrow H(u^{(n)}, u^{(\phi(n))}),$$

and such that, if $m \neq \phi(n)$, then, for some N',

$$\mathfrak{A}_\phi \ \& \ F^{(N')} \rightarrow \left(-H(u^{(n)}, u^{(m)}\right),$$

then ϕ may be said to be a computable function.

We cannot define general computable functions of a real variable, since there is no general method of describing a real number, but we can define a computable function of a computable variable. If n is satisfactory, let γ_n be the number computed by $\mathcal{M}(n)$, and let

$$\alpha_n = \tan\left(\pi(\gamma_n - \tfrac{1}{2})\right),$$

† If $\mathcal{M}$ computes γ, then the problem whether $\mathcal{M}$ prints 0 infinitely often is of the same character as the problem whether $\mathcal{M}$ is circle-free.

unless $\gamma_n = 0$ or $\gamma_n = 1$, in either of which cases $\alpha_n = 0$. Then, as n runs through the satisfactory numbers, α_n runs through the computable numbers†. Now let $\phi(n)$ be a computable function which can be shown to be such that for any satisfactory argument its value is satisfactory‡. Then the function f, defined by $f(\alpha_n) = \alpha_{\phi(n)}$, is a computable function and all computable functions of a computable variable are expressible in this form.

Similar definitions may be given of computable functions of several variables, computable-valued functions of an integral variable, etc.

I shall enunciate a number of theorems about computability, but I shall prove only (ii) and a theorem similar to (iii).

(i) A computable function of a computable function of an integral or computable variable is computable.

(ii) Any function of an integral variable defined recursively in terms of computable functions is computable. *I.e.* if $\phi(m, n)$ is computable, and r is some integer, then $\eta(n)$ is computable, where

$$\eta(0) = r,$$

$$\eta(n) = \phi\big(n, \eta(n-1)\big).$$

(iii) If $\phi(m, n)$ is a computable function of two integral variables, then $\phi(n, n)$ is a computable function of n.

(iv) If $\phi(n)$ is a computable function whose value is always 0 or 1, then the sequence whose n-th figure is $\phi(n)$ is computable.

Dedekind's theorem does not hold in the ordinary form if we replace "real" throughout by "computable". But it holds in the following form:

(v) If $G(\alpha)$ is a propositional function of the computable numbers and

$$(a) \quad (\exists \alpha)(\exists \beta)\{G(\alpha) \,\&\, (-G(\beta))\},$$

$$(b) \quad G(\alpha) \,\&\, (-G(\beta)) \rightarrow (\alpha < \beta),$$

and there is a general process for determining the truth value of $G(\alpha)$, then

† A function α_n may be defined in many other ways so as to run through the computable numbers.

‡ Although it is not possible to find a general process for determining whether a given number is satisfactory, it is often possible to show that certain classes of numbers are satisfactory.

there is a computable number ξ such that

$$G(\alpha) \to \alpha \leqslant \xi,$$

$$-G(\alpha) \to \alpha \geqslant \xi.$$

In other words, the theorem holds for any section of the computables such that there is a general process for determining to which class a given number belongs.

Owing to this restriction of Dedekind's theorem, we cannot say that a computable bounded increasing sequence of computable numbers has a computable limit. This may possibly be understood by considering a sequence such as

$$-1, -\tfrac{1}{2}, -\tfrac{1}{4}, -\tfrac{1}{8}, -\tfrac{1}{16}, \tfrac{1}{2}, \ldots.$$

On the other hand, (v) enables us to prove

(vi) If α and β are computable and $\alpha < \beta$ and $\phi(\alpha) < 0 < \phi(\beta)$, where $\phi(\alpha)$ is a computable increasing continuous function, then there is a unique computable number γ, satisfying $\alpha < \gamma < \beta$ and $\phi(\gamma) = 0$.

Computable convergence.

We shall say that a sequence β_n of computable numbers *converges computably* if there is a computable integral valued function $N(\epsilon)$ of the computable variable ϵ, such that we can show that, if $\epsilon > 0$ and $n > N(\epsilon)$ and $m > N(\epsilon)$, then $|\beta_n - \beta_m| < \epsilon$.

We can then show that

(vii) A power series whose coefficients form a computable sequence of computable numbers is computably convergent at all computable points in the interior of its interval of convergence.

(viii) The limit of a computably convergent sequence is computable.

And with the obvious definition of "uniformly computably convergent":

(ix) The limit of a uniformly computably convergent computable sequence of computable functions is a computable function. Hence

(x) The sum of a power series whose coefficients form a computable sequence is a computable function in the interior of its interval of convergence.

From (viii) and $\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \ldots)$ we deduce that π is computable.

From $e = 1 + 1 + \dfrac{1}{2!} + \dfrac{1}{3!} + \ldots$ we deduce that e is computable.

From (vi) we deduce that all real algebraic numbers are computable.

From (vi) and (x) we deduce that the real zeros of the Bessel functions are computable.

Proof of (ii).

Let $H(x, y)$ mean "$\eta(x) = y$", and let $K(x, y, z)$ mean "$\phi(x, y) = z$". $\mathfrak{A}_\phi$ is the axiom for $\phi(x, y)$. We take $\mathfrak{A}_\eta$ to be

$$\mathfrak{A}_\phi \,\&\, P \,\&\, \Big(F(x, y) \to G(x, y)\Big) \,\&\, \Big(G(x, y) \,\&\, G(y, z) \to G(x, z)\Big)$$

$$\&\, \Big(F^{(r)} \to H(u, u^{(r)})\Big) \,\&\, \Big(F(v, w) \,\&\, H(v, x) \,\&\, K(w, x, z) \to H(w, z)\Big)$$

$$\&\, \Big[H(w, z) \,\&\, G(z, t) \vee G(t, z) \to \Big(-H(w, t)\Big)\Big].$$

I shall not give the proof of consistency of $\mathfrak{A}_\eta$. Such a proof may be constructed by the methods used in Hilbert and Bernays, *Grundlagen der Mathematik* (Berlin, 1934), p. 209 *et seq.* The consistency is also clear from the meaning.

Suppose that, for some n, N, we have shown

$$\mathfrak{A}_\eta \,\&\, F^{(N)} \to H(u^{(n-1)}, u^{(\eta(n-1))}),$$

then, for some M,

$$\mathfrak{A}_\phi \,\&\, F^{(M)} \to K(u^{(n)}, u^{(\eta(n-1))}, u^{(\eta(n))}),$$

$$\mathfrak{A}_\eta \,\&\, F^{(M)} \to F(u^{(n-1)}, u^{(n)}) \,\&\, H(u^{(n-1)}, u^{(\eta(n-1))})$$

$$\&\, K(u^{(n)}, u^{(\eta(n-1))}, u^{(\eta(n))}),$$

and

$$\mathfrak{A}_\eta \,\&\, F^{(M)} \to [F(u^{(n-1)}, u^{(n)}) \,\&\, H(u^{(n-1)}, u^{(\eta(n-1))})$$

$$\&\, K(u^{(n)}, u^{(\eta(n-1))}, u^{(\eta(n))}) \to H(u^{(n)}, u^{(\eta(n))})].$$

Hence $$\mathfrak{A}_\eta \,\&\, F^{(M)} \to H(u^{(n)}, u^{(\eta(n))}).$$

Also $$\mathfrak{A}_\eta \,\&\, F^{(r)} \to H(u, u^{(\eta(0))}).$$

Hence for each n some formula of the form

$$\mathfrak{A}_\eta \,\&\, F^{(M)} \to H(u^{(n)}, u^{(\eta(n))})$$

is provable. Also, if $M' \geqslant M$ and $M' \geqslant m$ and $m \neq \eta(u)$, then

$$\mathfrak{A}_\eta \,\&\, F^{(M')} \to G(u^{\eta((n))}, u^{(m)}) \vee G(u^{(m)}, u^{(\eta(n))})$$

and

$$\mathfrak{U}_\eta \,\&\, F^{(M')} \to \Big[\{G(u^{(\eta(n))}, u^{(m)}) \vee G(u^{(m)}, u^{(\eta(n)}) \\ \& H(u^{(n)}, u^{(\eta(n))}\} \to \big(-H(u^{(n)}, u^{(m)})\big)\Big].$$

Hence $$\mathfrak{U}_\eta \,\&\, F^{(M')} \to \big(-H(u^{(n)}, u^{(m)})\big).$$

The conditions of our second definition of a computable function are therefore satisfied. Consequently η is a computable function.

Proof of a modified form of (iii).

Suppose that we are given a machine $\mathfrak{N}$, which, starting with a tape bearing on it ə ə followed by a sequence of any number of letters "F" on F-squares and in the m-configuration b, will compute a sequence γ_n depending on the number n of letters "F". If $\phi_n(m)$ is the m-th figure of γ_n, then the sequence β whose n-th figure is $\phi_n(n)$ is computable.

We suppose that the table for $\mathfrak{N}$ has been written out in such a way that in each line only one operation appears in the operations column. We also suppose that Ξ, Θ, $\overline{0}$, and $\overline{1}$ do not occur in the table, and we replace ə throughout by Θ, 0 by $\overline{0}$, and 1 by $\overline{1}$. Further substitutions are then made. Any line of form

$\mathfrak{A}$	a	$P\overline{0}$	$\mathfrak{B}$

we replace by

$\mathfrak{A}$	a	$P\overline{0}$	$\mathfrak{re}(\mathfrak{B}, \mathfrak{u}, h, k)$

and any line of the form

$\mathfrak{A}$	a	$P\overline{1}$	$\mathfrak{B}$

by

$\mathfrak{A}$	a	$P\overline{1}$	$\mathfrak{re}(\mathfrak{B}, \mathfrak{v}, h, k)$

and we add to the table the following lines:

$\mathfrak{u}$		$\mathfrak{pe}(\mathfrak{u}_1, 0)$
$\mathfrak{u}_1$	$R, Pk, R, P\Theta, R, P\Theta$	$\mathfrak{u}_2$
$\mathfrak{u}_2$		$\mathfrak{re}(\mathfrak{u}_3, \mathfrak{u}_3, k, h)$
$\mathfrak{u}_3$		$\mathfrak{pe}(\mathfrak{u}_2, F)$

and similar lines with $\mathfrak{v}$ for $\mathfrak{u}$ and 1 for 0 together with the following line

$\mathfrak{c}$	$R, P\Xi, R, Ph$	$\mathfrak{b}$.

We then have the table for the machine $\mathfrak{N}'$ which computes β. The initial m-configuration is $\mathfrak{c}$, and the initial scanned symbol is the second ə.

11. *Application to the Entscheidungsproblem.*

The results of § 8 have some important applications. In particular, they can be used to show that the Hilbert Entscheidungsproblem can have no solution. For the present I shall confine myself to proving this particular theorem. For the formulation of this problem I must refer the reader to Hilbert and Ackermann's *Grundzüge der Theoretischen Logik* (Berlin, 1931), chapter 3.

I propose, therefore, to show that there can be no general process for determining whether a given formula $\mathfrak{A}$ of the functional calculus **K** is provable, *i.e.* that there can be no machine which, supplied with any one $\mathfrak{A}$ of these formulae, will eventually say whether $\mathfrak{A}$ is provable.

It should perhaps be remarked that what I shall prove is quite different from the well-known results of Gödel†. Gödel has shown that (in the formalism of Principia Mathematica) there are propositions $\mathfrak{A}$ such that neither $\mathfrak{A}$ nor $-\mathfrak{A}$ is provable. As a consequence of this, it is shown that no proof of consistency of Principia Mathematica (or of **K**) can be given within that formalism. On the other hand, I shall show that there is no general method which tells whether a given formula $\mathfrak{A}$ is provable in **K**, or, what comes to the same, whether the system consisting of **K** with $-\mathfrak{A}$ adjoined as an extra axiom is consistent.

If the negation of what Gödel has shown had been proved, *i.e.* if, for each $\mathfrak{A}$, either $\mathfrak{A}$ or $-\mathfrak{A}$ is provable, then we should have an immediate solution of the Entscheidungsproblem. For we can invent a machine $\mathfrak{K}$ which will prove consecutively all provable formulae. Sooner or later $\mathfrak{K}$ will reach either $\mathfrak{A}$ or $-\mathfrak{A}$. If it reaches $\mathfrak{A}$, then we know that $\mathfrak{A}$ is provable. If it reaches $-\mathfrak{A}$, then, since **K** is consistent (Hilbert and Ackermann, p. 65), we know that $\mathfrak{A}$ is not provable.

Owing to the absence of integers in **K** the proofs appear somewhat lengthy. The underlying ideas are quite straightforward.

Corresponding to each computing machine $\mathcal{M}$ we construct a formula Un ($\mathcal{M}$) and we show that, if there is a general method for determining whether Un ($\mathcal{M}$) is provable, then there is a general method for determining whether $\mathcal{M}$ ever prints 0.

The interpretations of the propositional functions involved are as follows :

$R_{S_i}(x, y)$ is to be interpreted as "in the complete configuration x (of $\mathcal{M}$) the symbol on the square y is S".

† *Loc. cit.*

$I(x, y)$ is to be interpreted as "in the complete configuration x the square y is scanned".

$K_{q_m}(x)$ is to be interpreted as "in the complete configuration x the m-configuration is q_m.

$F(x, y)$ is to be interpreted as "y is the immediate successor of x".

Inst $\{q_i S_j S_k L q_l\}$ is to be an abbreviation for

$$(x, y, x', y') \Big\{ \Big(R_{S_j}(x, y) \,\&\, I(x, y) \,\&\, K_{q_i}(x) \,\&\, F(x, x') \,\&\, F(y', y) \Big) \rightarrow \Big(I(x', y') \,\&\, R_{S_k}(x', y) \,\&\, K_{q_l}(x') \,\&\, (z) \Big[F(y', z) \vee \Big(R_{S_j}(x, z) \rightarrow R_{S_k}(x', z) \Big) \Big] \Big) \Big\}.$$

Inst $\{q_i S_j S_k R q_l\}$ and Inst $\{q_i S_j S_k N q_l\}$

are to be abbreviations for other similarly constructed expressions.

Let us put the description of $\mathcal{M}$ into the first standard form of § 6. This description consists of a number of expressions such as "$q_i S_j S_k L q_l$" (or with R or N substituted for L). Let us form all the corresponding expressions such as Inst $\{q_i S_j S_k L q_l\}$ and take their logical sum. This we call Des $(\mathcal{M})$.

The formula Un $(\mathcal{M})$ is to be

$$(\exists u) \Big[N(u) \,\&\, (x) \Big(N(x) \rightarrow (\exists x') F(x, x') \Big) \,\&\, (y, z) \Big(F(y, z) \rightarrow N(y) \,\&\, N(z) \Big) \,\&\, (y) R_{S_0}(u, y) \,\&\, I(u, u) \,\&\, K_{q_1}(u) \,\&\, \mathrm{Des}(\mathcal{M}) \Big] \rightarrow (\exists s)(\exists t) [N(s) \,\&\, N(t) \,\&\, R_{S_1}(s, t)].$$

$[N(u) \,\&\, \ldots \,\&\, \mathrm{Des}\,(\mathcal{M})]$ may be abbreviated to $A(\mathcal{M})$.

When we substitute the meanings suggested on p. 259–60 we find that Un $(\mathcal{M})$ has the interpretation "in some complete configuration of $\mathcal{M}$, S_1 (*i.e.* 0) appears on the tape". Corresponding to this I prove that

(*a*) If S_1 appears on the tape in some complete configuration of $\mathcal{M}$, then Un $(\mathcal{M})$ is provable.

(*b*) If Un $(\mathcal{M})$ is provable, then S_1 appears on the tape in some complete configuration of $\mathcal{M}$.

When this has been done, the remainder of the theorem is trivial.

LEMMA 1. *If S_1 appears on the tape in some complete configuration of $\mathcal{M}$, then* $\mathrm{Un}(\mathcal{M})$ *is provable.*

We have to show how to prove $\mathrm{Un}(\mathcal{M})$. Let us suppose that in the n-th complete configuration the sequence of symbols on the tape is $S_{r(n,0)}, S_{r(n,1)}, \ldots, S_{r(n,n)}$, followed by nothing but blanks, and that the scanned symbol is the $i(n)$-th, and that the m-configuration is $q_{k(n)}$. Then we may form the proposition

$$R_{S_{r(n,0)}}(u^{(n)}, u) \,\&\, R_{S_{r(n,1)}}(u^{(n)}, u') \,\&\, \ldots \,\&\, R_{S_{r(n,n)}}(u^{(n)}, u^{(n)})$$
$$\&\, I(u^{(n)}, u^{(i(n))}) \,\&\, K_{q_{k(n)}}(u^{(n)})$$
$$\&\, (y)F\big((y, u') \vee F(u, y) \vee F(u', y) \vee \ldots \vee F(u^{(n-1)}, y) \vee R_{S_0}(u^{(n)}, y)\big),$$

which we may abbreviate to CC_n.

As before, $F(u, u') \,\&\, F(u', u'') \,\&\, \ldots \,\&\, F(u^{(r-1)}, u^{(r)})$ is abbreviated to $F^{(r)}$.

I shall show that all formulae of the form $A(\mathcal{M}) \,\&\, F^{(n)} \rightarrow CC_n$ (abbreviated to CF_n) are provable. The meaning of CF_n is "The n-th complete configuration of $\mathcal{M}$ is so and so", where "so and so" stands for the actual n-th complete configuration of $\mathcal{M}$. That CF_n should be provable is therefore to be expected.

CF_0 is certainly provable, for in the complete configuration the symbols are all blanks, the m-configuration is q_1, and the scanned square is u, *i.e.* CC_0 is

$$(y)\, R_{S_0}(u, y) \,\&\, I(u, u) \,\&\, K_{q_1}(u).$$

$A(\mathcal{M}) \rightarrow CC_0$ is then trivial.

We next show that $CF_n \rightarrow CF_{n+1}$ is provable for each n. There are three cases to consider, according as in the move from the n-th to the $(n+1)$-th configuration the machine moves to left or to right or remains stationary. We suppose that the first case applies, *i.e.* the machine moves to the left. A similar argument applies in the other cases. If $r\big(n, i(n)\big) = a$, $r\big(n+1, i(n+1)\big) = c$, $k\big(i(n)\big) = b$, and $k\big(i(n+1)\big) = d$, then $\mathrm{Des}(\mathcal{M})$ must include $\mathrm{Inst}\{q_a S_b S_d L q_c\}$ as one of its terms, *i.e.*

$$\mathrm{Des}(\mathcal{M}) \rightarrow \mathrm{Inst}\{q_a S_b S_d L q_c\}.$$

Hence $$A(\mathcal{M}) \,\&\, F^{(n+1)} \rightarrow \mathrm{Inst}\{q_a S_b S_d L q_c\} \,\&\, F^{(n+1)}.$$

But $$\mathrm{Inst}\{q_a S_b S_d L q_c\} \,\&\, F^{(n+1)} \rightarrow (CC_n \rightarrow CC_{n+1})$$

is provable, and so therefore is

$$A(\mathcal{M}) \,\&\, F^{(n+1)} \rightarrow (CC_n \rightarrow CC_{n+1})$$

and $$\left(A(\mathcal{M}) \,\&\, F^{(n)} \to CC_n\right) \to \left(A(\mathcal{M}) \,\&\, F^{(n+1)} \to CC_{n+1}\right),$$

i.e. $$CF_n \to CF_{n+1}.$$

CF_n is provable for each n. Now it is the assumption of this lemma that S_1 appears somewhere, in some complete configuration, in the sequence of symbols printed by $\mathcal{M}$; that is, for some integers N, K, CC_N has $R_{S_1}(u^{(N)}, u^{(K)})$ as one of its terms, and therefore $CC_N \to R_{S_1}(u^{(N)}, u^{(K)})$ is provable. We have then

$$CC_N \to R_{S_1}(u^{(N)}, u^{(K)})$$

and $$A(\mathcal{M}) \,\&\, F^{(N)} \to CC^N.$$

We also have

$$(\exists u) A(\mathcal{M}) \to (\exists u)(\exists u') \ldots (\exists u^{(N')}) \left(A(\mathcal{M}) \,\&\, F^{(N)}\right),$$

where $N' = \max(N, K)$. And so

$$(\exists u) A(\mathcal{M}) \to (\exists u)(\exists u') \ldots (\exists u^{(N')}) R_{S_1}(u^{(N)}, u^{(K)}),$$

$$(\exists u) A(\mathcal{M}) \to (\exists u^{(N)})(\exists u^{(K)}) R_{S_1}(u^{(N)}, u^{(K)}),$$

$$(\exists u) A(\mathcal{M}) \to (\exists s)(\exists t) R_{S_1}(s, t),$$

i.e. $\mathrm{Un}(\mathcal{M})$ is provable.

This completes the proof of Lemma 1.

LEMMA 2. *If* $\mathrm{Un}(\mathcal{M})$ *is provable, then* S_1 *appears on the tape in some complete configuration of* $\mathcal{M}$.

If we substitute any propositional functions for function variables in a provable formula, we obtain a true proposition. In particular, if we substitute the meanings tabulated on pp. 259–260 in $\mathrm{Un}(\mathcal{M})$, we obtain a true proposition with the meaning "S_1 appears somewhere on the tape in some complete configuration of $\mathcal{M}$".

We are now in a position to show that the Entscheidungsproblem cannot be solved. Let us suppose the contrary. Then there is a general (mechanical) process for determining whether $\mathrm{Un}(\mathcal{M})$ is provable. By Lemmas 1 and 2, this implies that there is a process for determining whether $\mathcal{M}$ ever prints 0, and this is impossible, by § 8. Hence the Entscheidungsproblem cannot be solved.

In view of the large number of particular cases of solutions of the Entscheidungsproblem for formulae with restricted systems of quantors, it

is interesting to express $\mathrm{Un}(\mathcal{M})$ in a form in which all quantors are at the beginning. $\mathrm{Un}(\mathcal{M})$ is, in fact, expressible in the form

$$(u)(\exists x)(w)(\exists u_1)\ldots(\exists u_n)\mathfrak{B}, \tag{I}$$

where $\mathfrak{B}$ contains no quantors, and $n = 6$. By unimportant modifications we can obtain a formula, with all essential properties of $\mathrm{Un}(\mathcal{M})$, which is of form (I) with $n = 5$.

Added 28 *August*, 1936.

APPENDIX.

Computability and effective calculability

The theorem that all effectively calculable (λ-definable) sequences are computable and its converse are proved below in outline. It is assumed that the terms "well-formed formula" (W.F.F.) and "conversion" as used by Church and Kleene are understood. In the second of these proofs the existence of several formulae is assumed without proof; these formulae may be constructed straightforwardly with the help of, *e.g.*, the results of Kleene in "A theory of positive integers in formal logic", *American Journal of Math.*, 57 (1935), 153-173, 219-244.

The W.F.F. representing an integer n will be denoted by N_n. We shall say that a sequence γ whose n-th figure is $\phi_\gamma(n)$ is λ-definable or effectively calculable if $1+\phi_\gamma(u)$ is a λ-definable function of n, *i.e.* if there is a W.F.F. M_γ such that, for all integers n,

$$\{M_\gamma\}(N_n) \text{ conv } N_{\phi_\gamma(n)+1},$$

i.e. $\{M_\gamma\}(N_n)$ is convertible into $\lambda xy.x\big(x(y)\big)$ or into $\lambda xy.x(y)$ according as the n-th figure of λ is 1 or 0.

To show that every λ-definable sequence γ is computable, we have to show how to construct a machine to compute γ. For use with machines it is convenient to make a trivial modification in the calculus of conversion. This alteration consists in using x, x', x'', ... as variables instead of $a, b, c, \ldots$. We now construct a machine $\mathfrak{L}$ which, when supplied with the formula M_γ, writes down the sequence γ. The construction of $\mathfrak{L}$ is somewhat similar to that of the machine $\mathfrak{K}$ which proves all provable formulae of the functional calculus. We first construct a choice machine $\mathfrak{L}_1$, which, if supplied with a W.F.F., M say, and suitably manipulated, obtains any formula into which M is convertible. $\mathfrak{L}_1$ can then be modified so as to yield an automatic machine $\mathfrak{L}_2$ which obtains successively all the formulae

into which M is convertible (cf. foot-note p. 252). The machine $\mathfrak{L}$ includes $\mathfrak{L}_2$ as a part. The motion of the machine $\mathfrak{L}$ when supplied with the formula M_γ is divided into sections of which the n-th is devoted to finding the n-th figure of γ. The first stage in this n-th section is the formation of $\{M_\gamma\}(N_n)$. This formula is then supplied to the machine $\mathfrak{L}_2$, which converts it successively into various other formulae. Each formula into which it is convertible eventually appears, and each, as it is found, is compared with

$$\lambda x\left[\lambda x'\left[\{x\}\left(\{x\}(x')\right)\right]\right], \quad i.e.\ N_2,$$

and with

$$\lambda x\left[\lambda x'[\{x\}(x')]\right], \quad i.e.\ N_1.$$

If it is identical with the first of these, then the machine prints the figure 1 and the n-th section is finished. If it is identical with the second, then 0 is printed and the section is finished. If it is different from both, then the work of $\mathfrak{L}_2$ is resumed. By hypothesis, $\{M_\gamma\}(N_n)$ is convertible into one of the formulae N_2 or N_1; consequently the n-th section will eventually be finished, *i.e.* the n-th figure of γ will eventually be written down.

To prove that every computable sequence γ is λ-definable, we must show how to find a formula M_γ such that, for all integers n,

$$\{M_\gamma\}(N_n) \text{ conv } N_{1+\phi_\gamma(n)}.$$

Let $\mathcal{M}$ be a machine which computes γ and let us take some description of the complete configurations of $\mathcal{M}$ by means of numbers, *e.g.* we may take the D.N of the complete configuration as described in § 6. Let $\xi(n)$ be the D.N of the n-th complete configuration of $\mathcal{M}$. The table for the machine $\mathcal{M}$ gives us a relation between $\xi(n+1)$ and $\xi(n)$ of the form

$$\xi(n+1) = \rho_\gamma\left(\xi(n)\right),$$

where ρ_γ is a function of very restricted, although not usually very simple, form: it is determined by the table for $\mathcal{M}$. ρ_γ is λ-definable (I omit the proof of this), *i.e.* there is a W.F.F. A_γ such that, for all integers n,

$$\{A_\gamma\}(N_{\xi(n)}) \text{ conv } N_{\xi(n+1)}.$$

Let U stand for

$$\lambda u\left[\left\{\{u\}(A_\gamma)\right\}(N_r)\right],$$

where $r = \xi(0)$; then, for all integers n,

$$\{U_\gamma\}(N_n) \text{ conv } N_{\xi(n)}.$$

It may be proved that there is a formula V such that

$$\{\{V\}(N_{\xi(n+1)})\}(N_{\xi(n)}) \begin{cases} \text{conv } N_1 & \text{if, in going from the } n\text{-th to the } (n+1)\text{-th complete configuration, the figure 0 is printed.} \\ \text{conv } N_2 & \text{if the figure 1 is printed.} \\ \text{conv } N_3 & \text{otherwise.} \end{cases}$$

Let W_γ stand for

$$\lambda u\left[\left\{\{V\}\left(\{A_\gamma\}\left(\{U_\gamma\}(u)\right)\right)\right\}\left(\{U_\gamma\}(u)\right)\right],$$

so that, for each integer n,

$$\left\{\{V\}(N_{\xi(n+1)})\right\}(N_{\xi(n)}) \text{ conv } \{W_\gamma\}(N_n),$$

and let Q be a formula such that

$$\left\{\{Q\}(W_\gamma)\right\}(N_s) \text{ conv } N_{r(s)},$$

where $r(s)$ is the s-th integer q for which $\{W_\gamma\}(N_q)$ is convertible into either N_1 or N_2. Then, if M_γ stands for

$$\lambda w\left[\{W_\gamma\}\left(\left\{\{Q\}(W_\gamma)\right\}(w)\right)\right],$$

it will have the required property†.

The Graduate College,
Princeton University,
New Jersey, U.S.A.

---

† In a complete proof of the λ-definability of computable sequences it would be best to modify this method by replacing the numerical description of the complete configurations by a description which can be handled more easily with our apparatus. Let us choose certain integers to represent the symbols and the m-configurations of the machine. Suppose that in a certain complete configuration the numbers representing the successive symbols on the tape are $s_1 s_2 \ldots s_n$, that the m-th symbol is scanned, and that the m-configuration has the number t; then we may represent this complete configuration by the formula

$$\left[[N_{s_1}, N_{s_2}, \ldots, N_{s_{m-1}}],\ [N_t, N_{s_m}],\ [N_{s_{m+1}}, \ldots, N_{s_n}]\right],$$

where $[a, b]$ stands for $\lambda u\left[\left\{\{u\}(a)\right\}(b)\right]$,

$[a, b, c]$ stands for $\lambda u\left[\left\{\left\{\{u\}(a)\right\}(b)\right\}(c)\right]$,

etc.